#1: Desk Accessories and System Resources

See also: The Resource Manager

Written by: Bryan Stearns February 25, 1985
Updated: March 1, 1988

This note formerly described a strategy for dealing with system resources from desk accessories. We no longer recommend calling `ReleaseResource` or `DetachResource` for a system resource. When you are done with a system resource, leave it alone; do not try to dispose or release it.

#2: Compatibility Guidelines

Written by: Cary Clark January 21, 1986
Scott Knaster
Modified by: Louella Pizzuti February 9, 1987
Updated: March 1, 1988

Apple has many enhancements planned for the Macintosh family of computers. To help ensure your software's compatibility with these enhancements, check each item in this note to be sure that you're following the recommendations.

If your software is written in a high-level language like Pascal or C and if you adhere to the guidelines listed in *Inside Macintosh*, many of the questions in this note won't concern you. If you develop in assembly language, you should read each question carefully. If you answer any question "yes," your software may encounter difficulty running on future Macintosh computers, and you should take the recommended action to change your software.

Do you depend on 68000 instructions which require that the processor be in supervisor mode?

In general, your software should not include instructions which depend on supervisor mode. These include modifying the contents of the status register. Most programs which modify the status register are only changing the Condition Code Register (CCR) half of the status register, so an instruction which addresses the CCR will work fine. Also, your software should not use the User Stack Pointer (USP) or turn interrupts on and off.

Do you have code which executes in response to an exception and relies on the position of data in the exception's local stack frame?

Exception stack frames vary on different microprocessors in the 68000 family, some of which may be used in future Macintosh computers. You should avoid using the `TRAP` instruction. **Note:** You can determine which microprocessor is installed by examining the low-memory global `CPUFlag` (a byte at `$12F`). These are the values:

CPUFlag	microprocessor
$00	68000
$01	68010
$02	68020
$03	68030

Do you use low-memory globals not documented in *Inside Macintosh*?

Other microprocessors in the 68000 family use the exception vectors in locations `$0` through `$FF` in different ways. No undocumented location below the system heap (`$100` through `$13FF`) is guaranteed to be available for use in future systems.

Do you make assumptions about the file system which are not consistent with both the original Macintosh File System and the Hierarchical File System?

Your applications should be compatible with both file systems. The easiest way to do this is to stick to the old files system trap calls (which work with both file systems) and avoid direct manipulation of data structures such as file control blocks and volume control blocks whenever possible.

Do you depend on the system or application heaps starting at a hard-coded address?

The starting addresses and the size of the system and application heaps has already changed (Macintosh vs. Macintosh Plus) and will change again in the future. Use the global `ApplZone` to find the application heap and `SysZone` to find the system heap. Also, don't count on the application heap zone starting at an address less than 65536 (that is, a system heap smaller than 64K).

Do you look through the system's queues directly?

In general, you should avoid examining queue elements directly. Instead, use the Operating System calls to manipulate queue elements.

Do you directly address memory-mapped hardware such as the VIA, the SCC, or the IWM?

You should avoid accessing this memory directly and use trap calls instead (disk driver, serial driver, etc.). Future machines may include a memory management unit (MMU) which may prevent access to memory-mapped hardware. Also, these memory-mapped devices may not be present on future machines. The addresses of these devices are likely to change, so if you must access the hardware directly, get the base address of the device from the appropriate low-memory global (obtainable from includes and interface files):

device	**global**
VIA	$1D4
SCCRd	$1D8
SCCWr	$1DC
IWM	$1E0

Do you assume the location or size of the screen?

The location, size, and bit depth of the screen is different in various machines. You can determine its location and size by examining the QuickDraw global variable `screenBits` on machines without Color QuickDraw. On machines with Color QuickDraw, the device list, described in the Graphics Devices chapter of *Inside Macintosh*, tells the location and size and bit depth of each screen, `screenBits` contains the location and size of the main device, and `GrayRgn` contains a region describing the shape and size of the desktop.

Does your software fail on some Macintosh models or on A/UX?

If so, you should determine the reason. Failure to run on all versions of the Macintosh may indicate problems which will prevent your software from working on future machines. Failture to run on A/UX, Apple's Unix for the Macintosh, also may indicate such problems.

Do you change master pointer flags of relocatable blocks directly with BSET or BCLR instructions?

In the future and on A/UX, all 32 bits of a master pointer may be used, with the flags byte moved elsewhere. Use the Memory Manager calls `HPurge`, `HNoPurge`, `HLock`, `HUnlock`, `HSetRBit`, `HClrRBit`, `HGetState`, and `HSetState` to manipulate the master pointer flags. (See the Memory Manager chapter of *Inside Macintosh Volume IV* for information on these calls.)

Do you check for 128K, 512K, and 1M RAM sizes?

You should be flexible enough to allow for non-standard memory sizes. This will allow your software to work in environments like MultiFinder.

Is your software incompatible with a third-party vendor's hardware?

If so, the incompatibility may prevent your software from working on future machines. You should research the incompatibility and try to determine a solution.

Do you rely on system resources being in RAM?

On most of our systems, some system resources are in ROM. You should not assume, for example, that you can regain RAM space by releasing system resources.

Does your software have timing-sensitive code?

Various Macintoshes run at different clock speeds, so timing loops will be invalid. You can use the trap call `Delay` for timing, or you can examine the global variable `Ticks`.

Do you have code which writes to addresses within the code itself?

A memory management unit (MMU) may one day prevent code from writing to addresses within code memory. Also, some microprocessors in the 68000 family cache code as it's encountered. Your data blocks should be allocated on the stack or in heap blocks separate from the code, and your code should not modify itself.

Do you rely on keyboard key codes rather than ASCII codes?

The various keyboards are slightly different; future keyboards may be different from them. For textual input, you should read ASCII codes rather than key codes.

Do you rely on the format of packed addresses in the trap dispatch table?

The trap dispatch table is different on various Macintoshes. There's no guarantee of the trap table's format in the future. You should use the system calls `GetTrapAddress` and `SetTrapAddress` to manipulate the trap dispatch table.

Do you use the Resource Manager calls `AddReference` or `RmveReference`?

These calls have been removed from the 128K ROM. They are no longer supported.

Do you store information in the application parameters area (the 32 bytes between the application and unit globals and the jump table)?

This space is reserved for use by Apple.

Do you depend on values in registers after a trap call, other than those documented in *Inside Macintosh*?

These values aren't guaranteed. The register conventions documented in *Inside Macintosh* will, of course, be supported. Often, you may not realize that your code is depending on these undocumented values, so check your register usage carefully.

Do you use the IMMED bit in File Manager calls?

This bit, which was documented in early versions of *Inside Macintosh* as a special form of File Manager call, actually did nothing for File Manager calls, and was used only for Device Manager calls. With the advent of the Hierarchical File System, this bit indicates that the call has a parameter block with hierarchical information.

Do you make assumptions about the number and size of disk drives?

There are now five sizes of Apple disks for the Macintosh (400K, 800K, and 20M, 40M, 80M), as well as many more from third-party vendors. You should use Standard File and File Manager calls to determine the number and size of disk drives.

Do you depend on alternate (page 2) sound or video buffers?

Some Macintoshes do not support alternate sound and video buffers.

Do you print by sending ASCII directly to the printer driver?

To retain compatibility with both locally-connected and AppleTalk-connected printers, you should print using Printing Managerr, as documented in *Inside Macintosh*.

Does your application fail when it's the startup application (i.e., without the Finder being run first)?

If so, you're probably not initializing a variable. If your application does not work as the startup application, you should determine why and fix the problem, since it may cause your application to fail in the future.

Macintosh Technical Notes

#3: Command-Shift-Number Keys

See also:	The Toolbox Event Manager Technical Note #110—MPW: Writing Standalone Code	
Written by:	Harvey Alcabes	March 3,1985
Modified by:	Ginger Jernigan	April 25,1985
Updated:		March 1, 1988

In the standard system, there are two Command-Shift-number key combinations that are automatically captured and processed by `GetNextEvent`. The combinations are:

Command-Shift-1	Eject internal disk
Command-Shift-2	Eject external disk

Numbers from 3 to 9 are also captured by `GetNextEvent`, but are processed by calling 'FKEY' resources. You can implement your own actions for Command-Shift-number combinations for numbers 5 to 9 by defining your own 'FKEY' resource. The routine must have no parameters. The ID of the resource must correspond to the number you want the routine to respond to. For example, if you want to define an action for Command-Shift-8, you would create an 'FKEY' resource with an ID of 8. The 'FKEY' resource should contain the code that you want to execute when the key is pressed.

The following Command-Shift-number key combinations are implemented with 'FKEY' resources in the standard System file.

Command-Shift-3	Save current screen as MacPaint file named Screen 0, Screen 1, ... Screen 9 (Works in one-bit mode only on Mac II)
Command-Shift-4	Print the active window (to an ImageWriter)
(with Caps Lock on)	Print the entire screen (to an ImageWriter)

#4: Error Returns from GetNewDialog

See also: The Dialog Manager

Written by: Russ Daniels April 4, 1985
Updated: March 1, 1988

When calling `GetNewDialog` to retrieve a dialog template from a previously opened resource file, how are error conditions indicated to the caller?

Unfortunately, they aren't. The Dialog Manager calls `GetResource` and assumes the returned value is good. Since the Dialog Manager doesn't check, you have two choices. Your first choice is to call `GetResource` for the dialog template, item list, and any resources needed by items in the item list yourself. But what do you do when you find the resources aren't there? Try to display an alert telling the user your application has been mortally wounded? What if resources needed for the alert aren't available?

The second, simpler alternative is to assure that the dialog template and other resources will be available when you build your product. This is really an adequate solution: If somebody uses a resource editor to remove your dialog template, you can hardly be blamed for its not executing properly.

A good debugging technique to catch this sort of problem is to put the value `$50FFC001` at absolute memory location 0 (the first long word of memory). If you do that, when the Dialog Manager tries to dereference the nil handle returned by the Resource Manager, you'll get an address error or bus error with some register containing `$50FFC001`. If you list the instructions around the program counter, you'll often see something like:

```
MOVE.L (A2),A1        ; in effect (0),A1
MOVE.L (A1),A1        ; the error occurs here
```

`GetNewWindow` and most of the other "GetSomething" calls will return nil if the "something" is not found.

#5: Using Modeless Dialogs from Desk Accessories

See also: The Toolbox Event Manager
The Dialog Manager
The Desk Manager

Written by: Russ Daniels April 4, 1985
Updated: March 1, 1988

When a desk accessory creates a window (including a modeless dialog window) it must set the windowKind to its refnum—a negative number. When the application calls `GetNextEvent`, the Event Manager calls `SystemEvent`, which checks to see if the event belongs to a desk accessory. `SystemEvent` checks the windowKind of the frontmost window, and uses the (negative) number for the refnum to make a control call, giving the desk accessory a shot at the event. Then `SystemEvent` returns TRUE, and `GetNextEvent` returns FALSE.

So, your desk accessory gets an event from `SystemEvent`. Since your window is a modeless dialog, you call `IsDialogEvent`, which mysteriously returns FALSE. What is going on?

Like `SystemEvent`, `IsDialogEvent` checks the windowKind of windows in the window list, looking for dialog windows. It does this by looking for windows with a windowKind of 2. In this case, it finds none, and does nothing.

The solution is to change the windowkind of your window to 2 before calling `IsDialogEvent`. This allows the Dialog Manager to recognize and handle the event properly. Before returning to `SystemEvent`, be sure to restore the windowKind. That way, when the application calls the Dialog Manager with the same event (the application should pass all events to Dialog Manager if it has any modeless dialogs itself), the Dialog Manager will ignore it.

Macintosh Technical Notes

#6: Shortcut for Owned Resources

See also: The Resource Manager
Technical Note #23—
Life With Font/DA Mover—Desk Accessories

Written by: Bryan Stearns May 10, 1986
Updated: March 1, 1988

To allow the Font/DA Mover to renumber desk accessories as needed when moving them between system files, desk accessories should use the "owned resource" protocol described in the Resource Manager chapter of *Inside Macintosh Volume I.*

All resource IDs in a desk accessory should be zero-based. At runtime, a routine can be called to find the current "base" value to add to a resource's zero-based value to get the actual current ID of that resource. Then, when a resource is needed, its zero-based value can be added to the resource base value, giving the actual resource ID to be used in future Resource Manager calls.

Here's the source to a handy routine to get the resource base value, `GetResBase`:

```
;FUNCTION GetResBase(driverNumber: INTEGER): INTEGER;
;
;GetResBase takes the driver number and returns the ID
;of the first resource owned by that driver. This is
;according to the private resource numbering convention
;documented in the Resource Manager.

GetResBase    FUNC

              MOVE.L    (SP)+,A0         ; Get return address
              MOVE.W    (SP)+,D0         ; Get driver number
              NOT.W     D0               ; Change to unit number
              ASL.W     #5,D0            ; Move it over in the word
              ORI.W     #$C000,D0        ; Add the magic bits
              MOVE.W    D0,(SP)          ; Return function result
              JMP       (A0)             ; and return

              END
```

#7: A Few Quick Debugging Tips

Written by: Jim Friedlander April 16, 1986
Updated: March 1, 1988

This presents a few tips which may make your debugging easier.

Setting memory location 0 to something odd

Dereferencing nil handles can cause real problems for an application. If location 0 (nil) is something even, the dereference will not cause an address error, and the application can run on for quite a while, making tracing back to the problem quite difficult. If location 0 contains something odd, such as `$50FFC001`, an address error will be generated immediately when a nil handle is dereferenced. On Macintoshes with 68020s, like the Mac II, this same value (`$50FFC001`) will cause a bus error. An address error or bus error will also be generated, of course, when the ROM tries to dereference a nil handle, such as when you call `HNoPurge(hndl)`, where `hndl` is nil.

Some versions of the TMON debugger set location 0 to 'NIL!' (`$4E494C21`) or `$50FFC001`. If you are using MacsBug, you should include code in your program that sets location 0. Of course, there is no need to ship your application with this code in it—it's just for debugging purposes. Old versions of the Finder used to set location 0 to the value $464F424A ('FOBJ'). On newer machines, newly launched applications get location 0 set to `$00F80000` by the Segment Loader.

Checksumming for slow motion mode

Entering the Macsbug command "`SS 400000 400000`" will cause Macsbug to do a checksum of the location `$400000` every time an instruction is executed. Checksum the ROM, because it will not change while your program is executing (the ROM may change in between launches of your application, but that's OK)! This will cause the Macintosh to go into slow motion mode. For example, you will need to hold down the mouse button for about 10 seconds to get a menu to pull down—you can see how the ROM draws menus, grays text, etc.

This technique is very handy for catching problems like multiple updates of your windows, redrawing scroll bars more than once, that troublesome flashing grow icon, etc. To turn slow motion mode off, simply enter MacsBug and type "`SS`".

TMON performs this function in a different way. Instead of calculating the checksum after each instruction, it only calculates checksums after each trap. You can checksum different amounts of the ROM depending on how much you want things to slow down.

Checksumming MemErr

A lot of programs don't call `MemError` as often as they should. If you are having strange, memory-related problems, one thing that you can do to help find them is to checksum on `MemErr` (the low memory global word at `$220`). In MacsBug, type "`SS 220 221`". In TMON, enter 220 and 221 as limits on the '`Checksum (bgn end) :`' line and on the line above, enter the range of traps you wish to have the checksum calculated after.
When `MemErr` changes, the debugger will appear, and you can check your code to make sure that you are checking `MemErr`. If not, you might have found a problem that could cause your program to crash!

Checksumming on a master pointer

Due to fear of moving memory, some programmers lock every handle that they create. Of course, handles need only be locked if they are going to be dereferenced **and** if a call will be made that can cause relocation. Unnecessarily locking handles can cause unwanted heap fragmentation. If you suspect that a particular memory block is moving on you when you have its handle dereferenced, you can checksum the master pointer (the handle you got back from `NewHandle` is the address of the master pointer). Your program will drop into the debugger each time your handle changes—that is, either when the block it refers to is relocated, or when the master pointer's flags byte changes.

#8: RecoverHandle Bug in AppleTalk Pascal Interfaces

See also: AppleTalk Manager

Written by: Bryan Stearns April 21, 1986
Updated: March 1, 1988

Previous versions of this note described a bug in the AppleTalk Pascal Interfaces. This bug was fixed in MPW 1.0 and newer.

#9: Will Your AppleTalk Application Support Internets?

Written by: Sriram Subramanian & Pete Helme April 1990
Written by: Bryan Stearns April 1986

This Technical Note discusses how AppleTalk applications should work across internets, groups of interconnected AppleTalk networks. It explains the differences between life on a single AppleTalk network and life on an internet.
Changes since March 1988: Removed the section on AppleTalk retry timers, as it is no longer accurate; see Technical Note #270, AppleTalk Timers Explained, for more information on retry timers.

You can read about internets (AppleTalk networks connect by one or more bridges) in *Inside AppleTalk.* What do you need to do about them?

Use a High-Level Network Protocol

Make sure you use the Datagram Delivery Protocol (DDP), or a higher AppleTalk protocol based on DDP, like the AppleTalk Transaction Protocol (ATP). Be warned that Link Access Protocol (LAP) packets do not make it across bridges to other AppleTalk networks. Also, don't broadcast; broadcast packets are not forwarded by bridges (broadcasting using protocols above LAP is discouraged, anyway).

Use Name Binding

As usual, use the Name Binding Protocol (NBP) to announce your presence on the network, as well as to find other entities on the network. Pay special attention to zone name fields; the asterisk (as in "MyLaser:LaserWriter:*") in a name you look up is now important; it means "my zone only" (see the Zone Information Protocol (ZIP) chapter of *Inside AppleTalk* for information on finding out what other zones exist). The zone field should always be an asterisk when registering a name.

Pay Attention to Network Number Fields

When handling the network addresses returned by `NBPLookUp` (or anyone else), don't be surprised if the network number field is non-zero.

Am I Running on an Internet?

The low-memory global `ABridge` is used to keep track of a bridge on the local AppleTalk network (NBP and DDP use this value). If `ABridge` is non-zero, then you're running on an internet; if it's zero, chances are, you're not (this is not guaranteed, however, due to the fact that the `ABridge` value is "aged", and if NBP hasn't heard from the bridge in a long time, the value is cleared).

Watch for Out-Of-Sequence and Non-Exactly-Once Requests

Due to a "race" condition on an internet, it's possible for an exactly-once ATP packet to slip through twice; to keep this from happening, send a sequence number as part of the data with each ATP packet; whenever you make a request, bump the sequence number, and never honor an old sequence number.

Further Reference:

- *Inside AppleTalk*
- *Inside Macintosh*, Volumes II & V, The AppleTalk Manager
- Technical Note #250, AppleTalk Phase 2 on the Macintosh
- Technical Note #270, AppleTalk Timers Explained

Macintosh Technical Notes

#10: Pinouts

See also: *Macintosh Hardware Reference Manual*
Technical Note #65—Macintosh Plus Pinouts

Written by: Mark Baumwell April 26, 1985
Modified: July 23, 1985
Updated: March 1, 1988

This note gives pinouts for Macintosh ports, cables, and other products.

Below are pinout descriptions for the Macintosh ports, cables, and various other products. Please refer to the Hardware chapter of *Inside Macintosh* and the *Macintosh Hardware Reference Manual* for more information, especially about power limits. Note that unconnected pins are omitted.

Macintosh Port Pinouts

Macintosh Serial Connectors (DB-9)

Pin	Name	Description/Notes
1	Ground	
2	+5V	See *Inside Macintosh* for power limits
3	Ground	
4	TxD+	Transmit Data line
5	TxD–	Transmit Data line
6	+12V	See Macintosh Hardware chapter for power limits
7	HSK	HandShaKe: CTS or TRxC, depends on Zilog 8530 mode
8	RxD+	Receive Data line; ground this line to emulate RS232
9	RxD–	Receive Data line

Macintosh Mouse Connector (DB-9)

Pin	Name	Description/Notes
1	Ground	
2	+5V	See *Inside Macintosh* for power limits
3	GND	Ground
4	X2	Horizontal movement line (connected to VIA PB4 line)
5	X1	Horizontal movement line (connected to SCC DCDA– line)
7	SW–	Mouse button line (connected to VIA PB3)
8	Y2	Vertical movement line (connected to VIA PB5 line)
9	Y1	Vertical movement line (connected to SCC DCDB– line)

Macintosh Keyboard Connector (RJ-11 Telephone-style jack)

Pin	Name	Description/Notes
1	Ground	
2	KBD1	Keyboard clock
3	KBD2	Keyboard data
4	+5V	See *Inside Macintosh* for power limits

Macintosh External Drive Connector (DB-19)

Pin	Name	Description/Notes
1	Ground	
2	Ground	
3	Ground	
4	Ground	
5	−12V	See *Inside Macintosh* for power limits
6	+5V	See *Inside Macintosh* for power limits
7	+12V	See *Inside Macintosh* for power limits
8	+12V	See *Inside Macintosh* for power limits
10	PWM	Regulates speed of the drive
11	PH0	Control line to send commands to the drive
12	PH1	Control line to send commands to the drive
13	PH2	Control line to send commands to the drive
14	PH3	Control line to send commands to the drive
15	WrReq−	Turns on the ability to write data to the drive
16	HdSel	Control line to send commands to the drive
17	Enbl2−	Enables the Rd line (else Rd is tri-stated)
18	Rd	Data actually read from the drive
19	Wr	Data actually written to the drive

Other Pinouts

Macintosh XL Serial Connector A (DB-25)

Pin	Name	Description/Notes
1	Ground	
2	TxD	Transmit Data line
3	RxD	Receive Data line
4	RTS	Request to Send
5	CTS	Clear To Send
6	DSR	Data Set Ready
7	Ground	
8	DCD	Data Carrier Detect
15	TxC	Connected to TRxCA
17	RxC	Connected to RTxCA
24	TEXT	Connected to TRxCA

Macintosh XL Serial Connector B (DB-25)

Pin	Name	Description/Notes
1	Ground	
2	TxD–	Transmit Data line
3	RxD–	Receive Data line
6	HSK/DSR	TRxCB or CTSB
7	Ground	
19	RxD+	Receive Data line
20	TXD+/DTR	connected to DTRB

Apple 300/1200 Modem Serial Connector (DB-9)

Modem	Name	Description/Notes
2	DSR	Output from modem
3	Ground	
5	RxD	Output from modem
6	DTR	Input to modem
7	DCD	Output from modem
8	Ground	
9	TxD	Input to modem

Apple ImageWriter Serial Connector (DB-25)

ImageWriter	Name	Description/Notes
1	Ground	
2	SD	Send Data; Output from ImageWriter
3	RD	Receive Data; Input to ImageWriter
4	RTS	Output from ImageWriter
7	Ground	
14	FAULT–	False when deselected; Output from ImageWriter
20	DTR	Output from ImageWriter

Apple LaserWriter AppleTalk Connector (DB-9)

LaserWriter	Name	Description/Notes
1	Ground	
3	Ground	
4	TxD+	Transmit Data line
5	TxD–	Transmit Data line
7	RXCLK	TRxC of Zilog 8530
8	RxD+	Receive Data line
9	RxD–	Receive Data line

Apple LaserWriter Serial Connector (DB-25)

LaserWriter	Name	Description/Notes
1	Ground	
2	TXD–	Transmit Data; Output from LaserWriter
3	RXD–	Receive Data; Input to LaserWriter
4	RTS–	Output from LaserWriter
5	CTS	Input to LaserWriter
6	DSR	Input to LaserWriter (connected to DCBB– of 8530)
7	Ground	
8	DCD	Input to LaserWriter (connected to DCBA– of 8530)
20	DTR–	Output from LaserWriter
22	RING	Input to LaserWriter

Macintosh Cable Pinouts

Note for the cable descriptions below:

The arrows ("→") show which side is an input and which is an output. For example, the notation "a → b" means that signal "a" is an output and "b" is an input.

When pins are said to be connected on a side in the Notes column, it means the pins are connected on that side of the connector.

Macintosh ImageWriter Cable
(part number 590-0169)

Macintosh (DB9)	Name			ImageWriter (DB25)	Notes
1	Ground			1	
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RD	3	RD = Receive Data
7	HSK	←	DTR	20	
8	RxD+	=	GND		Not connected on ImageWriter side
9	RxD–	←	SD	2	SD = Send Data

Macintosh Modem Cable (Warning! Don't use this cable to connect 2 Macintoshes!)
(part number 590-0197-A)

Macintosh (DB9)	Name			Modem (DB9)	Notes
3	Ground			3	pins 3, 8 connected on EACH side
5	TxD–	→	TxD	9	
6	+12V	→	DTR	6	
7	HSK	←	DCD	7	
8	No wire			8	
9	RxD–	←	RxD	5	

Macintosh to Macintosh Cable (Macintosh Modem Cable with pin 6 clipped on both ends.)

Macintosh (DB9)	Name			Macintosh (DB9)	Notes
3	Ground			3	pins 3, 8 connected on EACH side
5	TxD–	→	RxD–	9	
7	HSK	←	DCD	7	
8	No wire			8	
9	RxD–	←	TxD–	5	

Macintosh External Drive Cable
(part number 590-0183-B)

Macintosh (DB9)	Name	Sony Drive (20 Pin Ribbon)
1	Ground	1
2	Ground	3
3	Ground	5
4	Ground	7
6	+5V	11
7	+12V	13
8	+12V	15
10	PWM	20
11	PH0	2
12	PH1	4
13	PH2	6
14	PH3	8
15	WrReq–	10
16	HdSel	12
17	Enbl2–	14
18	Rd	16
19	Wr	18

Macintosh XL Null Modem Cable
(part number 590-0166-A)

Macintosh XL (DB25)	Name			DTE (DB25)	Notes
1	Ground			1	
2	TxD–	→	RxD	3	
3	RxD–	←	TxD	2	
4, 5	RTS,CTS	→	DCD	8	pins 4, 5 connected together
6	DSR	←	DTR	20	
7	Ground			7	
8	DCD	←	RTS, CTS	4, 5	pins 4, 5 connected together
20	DTR	→	DSR	6	

Macintosh to Non-Apple Product Cable Pinouts

Macintosh to IBM PC Serial Cable #1 (not tested)

Macintosh (DB9)	Name			IBM PC (DB25)	Notes
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RxD	3	
7	HSK	←	DTR	20	
8	RxD+	=	Ground		Not connected on IBM side
9	RxD–	←	TxD	2	
	CTS	←	RTS	4-5	pins 4, 5 connected on IBM side
	DSR	←	DCD,DTR	6-8-20	pins 6, 8, 20 connected on IBM side

Macintosh to IBM PC Serial Cable #2 (not tested)

Macintosh (DB9)	Name			IBM PC (DB25)	Notes
1	Ground			1	
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RxD	3	
9	RxD–	←	TxD	2	
	CTS	←	RTS	4-5	pins 4, 5 connected on IBM side
	DSR	←	DTR	6-8	pins 6, 8 connected on IBM side

#11: Memory–Based MacWrite Format

Revised: August 1989

This Technical Note formerly described the format of files created by MacWrite® 2.2.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the memory–based MacWrite 2.2 file format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.

#12: Disk–Based MacWrite Format

Revised: August 1989

This Technical Note formerly described the format of files created by MacWrite®, which is now published by CLARIS.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the disk–based MacWrite file format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.

#13: MacWrite Clipboard Format

Revised: August 1989

This Technical Note formerly described the clipboard format used by MacWrite®, which is now published by CLARIS.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the MacWrite clipboard format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.

Macintosh Technical Notes

#14: The INIT 31 Mechanism

See: The System Resource File
The Start Manager

Written by: Bryan Stearns March 13, 1986
Updated: March 1, 1988

This note formerly described things that are now covered in the System Resource File chapter of *Inside Macintosh Volume IV* and the Start Manager chapter of *Inside Macintosh Volume V*. Please refer to *Inside Macintosh*.

#15: Finder 4.1

Written by: Harvey Alcabes
Updated:

April 12, 1985
March 1, 1988

This note formerly described Finder 4.1, which is now recommended only for use with 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#16: MacWorks XL

Written by: Harvey Alcabes
Mark Baumwell

May 11, 1985
Updated: March 1, 1988

Earlier versions of this note described MacWorks XL, the system software that allowed you to use Macintosh applications on the Macintosh XL. Information specific to Macintosh XL machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#17: Low-Level Print Driver Calls

See also: The Print Manager

Written by: Ginger Jernigan April 14, 1986
Updated: March 1, 1988

This technical note has been replaced by information in *Inside Macintosh Volume V*. Please refer to the Print Manager chapter of *Inside Macintosh Volume V* for information on low-level print driver calls.

Macintosh Technical Notes

#18: TextEdit Conversion Utility

See also: Macintosh Memory Management: An Introduction
TextEdit

Written by: Harvey Alcabes — April 10, 1985
Updated: March 1, 1988

Text sometimes must be converted between a Pascal string and “pure” text in a handle. This note illustrates a way to do this using MPW Pascal.

Text contained in TextEdit records sometimes must be passed to routines which expect a Pascal string of type Str255 (a length byte followed by up to 255 characters). The following MPW Pascal unit can be used to convert between TextEdit records and Pascal strings:

```
UNIT TEConvert;

   {General utilities for conversion between TextEdit and strings}

   INTERFACE

      USES MemTypes,QuickDraw,OSIntf,ToolIntf;

      PROCEDURE TERecToStr(hTE: TEHandle; VAR str: Str255);
      {TERecToStr converts the TextEdit record hTE to the string str.}
      {If necessary, the text will be truncated to 255 characters.}

      PROCEDURE StrToTERec(str: Str255; hTE: TEHandle);
      {StrToTERec converts the string str to the TextEdit record hTE. }

   IMPLEMENTATION

      PROCEDURE TERecToStr(hTE: TEHandle; VAR str: Str255);

         BEGIN
            GetIText(hTE^^.hText, str);
         END;

      PROCEDURE StrToTERec(str: Str255; hTE: TEHandle);

         BEGIN
            TESetText(POINTER(ORD4(@str) + 1), ORD4(length(str)), hTE);
         END;

END.
```

#19: How To Produce Continuous Sound Without Clicking

Revised by: Jim Reekes June 1989
Written by: Ginger Jernigan April 1985

This Technical Note formerly described how to use the Sound Driver to produce continuous sound without clicking.
Changes since March 1988: The continuous sound technique is no longer recommended.

Apple currently discourages use of the Sound Driver due to compatibility issues. The hardware support for sound designed into the early Macintosh architecture was minimal. (Many things have changed since 1983–1984.) The new Macintosh computers contain a custom chip to provide better support for sound, namely the Apple Sound Chip (ASC). The ASC is present in the complete Macintosh II family as well as the Macintosh SE/30 and later machines. When the older hardware of the Macintosh Plus and SE are accessed, it is likely to cause a click. This click is a hardware problem. The software solution to this problem was to continuously play silence. This is not a real solution to the problem and is not advisable for the following reasons:

- The Sound Driver is no longer supported. There have always been, and still are, bugs in the glue code for `StartSound`.
- The Sound Driver may not be present in future System Software releases, or future hardware may not be able to support it. The Sound Manager is the application's interface to the sound hardware.
- The technique used to create a continuous sound should have only been used on a Macintosh Plus or SE, since these are the only models that have the "embarrassing click." Do not use this method on a Macintosh which has the Apple Sound Chip.
- Using the continuous sound technique, or the Sound Driver for that matter, will cause problems for the system and those applications that properly use the Sound Manager. Also realize that `_SysBeep`, which is a common routine that everything uses, is a Sound Manager routine.
- The continuous sound technique wastes CPU time by playing silence. With multimedia applications and the advent of MultiFinder, it is important to allow the CPU to do as much work as possible. The continuous sound technique used the CPU to continuously play silence, thus stealing valuable time from other, more important, jobs.

Further Reference:

- *The Sound Manager,* Interim Chapter by Jim Reekes, October 2, 1988
- Technical Note #180, MultiFinder Miscellanea

#20: Data Servers on AppleTalk

See also: The AppleTalk Manager
Inside LaserWriter

Written by: Bryan Stearns April 29, 1985
Updated: March 1, 1988

Many applications could benefit from the ability to share common data between several Macintoshes, without requiring a file server. This technical note discusses one technique for managing this AppleTalk communication.

There are four main classes of network "server" devices:

Device Servers, such as the LaserWriter, allow several users to share a single hardware device; other examples of this (currently under development by third parties) are modem servers and serial servers (to take advantage of non-intelligent printers such as the ImageWriter).

File Servers, such as AppleShare, which support file access operations over the network. A user station sends high-level requests over the network (such as "Open this file," "Read 137 bytes starting at the current file position of this file," "Close this file," etc.).

Block Servers, which answer to block requests over the network. These requests impart no file system knowledge about the blocks being passed, i.e., the server doesn't know which files are open by which users, and therefore cannot protect one user's open file from other users. Examples of this type of server are available from third-party developers.

Data Servers, which answer to requests at a higher level than file servers, such as "Give me the first four records from the database which match the following search specification." This note directs its attention at this type of server.

A data server is like a file server in that it responds to intelligent requests, but the requests that it responds to can be more specialized, because the code in the server was written to handle that specific type of request. This has several added benefits: user station processing can be reduced, if the data server is used for sorting or searching operations; and network traffic is reduced, because of the specificity of the requests passed over the network. The data server can even be designed to do printing (over the network to a LaserWriter, or on a local ImageWriter), given that it has the data and can be directed as to the format in which it should be printed.

ATP: The AppleTalk Transaction Protocol

ATP, the assured-delivery AppleTalk Transaction Protocol, can be used to support all types of server communications (the LaserWriter uses ATP for its communications!). Here is a possible scenario between two user stations ("Dave" and "Bill") and a data server station ("OneServer", a server of type "MyServer"). We've found that the "conversational" analogy is helpful when planning AppleTalk communications; this example is therefore presented as a conversation, along with appropriate AppleTalk Manager calls (Note that no error handling is presented, however; your application should contain code for handling errors, specifically the "half-open connection" problem described below).

Establishing the Connection

Each station uses `ATPLoad` to make sure that AppleTalk is loaded. The server station, since it wants to accept requests, opens a socket and registers its name using `NBPRegister`. The user stations use `NBPLookUp` to find out the server's network address. This looks like this, conversationally:

Server:	"I'm ready to accept requests!"	`ATPLoad`
		`OpenSocket`
		`NBPRegister`
		`ATPGetRequest`
		`ATPGetRequest`
		`ATPGetRequest`
			Opens AppleTalk
			Creates socket
			Assigns name to socket
			queue a few asynchronous calls, to be able to handle several users

Dave:	"Any 'MyServers' out there?"	`ATPLoad` `NBPLookup`	Opens AppleTalk look for servers, finds OneServer
Dave:	"Hey, MyServer! What socket should I talk to you on?"	`ATPRequest`	Ask the server which socket to use for further communications
Bill:	"Any 'MyServers' out there?"	`ATPLoad` `NBPLookup`	Opens AppleTalk look for servers, finds OneServer
Bill:	"Hey, MyServer! What socket should I talk to you on?"	`ATPRequest`	Ask the server which socket to use for further communications

Server:	"Hi, Dave! Use Socket N."	`ATPOpenSkt`
		`ATPResponse`
		`ATPGetRequest`
			Get a new socket for talking to Dave
			Send Dave the socket number
			Replace the used GetRequest

Server:	"Hi, Bill! Use socket M."	`ATPOpenSkt`
		`ATPResponse`
		`ATPGetRequest`
			Get a new socket for talking to Bill
			Send Bill the socket number
			Replace the used GetRequest

From this point on, the server knows that any requests received on socket N are from Dave, and those received on socket M are from Bill. The conversations continue, after a brief discussion of error handling.

Half-Open Connections

There is a possibility that one side of a connection could go down (be powered off, rebooted accidently, or simply crash) before the connection has been officially broken. If a user station goes down, the server must throw away any saved state information and close that user's open socket. This can be done by requiring that the user stations periodically "tickle" the server: every 30 seconds (for example) the user station sends a dummy request to the server, which sends a dummy response. This lets each side of the connection know that the other side is still "alive."

When the server detects that two intervals have gone by without a tickle request, it can assume that the user station has crashed, and close that user's socket and throw away any accumulated state information.

The user station should use a vertical-blanking task to generate these tickle requests asyncronously, rather than generating them within the `GetNextEvent` loop; this avoids problems with long periods away from `GetNextEvent` (such as when a modal dialog box is running). This task can look at the time that the last request was made of the server, and if it's approaching the interval time, queue an **asynchronous** request to tickle the server (it's important that any AppleTalk calls made from interrupt or completion routines be asynchronous).

If a user station's request (including a tickle request) goes unanswered, the user station should recover by looking for the server and reestablishing communications as shown above (beginning with the call to `NBPLookUp`).

More information about half-open connections can be found in the Printer Access Protocol chapter of *Inside LaserWriter,* available from APDA.

Using the Connection

The user stations Dave and Bill have established communications with the server, each on its own socket (note that the user stations have not had to open their own sockets, or register names of their own, to do this—the names we're using are merely for explanational convenience). They are also automatically tickling the server as necessary.

Now the user stations make requests of the server as needed:

Bill:	"I'd like to use the sales figures for this year."	`ATPRequest`	Bill opens a database.
Server:	"Ok, Bill."	`ATPResponse`	The server checks to make sure that no one else is using that database.
Dave:	"Hey, Server - I'm still here!"	`ATPRequest`	Dave notices that the interval time is approaching, and makes a tickle request.
Server:	"Ok, Dave."	`ATPResponse`	The server resets its "last time I heard from Dave".
Bill:	"Please print the figures for January thru June."	`ATPRequest`	Bill asks for specific data.
Server:	"Ok, Bill."	`ATPResponse`	The server does a database search sorts the results, and prints them on a local Imagewriter.
Dave:	"I'd like to use the sales figures for this year."	`ATPRequest`	Dave opens a database.
Server:	"Sorry, Dave, I can't do that. Bill is using that database."	`ATPResponse`	The server finds that Bill is using that data.

Closing the Connection

The user stations continue making requests of the server, until each is finished. The type of work being done by the server determines how long the conversation will last: since the number of sockets openable by the server is limited, it may be desirable to structure the requests in such a way that the average conversation is very short. It may also be necessary to have a (NBP named) socket open on the user station, if the server needs to communicate with the user on other than a request-response basis. Here is how our example connections ended:

Dave:	"Thank you, server, I'm done now. You've been a big help."	`ATPRequest`	Dave tells the server he's finished.
Server:	"Ok, Dave. Bye now."	`ATPResponse` `ATPCloseSkt`	The server kisses Dave goodbye. After the Response operation completes, the server closes the socket Dave was using.
		`ATPCloseSkt`	It also notices that Bill hasn't sent a request in more than two intervals, and closes Bill's socket, too.

The user station can forget about the socket it was using on the server; if it needs to talk with the server again, it starts at the `NBPLookUp` (just in case the server has moved, gone down and come up, etc.).

Macintosh Technical Notes

#21: QuickDraw's Internal Picture Definition

See also:
QuickDraw
Color QuickDraw
Using Assembly Language
Technical Note #59—Pictures and Clip Regions

Written by: Ginger Jernigan April 24, 1985
Modified by: Rick Blair November 15, 1986
Updated: March 1, 1988

This technical note describes the internal format of the QuickDraw picture data structure. This revision corrects some errors in the opcode descriptions and provides some examples.

This technical note describes the internal definition of the QuickDraw picture. The information given here only applies to QuickDraw picture format version 1.0 (which is always created by Macintoshes without Color QuickDraw). Picture format version 2.0 is documented in the Color QuickDraw chapter of *Inside Macintosh*. This information should not be used to write your own picture bottleneck procedures; if we add new objects to the picture definition, your program will not be able to operate on pictures created using standard QuickDraw. Your program will not know the size of the new objects and will, therefore, not be able to proceed past the new objects. (What this ultimately means is that you can't process a new picture with an old bottleneck proc.)

Terms

An *opcode* is a number that `DrawPicture` uses to determine what object to draw or what *mode* to change for subsequent drawing. The following list gives the opcode, the name of the object (or mode), the associated data, and the total size of the opcode and data. To better interpret the sizes, please refer to page I-91 of the Using Assembly Language chapter of *Inside Macintosh*. For types not described there, here is a quick list:

opcode	byte
mode	word
point	4 bytes
0..255	byte
–128..127	signed byte
rect	8 bytes
poly	10+ bytes (starts with word size for poly (incl. size word)
region	10+ bytes (starts with word size for region (incl. size word)

fixed point	long
pattern	8 bytes
rowbytes	word (always even)
bit data	rowbytes * (bounds.bottom - bounds.top) bytes

Each picture definition begins with a `picsize` (`word`), then a `picframe` (`rect`), and then the picture definition, which consists of a combination of the following opcodes:

Opcode	Name	Additional Data	Total Size (bytes)
00	NOP	none	1
01	clipRgn	rgn	1+rgn
02	bkPat	pattern	9
03	txFont	font (word)	3
04	txFace	face (byte)	2
05	txMode	mode (word)	3
06	spExtra	extra (fixed point)	5
07	pnSize	pnSize (point)	5
08	pnMode	mode (word)	3
09	pnPat	pattern	9
0A	thePat	pattern	9
0B	ovSize	point	5
0C	origin	dh, dv (words)	5
0D	txSize	size (word)	3
0E	fgColor	color (long)	5
0F	bkColor	color (long)	5
10	txRatio	numer (point), denom (point)	9
11	picVersion	version (byte)	2
20	line	pnLoc (point), newPt (point)	9
21	line from	newPt (point)	5
22	short line	pnLoc (point); dh, dv (-128..127)	7
23	short line from	dh, dv (-128..127)	3
28	long text	txLoc (point), count (0..255), text	6+text
29	DH text	dh (0..255), count (0..255), text	3+text
2A	DV text	dv (0..255), count (0..255), text	3+text
2B	DHDV text	dh, dv (0..255), count (0..255), text	4+text
30	frameRect	rect	9
31	paintRect	rect	9
32	eraseRect	rect	9
33	invertRect	rect	9
34	fillRect	rect	9
38	frameSameRect	rect	1
39	paintSameRect	rect	1
3A	eraseSameRect	rect	1
3B	invertSameRect	rect	1
3C	fillSameRect	rect	1
40	frameRRect	rect (ovalwidth, height; see 1, below)	9
41	paintRRect	rect (ovalwidth, height; see 1, below)	9
42	eraseRRect	rect (ovalwidth, height; see 1, below)	9

Opcode (cont.)	Name	Additional Data	Total Size (bytes)
43	invertRRect	rect (ovalwidth, height; see 1, below)	9
44	fillRRect	rect (ovalwidth, height; see 1, below)	9
48	frameSameRRect	rect	1
49	paintSameRRect	rect	1
4A	eraseSameRRect	rect	1
4B	invertSameRRect	rect	1
4C	fillSameRRect	rect	1
50	frameOval	rect	9
51	paintOval	rect	9
52	eraseOval	rect	9
53	invertOval	rect	9
54	fillOval	rect	9
58	frameSameOval	rect	1
59	paintSameOval	rect	1
5A	eraseSameOval	rect	1
5B	invertSameOval	rect	1
5C	fillSameOval	rect	1
60	frameArc	rect, startAngle, arcAngle	13
61	paintArc	rect, startAngle, arcAngle	13
62	eraseArc	rect, startAngle, arcAngle	13
63	invertArc	rect, startAngle, arcAngle	13
64	fillArc	rect, startAngle, arcAngle	13
68	frameSameArc	startAngle, arcAngle	5
69	paintSameArc	startAngle, arcAngle	5
6A	eraseSameArc	startAngle, arcAngle	5
6B	invertSameArc	startAngle, arcAngle	5
6C	fillSameArc	startAngle, arcAngle	5
70	framePoly	poly	1+poly
71	paintPoly	poly	1+poly
72	erasePoly	poly	1+poly
73	invertPoly	poly	1+poly
74	fillPoly	poly	1+poly
78	frameSamePoly	(not yet implemented—same as 70, etc.)	1
79	paintSamePoly	(not yet implemented)	1
7A	eraseSamePoly	(not yet implemented)	1
7B	invertSamePoly	(not yet implemented)	1
7C	fillSamePoly	(not yet implemented)	1
80	frameRgn	rgn	1+rgn
81	paintRgn	rgn	1+rgn
82	eraseRgn	rgn	1+rgn
83	invertRgn	rgn	1+rgn
84	fillRgn	rgn	1+rgn
88	frameSameRgn	(not yet implemented—same as 80, etc.)	1
89	paintSameRgn	(not yet implemented)	1
8A	eraseSameRgn	(not yet implemented)	1
8B	invertSameRgn	(not yet implemented)	1

Opcode (cont.)	Name	Additional Data	TotalSize (bytes)
8C	fillSameRgn	(not yet implemented)	1
90	BitsRect	rowBytes, bounds, srcRect, dstRect, mode, unpacked bitData	29+unpacked bitData
91	BitsRgn	rowBytes, bounds, srcRect, dstRect, mode, maskRgn, unpacked bitData	29+rgn+ bitData
98	PackBitsRect	rowBytes, bounds, srcRect, dstRect, mode, packed bitData for each row	29+packed bitData
99	PackBitsRgn	rowBytes, bounds, srcRect, dstRect, mode, maskRgn, packed bitData for each row	29+rgn+ packed bitData
A0	shortComment	kind(word)	3
A1	longComment	kind(word), size(word), data	5+data
FF	EndOfPicture	none	1

Notes

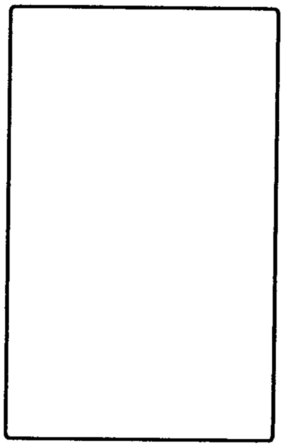
Rounded-corner rectangles use the setting of the ovSize point (see opcode $0B, above).

`OpenPicture` and `DrawPicture` set up a default set of port characteristics when they start. When drawing occurs, if the user's settings don't match the defaults, mode opcodes are generated. This is why there is usually a `clipRgn` code after the `picVersion`: the default clip region is an empty rectangle.

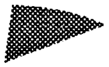
The only savings that the "same" opcodes achieve under the current implementation is for rectangles. `DrawPicture` keeps track of the last rectangle used and if a "same" opcode is encountered that requests a rectangle, the last rect. will be used (and no rectangle will appear in the opcode's data).

This last section contains some Pascal program fragments that generate pictures. Each section starts out with the picture itself (yes, they're dull) followed by the code to create and draw it, and concludes with a commented hex dump of the picture.

```
{variables used in all examples}

VAR
    err:      OSErr;
    ph:       PicHandle;
    h:        Handle;
    r:        Rect;
    smallr:   Rect;
    orgr:     Rect;
    pstate:   PenState; {are they in the Rose Bowl, or the state pen?}
```



```
I.    {Rounded-corner rectangle}
      SetRect(r, 20, 10, 120, 175);
      ClipRect(myWindow^.portRect);
      ph := OpenPicture(r);
      FrameRoundRect (r, 5, 4); {r,width,height}
      ClosePicture;
      DrawPicture(ph, r);

'PICT' (1)  0026 {size}  000A 0014 00AF 0078 {picFrame}
      1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
      0B 0004 0005 {ovSize point}  40 000A 0014 00AF 0078 {frameRRect rectangle}
      FF {fin}
```



```
II.   {Overpainted arc}
      GetPenState(pstate); {save}
      SetRect(r, 20, 10, 120, 175);
      ClipRect(myWindow^.portRect);
      ph := OpenPicture(r);
      PaintArc(r, 3, 45); {r,startangle,endangle}
      PenPat(gray);
      PenMode(patXor); {turn the black to gray}
      PaintArc(r, 3, 45); {r,startangle,endangle}
      ClosePicture;
      SetPenState(pstate); {restore}
      DrawPicture(ph, r);

data 'PICT' (2)  0036 {size}  000A 0014 00AF 0078 {picFrame}
      1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
      61 000A 0014 00AF 0078 0003 002D {paintArc rectangle,startangle,endangle}
      08 000A {pnMode patXor - note that the pnMode comes before the pnPat}
      09 AA55 AA55 AA55 AA55 {pnPat gray}
      69 0003 002D {paintSameArc startangle,endangle}
      FF {fin}
```

```
III. {CopyBits nopack, norgn, nowoman, nocry}
     GetPenState(pstate);
     SetRect(r, 20, 10, 120, 175);
     SetRect(smallr, 20, 10, 25, 15);
     SetRect(orgr, 0, 0, 30, 20);
     ClipRect(myWindow^.portRect);
     ph := OpenPicture(r);
     PaintRect(r);
     CopyBits (myWindow^.portBits, myWindow^.portBits,
                  smallr, orgr, notSrcXor, NIL);
     {note: result BitMap is 8 bits wide instead of the 5 specified by smallr}
     ClosePicture;
     SetPenState(pstate); {restore the port's original pen state}
     DrawPicture(ph, r);

data 'PICT' (3)  0048 {size}  000A 0014 00AF 0078 {picFrame}
     1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
     31 000A 0014 00AF 0078 {paintRect rectangle}
     90 0002 000A 0014 000F 001C {BitsRect rowbytes bounds (note that bounds is
                                    wider than smallr)}
     000A 0014 000F 0019 {srcRect}
     0000 0000 0014 001E {dstRect}
     00 06 {mode=notSrcXor}
     0000 0000 0000 0000 0000 {5 rows of empty bitmap (we copied from a
                                 still-blank window)}
     FF {fin}
```

#22: TEScroll Bug

See also: TextEdit
Technical Note #131—TextEdit Bugs

Written by: Bryan Stearns April 21, 1986
Updated: March 1, 1988

A bug in TextEdit causes the following problem: a call to TEScroll with no horizontal or vertical displacement (that is, both dh and dv set to zero) results in disappearance of the insertion point. Since such calls do nothing, they should be avoided:

```
IF (dh <> 0) OR (dv <> 0) THEN TEScroll(dh,dv,myTEHandle);
```

#23: Life With Font/DA Mover—Desk Accessories

See also: The Resource Manager
Technical Note #6—Shortcut for Owned Resources

Written by: Ginger Jernigan April 25, 1985
Updated: March 1, 1988

This technical note describes how to make sure that your desk accessory will work after being moved by Font/Desk Accessory Mover.

If you want your desk accessory to work properly after being moved by the Font/DA Mover, there are some eccentricities that you need to be aware of. When the Font/DA Mover moves a desk accessory, it renumbers to avoid conflicts in ID numbers. It will also renumber all of your desk accessory's owned resources. See the Resource Manager chapter of *Inside Macintosh* for more information on owned resources.

Since these owned resources are renumbered, your code will need to calculate the resource ID of any owned resource it uses. For example, if your desk accessory has an owned 'DLOG' resource, and calls `GetNewDialog` with the ID you assigned to it originally, the Resource Manager will not find it. The solution is that every time your desk accessory references an owned resource, it must figure out (at execution time) the ID of the resource according to the current driver resource ID.

When the Font/DA Mover renumbers, it does its best to keep resources pointing to each other properly. This means that it tries to renumber resource IDs embedded in other resources as well as the resources themselves. For example, the reference to a 'DITL' within a 'DLOG' or 'ALRT' resource gets changed automatically. Font/DA Mover knows about the standard embedded resource IDs in most of the standard resources, but if you define your own, the Font/DA Mover won't be able to renumber them for you. The embedded resource IDs which the Font/DA Mover knows about are listed below.

Note that certain resources can never be owned, because their resource IDs are restricted to a certain range. One such example is a WDEF. Since the ID of a WDEF is specified along with a four bit variation code, the range of WDEF IDs that can be used is 0-16363. Since none of this falls within the owned resource ID range, WDEFs cannot be owned. For the same reason, MDEFs, CDEFs, and MBDFs can't be owned either.

As a rule of thumb, before you ship a desk accessory, move it to a disk with another desk accessory of the same ID. This will cause the Font/DA Mover to renumber your desk accessory. If the moved copy doesn't work, then there is probably something wrong with the way you are handling your owned resources.

Embedded resources known by Font/DA Mover

These are all true for Font/DA Mover 3.3 and newer:

- references to 'DITL' resources in 'DLOG'/'ALRT' resources
- references to 'ICON', 'PICT', 'CTRL' in 'DITL' resources
- references to 'MENU' resources inside the resources themselves (menuID field)
- references to 'MENU' resources in 'MBAR' resources

Anything not on this list has to be fixed by the desk accessory.

By the way...

Before Font/DA Mover, desk accessories could have an ID in the range 12 to 31. Now, and in the future, desk accessories can only have IDs in the range 12 to 26, because Font/DA Mover will only assign numbers in this range. Numbers 27 thru 31 are reserved.

Macintosh Technical Notes

#24: Available Volumes

See also: The File Manager

Written by: Bryan Stearns April 26, 1985
Modified by: Bryan Stearns October 15, 1985
Updated: March 1, 1988

Standard File lets the user select one file from any available volume; it is sometimes necessary for an application to find which volumes are present. This technical note gives the proper method of accomplishing this.

There is a little-noticed feature of the low-level file manager call `PBHGetVInfo` which allows specification of a "volume index" to select the volume. This volume index selects the nth volume in the VCB queue. The following function uses `PBHGetVInfo` to find out about a given volume. In MPW Pascal:

```
FUNCTION GetIndVolume(whichVol: INTEGER; VAR volName: Str255;
                      VAR volRefNum: INTEGER): OSErr;

{Return the name and vRefNum of volume specified by whichVol.}

  VAR
    volPB : HParamBlockRec;
    error : OSErr;

  BEGIN
    WITH volPB DO BEGIN          {makes it easier to fill in!}
      ioNamePtr  := @volName;   {make sure it returns the name}
      ioVRefNum  := 0;          {0 means use ioVolIndex}
      ioVolIndex := whichVol;   {use this to determine the volume}
    END; {with}
    error := PBHGetVInfo(@volPB,false); {do it}
    IF error = noErr THEN BEGIN        {if no error occurred }
      volRefNum := volPB.ioVRefNum;   {return the volume reference}
    END; {if no error}
    {other information is available from this record; see the FILE}
    {Manager's description of PBHGetVInfo for more details...}
    GetIndVolume := error; {return error code}
  END;
```

In MPW C:

```
OSErr GetIndVolume(whichVol,volName,volRefNum)
short int whichVol;
char  *volName;
short int *volRefNum;

{
      /*Return the name and vRefNum of volume specified by whichVol.*/

      HVolumeParam      volPB;
      OSErr             error;


      volPB.ioNamePtr = volName; /*make sure it returns the name*/
      volPB.ioVRefNum = 0;       /*0 means use ioVolIndex*/
      volPB.ioVolIndex = whichVol; /*use this to determine the volume*/

      error = PBHGetVInfo(&volPB,false); /*do it*/
      if (error == noErr)    /*if no error occurred */
            *volRefNum = volPB.ioVRefNum; /*return the volume reference*/

      /*other information is available from this record; see the FILE*/
      /*Manager's description of PBHGetVInfo for more details...*/

      return(error);    /*always return error code*/
} /* GetIndVolume */
```

To find out about all volumes on-line, you can call this routine several times, starting at `whichVol` := 1 and incrementing `whichVol` until the routine returns `nsvErr`.

#25: Don't Depend on Register A5 Within Trap Patches

See also: The Operating System Utilities

Written by: Bryan Stearns June 25, 1986
Updated: March 1, 1988

Future software may allow desk accessories to have their own globals by changing register A5 when the accessory is entered and exited. This can cause problems for applications that patch traps without following certain rules.

If your application patches any traps, it's important that the patches not depend on register A5. This is because you may have intercepted a trap used by a desk accessory.

If you need access to your globals within your patch, you can save A5 (on the stack, perhaps), load A5 from the low-memory global CurrentA5 (this is guaranteed to be correct for your application), do whatever you have to do within your patch, then restore A5 on the way out. Note that if you make any traps within your patch (or call the "real" version of the routine you patched), you should restore the caller's A5 before doing so.

There are several ways of depending on A5 within a patch that you should watch out for:

- Are you making any references to your global variables, or those of any units that you're using, such as thePort from QuickDraw? These are accessed with A5-relative references with negative offsets.
- Are you making any inter-segment subroutine calls? These are accessed with A5-relative references with positive offsets.
- Are you using any system calls (either traps or "glue" routines) which will depend on A5 during their execution? In this case, you need to be sure that you restore the caller's A5 before executing the call.

To be safest, patched traps should follow the same rules as interrupt handlers.

Note

In general, applications should not have to patch any traps, and risk compatibility problems if they do! If you'd like help in removing your dependence on patching, please contact Macintosh Developer Technical Support.

Macintosh
Technical Notes

Developer Technical Support

#26: Fond of FONDs

Written by: Joseph Maurer

May 1992

This Technical Note takes the place of Tech Note #26, “Character vs. String Operations in QuickDraw” by Bryan Stearns (March 1988), which pointed out the possible differences between the results of a `StringWidth` call and successive calls to `CharWidth`. This Note updates and brings into a broader context the issues related to text measuring. It also provides additional documentation on font family resources (`'FOND'`s), and addresses various other frequently asked questions related to the Font Manager. For reasons of consistency and easier reference, much of the contents of Technical Notes #191, “Font Names,” #198, “Font/DA Mover, Styled Fonts, and `'NFNT'`s,” and #245, “Font Family Numbers,” have been updated and worked into this Note as well.

Introduction

Every Macintosh developer needs to draw text in a GrafPort, and to specify typeface, size, and style. In most cases, there are no problems, and application developers don’t need to have in-depth knowledge of the Font Manager’s inner workings and the data structures involved. Sometimes, however, the results on the screen or on printed output may be different from what you expected. Then, usually, DTS comes into play to figure out what the problem is and how to fix it. This Note is based on sharp developer questions from the last year or so, which point mainly at shortcomings of the existing Font Manager architecture, inconsistencies in its data structures, and missing details in the documentation.

We’ll start with a historical overview, which discusses the introduction of font family description resources (`'FOND'`s) back in 1986, explains the consequences of non-proportionally scaling fonts, and covers non-registration and volatility of font family numbers.

We will then deal with the Font/DA Mover and the built-in “Mover” of the Finder in System 7. We discuss a number of not-so-well-known aspects of moving fonts in and out of a suitcase file, and recommend that you altogether abandon the resource type `'FONT'`. We'll also comment on font names, and show you how to put separate stylistic variants of a typeface together into one font family. And we provide documentation on the `ffVersion` field of a `'FOND'` (accompanied by a disclaimer and another piece of irritating information).

The main body of this Note addresses how the Font Manager works in the `FMSwapFont` context, and gives information on the scaling factors in the `FMOutput` structure and on the changes introduced by TrueType. We again took the examples of unexpected behavior (under certain circumstances) from developer questions. Thanks for helping document this!

Determining the width of text, as required for line layout, is sometimes trickier than you might think. We will document the effects of `SetFractEnable` in more detail and mention some more line layout problems.

Finally, this Note includes sample code that puts the `OutlineMetrics` call to work, and determines text bounding boxes for bitmap fonts.

Some FOND Background

Originally (*Inside Macintosh* Volume I, Chapter 7), all font-related data was contained in resources of type `'FONT'`. For a font number within the range 0....255, and a font size restricted to less than 128, the (unnamed) `'FONT'` resource with an ID:

128*(font number) + (font size)

contained the bitmap font strike, while the `'FONT'` resource with ID = 128*(font number), corresponding to font size 0, did not contain any data, but its resource name provided the font family name. QuickDraw took care of stylistic variants like italic, bold, shadow, and so on; if a user had a specifically fine-tuned font strike for a stylistic variant, QuickDraw would **not** automatically substitute it when drawing text.

For aesthetic reasons, bitmap fonts for different sizes were usually designed with widths non-proportional to the point size. For example, the text *"Show the difference in text widths"* drawn with Courier 9 measures 170 pixels, whereas the same text drawn with Courier 18 measures 374 pixels, which is 10% more than you expect. (By the way, this is bad news for the ImageWriter printer driver. When "Best" mode (144 dpi) is selected and text in Courier 9 is to be printed, the printer driver uses Courier 18 to render the 9-point font size on the paper at twice the screen resolution, and obviously has big trouble compensating for the 10% difference in text width.)

On the other hand, given that only integer character widths (in QuickDraw's 72 dpi units) are possible, proportional font scaling is compromised anyway. Accumulated rounding errors in text measuring, particularly for scaled fonts, contribute to the headaches of many Macintosh programmers. The computed text widths (vital for positioning text precisely and for line layout algorithms to justify text) sometimes change quite abruptly when the user removes or adds certain font sizes.

The introduction of the LaserWriter, and the success of Macintosh in the desktop publishing arena, required an extension of the original Font Manager architecture. This extension is based on the concept of "font family description" resources of type `'FOND'`, and on a new resource type `'NFNT'` for the data of the existing `'FONT'` resources (see *Inside Macintosh* Volume IV, Chapter 5).

The `'FOND'` resource stores size-independent information about the font family, and its resource ID is the font number (in the range 0...32767). The resource name of the `'FOND'` is the font name, and it contains a variable-length **font association table**, which references the font strikes belonging to a specific font family. These references include size, style, and resource ID of the `'NFNT'` or `'FONT'` resource containing the bitmap font data. TrueType fonts were retrofitted into this scheme, and are identified as font strike resources for point size zero. Any reference to point size zero refers to a resource of type `'sfnt'`.

Note: The range 0...32767 for font numbers is subdivided into ranges for the various script systems (see *Inside Macintosh* Volume VI, pages 13-8 and 14-22, and Technical Note #242, "Fonts and the Script Manager"). This restricts the range of font numbers for the Roman script to 0...16383, with 0, 1, and 16383 reserved for the system.

Since Apple originally intended fonts to be referenced by their font family numbers, DTS attempted to register those numbers (see *Inside Macintosh* Volume I, page 219 and Volume IV, page 31). This failed—not only because the number of fonts registered grew greater than the number of font family numbers available, but also because the Font/DA Mover (version 3.8, shipped with System 6), and the "Mover" built into the System 7 Finder resolve conflicts between font IDs (which happened anyway!) by renumbering the fonts on-the-fly. There is no font ID registration any more—except for the very special case of Japanese Kanji `'FOND'`–`'fbit'` IDs, and potentially for Korean, Chinese and other double-byte fonts.

As early as April 1988, Technical Note #191, "Font Names," recommended the use of font names rather than font family numbers. Since then, the recommendation has been reinforced in *Inside Macintosh* Volume VI, page 12-16. Fortunately, most applications have been good about following this recommendation. Unfortunately, some exceptions remain, even in Apple's own software. QuickDraw Pictures created without 32-Bit QuickDraw refer to fonts by font family number only!

For obvious reasons of upward compatibility (to maintain existing fonts, and to avoid reflowing of existing documents), the introduction of `'FOND'`s did not solve all the problems. This is what this Note is all about.

Moofing Fonts

The Font/DA Mover utility has evolved into version 4.1, which knows about `'sfnt'`s. It is available on the *Developer CD Series* disc, path "Tools & Apps (Moof!): Misc. Utilities:". The Finder in System 7 incorporates its own "Mover" (see *Inside Macintosh* Volume VI, page 9-33), which makes the Font/DA Mover redundant for System 7 users.

Given the combinatorial explosion of all imaginable situations with `'FOND'`s, `'FONT'`s, `'NFNT'`s and `'sfnt'`s, and stylistic variations of fonts belonging to the same family, the font moving job deserves respect. The following notes cover some less well-known aspects of this business.

- If an old "standalone" `'FONT'` (without corresponding `'FOND'` resource) is moved into a suitcase file, Font/DA Mover or the System 7 Mover creates a minimal `'FOND'` resource on-the-fly. This `'FOND'` has no tables, and nearly all its fields are zeroed. The System 7 Finder also converts the resource type from `'FONT'` to `'NFNT'`; unfortunately, the Font/DA Mover keeps the resource type `'FONT'`.

  Note: While it is perfectly legal to have `'FOND'`s continue to reference the older `'FONT'` type, DTS recommends that you avoid `'FONT'`s. Accessing `'FONT'`s is much slower, since the Font Manager always looks for `'FOND'`s and `'NFNT'`s first. More importantly, `'FONT'`s are troublemakers if an application comes with its own font in its resource fork. Imagine an application that includes a private `'FOND'` which references a `'FONT'` in its resource fork by resource ID. When the Font Manager wants to load the font resource, it first looks for a resource of type `'NFNT'` with this same resource ID. If there's an `'NFNT'` in the System file with the same resource ID, the Font Manager will pick it instead of the `'FONT'` from the application's resource fork. This happens more often than you'd like to think!

- Under the current font architecture, the font name is the resource name of the `'FOND'` resource (let's forget about `'FONT'`s altogether), so the font name can be any Pascal string. Unfortunately, this conflicts with the 31-character limitation of a file name when the System 7 Finder derives the file name of a movable font file (*Inside Macintosh* Volume VI, page 9-34) from the font name. Some third-party fonts come with font names long enough to cause trouble. You may also see this problem when trying to open a suitcase if the Finder can't generate distinct names for all of the fonts in the suitcase; the Finder may say the suitcase is "damaged" when it is not.

  Note: Each TrueType `'sfnt'` resource contains a Naming Table (see *The TrueType™ Font Format Specification*, APDA™ M0825LL/A) which provides nearly unrestricted font naming capabilities, to accommodate the needs of font manufacturers. A forthcoming Macintosh Technical Note on TrueType Naming Tables gives additional information.

- QuickDraw and the current Font Manager have no provision for stylistic variants like "light," "medium," "demi," "book," "black," "heavy," "extra," "ultra," etc., used in the context of professional typesetting. Therefore, each of these variants comes with a separate font family resource. Probably for reasons of consistency, the "italic" variants have their own font family resources as well. Unfortunately, unless each `'FOND'` references both the "plain" and the "italic" font strikes, QuickDraw will no longer know a customized italic font strike exists.

  It is fairly easy, using System 7 and ResEdit, to merge two font families (named, for exmaple, "myFont" and "myFont italic") into one. This way, QuickDraw will automatically use the pre-designed italic font strike instead of creating one algorithmically. Follow these convenient steps:

  1. Make sure there is no resource ID conflict between the `'NFNT'`s and `'sfnt'`s belonging to both families.
  2. Make sure the style bits for italic are set in the font association table of "myFont italic."
  3. From ResEdit's File menu, "Get Info..." on the "myFont" `'FOND'` resource. Write down the resource ID of the "myFont" `'FOND'`.
  4. From ResEdit's File menu, "Get Info..." on the "myFont italic" `'FOND'`. Change its resource ID to be identical to the one you wrote down in step 3. Change its resource name to "myFont."
  5. Use the Finder in System 7 to move the contents of the "myFont italic" suitcase into the original "myFont" suitcase. It will merge all constituents into one font association table, and thus enable transparent substitution of the right font for QuickDraw's italic style.

Version Numbers

The `'FOND'` structure (see *Inside Macintosh* Volume IV, page 45, "FamRec") contains a field `ffVersion`, and inquiring minds naturally want to know more about it. Before anything else, however, please read the following disclaimer:

Disclaimer: The Font Manager does not check version numbers in a `'FOND'`, and we recommend that you not rely on the (intentionally vague) statements below, but rather analyze the data in the `'FOND'` independently.

Currently, values 0...3 may appear in the ffVersion field, with the following intended interpretations:

Version 0: Usually indicates that the 'FOND' has been created on the fly by the Font/DA Mover (or the System 7 Finder). But the 'FOND' for Palatino on the distribution disks of System 7 is a counterexample.

Version 1: Obviously indicates the first version when 'FOND's came out (*Inside Macintosh* Volume IV, page 36).

Version 2: Corresponds to the extension of the 'FOND' format documented in *Inside Macintosh* Volume V, page 185 (which does not mean that the 'FOND' actually contains a bounding box table).

Version 3: The 'FOND' is supposed to contain a bounding box table.

This brings up an annoying fact. All measurement values (referring to a hypothetical 1-point font) in the 'FOND' are in a 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. You would expect that negative values (like for ffDescent, or in the kerning tables) are represented in the usual two's-complement format, such that standard binary arithmetic applies. This is mostly true, but not always. Again, Palatino is a counterexample (and probably not the only one). To our knowledge, version 0 and version 1 'FOND's have negative values represented in a format where the most significant bit is the sign bit, and the rest represents the absolute value. However, there is nothing in the system software that enforces this, so counterexamples may exist.

Warning: Don't rely on the version number, but include sanity checks for the negative values in a 'FOND' instead! The following Pascal function shows how this can be done:

```
FUNCTION Check4p12Value(n: Integer): Integer;
{ n is a 4.12 fixed-point value; i.e., its "real" value is n/4096.    }
{ If n is "unreasonably negative," interpret the most significant bit }
{ as sign bit, and convert to the usual two's complement format.      }

   BEGIN
      IF n < $8FFF THEN { means: (4.12-interpretation of n) is below - 7 }
         Check4p12Value := - BitAnd(n,$7FFF)
     { i.e., mask sign bit, and take negative of absolute value }
      ELSE
         Check4p12Value := n;
   END;
```

In the Heart of the Font Manager

Swapping Fonts

As stated in *Inside Macintosh,* there is only one contact between QuickDraw and the Font Manager: the FMSwapFont function. Each of the three QuickDraw text *measuring* functions (CharWidth, StringWidth and TextWidth) always ends up in the QuickDraw bottleneck procedure QDProcs.txMeasProc. Each of the three QuickDraw text *drawing* procedures (DrawChar, DrawString and DrawText) always ends up in the QDProcs.textProc bottleneck procedure. Any reasonable textProc (like StdText) needs to call the currently-installed text measuring bottleneck procedure before actually rendering the text. And what does any reasonable text measuring bottleneck procedure (like StdTxMeas) do first, before anything else? It calls `FMSwapFont`, to make sure we are talking about the right font and its properties! (To be precise, `GetFontInfo` and `FontMetrics` are the other calls that make sure the right font is swapped in and set up, without requiring you to call `FMSwapFont` explicitly.)

Responding to a font request is a lot of work, and `FMSwapFont` has been optimized to return as quickly as possible if the request is the same as the previous one. Building the **global width table** (see *Inside Macintosh* Volume IV, page 41) is among the more time-consuming tasks related to `FMSwapFont`; this is why the Font Manager maintains a cache of up to 12 width tables.

Inside Macintosh Volume I, page 220 documents the Font Manager's choice when a font of the requested size is not available. However, some consequences or additional features have occasionally been a surprise to developers (and users as well).

Scaling Factors in FMOutPut and StdTxMeas

Let's suppose you have only a 12-point bitmap version of Palatino, and don't have any Palatino outline fonts. When you request Palatino 18, QuickDraw sets up the `FMInput` record with `size = 18` and `numer = denom = Point($00010001)`.On return, the `FMOutput` record contains the handle to the font record to use (the `'NFNT'` with the Palatino 12 bitmap font strike), and indicates the scaling factors QuickDraw will have to use to produce the desired text point size in `FMOutput.numer` and `FMOutput.denom`. In this example, that ratio is 3/2.

Note that these are also the values returned in `StdTxMeas` (*Inside Macintosh* Volume I, page 199) if you call the procedure with `numer = denom = Point($00010001)`. Why? Because `StdTxMeas` calls `FMSwapFont`, as explained under "Swapping Fonts." `StdTxMeas` does **not** apply these scaling factors to the text it measures. In our example, it would measure Palatino 12 and return `numer` and `denom` in the ratio 3/2 to tell you that your application must multiply the results by these values to get the correct measurements for Palatino 18. This has surprised more than one programmer who didn't expect `numer` and `denom` to change!

By the way, the Font Manager always normalizes the scaling factors as fractions numer/denom such that the denominator is `equal to 256`. In our example, the real numbers returned by `FMSwapFont` or `StdTxMeas` are `numer = 384` and `denom = 256`.

Warning: If the scaling factors `numer` and `denom` passed to `StdTxMeas`, `StdText` (see *Inside Macintosh* Volume I, pages 198 and 199), or in the `FMInput` record to `FMSwapFont` are such that `txSize*numer.v/denom.v` is less than 0.5 and rounds to 0, and if there is more than one `'sfnt'` resource referenced in the font association table, then the current Font Manager may get confused and return results for the wrong font strike.

TrueType Always Has the Right Size

The default value of `outlinePreferred` is `FALSE`. If you have bitmap fonts for Palatino 12 and Palatino 14 in your system as well as a Palatino TrueType font, then requests for Palatino 12 or Palatino 14 are fulfilled with the bitmap fonts, but requests for any other size are fulfilled with the TrueType font. In particular, if you (or, for example, a printer driver) need Palatino 12 scaled by 2, the Font Manager will actually look for Palatino 24 and return the outline font, regardless of the setting of `outlinePreferred`. Even if you wanted the bitmap font doubled for exact "what-you-see-is-what-you-get" text placement, you're out of luck—you get the TrueType font, which may have very different font metrics or character shapes.

If the Font Manager uses an outline font to fulfill a given font request, the `IsOutline` function returns `TRUE`. Interestingly, this does not imply that `RealFont` returns `TRUE` as well. If the text size is smaller than the value `lowestRecPPEM` ("smallest readable size in pixels") in the `'head'` font header in the TrueType font (see *The TrueType Font Format Specification*, version 1.0, page 227), then `RealFont` returns `FALSE`!

First Size, Then Style—or: To Be or Not to Be Outline

When the Font Manager walks the font association table of a `'FOND'` to look for a font strike of a specified size and style, it stops at the first font of the right size. Only if you requested a stylistic variant (like bold or italic) does it take a closer look at the fonts of the same size. It does this by putting weights on the various style bits (for example, 8 for italic, 4 for bold, 3 for outline) and choosing the font strike whose style weight most closely matches the weight of the requested style. All this is fine when only bitmap fonts are available. With the presence of TrueType outlines, however, the results are not always as expected, depending on the font configuration installed.

Let's look at a few examples:

Example 1: Let's suppose you have the bitmap font Times 12 (Normal) and the TrueType fonts Times (Normal), Times Italic and Times Bold in your system. If you request Times 14 Italic or Times 14 Bold, it's rendered from the Times Italic or Times Bold TrueType fonts. However, if you ask for Times 12 Italic or Times 12 Bold, and your system has the default setting of `outlinePreferred = FALSE`, the Font Manager decides to take the Times 12 bitmap and let QuickDraw algorithmically slant it (for italics) or smear it (for bold).

Example 2: Let's suppose you want to draw big, bold Helvetica characters and there are no existing bitmaps for the size you want. If the Helvetica Bold TrueType outlines are available, the Font Manager chooses them and the only surprise in text rendering will be a pleasant one. If there is no Helvetica Bold TrueType font, however (like in the machine of your customer, who kept only the normal Helvetica TrueType font in his system), then the characters are rendered using the normal Helvetica outlines and, in a second step, QuickDraw applies its horizontal 1-pixel "smearing" to simulate the bold stylistic variant. The result is very different (and rather an unpleasant surprise).

Example 3: Admittedly, this is less likely (but it has happened). Let's suppose somebody decides to rip the Times TrueType outline out of the System file (don't ask me why—I don't know). He forgets to take the Times Italic TrueType outline away as well. The next time he draws text in Times (Normal), in a size for which there is no bitmap font (or if `outlinePreferred = TRUE`), the Font Manager goes for an `'sfnt'`, and the text shows up in *italic* (what a surprise!).

Unfortunately, given the current implementation of the Font Manager, there are no solutions to the problems illustrated above—other than asking users of your application to install the fonts you recommend. The only way to anticipate these potential surprises from within your application is to look into the 'FOND's font association table. You can't depend on the IsOutline function because it returns TRUE as soon as the Font Manager stops at an 'sfnt', in its first pass through the font association table—regardless of subsequent stylistic variations. This means, for example, if you ask for Helvetica Bold and IsOutline returns TRUE, you don't know if you got the Helvetica Bold TrueType font or if QuickDraw "smeared" the Helvetica (Plain) TrueType font.

Where Do the Widths Come From?

Text measuring (for example, for precise text placement in forms with bounding boxes) and most line layout algorithms for justified text rely heavily on the character widths contained in the global width table. Given that under the current font architecture, we may easily have three or more different width tables for the same font specification (the non-proportional integer widths attached to the 'NFNT', the fractional widths contained in the 'FOND', and the fractional widths provided by the 'sfnt'), it is important to understand where the widths come from in any case.

Since SetFractEnable was introduced (*Inside Macintosh* Volume IV, page 32 and Volume V, page 180), its setting TRUE or FALSE was supposed to give predictable effects. If it's FALSE, the Font Manager takes the integer widths from the 'NFNT'; if it's TRUE, it takes the fractional widths from the 'FOND'. Unfortunately, there are some additional details and side effects that are not well known.

- The Font Manager looks at bit 14 of the ffFlags field in the 'FOND' (see *Inside Macintosh* Volume IV pages 36 and 37). If it is set (like it is for Courier), the fractional widths from the 'FOND' are *never* used.
- If SetFractEnable is TRUE and you request a stylistic variation like bold or italic, the Font Manager looks at bits 12 and 13 of the ffFlags field to decide how different widths or extra widths for the stylistic variants have to be used. What it decides is documented in the "Font Manager" chapter of *Inside Macintosh Preview*, located on the *Developer CD Series* discs.
- Given that it is not possible to set the pen to a fractional position, precise text positioning with fractional widths enabled is always compromised because of (accumulated) rounding errors.
- QuickDraw distributes the accumulated rounding errors across characters within a string (instead of adding it at the end of the drawn text). This results in poor text quality on the screen, and in problems when calculating the position of the insertion point between characters.
- The LaserWriter driver watches what you pass to SetFractEnable. Passing TRUE to SetFractEnable disables some of the LaserWriter driver's line layout features, assuming that the programmer intends to control text placement manually. Explicitly passing FALSE to SetFractEnable achieves different results than using the default value of FALSE—Font Substitution behaves differently, for example. These effects are sometimes Not What You Wanted.
- On non-32-Bit-QuickDraw systems, SetFractEnable is not recorded in pictures. This affects the line layout of text reproduced through DrawPicture if the picture was created with fractional widths enabled.

In systems with TrueType, quite naturally the widths *always* come from the 'sfnt' when the Font Manager uses a TrueType font. If fractEnable is FALSE, hand-tuned integer character widths for specific point sizes come from the 'hdmx' table in the 'sfnt'. If fractEnable is FALSE and no 'hdmx' table is present or it contains no entries for the desired point size, the fractional character widths from the 'sfnt' are rounded to integral values.

More Line Layout Problems

The routines `SpaceExtra` (*Inside Macintosh* Volume I, page 172) and `CharExtra` (*Inside Macintosh* Volume V, page 77; available only in color GrafPorts) are intended to help you draw fully justified text. This works fine on the screen, but not all printer drivers are smart enough to use these settings appropriately under all circumstances. In particular, if you pass `TRUE` to `SetFractEnable`, or if you turn the LaserWriter driver's line layout algorithm off (by means of the picture comment `LineLayoutOff;` see Macintosh Technical Note #91), or if font substitution is enabled and actually occurs, it is better not to rely on `SpaceExtra` and `CharExtra` when printing fully justified text. Instead, keep the LaserWriter driver's line layout adjustments off, and calculate the placement of your text (word by word, or even character by character) yourself.

Putting Text Into Boxes

TrueType fonts came to the Macintosh together with seven new Font Manager routines (as documented in *Inside Macintosh* Volume VI, Chapter 12). The `OutlineMetrics` function is certainly the most sophisticated of these, and sample code illustrating its usage may be helpful. The following procedure `DrawBoxedString` assumes that the new outline calls (*Inside Macintosh* Volume VI, Chapter 12) are available, and that `IsOutline` returns `TRUE` for the current port setting.

```
PROCEDURE DrawBoxedString(pt: Point; s: Str255);
{ Draw string s at pen position (pt.h, pt.v), and show each character's bounding box. }

   CONST
      kOneOne = $00010001;

   VAR
      advA: FixedPtr;
      lsbA: FixedPtr;
      bdsA: RectPtr;
      err,i,yMin,yMax,leftEdge,temp: Integer;
      numer,denom: Point;
      advance,lsb: Fixed;
      r: Rect;

   BEGIN
      numer := Point(kOneOne);
      denom := Point(kOneOne); { unless you want to draw with scaling factors
                                 .... }
      MoveTo(pt.h,pt.v);
      DrawString(s);
{ This is for the pleasure of your eyes only - in practice, you would probably }
{ first look at the metrics, and then decide where and how to draw the string! }
      advA := FixedPtr(NewPtr(Length(s) * SizeOf(Fixed)));
      lsbA := FixedPtr(NewPtr(Length(s) * SizeOf(Fixed)));
      bdsA := RectPtr(NewPtr(Length(s) * SizeOf(Rect)));
      { Please, check for NIL pointers here! }
      err := OutlineMetrics(Length(s),@s[1],numer,denom,yMax,yMin,advA,lsbA,
                            bdsA);
      advance := 0;
      FOR i := 1 TO Length(s) DO { for each character }
         BEGIN
         { Add accumulated advanceWidth and leftSideBearing of current glyph }
         { horizontally to starting point. }
         leftEdge := pt.h + Fix2Long(advance + lsbA^);
      r := bdsA^; { The bounding box rectangle is in TrueType coordinates. }
      temp := r.bottom; { need to flip it "upside down" }
      r.bottom := - r.top;
      r.top := - temp;
      OffsetRect(r,leftEdge,pt.v);
      FrameRect(r); { This is the glyph's bounding box. }
      advance := advance + advA^;
      { "Advance" is Fixed, to avoid accumulation of rounding errors. }
      { Now, bump pointers for next glyph. }
      bdsA := RectPtr(ord4(bdsA) + SizeOf(Rect));
      advA := FixedPtr(ord4(advA) + SizeOf(Fixed));
      lsbA := FixedPtr(ord4(lsbA) + SizeOf(Fixed));
   END;
   DisposPtr(Ptr(advA));
   DisposPtr(Ptr(lsbA));
   DisposPtr(Ptr(bdsA));
END; { DrawBoxedString }
```

`OutlineMetrics` exists because many developers need pixel-precise information on placement and bounding boxes, often on a character-by-character basis. Unfortunately, there is no similar facility for text drawing with bitmap fonts. Worse, under certain circumstances, italicized or shadowed (or both) bitmap fonts are sometimes poorly clipped, particularly for scaled sizes. Cosmetic workarounds include adding a space character to strings drawn in italic. You might also draw the text off-screen first (in order to determine the bounding box of the black pixels) and use `CopyBits` to copy the text onto the screen—but using `CopyBits` for text is usually bad for printing.

The existing documentation on the `FMOutput` and global width table structures (*Inside Macintosh* Volume I, page 227 and Volume IV, page 41) suggests it's possible to devise a routine for determining a fairly precise text bounding box for bitmap fonts. The procedure below, `BitmapTextBoundingBox`, is a first attempt. It assumes that TrueType is unavailable, or that the `IsOutline` call returned `FALSE` for the current port settings. While the returned bounding box is not always "tight," be careful before modifying the algorithm and shrinking the resulting bounding box—bitmap fonts just don't contain enough precise information for an exact bounding box, and different bitmap fonts and different sizes may require different adjustments.

```
PROCEDURE TextBoundingBox(s: Str255; numer,denom: Point; VAR box: Rect);

   CONST
      FMgrOutRec = $998; { FMOutRec starts here in low memory }
      tabFont = 1024;
      { global width table offset for font record handle, see IM IV-41 }

   TYPE
      FontRecPtr = ^FontRec;

   VAR
      hScale,vScale: Fixed;
      err,intWidth,kernAdjust: Integer;
      xy: Point;
      info: FontInfo;  { only for StdTxMeas; we'll use FontMetrics }
      fm: FMetricRec;  { see Inside Macintosh, IV-32 }
      fmOut: FMOutput;
      h: Handle;

   BEGIN
      intWidth := StdTxMeas(ord(s[0]),@s[1],numer,denom,info);
      { calls FMSwapFont and everything - }
      { StdTxMeas returns possibly modified scaling factors numer, denom }
      hScale := FixRatio(numer.h,denom.h);
      vScale := FixRatio(numer.v,denom.v);
      { These are the scaling factors QuickDraw uses     }
      { in "stretching" the available character bitmaps }
      fmOut := FMOutPtr(FMgrOutRec)^;
      { has been filled by the most recent FMSwapFont,   }
      { implicitly called by StdTxMeas  }
      SetRect(box,0, - info.ascent,intWidth,info.descent);
      { bounding box for unscaled plain text }
      IF (italic IN thePort^.txFace) AND (fmOut.italic <> 0) THEN BEGIN
      { the following is heuristics … )
         box.right := box.right + (info.ascent + info.descent - 1) *
                      fmOut.italic DIV 16;
         FontMetrics(fm);
         HLock(fm.WTabHandle); { We'll point to global WidthTable. }
         h := Handle(LongPtr(ord4(fm.WTabHandle^) + tabFont)^);
         { Be sure it's a handle to a 'NFNT' or 'FONT' ! }
         kernAdjust := FontRecPtr(h^)^.kernMax;
         OffsetRect(box, - kernAdjust,0);
         HUnlock(fm.WTabHandle);
      END;
      IF (bold IN thePort^.txFace) AND (fmOut.bold <> 0) THEN
         box.right := box.right + fmOut.bold - fmOut.extra;
      IF (outline IN thePort^.txFace) THEN InsetRect(box, - 1, - 1);
      IF (shadow IN thePort^.txFace) AND (fmOut.shadow <> 0) THEN BEGIN
         IF fmOut.shadow > 3 THEN fmOut.shadow := 3;
         box.right := box.right + fmOut.shadow;
         box.bottom := box.bottom + fmOut.shadow;
         InsetRect(box, - 1, - 1);
      END;
      { Now scale the box (more or less) as QuickDraw would do. }
      { Note that some of the adjustments are based on trial and error… }
      box.top := FixRound(FixMul(Long2Fix(box.top),vScale));
      box.left := FixRound(FixMul(Long2Fix(box.left),hScale)) - 1;
      box.bottom := FixRound(FixMul(Long2Fix(box.bottom),vScale)) + 1;
      box.right := FixRound(FixMul(Long2Fix(box.right),hScale)) + 1;
      GetPen(xy);
      OffsetRect(box,xy.h,xy.v);
   END;
```

Conclusion

At the time when the original Font Manager architecture was designed, based on QuickDraw's hard-coded 72 dpi resolution, nobody could anticipate that some years later, the Macintosh would be used to tackle professional typesetting projects. Several advanced page layout applications managed to work around the "built-in" limitations, at high development costs, and some compatibility and performance problems. In many other cases, however, those limitations caused questions to DTS and unsatisfying compromises. This Note can't do much more than explain the state of affairs; the real solution to the problems must come from a redesigned foundation. TrueType leads the way and already fulfills many of the requirements; everything else is getting closer and closer.

Further Reference:

- *Inside Macintosh*, Volume I, Chapter 7, The Font Manager
- *Inside Macintosh*, Volume IV, Chapter 5, The Font Manager
- *Inside Macintosh*, Volume V, Chapter 9, The Font Manager
- *Inside Macintosh*, Volume VI, Chapter 12, The Font Manager
- *New & Improved Inside Macintosh*, Imaging: The Font Manager. *Developer CD Series* disc, path Developer Essentials: Technical Docs: Inside Macintosh Preview
- Macintosh Technical Note #91, Picture Comments—The Real Deal
- Macintosh Technical Note #191, Font Names
- Macintosh Technical Note #242, Fonts and the Script Manager
- Macintosh Technical Note #245, Font Family Numbers
- *Apple LaserWriter Reference,* Chapter 2, Working With Fonts (Addison-Wesley, 1988)
- Adobe Technical Note #0091 (PostScript Developer Support Group), Macintosh FOND Resources

PostScript and Adobe are registered trademarks of Adobe Systems Incorporated.
Helvetica and Palatino are registered trademarks of Linotype AG and/or its subsidiaries.

Velocio is **not** a trademark of the author.

Developer Technical Support

#27: MacDraw's PICT File Format

Revised: August 1989
Written by: Ginger Jernigan August 1986

This Technical Note formerly described the PICT file format used by MacDraw® and the picture comments the MacDraw used to communicate with the LaserWriter driver.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the PICT file format used by MacDraw, which is now published by CLARIS. For information on MacDraw (its specific use of the PICT format) and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

Inside Macintosh, Volume V–39, Color QuickDraw and Technical Note #21, QuickDraw's Internal Picture Format, now document the PICT file format. Technical Note #91, Optimizing for the LaserWriter—Picture Comments, now documents the picture comments which the LaserWriter driver supports.

Further Reference:

- *Inside Macintosh*, Volume V–39, Color QuickDraw
- Technical Note #21, QuickDraw's Internal Picture Format
- Technical Note #91, Optimizing for the LaserWriter—Picture Comments

MacDraw is a registered trademark of CLARIS Corporation.

#28: Finders and Foreign Drives

Written by: Ginger Jernigan May 7, 1984
Updated: March 1, 1988

This technical note describes the differences in the way the 1.1g, 4.1, 5.0 and newer Finders communicate with foreign (non-Sony) disk drives.

Identifying Foreign Drives

Non-Sony disk drives can send an icon and a descriptive string to the Finder; this icon is used on the desktop to represent the drive. The string is displayed in the “Get Info” box for any object belonging to that disk. When the Finder notices a non-Sony drive in the VCB queue, it will issue 1 or 2 control calls to the disk driver to get the icon and string.

Finder 1.1g issues one control call to the driver with `csCode` = 20 and the driver returns the icon ID in `csParam`. This method has problems because the icon ID is tied to a particular system file. So, if the Finder switch-launches to a different floppy, the foreign disk's icon reverts to the Sony's.

Finders 4.1 and newer issue a newer control call and, if that fails, they issue the old Control call. The new call has `csCode` = 21, and the driver should return a pointer in `csParam`. The pointer points to an 'ICN#' followed by a 1 to 31 byte Pascal string containing the descriptor. This implies that the icon and the string must be part of the disk driver's code because only the existence of the driver indicates that the disk is attached.

This has implications about the translation of the driver for overseas markets, but the descriptor will usually be a trademarked name which isn't translated. However, the driver install program could be made responsible for inserting the translated name into the driver.

Drivers should respond to both control calls if compatibility with both Finders is desired.

Formatting Foreign Drives

When the user chooses the Erase Disk option in the Finder, a non-Sony driver needs to know that this has happened so it can format the disk. Finder 4.1 and newer notify the driver that the drive needs to be formatted and verified. They first issue a `Control` call to the driver with the `csCode` = 6 to tell the disk driver to format the drive. Then they issue a `Control` call with a `csCode` = 5 to tell the driver to verify the drive.

Other Nifty Things to Know About

Finders 4.1 and newer also permit the user to drag any online disk to the trash can. The Finder will clean up the disk state, issue an `Eject` call followed by an `Unmount` call to the disk and then, an event loop later, reclaim all the memory. This means any program/accessory used to mount volumes should reconcile its private data, menus, etc. to the current state of the VCB queue. These Finders also notice if a volume disappears and will clean up safely. But, because of a quirk in timing, a mount manager cannot unmount one volume then mount another immediately; it must wait for the Finder to loop around and clean up the first disk before it notices the second. (It should have cleaned up old ones before it notices new ones, but it doesn't.)

Finders 5.0 and newer allow you to drag the startup disk to the trash; Finder 4.1 just ignored you. Finders 5.0 and newer take the volume offline as if you had chosen Eject.

#29: Resources Contained in the Desktop File

See also: The Finder Interface

Written by: Ginger Jernigan May 7, 1985
Modified by: Ginger Jernigan December 2, 1985
Updated: March 1, 1988

This technical note describes the resources found in the Desktop file. **Note:** Don't base anything critical on the format of the Desktop file. AppleShare already uses another scheme; AppleShare volumes don't have Desktop files. The format of this file can, and probably will, change in the future.

The Desktop file contains almost the same resources for both the Macintosh File System (MFS) and the Hierarchical File System (HFS). This technical note describes the resources found in both. This information is for reading only. This means your application can read it but it should NEVER write out information of its own, because the Finder, as well as Macintosh Developer Technical Support, won't like it.

The Desktop is a resource file which contains the folder information on an MFS volume, the "Get Info" comments, the application bundles, 'FREF's and 'ICN#'s, and information concerning the whereabouts of applications on an HFS disk. Everything except the comments are preloaded when the desktop is opened, making it easier for the Finder to find things.

The contents of the Desktop file are described below. The resource types are the same for both MFS and HFS volumes unless otherwise stated.

'APPL': This resource type is used by the HFS to locate applications. This is used by the Finder to locate the right application when a document is opened. Each application is identified by the creator, the directory number, and the application name. This is used only by HFS.

'BNDL': This resource type contains a copy of all of the bundles for all of the applications that are either on the disk or are the creators of documents that are on the disk. This is used by the Finder to find the right icons for documents and applications. If you have a document whose creator the Finder has not seen yet, it will not be in the Desktop file and the default document icon will be used.

'FREF': This contains a copy of all of the FREFs referenced in the bundles.

'FCMT': This resource contains all of the "Get Info" comments for applications and documents. On MFS volumes the ID is a hash of the object's name. The hashing algorithm is as follows:

```
; FUNCTION HashString(str: Str255): INTEGER;

; The ID for the FCMT returned in function result

HashString
        MOVE.L    (SP)+,A0        ; get return address
        MOVE.L    (SP)+,A1        ; get string pointer

        MOVEQ     #0,D0           ; get string length
        MOVE.B    (A1)+,D0

        MOVEQ     #0,D2           ; accumulate ID here
@2
        MOVE.B    (A1)+,D1        ; get next char
        EOR.B     D1,D2           ; XOR in
        ROR.W     #1,D2           ; stir things up
        BMI.S     @1              ; ID must be negative
        NEG.W     D2
@1
        SUBQ.W    #1,D0           ; loop until done
        BNE.S     @2              ; until end of string

        MOVE      D2,(SP)         ; return the hashed code
        JMP       (A0)
```

For HFS volumes, the ID of the resource is randomly generated using `UniqueID`. To find the ID of the comment for a file or directory call `PBGetCatInfo`. The comment ID for a file is kept in `ioFlXFndrInfo.fdComment`. The comment ID for a directory is kept in `ioDrFndrInfo.frComment`.

'FOBJ': This resource type contains all of the folder information for an MFS volume. The format of this resource is not available. This is only in an MFS volume's Desktop file.

'ICN#': This resource type contains a copy of all of the 'ICN#' resources referenced in the bundles and any others that may be present.

'STR ': This is a string that identifies the version of the Finder, but it isn't always correct.

Creators: A resource with a type equal to the creator of each application with a bundle is stored in the Desktop file for reference purposes only. The data stored in these resources is for the Finder's use only.

Be aware that if a resource is copied from an application resource file and there is an ID conflict, the Finder will renumber the resource in the Desktop file.

#30: Font Height Tables

See Also: The Font Manager
The Resource Manager

Written by: Gene Pope April 25, 1986
Updated: March 1, 1988

This technical note describes how the Font Manager (except in 64K ROMs) calculates height tables for fonts and how you can force recalculation.

In order to expedite the processing of fonts, the Font Manager (in anything newer than the 64K ROMs) calculates a height table for all of the characters in a font when the font is first loaded into memory. This height table is then appended to the end of the font resource in memory; if some program (such as a font editor) subsequently saves the font, the height table will be saved with the font and will not have to be built again. This is fine for most cases except, for example, when the tables really should be recalculated, such as in a font editor when the ascent and/or descent have changed.

The following is an example of how to eliminate the height table from a font:

```
IF (BitAnd(hStrike^^.fontTyp,$1)=1) THEN BEGIN {We have a height table}
   {Truncate the height table}
   SetHandleSize(Handle(hStrike),GetHandleSize(Handle(hStrike)-
                 (2*(hStrike^^.lastChar-hStrike^^.firstChar)+3)));
   {We no longer have a height table so set the flag to indicate that}
   hStrike^^.format := BitAnd(hStrike^^.fontType,$FFFFFFFE);
END;
```

In MPW C:

```
if ((**hStrike).fontType & 0x1 ==1) { /*We have a height table*/
   /*Truncate the height table*/
   SetHandleSize((Handle)hStrike,GetHandleSize((Handle)hStrike)-
                 (2*((**hStrike).lastChar-(**hStrike).firstChar)+3));
   /*We no longer have a height table so set the flag to indicate that*/
   (**hStrike).fontType = (**hStrike).fontType & 0xFFFFFFFE;
}
```

where `hStrike` is a handle to the ‘FONT’ or ‘NFNT’ resource (handle to a `FontRec`).

Note: After the height table has been eliminated, the modified font should be saved to disk (with `ChangedResource` and `WriteResource`) and purged from memory (using `ReleaseResource`). This is an important step, because the Font Manager does not expect other code to go behind its back removing height tables that it has calculated.

Macintosh
Technical Notes

REVIEW DRAFT

Developer Technical Support

#31A: GestaltWaitNextEvent

Revised by: C.K. Haun <TR> April 1 1992

This Technical Note discusses a new Event Manager call in Macintosh System Software.

The Changing World

The Macintosh operating environment is changing rapidly. Modular system software, dynamically linked libraries, plug and play hardware, all add up to a confusing environment for the application programmer.

To dispel this confusion, it is essential that an application *always* know what features are available for its use. The user experience will be greatly enhanced when the user can drop a new system extension into their System Folder and immediately use it in all applications.

To allow this, a new function (provided as a system extension) has been added to System 7 and later, `GestaltWaitNextEvent`.

The best way to explain GWNE is to see it in action. The function prototype for `GWNE` is:

```
pascal EventReturnStructHandle GestaltWaitNextEvent(EventMaskHandle
theMask,SleepHandle sleepValue,GestaltAvailableHandle
featuresAvailable,GestaltAvailableHandle minimumNeeded,GWNECallbackHandle
myCallBack);
```

The first thing you'll notice is that the mouse region parameter is missing. No one could ever figure this out, so it's been dropped.

There are six new structures defined for this call.

The first is the `EventReturnStruct`. Since you never know what features may be connected to your Mac, you can never be certain what events you'll get back. Also, it is possible to get multiple events simultaneously, depending on the types of devices and extensions the user has installed So this variable structure has been created to let you know what happened during the event call.

```
struct EventReturnStruct{
      unsigned long         NumberOfEvents;
      struct EventRecord2   **theEvents;
};
```

where EventRecord2 is:

```
struct EventRecord2 {
      unsigned long     typeOfEvent;
      Handle            eventData;
      DateTimeRec       eventTime;
      EventRecord2      **nextEvent;
};
```

When GWNE returns, you will then walk through the linked list of EventRecord2 structures, examining the event type and parsing the data in the eventData field as appropriate for that event. The numberOfEvents parameter is available to quickly determine how many events have occurred. Since it is possible for you to get up to 4294967295 events per GWNE call (or up to available memory) it may be appropriate to display a watch cursor or 'please wait' dialog after returning from GWNE.

Also please note that each event contains a DateTimeRec structure. Ticks are not enough for some events, for example if the SubSpace manager (see *develop* issue 7) is installed, the normal starting point of Jan 1 1904 is not adequate, since events posted many millennia earlier or later may also be queued to your machine. Please see the specific event source documentation for explanation of this record for specific events.

The next new structure is the EventMaskStruct. This is necessary since there is a large amount of possible events (again, up to 4294967295) that you may be interested in, and they may have different masking needs.

```
struct EventMaskStruct{
      unsigned long            typeOfEvent;
      Handle                   eventAcceptParameters;
      Handle                   eventRefuseParameters;
      struct EventMaskStruct   **nextEventMask;
};
```

You'll note that you can pass reasons both for accepting or refusing any event, the contents of these handles is determined by the eventType field.

Warning: You *must* pass a handle in both eventAcceptParameters and eventRefuseParameters. Failure to do so may cause an event not intended for your computer to be accepted.

The old sleep value has also changed. The new structure SleepHandle defines not only how long you'll sleep, but also if you should wake up for any specified event. This gives you much more flexibility to customize your application to meet the real needs of your customers.

```
struct SleepStruct{
      unsigned long            typeOfEvent;
      Handle                   eventWakeParameters;
      Handle                   eventStayAsleepParameters;
      EventMaskHandle          XOREvents;
      EventMaskHandle          ANDEvents;
      EventMaskHandle          OREvents;
      EventMaskHandle          NOTEvents;
      struct SleepStruct       **nextSleep;
};
```

The new sleep structure gives you much finer control over what you wish to wake up for. Besides passing the wake up parameters and stay sleeping parameters (the definition of these parameters is determined by the event number) you also pass handles to the events that may relate to the event you are concerned about.

For example, you pass a SleepStruct for a kMonitorMoved event that specifies that you should only be awakened if the monitor moved more that 75 degrees vertically, but stay sleeping if the move angle exceeds 90 degrees vertical. This may be all that is required, but you may also be concerned about *what* caused that to happen. If you pass an event mask for a `kCatJumpedOnMontior` as one of the `ANDEvent` parameters, then you will be wakened if the 75-90 tilt is the result of the `kCatJumpedOnMontior`. If there are some simultaneous events that you *don't* care about, pass them in the `NOTEvents`. In this case, you may pass a `kEarthQuakeEvent` mask with a value of `kLessThanRichter4.0` as a parameter. This would indicate that you want to be wakened *if* monitor moved more that 75 degrees vertically, but stay sleeping if the move angle exceeds 75 degrees vertical *and* this was not caused by a small earthquake.

A few experiments will make this clear, and you'll be glad to have the control you have.

The next new parameter is the `GestaltAvailableHandle`, this will return to you a list of current system features. This will allow you to dispatch rapidly to the appropriate routine when the user adds or deletes a system feature.

```
struct GestaltAvailable {
        Boolean             changed;
        Boolean             added;
        FeatureStruct       **addedFeatures;
        Boolean             removed;
        FeatureStruct       **removedFeatures;
};
```

where FeatureStruct is:

```
struct FeatureStruct{
        OSType                      selector;
        long                        response;
        OSErr                       result;
        struct FeatureStruct        **nextFeature;
};
```

The `selector` is self-explanatory. `Response` and `result` are included here, because `GWNE` will automatically call all the currently installed Getstalt selectors during its call.

The next parameter to GWNE is another `GestaltAvailableHandle`. This record specifies the minimum requirements your application has to be awakened again.

While we hope every application is rewritten to take advantage of every possible system configuration dynamically, we understand that there are some smaller shops where this will not be possible for a few months after GWNE goes into general use. For example, there may be some applications that will take a while to revise to continue working when the user removes QuickDraw from the system.

If this is the case for your application, in this parameter all the features that you need to run in `minimumNeeded`.

Note: `Please do not abuse this feature.` If your application is too picky and not ready to handle many different configurations, it is possible for you to call GWNE and never return. The user would be confused by this.

The final new structure is the `GWNECallbackHandle`

```
struct GWNECallbackHandle{
        VoidProcPtr callBack;
        FeatureStruct      **featuresNeeded;
```

```
      short minimumCallBackMinutes;
};
```

Because of the power of GWNE , it sometimes takes a longer time to complete than the older WNE routine. If you would like to take some periodic action during a GWNE call, pass this structure. GWNE will call your callBack proc when the amount of minutes specified in `minimumCallBackMinutes` has elapsed *if* the feature set you defined in `featuresNeeded` is available.

Cautionary Notes

Obviously `GWNE` is going to take a little more time than the older `WaitNextEvent` call. Also, `GWNE` disables interrupts for the duration of the call to prevent new selectors and features from being added while the call is in progress.

This should not be a problem for a well-behaved application, if you are checking Ticks instead of incrementing a variable during interrupt time you will not be affected

Note: `TickCount` now returns minutes, not sixtieths of a second.

We have determined the text editing applications may experience difficulty blinking an insertion point if the user has a great many features installed. We cannot fix this in current System Software, but all new hardware projects will be designed with a 'LCD Shutter' over the display, cycling once every 1.3 seconds. This will simulate the effect of a blinking cursor by blinking the whole screen regularly.

Determining if `GWNE` is available

At this writing, `GWNE` is designed to be a system extension, and there are no plans to incorporate it in core system software. Incorporating it in the core software would limit its effectiveness.

This means that determining its availability is problematic. You must call `GWNE` to determine if `GWNE` is available. We recommend the following code:

```
// Prior to calling GWNE, copy all RAM to disk to allow recovery if call fails

   CopyMachineRAMToDisk();  /* your routine */

// Install a bus error handler.  This will point to the code immediatly after
// the GWNE call
   InstallMyBusError();

// Call GWNE

myEvts=GestaltWaitNextEvent(myMaskHandle,mySleepHandle,returnedFeatureSet,mini
mumFeaturesNeeded,callBackHandle);
   if (didBusError){

// this flag will be set by your bus error handler.  If it is set,then GWNE is
// not currently installed.  Reload memory from disk
   CopyDiskImageBackToRAM();  /* your routine */

       CallWaitNextEvent();   // default to calling WNE
   }
```

NOTE: You cannot assume that GWNE will never be available if it was not available one time. The user may install or remove it at any time, so you must write your event loop in this fashion.

Conclusion:

GestaltWaitNextEvent answers the prayers of developers, and the needs of users. It gives a well defined, consistent interface to a fluid environment.

Obviously, existing applications will need some rewriting to become fully GWNE aware. We expect incorporation will take up to two weeks, and re-writing your code to be 'any feature aware' may take slightly longer. However, it will be worth the effort.

Further Reference:

- *Inside Macintosh*, Volume VII-XXIII, Possible Event Codes References

#32: Reserved Resource Types

See: The Resource Manager

Written by: Scott Knaster May 13, 1985
Updated: March 1, 1988

Your applications and desk accessories can create their own resource types. To avoid using type names which have been or will be used in the system, Apple has reserved all resource type names which consist entirely of spaces ($20), lower-case letters ($61 through $7A), and “international” characters (greater than $7F).

In addition Apple has reserved a number of resource types which contain upper-case letters and the “#” character. For a list of these resource types, see The Resource Manager Chapter of *Inside Macintosh* (starting with *Volume V*).

Macintosh Technical Notes

#33: ImageWriter II Paper Motion

Written by: Ginger Jernigan — April 30, 1986
Updated: March 1, 1988

The purpose of this technical note is to answer the many questions asked about why the paper moves the way it does on the ImageWriter II.

Many people have asked why the paper is rolled backward at the beginning of a Macintosh print job on the ImageWriter II. First, note that this only happens with pin-feed paper (i.e. not with hand-feed or the sheet-feeder) and only at the beginning of a job.

It is not a bug, and it is not malicious programming. It is simply that users are told in the manual to load pin-feed paper with the top edge at the pinch-rollers, making it easy to rip off the printed page(s) without wrecking the paper that is still in the printer or having to roll the paper up and down manually. At the end of every job, the software makes sure that the paper is left in this position, leaving the print-head roughly an inch from the edge. If something is to be printed higher than that, the paper has to be rolled backwards.

As you are probably aware, the "printable rectangle" (`rPage`) reported to the application by the print code begins 1/2 inch from the top edge, not one inch. The reason for that is that we want a document to print exactly the same way whether you are printing on the ImageWriter I or II. On the ImageWriter I, the paper starts with the print-head 1/2 inch from the top edge, so the top of `rPage` is at that position for both printers.

There is no way to eliminate the reverse-feed action, because the user would have to load the paper a different way AND the software would have to know that this was done.

Incidentally, in addition to the paper motion described above, there is also the "burp." This is a 1/8-inch motion back and forth to take up the slop in the printer's gear-train. It is needed on the old-model printer, and there is debate about whether or not it's needed on ALL ImageWriter IIs, or only some, or none. The burp has been in and out of the ImageWriter II code in various releases; right now it's in.

Macintosh Technical Notes

#34: User Items in Dialogs

See also: *Inside Macintosh*, The Dialog Manager

Written by: Bryan Stearns May 29, 1985
Updated: March 1, 1988
Revised by: Jim Reekes October 1, 1988

The Dialog Manager does not go into detail about how to manage user items in dialogs; this Technical Note describes the process.
Changes since March 1, 1988: Added MPW C 3.0 code, added a `_SetPort` call to the Pascal example, and noted the necessity and meaning of `enabled` items.

To use a `userItem` with the Dialog Manager, you must define a dialog, load the dialog and install your `userItem`, and respond to events which relate to your `userItem`. If your application wants to receive mouse clicks in the `userItem`, then you must set the item to `enabled`.

Defining a Dialog Box with a userItem

You should define the dialog box in your resource file as follows. Note that it is defined as invisible, since we have to play with the `userItem` before we can draw it.

```
resource 'DLOG' (1001) {                    /* type/ID for box */
   {100,100,300,400},                       /* rectangle for window */
   dBoxProc, invisible, noGoAway, 0x0,      /* note it is invisible */
   1001,
   "Test Dialog"
};

resource 'DITL' (1001) {                    /* matching item list */
   {
      {160, 190, 180, 280},                 /* rectangle for button */
         button { enabled, "OK" };          /* an OK button */
      {104, 144, 120, 296},                 /* rectangle for item */
         userItem { enabled }               /* a user item! */
   }
};
```

Loading and Preparing to Show the Dialog Box

Before we can actually show the dialog box to the user, we need two support routines. The Dialog Manager calls the first procedure whenever we need to draw our `userItem`. You should install it (as shown below) after calling `_GetNewDialog` but before calling `_ShowWindow`. This first procedure simply draws the `userItem`.

In MPW Pascal:

```
PROCEDURE MyDraw(theDialog: DialogPtr; theItem: INTEGER);

          VAR
             iType : INTEGER;                    {returned item type}
             iBox  : Rect;                       {returned bounds rect}
             iHdl  : Handle;                     {returned item handle}

          BEGIN
             GetDItem(theDialog,theItem,iType,iHdl,iBox); {get the box}
             FillRect(iBox,ltGray);              {fill with light gray}
             FrameRect(iBox);                    {frame it}
          END; {MyDraw}
```

In MPW C 3.0:

```
       pascal void MyDraw(theDialog,theItem)
       DialogPtr      theDialog;
       short int      theItem;

       {
              short int      iType;              /*returned item type*/
              Rect           iBox;               /*returned bounds rect*/
              Handle         iHdl;               /*returned item handle*/

              GetDItem(theDialog,theItem,&iType,&iHdl,&iBox); /*get the box*/
              FillRect(&iBox,qd.ltGray);         /*fill with light gray*/
              FrameRect(&iBox);                  /*frame it*/
       } /*MyDraw*/
```

The other necessary procedure is a filter procedure (filterProc) that the Dialog Manager calls whenever _ModalDialog receives an event (this only applies when calling _ModalDialog; modeless dialogs are covered below). The default filterProc looks for key-down and auto-key events and simulates pressing the OK button (or whatever else is item 1) if the user has pressed either the Return key or the Enter key. To support a userItem, the filterProc must handle events for any userItem items in the dialog in addition to performing the default filterProc tasks. The following short filterProc supports these types of items; when the user clicks in the userItem, the filterProc inverts it.

In MPW Pascal:

```
       FUNCTION MyFilter(theDialog: DialogPtr; VAR theEvent: EventRecord;
                         VAR itemHit: INTEGER): BOOLEAN;
          CONST
             enterKey    = 3;
             returnKey   = 13;

          VAR
             mouseLoc : Point;                   {we'll play w/ mouse}
             key      : SignedByte;              {for enter/return}
             iBox     : Rect;                    {returned boundsrect}
             iHdl     : Handle;                  {returned item handle}
             iType, itemHit : INTEGER;           {returned item and type}

          BEGIN
             SetPort(theDialog);
             MyFilter := FALSE;                  {assume not our event}
```

```
   CASE theEvent.what OF                  {which event?}
      keyDown,autoKey: BEGIN              {he hit a key}
         key := SignedByte(event.message); {get keycode}
         IF (key = enterKey) OR (key = returnKey ) THEN BEGIN
            MyFilter := TRUE;             {we handled it}
            itemHit := 1;                 {he hit the 1st item}
         END;                             {test CR or Enter}
      END;                                {keydown}
      mouseDown: BEGIN                    {he clicked}
         mouseLoc := theEvent.where;      {get the mouse pos'n}
         GlobalToLocal(mouseLoc);         {convert to local}
         GetDItem(theDialog,2,iType,iHdl,iBox); {get our box}
         IF PtInRect(mouseLoc,iBox) THEN BEGIN {he hit our item}
            InvertRect(iBox);
            MyFilter := TRUE;             {we handled it}
            itemHit := 2;                 {he hit the userItem}
         END;                             {if he hit our userItem}
      END;                                {mousedown}
   END;                                   {event case}
END;                                      {MyFilter}
```

In MPW C 3.0:

```
pascal Boolean MyFilter(theDialog,theEvent,itemHit)
DialogPtr      theDialog;
EventRecord    *theEvent;
short int      *itemHit;

#define enterKey      3;                    /*the enter key*/
#define returnKey     13;                   /*the return key*/

{
       char           key;                  /*for enter/return*/
       short int      iType;                /*returned item type*/
       Rect           iBox;                 /*returned boundsrect*/
       Handle         iHdl;                 /*returned item handle*/
       Point          mouseLoc;             /*we'll play w/ mouse*/

       SetPort(theDialog);
       switch (theEvent->what)              /*which event?*/
       {

              case keyDown:
              case autoKey: /*he hit a key*/
                     key = theEvent->message; /*get ascii code*/
                     if ((key == enterKey) || (key == returnKey))
                     {                     /*he hit CR or Enter*/
                            *itemHit = 1; /*he hit the 1st item*/
                            return(true); /*we handled it*/
                     } /*he hit CR or enter*/
                     break;                /* case keydown, case autoKey */
              case mouseDown:              /*he clicked*/
                     mouseLoc = theEvent->where;  /*get the mouse pos'n*/
                     GlobalToLocal(&mouseLoc);    /*convert to local*/
                     GetDItem(theDialog,2,&iType,&iHdl,&iBox); /*get our box*/
                     if (PtInRect(mouseLoc,&iBox))
                     {                     /*he hit our item*/
                            InvertRect(&iBox);
                            *itemHit = 2; /*he hit the userItem*/
                            return(true); /*we handled it*/
                     } /*if he hit our userItem*/
                     break; /*case mouseDown */
       } /*event switch*/
       return(false); /* we're still here, so return false
                            (we didn't handle the event) */
} /*MyFilter*/
```

Invoking the Dialog Box

When we need this dialog box, we load it into memory as follows:

In MPW Pascal:

```
PROCEDURE DoOurDialog;

   VAR
      myDialog : DialogPtr;                   {the dialog pointer}
      iType, itemHit : INTEGER;               {returned item type}
      iBox      : Rect;                       {returned boundsRect}
      iHdl      : Handle;                     {returned item Handle}

   BEGIN
      myDialog := GetNewDialog(1001,nil,POINTER(-1)); {get the box}
      GetDItem(myDialog,2,iType,iHdl,iBox); {2 is the item number}
      SetDItem(myDialog,2,iType,@myDraw,iBox); {install draw proc}
      ShowWindow(theDialog);                  {make it visible}
      REPEAT
         ModalDialog(@MyFilter, itemHit ); {let dialog manager run it}
      UNTIL itemHit = 1;                      {until he hits ok.}
      DisposDialog(myDialog);                 {throw it away}
   END;                                       {DoOurDialog}
```

In MPW C 3.0:

```
void DoOurDialog()

{

       DialogPtr      myDialog;      /*the dialog pointer*/
       short int      iType;         /*returned item type*/
       short int      itemHit;       /*returned from ModalDialog*/
       Rect           iBox;          /*returned boundsRect*/
       Handle         iHdl;          /*returned item Handle*/

       myDialog = GetNewDialog(1001,nil,(WindowPtr)-1); /*get the box*/
       GetDItem(myDialog,2,&iType,&iHdl,&iBox); /*2 is the item number*/
       SetDItem(myDialog,2,iType,MyDraw,&iBox); /*install draw proc*/
       ShowWindow(myDialog);         /*make it visible*/

       while (itemHit != 1) ModalDialog(MyFilter, &itemHit);
       DisposDialog(myDialog);       /*throw it away*/
}                                    /*DoOurDialog*/
```

Using userItem Items with Modeless Dialogs

If you are using `userItem` items in modeless dialog box, the Dialog Manager will call the draw procedure when `_DialogSelect` receives an update event for the dialog box. When the user clicks on your `userItem` and it is `enabled`, `_DialogSelect` will return `TRUE`. The `itemHit` will be equal to the item number of your `userItem`. Your code can then handle this like the mouse-down event case in the example above.

#35: DrawPicture Problem

Written by: Mark Baumwell June 19, 1986
Updated: March 1, 1988

This note formerly described a problem with DrawPicture that occurred only on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#36: Drive Queue Elements

See also: The File Manager
The Device Manager

Written by: Bryan Stearns June 12, 1985
Updated: March 1, 1988

This note expands on *Inside Macintosh*'s definition of the drive queue, which is given in the File Manager chapter.

As shown in *Inside Macintosh*, a drive queue element has the following structure:

```
DrvQEl = RECORD
    qLink:     QElemPtr;  {next queue entry}
    qType:     INTEGER;   {queue type}
    dQDrive:   INTEGER;   {drive number}
    dQRefNum:  INTEGER;   {driver reference number}
    dQFSID:    INTEGER;   {file-system identifier}
    dQDrvSz:   INTEGER;   {number of logical blocks on drive}
    dQDrvSz2:  INTEGER;   {additional field to handle large drive size}
END;
```

Note that dQDrvSz2 is only used if qType is 1. In this case, dQDrvSz2 contains the high-order word of the size, and dQDrvSz contains the low-order word.

Inside Macintosh also mentions four bytes of flags that preced each drive queue entry. How are these flags accessed? The flags begin 4 bytes before the address pointed to by the `DrvQElPtr`. In assembly language, accessing this isn't a problem:

```
MOVE.L  -4(A0),D0    ;A0 = DrvQElPtr; get drive queue flags
```

If you're using Pascal, it's a little more complicated. You can get to the `flags` with this routine:

```
FUNCTION DriveFlags(aDQEPtr: DrvQElPtr): LONGINT;

  VAR
    flagsPtr : ^LONGINT; {we'll point at drive queue flags with this}

  BEGIN
    {subtract 4 from the DrvQElPtr, and get the LONGINT there}
    flagsPtr := POINTER(ORD4(aDQEPtr) - 4);
    DriveFlags := flagsPtr^;
  END;
```

From MPW C, you can use:

```
long DriveFlags(aDQEPtr)
DrvQElPtr    aDQEPtr;

{ /* DriveFlags */
   return(*((long *)aDQEPtr - 1));  /* coerce flagsPtr to a (long *)
                                       so that subtracting 1 from it
                                       will back us up 4 bytes */
} /* DriveFlags */
```

Creating New Drives

To add a drive to the drive queue, assembly-language programmers can use the function defined below. It takes two parameters: the driver reference number of the driver which is to "own" this drive, and the size of the new drive in blocks. It returns the drive number created. It is vital that you **not** hard-code the drive number; if the user has installed other non-standard drives in the queue, the drive number you're expecting may already be taken. (Note that the example function below arbitrates to find an unused drive number, taking care of this problem for you. Also, note that this function doesn't mount the new volume; your code should take care of that, calling the Disk Initialization Package to reformat the volume if necessary).

```
AddMyDrive  PROC      EXPORT
;-----------------------------------------------------------------------
;FUNCTION AddMyDrive(drvSize: LONGINT; drvrRef: INTEGER): INTEGER;
;-----------------------------------------------------------------------
;Add a drive to the drive queue. Returns the new drive number, or a negative
;error code (from trying to allocate the memory for the queue element).
;-----------------------------------------------------------------------
DQESize     EQU       18           ;size of a drive queue element
;We use a constant here because the number in SysEqu.a doesn't take into
;account the flags LONGINT before the element, or the size word at the end.
;-----------------------------------------------------------------------
StackFrame  RECORD    {link},DECR
result      DS.W      1            ;function result
params      EQU       *
drvSize     DS.L      1            ;drive size parameter
drvrRef     DS.W      1            ;drive refNum parameter
paramSize   EQU       params-*
return      DS.L      1            ;return address
link        DS.L      1            ;saved value of A6 from LINK
block       DS.B      ioQElSize    ;parameter block for call to MountVol
linkSize    EQU       *
            ENDR
;-----------------------------------------------------------------------
         WITH      StackFrame      ;use the offsets declared above

         LINK      A6,#linkSize    ;create stack frame

         ;search existing drive queue for an unused number

         LEA       DrvQHdr,A0      ;get the drive queue header
         MOVEQ     #4,D0           ;start with drive number 4
```

```
CheckDrvNum
          MOVE.L    qHead(A0),A1    ;start with first drive
CheckDrv
          CMP.W     dqDrive(A1),D0 ;does this drive already have our number?
          BEQ.S     NextDrvNum      ;yep, bump the number and try again.
          CMP.L     A1,qTail(A0)    ;no, are we at the end of the queue?
          BEQ.S     GotDrvNum       ;if yes, our number's unique! Go use it.
          MOVE.L    qLink(A1),A1    ;point to next queue element
          BRA.S     CheckDrv        ;go check it.

NextDrvNum
          ;this drive number is taken, pick another

          ADDQ.W    #1,D0           ;bump to next possible drive number
          BRA.S     CheckDrvNum     ;try the new number

GotDrvNum
          ;we got a good number (in D0.W), set it aside

          MOVE.W    D0,result(A6)   ;return it to the user

          ;get room for the new DQE

          MOVEQ     #DQESize,D0     ;size of drive queue element, adjusted
          _NewPtr   sys             ;get memory for it
          BEQ.S     GotDQE          ;no error...continue
          MOVE.W    D0,result(A6)   ;couldn't get the memory! return error
          BRA.S     FinishUp        ;and exit

GotDQE
          ;fill out the DQE

          MOVE.L    #$80000,(A0)+   ;flags: non-ejectable; bump past flags

          MOVE.W    #1,qType(A0)    ;qType of 1 means we do use dQDrvSz2
          CLR.W     dQFSID(A0)      ;"local file system"
          MOVE.W    drvSize(A6),dQDrvSz2(A0)  ;high word of number of blocks
          MOVE.W    drvSize+2(A6),dQDrvSz(A0) ;low word of number of blocks

          ;call AddDrive

          MOVE.W    result(A6),D0   ;get the drive number back
          SWAP      D0              ;put it in the high word
          MOVE.W    drvrRef(A6),D0 ;move the driver refNum in the low word
          _AddDrive                 ;add this drive to the drive queue

FinishUp
          UNLK      A6              ;get rid of stack frame
          MOVE.L    (SP)+,A0        ;get return address
          ADDQ      #paramSize,SP   ;get rid of parameters
          JMP       (A0)            ;back to caller
;-----------------------------------------------------------------------
          ENDPROC
```

#37: Differentiating Between Logic Boards

See: Technical Note #129—SysEnvirons

Written by: Mark Baumwell June 19, 1986
Updated: March 1, 1988

Earlier versions of this note are obsoleted by existence of `SysEnvirons`, which is documented in Technical Note #129.

Macintosh Technical Notes

#38: The ROM Debugger

Written by: Louella Pizzuti June 20, 1986
Updated: March 1, 1988

The debugger in ROM (not present on the Macintosh 128, Macintosh 512, or Macintosh XL) recognizes the following commands:

`PC [expr]` (program counter)

Typing `PC` on a line by itself displays the program counter. Typing `PC 50000` sets the program counter to `$50000`.

`SM [address [number(s)]]` (set memory)

Typing `SM` on a line by itself displays the next 96 bytes of memory. Typing `SM 50000` will display memory starting at `$50000`. Typing `SM 50000 4849 2054 6865 7265 2120` will set memory starting at `$50000` to `$4849...` Subsequently hitting Return will increment the display a screen at a time.

`DM [address]` (display memory)

Typing `DM` on a line by itself displays the next 96 bytes of memory. Typing `DM 50000` will display memory at `$50000`. Subsequently hitting Return will increment the display a screen at a time.

`SR [expr]` (status register)

Typing `SR` on a line by itself displays the status register. Typing `SR 2004` sets the status register to `$2004`.

`TD` (total display)

Displays memory at the “magic” location `$3FFC80`, which contains the current values of the registers. The registers are displayed in the following order: `D0-D7`, `A0-A7`, `PC`, `SR`.

`G [address]` (go)

Executes instructions starting at address. If `G` is typed on a line by itself, execution begins at the address indicated by the program counter.

Note: If you want to exit to the shell, you just need to type: `SM 0 A9F4`, then `G 0`

Note: If you crash into the debugger and the system hangs, try turning off your modem.

#39: Segment Loader Patch

Written by:	Russ Daniels Bryan Stearns	August 1, 1985
Modified by:	Jim Friedlander	November 15, 1986
Updated:		March 1, 1988

This note formerly described a patch to the Segment Loader for 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#40: Finder Flags

See also: The File Manager

Written by: Jim Friedlander June 16, 1986
Modified by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

This revision corrects the meanings of bits 6 and 7, which were interchanged in the older version of this technical note. ResEdit uses these bits incorrectly in versions older than 1.2.

The Finder keeps and uses a series of file information flags for each file. These flags are located in the `fdFlags` field (a word at offset `$28` into an `HParamBlockRec`) of the `ioFlFndrInfo` record of a parameter block. They may change with newer versions of the Finder. Finders 5.4 and newer assign the following meanings to the flags:

Bit	Meaning
0	Set if file/folder is on the desktop (Finder 5.0 and later)
1	bFOwnAppl (used internally)
2	reserved (currently unused)
3	reserved (currently unused)
4	bFNever (never SwitchLaunch) (not implemented)
5	bFAlways (always SwitchLaunch)
6	Set if file is a shareable application
7	reserved (used by System)
8	Inited (seen by Finder)
9	Changed (used internally by Finder)
10	Busy (copied from File System busy bit)
11	NoCopy (not used in 5.0 and later, formerly called BOZO)
12	System (set if file is a system file)
13	HasBundle
14	Invisible
15	Locked

Macintosh
Technical Notes

Developer Technical Support

#41: Drawing Into an Off-Screen Bitmap

Revised by: Jon Zap & Forrest Tanaka June 1990
Written by: Jim Friedlander & Ginger Jernigan July 1985

This Technical Note provides an example of creating an off-screen bitmap, drawing to it, and then copying from it to the screen.

Changes since April 1990: Clarified the section on window updates with off-screen bitmaps to explicitly limit these updates to your own windows.

The following is an example of creating and drawing to an off-screen bitmap, then copying from it to an on-screen window. We supply this example in both MPW Pascal and C.

MPW Pascal

First, let's look at a general purpose function to create an off-screen bitmap. This function creates the `GrafPort` on the heap. You could also create it on the stack and pass the uninitialized structure to a function similar to this one.

```
FUNCTION CreateOffscreenBitMap(VAR newOffscreen:GrafPtr; inBounds:Rect) : BOOLEAN;

VAR
  savePort  : GrafPtr;
  newPort   : GrafPtr;

BEGIN
  GetPort(savePort);        {need this to restore thePort after OpenPort changes it}

  newPort := GrafPtr(NewPtr(sizeof(GrafPort)));    {allocate the GrafPort}
  IF MemError <> noErr THEN BEGIN
    CreateOffscreenBitMap := false;                {failed to allocate it}
    EXIT(CreateOffscreenBitMap);
  END;
  {
  the OpenPort call does the following . . .
    allocates space for visRgn (set to screenBits.bounds) and clipRgn (set wide open)
    sets portBits to screenBits
    sets portRect to screenBits.bounds
    etc. (see IM I-163,164)
    side effect: does a SetPort(offScreen)
  }
  OpenPort(newPort);
  {make bitmap exactly the size of the bounds that caller supplied}
  WITH newPort^ DO BEGIN {portRect, clipRgn, and visRgn are in newPort}
    portRect := inBounds;
    RectRgn(clipRgn, inBounds);        {avoid wide-open clipRgn, to be safe}
    RectRgn(visRgn, inBounds);         {in case inBounds is > screen bounds}
  END;
  WITH newPort^.portBits DO BEGIN       {baseAddr, rowBytes and bounds are in newPort}
    bounds := inBounds;
    {rowBytes is size of row  It must be rounded up to even number of bytes}
    rowBytes := ((inBounds.right - inBounds.left + 15) DIV 16) * 2;

    {number of bytes in BitMap is rowBytes * number of rows}
    {see note at end of Technical Note about using _NewHandle rather than _NewPtr}
    baseAddr := NewPtr(rowBytes * LONGINT(inBounds.bottom - inBounds.top));
  END;
  IF MemError <> noErr THEN BEGIN     {see if we had enough room for the bits}
    SetPort(savePort);
    ClosePort(newPort);               { dump the visRgn and clipRgn }
    DisposPtr(Ptr(newPort));          { dump the GrafPort}
    CreateOffscreenBitMap := false;
  END
  ELSE BEGIN
    { since the bits are just memory, let's erase them before we start }
    EraseRect(inBounds);             {OpenPort did a SetPort(newPort)}
    newOffscreen := newPort;
    SetPort(savePort);
    CreateOffscreenBitMap := true;
  END;
END;
```

Here is the procedure to get rid of an off-screen bitmap created by the previous function:

```
PROCEDURE DestroyOffscreenBitMap(oldOffscreen : GrafPtr);
BEGIN
  ClosePort(oldOffscreen);                        { dump the visRgn and clipRgn }
  DisposPtr(oldOffscreen^.portBits.baseAddr);     { dump the bits }
  DisposPtr(Ptr(oldOffscreen));                   { dump the port }
END;
```

Now that you know how to create and destroy an off-screen bitmap, let’s go through the motions of using one. First, let’s define a few things to make the _NewWindow call a little clearer.

```
CONST
  kIsVisible    = true;
  kNoGoAway     = false;
  kMakeFrontWindow = -1;
  myString      = 'The EYE';  {string to display}
```

Here’s the body of the test code:

```
VAR
  offscreen  : GrafPtr;     {our off-screen bitmap}
  ovalRect   : Rect;        {used for example drawing}
  myWBounds  : Rect;        {for creating window}
  OSRect     : Rect;        {portRect and bounds for off-screen bitmap}
  myWindow   : WindowPtr;

BEGIN
  InitToolbox;                        {exercise left to the reader}

  myWBounds := screenBits.bounds;     { size of main screen }
  InsetRect(myWBounds, 50,50);        { make it fit better }
  myWindow := NewWindow(NIL, myWBounds, 'Test Window', kIsVisible,
                        noGrowDocProc, WindowPtr(kMakeFrontWindow), kNoGoAway, 0);

  IF NOT CreateOffscreenBitMap(offscreen,myWindow^.portRect) THEN BEGIN
    SysBeep(1);
    ExitToShell;
  END;
  { Example drawing to our off-screen bitmap }
  SetPort(offscreen);
  OSRect := offscreen^.portRect;    { offscreen bitmap's local coordinate rect }
  ovalRect := OSRect;
  FillOval(ovalRect, black);
  InsetRect(ovalRect, 1, 20);
  FillOval(ovalRect, white);
  InsetRect(ovalRect, 40, 1);
  FillOval(ovalRect, black);
  WITH ovalRect DO
    MoveTo((left+right-StringWidth(myString)) DIV 2, (top+bottom-12) DIV 2);
  TextMode(srcXor);
  DrawString(myString);

  { copy from the off-screen bitmap to the on-screen window.  Note that in this
  case the source and destination rects are the same size and both cover the
  entire area.  These rects are allowed to be portions of the source and/or
  destination and do not have to be the same size.  If they are not the same size
  then _CopyBits scales the image accordingly
  }
  SetPort(myWindow);
  CopyBits(offscreen^.portBits, myWindow^.portBits,
           offscreen^.portRect, myWindow^.portRect, srcCopy, NIL);

  DestroyOffscreenBitMap(offscreen);    {remove the evidence}

  WHILE NOT Button DO;                  {give user a chance to see the results}
END.
```

MPW C

First, let's look at a general purpose function to create an off-screen bitmap. This function creates the `GrafPort` on the heap. You could also create it on the stack and pass the uninitialized structure to a function similar to this one.

```
Boolean CreateOffscreenBitMap(GrafPtr *newOffscreen, Rect *inBounds)
{
  GrafPtr savePort;
  GrafPtr newPort;

  GetPort(&savePort);    /* need this to restore thePort after OpenPort */

  newPort = (GrafPtr) NewPtr(sizeof(GrafPort));    /* allocate the grafPort */
  if (MemError() != noErr)
    return false;                 /* failed to allocate the off-screen port */
  /*
  the call to OpenPort does the following . . .
    allocates space for visRgn (set to screenBits.bounds) and clipRgn (set wide open)
    sets portBits to screenBits
    sets portRect to screenBits.bounds
    etc. (see IM I-163,164)
    side effect: does a SetPort(&offScreen)
  */
  OpenPort(newPort);
  /* make bitmap the size of the bounds that caller supplied */
  newPort->portRect = *inBounds;
  newPort->portBits.bounds = *inBounds;
  RectRgn(newPort->clipRgn, inBounds);    /* avoid wide-open clipRgn, to be safe  */
  RectRgn(newPort->visRgn, inBounds);     /* in case newBounds is > screen bounds */

  /* rowBytes is size of row, it must be rounded up to an even number of bytes */
  newPort->portBits.rowBytes = ((inBounds->right - inBounds->left + 15) >> 4) << 1;
  /* number of bytes in BitMap is rowBytes * number of rows */
  /* see notes at end of Technical Note about using _NewHandle rather than _NewPtr */
  newPort->portBits.baseAddr =
           NewPtr(newPort->portBits.rowBytes * (long) (inBounds->bottom - inBounds->top));
  if (MemError()!=noErr) {    /* check to see if we had enough room for the bits */
    SetPort(savePort);
    ClosePort(newPort);       /* dump the visRgn and clipRgn */
    DisposPtr((Ptr)newPort);  /* dump the GrafPort */
    return false;             /* tell caller we failed */
    }
  /* since the bits are just memory, let's clear them before we start */
  EraseRect(inBounds);      /* OpenPort did a SetPort(newPort) so we are ok */
  *newOffscreen = newPort;
  SetPort(savePort);
  return true;                /* tell caller we succeeded! */
}
```

Here is the function to get rid of an off-screen bitmap created by the previous function:

```
void DestroyOffscreenBitMap(GrafPtr oldOffscreen)
{
  ClosePort(oldOffscreen);                    /* dump the visRgn and clipRgn */
  DisposPtr(oldOffscreen->portBits.baseAddr); /* dump the bits */
  DisposPtr((Ptr)oldOffscreen);               /* dump the port */
}
```

Now that you know how to create and destroy an off-screen bitmap, let's go through the motions of using one. First, let's define a few things to make the `_NewWindow` call a little clearer.

```
#define kIsVisible true
#define kNoGoAway false
#define kNoWindowStorage 0L
#define kFrontWindow ((WindowPtr) -1L)
```

Here's the body of the test code:

```
main()
{
  char* myString = "\pThe EYE";  /* string to display */

  GrafPtr   offscreen;           /* our off-screen bitmap */
  Rect      ovalRect;            /* used for example drawing */
  Rect      myWBounds;           /* for creating window */
  Rect      OSRect;              /* portRect and bounds for off-screen bitmap*/
  WindowPtr myWindow;

  InitToolbox();                 /* exercise for the reader */
  myWBounds = qd.screenBits.bounds;  /* size of main screen */
  InsetRect(&myWBounds, 50,50);  /* make it fit better */
  myWindow = NewWindow(kNoWindowStorage, &myWBounds, "\pTest Window", kIsVisible,
                      noGrowDocProc, kFrontWindow, kNoGoAway, 0);
  if (!CreateOffscreenBitMap(&offscreen, &myWindow->portRect)) {
    SysBeep(1);
    ExitToShell();
    }
  /* Example drawing to our off-screen bitmap*/
  SetPort(offscreen);
  OSRect = offscreen->portRect;  /* offscreen bitmap's local coordinate rect */
  ovalRect = OSRect;
  FillOval(&ovalRect, qd.black);
  InsetRect(&ovalRect, 1, 20);
  FillOval(&ovalRect, qd.white);
  InsetRect(&ovalRect, 40, 1);
  FillOval(&ovalRect, qd.black);
  MoveTo((ovalRect.left + ovalRect.right - StringWidth(myString)) >> 1,
         (ovalRect.top + ovalRect.bottom - 12) >> 1);
  TextMode(srcXor);
  DrawString(myString);

  /* copy from the off-screen bitmap to the on-screen window.  Note that in this
  case the source and destination rects are the same size and both cover the
  entire area.  These rects are allowed to be portions of the source and/or
  destination and do not have to be the same size.  If they are not the same size
  then _CopyBits scales the image accordingly.
  */
  SetPort(myWindow);
  CopyBits(&offscreen->portBits, &(*myWindow).portBits,
           &offscreen->portRect, &(*myWindow).portRect, srcCopy, 0L);

  DestroyOffscreenBitMap(offscreen);    /* dump the off-screen bitmap */
  while (!Button());     /* give user a chance to see our work of art */
}
```

Comments

In the example code, the bits of the `BitMap` structure, which are pointed to by the `baseAddr` field, are allocated by a `_NewPtr` call. If your off-screen bitmap is close to the size of the screen, then the amount of memory needed for the bits can be quite large (on the order of 20K for the Macintosh SE or 128K for a large screen). This is quite a lot of memory to lock down in your heap and it can easily lead to fragmentation if you intend to keep the off-screen bitmap around for any length of time. One alternative that lessens this problem is to get the bits via `_NewHandle` so the Memory Manager can move them when necessary. To implement this approach, you need to keep the handle separate from the `GrafPort` (for example, in a structure that combines a `GrafPort` and a `Handle`). When you want to use the off-screen bitmap you would then lock the handle and put the dereferenced handle into the `baseAddr` field. When you are not using the off-screen bitmap you can then unlock it.

This example does not demonstrate one of the more typical uses of off-screen bitmaps, which is to preserve the contents of windows so that after a temporary window or dialog box obscures part of your windows and is then dismissed, you can quickly handle the resulting update events without recreating all of the intermediate drawing commands.

Make sure you only restore the pixels within the content regions of your own windows in case the temporary window partly obscures windows belonging to other applications or to the desktop. Another application could change the contents of its windows while they are behind your temporary window, so you cannot simply restore all the pixels that were behind the temporary window because that would restore the old contents of the other application's windows. Instead, you could keep keep an off-screen bitmap for each of your windows and then restore them by copying each bit map into the corresponding window's ports when they get their update events.

An alternate method is to make a single off-screen bitmap that is as large as the temporary window and a region that is the union of the content regions of your windows. Before you display the temporary window, copy the screen into the off-screen bit map using the region as a mask. After the temporary window is dismissed, restore the obscured area by copying from the off-screen bit map into a copy of the Window Manager port, and use the region as a mask. If the region has the proper shape and location, it prevents _CopyBits from drawing outside of the content regions of your windows. See Technical Note #194, WMgrPortability for details about drawing across windows.

In some cases it can be just as fast and convenient to simply define a picture (PICT) and then draw it into your window when necessary. There are cases, however, such as text rotation, where it is advantageous to do the drawing off the screen, manipulate the bit image, and then copy the result to the visible window (thus avoiding the dangers inherent in writing directly to the screen). In addition, this technique reduces flicker, because all of the drawing done off the screen appears on the screen at once.

It is also important to realize that, if you plan on using the pre-Color QuickDraw eight-color model, an off-screen bitmap loses any color information and you do not see your colors on a system that is capable of displaying them. In this case you should either use a PICT to save the drawing information or check for the presence of Color QuickDraw and, when it is present, use a PixMap instead of a BitMap and the color toolbox calls (*Inside Macintosh*, Volume V) instead of the standard QuickDraw calls (*Inside Macintosh*, Volume I).

You may also want to refer to the OffScreen library (DTS Sample Code #15) which provides both high- and low-level off-screen bitmap support for the 128K and later ROMs. The OffSample application (DTS Sample Code #16) demonstrates the use of this library.

Further Reference:

- *Inside Macintosh*, Volumes I & IV, QuickDraw
- *Inside Macintosh*, Volume V, Color QuickDraw
- Technical Note #120, Drawing Into an Off-Screen Pixel Map
- Technical Note #194, WMgrPortability
- DTS Macintosh Sample Code #15, OffScreen & #16, OffSample

Macintosh Technical Notes

#42: Pascal Routines Passed by Pointer

See also: Macintosh Memory Management: An Introduction

Written by: Scott Knaster July 22, 1985
Updated: March 1, 1988

Routines passed by pointer are used in many places in conjunction with Macintosh system routines. For example, filter procedures for modal dialogs are passed by pointer, as are controls' action procedures (when calling `TrackControl`), and I/O completion routines.

If you're using MPW Pascal, the syntax is usually

```
partCode := TrackControl(theControl, startPt, @MyProc)
```

where `MyProc` is the procedure passed by pointer (using the @ symbol).

Because of the way that MPW Pascal (and some other compilers) construct stack frames, any procedure or function passed by pointer **must not** have its declaration nested within another procedure or function. If its declaration is nested, the program will crash, probably with an illegal instruction error. The following example demonstrates this:

```
PROGRAM CertainDeath;

   PROCEDURE CallDialog;

      VAR
         x : INTEGER;

      FUNCTION MyFilter(theDialog: DialogPtr; VAR theEvent: EventRecord;
                        VAR itemHit: INTEGER): Boolean;
      {note that MyFilter's declaration is nested within CallDialog}

      BEGIN {MyFilter}
        {body of MyFilter}
      END; {MyFilter}

   BEGIN {CallDialog}
      ModalDialog(@MyFilter,itemHit) {<------------ will crash here}
   END; {CallDialog}

BEGIN {main program}
      CallDialog;
END.
```

#43: Calling LoadSeg

See also: The Segment Loader

Written by: Gene Pope October 15, 1985
Updated: March 1, 1988

Earlier versions of this note described a way to call the `LoadSeg` trap, which is used internally by the Segment Loader. We no longer recommend calling `LoadSeg` directly.

#44: HFS Compatibility

See also: The File Manager

Written by: Jim Friedlander October 9, 1985
Modified by: Scott Knaster December 5, 1985
Jim Friedlander
Updated: March 1, 1988

This technical note tells you how to make sure that your applications run under the Hierarchical File System (HFS).

The Hierarchical File System (HFS) provides fast, efficient management of larger volumes than the original Macintosh File System (MFS). Since HFS is hierarchical, HFS folders have a meaning different from MFS folders. In MFS, a folder has only graphical significance—it is only used by the Finder as a means of visually grouping files. The MFS directory structure is actually flat (all files are at the 'root' level). Under HFS, a folder is a directory that can contain files and other directories.

A folder is accessed by use of a `WDRefNum` (Working Directory reference number). Calls that return a `vRefNum` when running under MFS may return a `WDRefNum` when running under HFS. You may use a `WDRefNum` wherever a `vRefNum` may be used.

In order to provide for compatibility with software written for MFS, the HFS calls that open files search both the default directory and the directory that contains the System and the Finder (HFS marks this last directory so it always knows where to look for the System and the Finder).

Your goal should be to write programs that are file system independent. Your programs should not only be able to access files on other volumes, but also files that are in other directories. Accomplishing this is not difficult—most applications that were written for MFS work correctly under HFS. If you find that your current applications do not run correctly under HFS, you should check to see if you are doing any of the following five things:

Are you using Standard File?

This is very important to ensure that your application will run correctly under HFS. HFS uses an extended Standard File, which allows the user to select from files in different directories. This increased functionality was implemented without changing Standard File's external specification—the only difference is that `SFReply.vRefNum` can now be a `WDRefNum`. Please note that using Standard File's dialog hook and filter procs or adding controls of your own will not cause compatibility problems with HFS.

Existing applications that use Standard File properly run without modification under HFS. Applications that take the SFReply.vRefNum and convert that to a volume name, then append it to SFReply.fName (as in #2 below) do not function correctly under HFS—the user can only open files in the root directory. If you call Open with SFReply.vRefNum and SFReply.fName, everything will work correctly. Remember, SFReply.vRefNum may be a WDRefNum . Using Standard File will virtually guarantee that your application will be compatible with MFS, HFS, and future file systems.

Are you concatenating volume names to file names, i.e. using file names of the form VOLUME:fileName?

Applications that do this do not work correctly under HFS (in fact, they do not even run correctly under MFS). Instead of this, use a vRefNum to access a volume or a directory. Fully qualified pathnames (such as volume:folder1:folder2:filename) work correctly, but we don't recommend that you use them. Please don't ever make a user type in a full pathname!

Are you searching directories for files using a loop such as
FOR index:= 1 to ioVNmFls DO ...
where ioVNmFls was returned from a PBGetVinfo call?

This technique should not be used. Instead, use repeated calls to PBGetFInfo using ioFDirIndex until fnfErr is returned. Indexed calls to PBGetFInfo will return files in the directory specified by the vRefNum that you put in the parameter block.

Are you assuming that a vRefNum will actually refer to a volume?

A vRefNum can now be a WDRefNum. A WDRefNum indicates which working directory (folder) a file is in, not which volume the file is on. Don't think of a vRefNum as a way to access a volume, but rather as a means of telling the file system where to find a file.

Are you walking through the VCB queue?

You should let us do the walking for you. Using indexed calls to PBGetVInfo will allow you to get information about any mounted volume. You shouldn't walk through the VCB queue because it changed for HFS and might change in the future. The routines that we supply will correctly access information in the VCB queue.

Are you using the file system's "IMMED" bit? (assembly language only)

Inside Macintosh describes bit 9 of the trap word as the immediate bit. In fact, setting this bit under MFS did not work as documented; it did not have the desired effect of bypassing the file I/O queue. Under HFS, this bit is used; it distinguishes HFS varieties of calls from MFS varieties. For example, the PBOpen call has this bit clear; PBHOpen has it set. Therefore, you must be sure that your file system calls do not use this bit as the immediate bit.

#45: *Inside Macintosh* Quick Reference

Compiled by: Jim Friedlander August 2, 1985
Updated: March 1, 1988

This note formerly listed the traps from *Inside Macintosh Volumes I-III*. Better references are now available elsewhere.

#46: Separate Resource Files

See also: The Resource Manager

Written by: Bryan Stearns October 16, 1985
Updated: March 1, 1988

During application development, you use a resource compiler (RMaker or Rez) to convert a resource definition file into an executable application. You rarely change anything but your CODE resources during development, and the resource compiler spends a lot of time compiling other resources which have not changed since they were originally created.

To save time, some developers have adopted the technique of storing all of these "static" resources in a separate resource file. This file should be placed on the same volume as your application; when your application starts up, use `OpenResFile` to open the separate file. This will cause the resource map for the separate file to be searched before the normal application resource file's map (which now contains mostly CODE resources, along with any brand-new resources still being tested).

This will have little or no effect on the rest of your program. Any time that a resource is needed, both resource files will be searched automatically so you **don't** need to change each `GetResource` call. (Actually, having the extra resource file open has a minor impact on memory management, and uses one more file-control block; unless you're using a lot of open files at once, or are running at the limits of available memory without segmentation, this shouldn't affect you.)

Once your application is close to being finished, you can use ResEdit to move all the resources back into the main application file, and remove the extra `OpenResFile` at the beginning of your application. You should do this for any major release (alpha, beta, and any other 'heavy-testing' releases). Other minor modifications (such as fine-tuning dialog box item positions) may also be done with ResEdit at this time.

The only catch is that you must be careful if your application adds resources to its own resource file. Most applications do not do this (it's not really a great idea, and causes problems with file servers).

#47: Customizing Standard File

See also: The Standard File Package

Written by: Jim Friedlander October 11, 1985
Updated: March 1, 1988

This note contains an example program that demonstrates how SFPGetFile can be customized using the dialog hook and file filter functions.

SFPGetFile's dialog hook function and file filter function enable you to customize SFPGetFile's behavior to fit the needs of your application. This technical note consists primarily of a short example program that

1) changes the title of the Open button to 'MyOpen',
2) adds two radio buttons so that the user can choose to display either text files or text files and applications.
3) adds a quit button to the SFPGetFile dialog,

All this is done in a way so as to provide compatibility with the Macintosh File System (MFS), the Hierarchical File System (HFS) and (hopefully) future systems. If you have any questions as you read, the complete source of the demo program and the resource compiler input file is provided at the end of this technical note.

Basically, we need to do three things: add our extra controls to the resource compiler input file, write a dialog hook function, and write a file filter function.

Modifying the Resource Compiler Input File

First we need to define a dialog in our resource file. It will be DLOG #128:

```
CONST myDLOGID = 128;
```

and it's Rez description is:

```
resource 'DLOG' (128, purgeable) {
    {0, 0, 200, 349},
    dBoxProc, invisible, noGoAway,
    0x0,
    128,
    "MyGF"
};
```

The above coordinates (0 0 200 349) are from the standard Standard File dialog. If you need to change the size of the dialog to accommodate new controls, change these coordinates. Next we need to add a DITL in our resource file that is the same as the standard HFS DITL #–4000 except for one item. We need to change the left coordinate of UserItem #4, or part of the dialog will be hidden if we're running under MFS:

```
/* [4] */
/* left coordinate changed from 232 to 252 so program will
   work on MFS */
{39, 252, 59, 347},
UserItem {
    disabled
};
```

None of the other items of the DITL should be changed, so that your program will remain as compatible as possible with different versions of Standard File. Finally, we need to add three items to this DITL, two radio buttons and one button (to serve as a quit button)

```
/* [11] textButton */
{1, 14, 20, 142},
RadioButton {
    enabled,
    "Text files only"
};
/* [12] textAppButton */
{19, 14, 38, 176},
RadioButton {
    enabled,
    "Text and applications"
};
/* [13] quitButton */
{6, 256, 24, 336},
Button {
    enabled,
    "Quit"
}
```

Because we've added three items, we need also need to change the item count for the DITL from 10 to 13. We also include the following in our resource file:

```
resource 'STR#' (256) {
    {/* array StringArray: 1 elements */
        /* [1] */
        "MyOpen"
    }
};
```

That's all there is to modify in the resource file.

The Dialog Hook

We will be calling SFPGetFile as follows:

```
SFPGetFile (wher, '', @SFFileFilter, NumFileTypes,
            MyFileTypes, @MySFHook, reply, myDLOGID,nil);
```

Notice that we're passing @MySFHook to Standard File. This is the address of our dialog hook routine. Our dialog hook is declared as:

```
FUNCTION MySFHook(MySFitem: INTEGER; theDialog: DialogPtr):INTEGER;
```

A dialog hook routine allows us to see every item hit before standard file acts on it. This allows us to handle controls that aren't in the standard SFPGetFile's DITL or to handle standard controls in non-standard ways. The dialog hook in this example consists of a case statement with MySFitem as the case selector. Before SFPGetFile displays its dialog, it calls our dialog hook, passing it a –1 as MySFitem. This gives us a chance to initialize our controls. Here we will set the textAppButton to off and the textButton to on:

```
GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
SetCtlValue(controlHandle(itemToChange),btnOff);
GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
SetCtlValue(controlHandle(itemToChange),btnOn);
```

and we can also change the title of an existing control. Here's how we might change the title of the Open button using a string that we get from a resource file:

```
GetIndString(buttonTitle,256,1);
If buttonTitle <> '' then Begin           { if we really got the resource}
    GetDItem(theDialog,getOpen,itemType,itemToChange,itemBox);
    SetCtitle(controlHandle(itemToChange),buttonTitle);
End; {if}          {if we didn't get the resource, don't change the title }
```

Upon completion of our routine that handles the –1, we return a –1 to standard file:

```
MySFHook:= MySFItem;                {pass back the same item we were sent}
```

We now have a SFPGetFile dialog displayed that has a quit button and two radio buttons (the textOnly button is on, the TextApp button is off). In addition, the standard Open button has been renamed to MyOpen (or whatever STR is the first string in STR# 256). This was all done before SFPGetFile displayed the dialog. Once our hook is exited, SFPGetFile displays the dialog and calls ModalDialog.

When the user clicks on an item in the dialog, our hook is called again. We can then take appropriate actions, such as highlighting the textButton and un-highlighting the textAppButton if the user clicks on the textButton. At this time, we can also update a global variable (textOnly) that we will use in our file filter function to tell us which files to display. Notice that we can redisplay the file list by returning a 101 as the result of MySFHook. (Standard File for Systems newer than 4.3 will also read the low memory globals, CurDirStore and SFSaveDisk, and switch directories when necessary if a 101 is returned as the result. Thus, you can point Standard File to a new directory, or a new disk.) For example, when the textButton is hit we turn the textAppButton off, turn the textButton on, update the global variable textOnly, and tell SFPGetFile to redisplay the list of files the user can choose from:

```
if not textOnly then Begin     {if textOnly was turned off, turn it on now}
    GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
    SetCtlValue(controlHandle(itemToChange),btnOff);
    GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
    SetCtlValue(controlHandle(itemToChange),btnOn);
    textOnly:=TRUE;      {toggle our global variable for use in the filter}
    MySFHook:= reDrawList;{101}       {we must tell SF to redraw the list}
End;  {if not textOnly}
```

If our quit button is hit, we can pass SFPGetFile back the cancel button:

```
MySFHook:= getCancel;
```

If one of SFPGetFile's standard items is hit, it is very important to pass that item back to SFPGetFile:

```
MySFHook:= MySFItem; {pass back the same item we were sent}
```

The File Filter

Remember, we called SFPGetFile as follows:

```
SFPGetFile (wher, '', @SFFileFilter, NumFileTypes,
            MyFileTypes, @MySFHook, reply,myDLOGID,nil);
```

Notice that we're passing @SFFileFilter to SFPGetFile. This is the address of our file filter routine. A file filter is declared as:

```
FUNCTION SFFileFilter (p: ParmBlkPtr): BOOLEAN;
```

A file filter routine allows us to control which files SFPGetFile will display for the user. Our file filter is called for every file (of the type(s) specified in the typelist) on an MFS disk, or for every file (of the type(s) specified in the typelist) in the current directory on an HFS disk. In addition, SFPGetFile displays HFS folders for us automatically. Our file filter selects which files should appear in the dialog by returning FALSE for every file that should be shown and TRUE for every file that shouldn't.

For example, using our global variable textOnly (which we set in our dialog hook, remember?):

```
FUNCTION SFFileFilter(p:parmBlkPtr):boolean;

Begin {SFFileFilter}
  SFFileFilter:= TRUE;                          {Don't show it -- default}

  if textOnly then
      if p^.ioFlFndrInfo.fdType = 'TEXT' then
          SFFileFilter:= FALSE                      {Show TEXT files only}
      else Begin
      End  {dummy else}
  else
      if (p^.ioFlFndrInfo.fdType = 'TEXT') or
               (p^.ioFlFndrInfo.fdType = 'APPL') then
                    SFFileFilter:= FALSE;       { show TEXT or APPL files}
End;  {SFFileFilter}
```

SFPGetFile calls the file filter after it has called our dialog hook. Please remember that the filter is passed every file of the types specified in the typelist (MyFileTypes). If you want your application to be able to choose from all files, pass SFPGetFile a –1 as numTypes. For information about parameters to SFPGetFile that haven't been discussed in this technical note, see the Standard File Package chapter of *Inside Macintosh*.

That's all there is to it!! Now that you know how to modify SFPGetFile to suit your needs, please don't rush off and load up the dialog window with all kinds of controls and text. Please make sure that you adhere to Macintosh interface standards. Similar techniques can be used with SFGetFile, SFPutFile and SFPPutFile.

The complete source of the demo program and of the resource compiler input file follows:

MPW Pascal Source

```
{$R-}

{Jim Friedlander    Macintosh Technical Support      9/30/85}

program SFGetDemo;

USES
     MemTypes,
     QuickDraw,
     OSIntf,
     ToolIntf,
     PackIntf;
{$D+}

CONST
  myDLOGID = 128;               {ID of our dialog for use with SFPGetFile}

VAR
  wher: Point;                  { where to display dialog }
  reply: SFReply;               { reply record }
  textOnly: BOOLEAN;            { tells us which files are currently being displayed}
  myFileTypes: SFTypeList;      { we won't actually use this }
  NumFileTypes: integer;


{-----------------------------------------------------------------------------------}
FUNCTION MySFHook(MySFitem:integer; theDialog:DialogPtr): integer;

CONST
  textButton                    = 11;     {DITL item number of textButton}
  textAppButton                 = 12;     {DITL item number of textAppButton}
  quitButton                    = 13;     {DITL item number of quitButton}


  stayInSF                      =  0;     {if we want to stay in SF after getting an Open hit,
                                            we can pass back a 0 from our hook  (not used in
                                            this example) }
  firstTime                     = -1;     {the first time our hook is called, it is passed a
                                            -1}

{The following line is the key to the whole routine -- the magic 101!!}
  reDrawList                    = 101;    {returning 101 as item number will cause the
                                            file list to be recalculated}
  btnOn                         = 1;      {control value for on}
  btnOff                        = 0;      {control value for off}

VAR
  itemToChange: Handle;                   {needed for GetDItem and SetCtlValue}
  itemBox:Rect;                           {needed for GetDItem}
  itemType:integer;                       {needed for GetDItem}
  buttonTitle: Str255;                    {needed for GetIndString}

Begin {MySFHook}
  case MySFItem of

    firstTime: Begin                      { before the dialog is drawn, our hook gets called
                                            with a -1 (firstTime) as the item so we can change
                                            things like button titles, etc. }
```

```
{Here we will set the textAppButton to OFF, the textButton to ON}
        GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
        SetCtlValue(controlHandle(itemToChange),btnOff);
        GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
        SetCtlValue(controlHandle(itemToChange),btnOn);

        GetIndString(buttonTitle,256,1);
                                    {get the button title from a resource file}
        If buttonTitle <> '' then Begin    { if we really got the resource}
            GetDItem(theDialog,getOpen,itemType,itemToChange,itemBox); {get a handle to the
                                                                        open button}
            SetCtitle(controlHandle(itemToChange),buttonTitle);
        End; {if}                   {if we can't get the resource, we just won't change
                                      the open button's title}
        MySFHook:= MySFItem;        {pass back the same item we were sent}
    End;  {firstTime}


{Here we will turn the textAppButton OFF, the textButton ON and redraw the list}
    textButton: Begin
        if not textOnly then Begin
            GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),btnOff);
            GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),btnOn);
            textOnly:=TRUE;
            MySFHook:= reDrawList;         {we must tell SF to redraw the list}
        End;  {if not textOnly}
    End;  {textOnlyButton}

{Here we will turn the textButton OFF, the textAppButton ON and redraw the list}
    textAppButton: Begin
        if textOnly then Begin
            GetDItem(theDialog,TextButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),BtnOff);
            GetDItem(theDialog,TextAppButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),BtnOn);
            TextOnly:=FALSE;
            MySFHook:= reDrawList;         {we must tell SF to redraw the list}
          End;  {if not textOnly}
        End;  {textAppButton}

    quitButton:  MySFHook:= getCancel;     {Pass SF back a 'cancel button'}

{!!!!very important !!!! We must pass SF's 'standard' item hits back to SF}
    otherwise  Begin
                 MySFHook:= MySFItem;      { the item hit was one of SF's standard items... }
    End;  {otherwise}                      { so just pass it back}
  End;  {case}
End;  {MySFHook}

{----------------------------------------------------------------------------}
```

```
FUNCTION SFFileFilter(p:parmBlkPtr):boolean; {general strategy -- check value of global var
                                              textOnly to see which files to display}

Begin {SFFileFilter}
  SFFileFilter:= TRUE;                       {Don't show it -- default}

  if textOnly then
      if p^.ioFlFndrInfo.fdType = 'TEXT' then
          SFFileFilter:= FALSE               {Show it}
      else Begin
      End  {dummy else}
  else
      if (p^.ioFlFndrInfo.fdType = 'TEXT') or (p^.ioFlFndrInfo.fdType = 'APPL') then
          SFFileFilter:= FALSE;              {Show it}
End;  {SFFileFilter}


{-----------------------------------------------------------------------------------}

Begin {main program}
   InitGraf (@thePort);
   InitFonts;
   InitWindows;
   TEInit;
   InitDialogs (nil);

   wher.h:=80;
   wher.v:=90;
   NumFileTypes:= -1;                        {Display all files}

{ we don't need to initialize MyFileTypes, because we want to get a chance to filter every file
  on the disk in SFFileFilter - we will decide what to show and what not to. If you want to
  filter just certain types of files by name, you would set up MyFileTypes and NumFileTypes
  accordingly}

   repeat
       textOnly:= TRUE;{each time SFPGetFile is called, initial display will be text-only
                               files}
       SFPGetFile (wher, '', @SFFileFilter, NumFileTypes, MyFileTypes, @MySFHook,
                               reply,myDLOGID,nil);
   until reply.good = FALSE;
          {until we get a cancel button hit ( or a Quit button -- thanks to our dialog hook ) }
End.
```

MPW C Source

```
#include <Types.h>
#include <Quickdraw.h>
#include <Resources.h>
#include <Fonts.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Events.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Files.h>
#include <Controls.h>
#include <ToolUtils.h>
```

```
     /*DITL item number of textButton*/
#define     textButton          11

     /*DITL item number of textAppButton*/
#define     textAppButton  12

     /*DITL item number of quitButton*/
#define     quitButton          13

     /*if we want to stay in SF after getting an Open hit, we can pass back a 0
    from our hook  (not used in this example) */
#define     stayInSF            0

     /*the first time our hook is called, it is passed a -1*/
#define     firstTime     -1

     /*The following line is the key to the whole routine -- the magic 101!!*/
     /*returning 101 as item number will cause the file list to be recalculated*/
#define     reDrawList     101

     /*control value for on*/
#define     btnOn          1

     /*control value for off*/
#define     btnOff         0

     /*resource ID of our DLOG for SFPGetFile*/
#define myDLOGID    128

Boolean             textOnly;           /* tells us which files are currently being
displayed*/

main()
{    /*main program*/

          pascal short MySFHook();
          pascal Boolean flFilter();

          Point         wher;                 /* where to display dialog */
          SFReply       reply;
     /* reply record */
          SFTypeList    myFileTypes;
     /* we won't actually use this */
          short int     NumFileTypes = -1;

   InitGraf(&qd.thePort);
   InitFonts();
   FlushEvents(everyEvent, 0);
   InitWindows();
   TEInit();
   InitDialogs(nil);
   InitCursor();


  wher.h=80;
  wher.v=90;
```

```
/* we don't need to initialize MyFileTypes, because we want to get a chance to filter every
file on  the disk in flFilter - we will decide what to show and what not to. if you want to
filter just certain types of files by name, you would set up MyFileTypes and NumFileTypes
accordingly*/

   do
   {    textOnly= true;     /*each time SFPGetFile is called, initial display will be
     text-only files*/
       SFPGetFile(&wher, "",flFilter, NumFileTypes,myFileTypes,MySFHook, &reply,myDLOGID,nil);
   }while (reply.good);    /*until we get a cancel button hit ( or a Quit button in this case )
*/
} /* main */


pascal short MySFHook(MySFItem,theDialog)
short MySFItem;
DialogPtr theDialog;

{

Handle  itemToChange;              /*needed for GetDItem and SetCtlValue*/
Rect  itemBox;                     /*needed for GetDItem*/
short itemType;                   /*needed for GetDItem*/
char  buttonTitle[256];           /*needed for GetIndString*/

  switch (MySFItem)
  {
     case firstTime:
             /* before the dialog is drawn, our hook gets called with a -1 (firstTime)...*/
             /* as the item so we can change things like button titles, etc. */
             /*Here we will set the textAppButton to OFF, the textButton to ON*/
             GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
             SetCtlValue(itemToChange,btnOff);
             GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
             SetCtlValue(itemToChange,btnOn);

             GetIndString((char *)buttonTitle,256,1);
                     /*get the button title from a resource file*/
             if (buttonTitle[0] != 0)     /* check the length of the p-string to
                                           see if we really got the resource*/
             {
                     GetDItem(theDialog,getOpen,&itemType,&itemToChange,&itemBox); /*get a
                                          handle to the open button*/
                     SetCTitle(itemToChange,buttonTitle);
             }  /*if we can't get the resource, we just won't change the open button's title*/
             return MySFItem;      /*pass back the same item we were sent*/
             break;

/*Here we will turn the textAppButton OFF, the textButton ON and redraw the list*/
    case textButton:
         if (!textOnly)
             {
                     GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
                     SetCtlValue(itemToChange,btnOff);
                     GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
                     SetCtlValue(itemToChange,btnOn);
                     textOnly=true;
                     return(reDrawList);
                    /*we must tell SF to redraw the list*/
             }  /*if !textOnly*/
             return MySFItem;
             break;
```

```
 /*Here we will turn the textButton OFF, the textAppButton ON and redraw the list*/
     case textAppButton:
         if (textOnly)
               {
                     GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
                     SetCtlValue(itemToChange,btnOff);
                     GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
                     SetCtlValue(itemToChange,btnOn);
                     textOnly=false;
                     return(reDrawList);
                    /*we must tell SF to redraw the list*/
              }  /*if not textOnly*/
              return MySFItem;       /*pass back the same item we were sent*/
              break;

     case quitButton:
              return(getCancel);
            /*Pass SF back a 'cancel button'*/

 /*!!!!!!very important !!!!!!!! We must pass SF's 'standard' item hits back to SF*/
     default:
              return(MySFItem);     /* the item hit was one of SF's standard items... */
   }  /*switch*/
      return(MySFItem);     /* return what we got */
 }  /*MySFHook*/


 pascal Boolean flFilter(pb)
 FileParam    *pb;

 {

 /* is this gross or what??? */
 return((textOnly) ? ((pb->ioFlFndrInfo.fdType) != 'TEXT') :
       ((pb->ioFlFndrInfo.fdType) != 'TEXT') &&
       ((pb->ioFlFndrInfo.fdType) != 'APPL'));
 }  /*flFilter*/
```

Rez Input File

```
#include "types.r"

resource 'STR#' (256) {
  {
    "MyOpen"
  }
};

resource 'DLOG' (128, purgeable) {
  {0, 0, 200, 349},
  dBoxProc,
  invisible,
  noGoAway,
  0x0,
  128,
  "MyGF"
};
```

```
resource 'DITL' (128, purgeable) {
  {
    /* [1] */
    {138, 256, 156, 336},
    Button { enabled, "Open" };
    /* [2] */
    {1152, 59, 1232, 77},
    Button { enabled, "Hidden" };
    /* [3] */
    {163, 256, 181, 336},
    Button { enabled, "Cancel" };
    /* [4] */
    {39, 252, 59, 347},
    UserItem { disabled };
    /* [5] */
    {68, 256, 86, 336},
    Button { enabled, "Eject" };
    /* [6] */
    {93, 256, 111, 336},
    Button { enabled, "Drive" };
    /* [7] */
    {39, 12, 185, 230},
    UserItem { enabled };
    /* [8] */
    {39, 229, 185, 245},
    UserItem { enabled };
    /* [9] */
    {124, 252, 125, 340},
    UserItem { disabled };
    /* [10] */
    {1044, 20, 1145, 116},
    StaticText { disabled, "" };
    /* [11] */
    {1, 14, 20, 142},
    RadioButton { enabled, "Text files only" };
    /* [12] */
    {19, 14, 38, 176},
    RadioButton { enabled, "Text and applications" };
    /* [13] */
    {6, 256, 24, 336},
    Button { enabled, "Quit" }
  }
};
```

Macintosh Technical Notes

#48: Bundles

See also: The Finder Interface

Written by: Ginger Jernigan November 1, 1985
Updated: March 1, 1988

This note describes what a bundle is and how to create one.

A bundle is a collection of resources. Bundles can be used for a number of different purposes, and are currently used by the Finder ito tie an icon to a file type, allowing your application or data file to have its own icon.

How to Create a Bundle

A bundle is a collection of resources. To make a bundle for finder icons, we need to set up four types of resources: an ICN#, an FREF, a creator STR and a BNDL.

The ICN# resource type is an icon list. Each ICN# resource contains one or more icons, on after another. For Finder bundle icons, there are **two** icons in each ICN#: one for the icon itself and one for the mask. In our sample bundle, we have two file types, each with its own icon. To define the icons for these files we would enter this into our Rez input file:

```
resource 'ICN#' (732) { /* first icon: the ID number can be anything */
   {                    /* first, the icon */
      $"FF FF FF FF"    /* each line is 4 bytes (32 bits) */
      $"F0 09 CD DD"    /* 32 lines total for icon */
      ...
      $"FF FF FF FF"    /* 32nd line of icon */
      ,                 /* now, the mask */
      $"FF FF FF FF"    /* 32 lines total for mask */
      $"FF FF FF FF"
      ...
      $"FF FF FF FF"    /* 32nd line of mask*/
   }
};
resource 'ICN#' (733) { /* second icon */
   {
      $"FF FF FF FF"
      ...
      ,
      $"FF FF FF FF"
      ...
   }
};
```

Now that we've defined our icons we can set up the FREFs. An FREF is a file type reference; you need one for each file type that has an icon. It ties a file type to a local icon resource ID. This will be mapped by the BNDL onto an actual resource ID number of an ICN# resource. Our FREFs will look like this:

```
resource 'FREF' (816) { /* file type reference for application icon */
   {
      'APPL', 605, /* the type is APPL(ication), the local ID is 605 */
      ""           /* this string should be empty (it is unused) */
   }
};

resource 'FREF' (816) { /* file type reference for a document icon */
   {
      'TEXT', 612, /* the type is TEXT, the local ID is 612 */
      ""           /* this string should be empty (it is unused) */
   }
};
```

The reason that you specify the local ID, rather than the actual resource ID of the ICN# is that the Finder will copy all of the bundle resources into the Desktop file and renumber them to avoid conflicts. This means that the actual IDs will change, but the local IDs will remain the same.

Every application (or other file with a bundle) has a unique four-character signature. The Finder uses this to identify an application. The creator resource that contains a single string, and should be defined like this:

```
type 'MINE' as 'STR '; /* MINE is the signature */
resource 'MINE' (0) { /* the creator resource ID must be 0 */
   "MyProgram 1.0 Copyright 1988"
};
```

Now for the BNDL resource. The BNDL resource associates local resource IDs with actual resource IDs, and also tells the Finder what file types exist, and which ICN#s and FREFs are part of the bundle. The resource looks like this:

```
resource 'BNDL' (128) { /* the bundle resource ID should be 0 */
   'MINE', /* signature of this application */
   0, /* the creator resource ID (this must be 0) */
   {
      'ICN#', /* local resource ID mapping for icons */
      {
         605, 732, /* ICN# local ID 605 maps to 732 */
         612, 733  /* ICN# local ID 612 maps to 733 */
      },
      'FREF', /* local resource ID mapping for file type references */
      {
         523, 816, /* FREF local ID 523 maps to 816 */
         555, 817  /* FREF local ID 555 maps to 817 */
      },
```

When you are in the Finder, your application, type APPL (FREF 816), will be displayed with icon local ID 605 (from the FREF resource). This is ICN# 732. Files of type TEXT (FREF 817) created by your application will be displayed with icon local ID 612 (from the FREF resource). This is ICN# 733.

How the Finder Uses Bundles

If a file has the bundle bit set, but the bundle isn't in the Desktop file, the Finder looks for a BNDL resource. If the BNDL resource matches the signature of theapplication, the Finder then makes a copy of the bundle and puts it in the Desktop file. The file is then displayed with its associated icon.

If a file has lost its icon (it's on a disk without the file containing bundle and the Desktop file doesn't contain the bundle), then it will be displayed with the default document icon until the Finder encounters a copy of the file that contains the right bundle. The Finder then makes a copy of the application's bundle (renumbering resources if necessary) and places it in the Desktop file of that disk.

Problems That May Arise

There are times when you have set up these resource types properly but the icon is either the wrong one or it has defaulted to the standard application or data file icon. There are a number of possible reasons for this.

If you are using the Macintosh-based RMaker, the first thing to check is whether there are any extraneous spaces in your resource compiler input file. The Macintosh-based RMaker is very picky about extra spaces.

If your icon is defaulting to the standard icon, check to see that the bundle bit is set. If the bundle bit isn't set , the Finder doesn't know to place the bundle in the Desktop file. If it isn't in the Desktop file, the Finder displays the file with a default icon.

If you changed the icon and remade the resource file, but the file still has the same old icon when displayed in the Finder. The old icon is still in the Desktop file. The Finder doesn't know that you've changed it, so it uses what it has. To get it to use the new icon you need to rebuild the Desktop file. To force the Finder to rebuild the Desktop file, you can hold down the Option and Command keys on startup or on insertion of the disk in question if it isn't the boot disk. The Finder will ask whether or not you want to rebuild the desktop (meaning the Desktop file).

Have a bundle of fun!

Macintosh Technical Notes

#50: Calling SetResLoad

See also: The Resource Manager
Technical Note #1—DAs and System Resources

Written by: Jim Friedlander October 25, 1985
Updated: March 1, 1988

Calling `SetResLoad(FALSE)` can be useful if you need to get a handle to a resource, without causing the resource to be loaded from disk if it isn't already in memory. This technique is used in Technical Note #1. `SetResLoad` changes the value of the low-memory global `ResLoad` (at location `$A5E`).

It is very important that your program not leave `ResLoad` set to `FALSE` when it exits. Doing this will cause the system to reboot or crash when it does a `GetResource` call for the next code segment to be loaded (usually the Finder). The system will crash because `GetResource` will not actually load the code from disk when `ResLoad` is `FALSE`.

So, make sure that you call `SetResLoad(TRUE)` before exiting your program.

#51: Debugging With PurgeMem and CompactMem

See also: The Memory Manager

Written by: Jim Friedlander October 19, 1985
Updated: March 1, 1988

If you are having problems finding bugs like handles that aren't locked down when they should be, or resources that aren't there when they're supposed to be, there is a handy technique for forcing these problems to the surface. Every time through the main event loop call:

```
    PurgeMem(MaxSize);                                    {MaxSize = $800000}
    size:= CompactMem(MaxSize);
```

PurgeMem will purge all purgeable blocks and CompactMem will rearrange the heap, trying to find a contiguous free block of MaxSize bytes. Obviously, this will move things around quite a bit, so, if there are any unlocked handles that you have de-referenced, you will find out about them very quickly.

Don't be alarmed when you see the performance of your program deteriorate drastically—it's because lots of resources are being loaded and purged every time through the main event loop. You might want to have a debugging menu item that toggles between glacial and normal execution speeds.

Please be sure to remove these two lines from any code that you ship!! In fact, neither of these two calls should normally be made from your application. They tend to undo work that has been done by the Memory and Resource Managers.

#52: Calling _Launch From a High-Level Language

Revised by: Rich Collyer April 1989
Written by: Jim Friedlander November 1985

This Technical Note formerly discussed calling `_Launch` from a high-level language which allows inline assembly code.
Changes since March 1988: Merged contents into Technical Note #126.

This Note formerly discussed calling `_Launch` from a high-level language. The information on calling `_Launch` is now contained in Technical Note #126, Sub(Launching) From a High-Level Language, which also covers sublaunching other applications.

Macintosh Technical Notes

#53: MoreMasters Revisited

See also: The Memory Manager

Written by: Jim Friedlander October 28, 1985
Updated: March 1, 1988

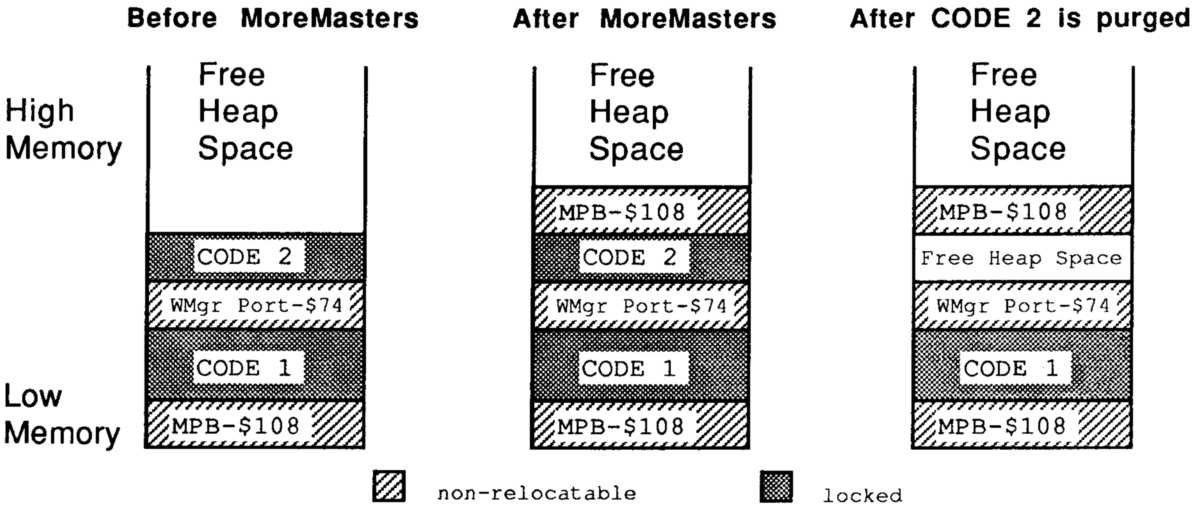
`MoreMasters` should be called from CODE segment 1. The number of master pointers that a program needs can be determined empirically. `MoreMasters` can be tricked into creating the exact number of master pointers desired.

If you ask Macintosh programmers when and how many times `MoreMasters` should be called, you will get a variety of answers, ranging from "four times in the initialization segment" to "once, anywhere." As you might suspect, the answer is somewhat different from either of these.

`MoreMasters` allocates a block of master pointers in the current heap zone. In the application heap, a block of master pointers consists of 64 master pointers; in the system heap, a block consists of 32 master pointers. Since master pointer blocks are **non-relocatable**, we want to be sure to allocate them early. The system will allocate one master pointer block as your program loads. It's the first object in the application heap—its size is $108 bytes.

A lot of programmers call `MoreMasters` from an "initialization" segment, but as we shall see, that's not such a good idea. The problem occurs when we unload our "initialization" segment and it gets purged from memory.

The following diagrams of the application heap illustrate what happens if we call MoreMasters from CODE segment 2 (MPB stands for Master Pointer Block):



Notice that we now have some heap fragmentation—not serious, but it can be avoided by making all MoreMasters calls in CODE segment 1. Because InitWindows creates the Window Manager Port (WMgrPort), it should also be called from CODE segment 1. Both MoreMasters and InitWindows should be called before another CODE segment is loaded, or the non-relocatable objects they allocate will be put above the CODE segment and you'll get fragmentation when the CODE segment is purged. If you want to call an initialization segment before calling MoreMasters and InitWindows, make sure that you unload it before you call either routine.

Now that we know when to call MoreMasters, how many times do we call it? The answer depends on your application. If you don't call MoreMasters enough times, the system will call it when it needs more master pointers. This can happen at very inconvenient times, causing heap fragmentation. If you call MoreMasters too often, you can be wasting valuable memory. This is preferable, however, to allocating too few master pointer blocks!

The number of times you should call MoreMasters can be empirically determined. Once your application is almost finished, remove all MoreMasters calls. Exercise your application as completely as possible, opening windows, using handles, opening desk accessories, etc. You can then go in with a debugger and see how many times the system called MoreMasters. You do that by counting the non-relocatables of size $108. Due to Memory Manager size correction, the master pointer blocks can also have a size of $10C or $110 bytes. You should give yourself about 20% leeway — that is, if the system called MoreMasters 10 times for you, you should call it 12 times. If you're more cautious, you might want to call MoreMasters 15 times.

Another technique that can save time at initialization is to calculate the number of master pointers you will need, then set the `MoreMast` files of the heap zone header to that number, and then call `MoreMasters` once:

```
PROCEDURE MyMoreMasters(numMastPtrs : INTEGER);

VAR
   oldMoreMast : INTEGER;      {saved value of MoreMast}
   zone        : THz;          {heap zone}

BEGIN
   zone := GetZone;            {get the heap zone}
   WITH zone^ DO BEGIN
      oldMoreMast := MoreMast; {get the old value of MoreMast}
      MoreMast := numMastPtrs; {put the value we want in the zone header}
      MoreMasters;             {allocate the master pointers}
      MoreMast := oldMoreMast; {restore the old value of MoreMast}
   END;
END;
```

In MPW C:

```
void MyMoreMasters(numMastPtrs)
short numMastPtrs;

{ /* MyMoreMasters */
   short     oldMoreMast;            /* saved value of MoreMast*/
   THz       oZone;                  /* heap zone*/

   oZone = GetZone();                /* get the heap zone*/
   oldMoreMast = oZone->moreMast;    /* get the old value of MoreMast*/
   oZone->moreMast = numMastPtrs;    /* put the value we want in the
                                        zone header */
   MoreMasters();                    /*allocate the master pointers*/
   oZone->moreMast = oldMoreMast;    /*restore the old value of MoreMast*/
} /* MyMoreMasters */
```

Macintosh Technical Notes

#54: Limit to Size of Resources

Written by:	Jim Friedlander	October 23, 1985
Updated:		March 1, 1988

This note formerly described a bug in `WriteResource` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#55: Drawing Icons

See also: QuickDraw
Toolbox Utilities

Written by: Jim Friedlander October 21, 1985
Updated: March 1, 1988

Using resources of type ICON allows drawing of icons in `srcOr` mode. Using resources of type ICN# allows for more variety when drawing icons.

There are two different kinds of resources that contain icons: ICON and ICN#. An ICON is a 32 by 32 bit image of an icon and can be drawn using the following Toolbox Utilities calls:

```
MyIconHndl:= GetIcon(iconID);
PlotIcon(destRect,iconID);
```

While very convenient, this method only allows the drawing of icons in SrcOr mode (as in the MiniFinder). The Finder uses resources of type ICN# to draw icons on the desktop. Because the Finder uses ICN#s, it can draw icons in a variety of ways.

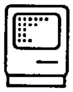
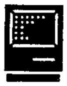
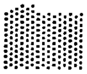
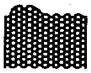
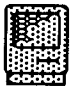
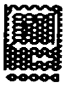
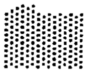
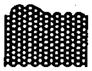
An ICN# resource is a list of 32 by 32 bit images that are grouped together. Common convention has been to group two 32 by 32 bit images together in each ICN#. The first image is the actual icon, the second image is the mask for the icon. To get a handle to an ICN#, we would use something like this:

```
TYPE
  iListHndl    = ^iListPtr;
  iListPtr     = ^iListStruct;
  iListStruct  = record
                   icon : packed array[0..31] of Longint;
                   mask : packed array[0..31] of Longint;
                 End; {iListStruct}

VAR
   myILHndl        : iListHndl;                     {handle to an ICN#}
   iBitMap         : BitMap;                       {BitMap for the icon}
   mBitMap         : BitMap;                       {BitMap for the mask}

MyILHndl:= iListHndl(GetResource('ICN#',iconID));
if MyILHndl = NIL then HandleError; { and exit or whatever is appropriate}
```

Once we have a handle to the icons, we need to set up two bitMaps that we will be using later in CopyBits:

```
   SetRect(icnRect,0,0,32,32);                  { define the icon's 'bounds'}
   With iBitMap do Begin
       baseAddr:= @MyILHndl^^.icon;
       rowbytes:= 4;                                             { 4 * 8 =32}
       bounds:= icnRect;
   End; {with}
   With mBitMap do Begin
       baseAddr:= @MyILHndl^^.mask;
       rowbytes:= 4;
       bounds:= icnRect;
   End; {with}
```

Icons can represent desktop objects that are either selected or not. Folder and volume icons can either be open or not. The object (or the volume it is on) can either be online or offline. The Finder draws icons using all permutations of open, selected and online:

	Non-Open Non-Selected	Non-Open Selected	Open Non-Selected	Open Selected
Online				
Offline				

Drawing icons as non-open is basically the same for online and offline volumes. We need to punch a hole in the desktop for the icon. This is analogous to punching a hole in dough with an irregular shaped cookie-cutter. We can then sprinkle jimmies* all over the cookie and they will only stick in the area that we punched out (the mask). We do this by copyBitsing the mask onto the desktop (whatever pattern) to our destRect. For non-open, non-selected icons:

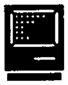
we use the SrcBic mode so that we punch a white hole:

```
   SetRect(destRect,left,top,left+32,top+32);
   CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);
```

Then we XOR in the icon:

```
   CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcXor,NIL);
```

That's all there is to drawing an icon as non-open, non-selected. To draw the icon as non-open, selected:



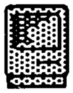
we will OR in the mask, causing a mask-shaped BLACK hole to be punched in the desktop:

```
    CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcOr,NIL);
```

Then, as before, we XOR in the icon:

```
    CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcXOr,NIL);
```

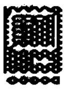
To draw icons as non-opened for offline volumes:



we need to do a little more work. We need to XOR a ltGray pattern into the boundsRect of the icon. We will then punch the hole, draw the icon and then XOR out the ltgray pattern that does not fall inside the mask. So, to draw the icon as offline, non-open, non-selected we would:

```
    GetPenState(OldPen);             {save the pen state so we can restore it}
    PenMode(patXor);
    PenPat(ltGray);
    PaintRect(destRect);                 {paint a ltGray background for icon}

    CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);   {punch}
    PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                        {filled with ltGray}
    CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcOr,NIL);  { OR in }
                                                 { the icon to the ltGray mask}
    SetPenState(OldPen);                               {restore the old pen state}
```

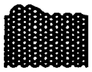
To draw the icon as offline, non-open, selected:



we would use a similar approach:

```
    GetPenState(OldPen);            { save the pen state so we can restore it}
    PenMode(patXor);
    PenPat(dkGray);                { the icon is selected, so we need dkGray }
    PaintRect(destRect);                { paint a dkGray background for icon }

    CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);   {punch}
    PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                        {filled with dkGray}
    CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL); {BIC the}
                                                    {icon to the dkGray mask}
    SetPenState(OldPen);                               {restore the old pen state}
```

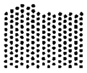
Drawing the opened icons requires one less step. We don"t have to `CopyBits` the icon in, we just use the mask. Online and offline icons are drawn the same way. To draw icons as open, selected:



we do the following:

```
GetPenState(OldPen);             {save the pen state so we can restore it}
PenMode(patXor);
PenPat(dkGray);                { the icon is selected, so we need dkGray }
PaintRect(destRect);                 { paint a dkGray background for icon}
CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);   {punch}
PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                      {filled with dkGray}
SetPenState(OldPen);                          {restore the old pen state}
```

To draw icons as open, non-selected:



we just need to change one line from above. Instead of XORing with a dkGray pattern, we use a ltGray pattern:

```
PenPat(ltGray);            { the icon is non-selected, so we need ltGray }
```

These techniques will work on any background, window-white or desktop-gray and all patterns in between. Have fun.

* *jimmies* : little bits of chocolate

#56: Break/CTS Device Driver Event Structure

See also: The Device Manager
Serial Drivers
Zilog Z8030/Z8530 SCC Serial Communications Controller Technical Manual

Written by: Mark Baumwell

December 2, 1986
Updated: March 1, 1988

This technical note documents the event record information that gets passed when the serial driver posts an event for a break/CTS status change.

The serial driver can be programmed to post a device driver event upon encountering a break status change or CTS change (via the `SerHShake` call). The structure of device driver events is driver-specific. This technical note documents the event record information that gets passed when the serial driver posts a device driver event for a break/CTS status change.

When the event is posted, the message field of the event record will be a long word (four bytes). The most significant byte will contain the value of SCC Read Register 0 (see below for the relevant Read Register 0 values). The next byte will contain the changed (since the last interrupt) bits of the SCC read register 0. The lower two bytes (word) will contain the `DCtlRefNum`.

The values for Read Register 0 are as follows:

- If a break occurred, bit 7 will be set.
- If CTS changed, bit 5 will reflect the state of the CTS pin (0 means the handshake line is asserted and that it is OK to transmit).

We discourage posting these events because interrupts would be disabled for a long time while the event is being posted. However, it is possible to detect a break or read the value of the CTS line in another way. A break condition will **always** terminate a serial driver input request (but not an output request), and the error breakRecd (–90) will be returned. (This constant is defined in the SysEqu file.) You could therefore detect a break by checking the returned error code.

The state of the CTS line can be checked by making a `SerStatus` call and checking the value of the `ctsHold` flag in the `SerStaRec` record. See the Serial Drivers chapter of *Inside Macintosh* for details.

#57: Macintosh Plus Overview

See: *Inside Macintosh Volume IV*

Written by: Scott Knaster January 8, 1986
Updated: March 1, 1988

This note was originally meant as interim Macintosh Plus documentation and has been replaced by *Inside Macintosh Volume IV*, which is more complete and more accurate.

#58: International Utilities Bug

Written by: Jim Friedlander January 24, 1986
Updated: March 1, 1988

This note formerly described a bug in System 2.0, which is now recommended only for use with 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#59: Pictures and Clip Regions

See also: QuickDraw

Written by: Ginger Jernigan January 16, 1986
Updated: March 1, 1988

This note describes a problem that affects creation of QuickDraw pictures.

When a `GrafPort` is created, the fields in the `GrafPort` are given default values; one of these is the clip region, which is set to the rectangle (–32767, –32767, 32767, 32767). If you create a picture, then call `DrawPicture` with a destination rectangle that is not the same size as the `picFrame` without ever changing the default clip region, nothing will be drawn.

When the picture frame is compared with the destination rectangle and the picture is scaled, the clip region is scaled too. In the process of scaling, the clip region you end up overflows and becomes empty, and your picture doesn't get drawn. If you call `ClipRect(thePort^.portRect)` before you record the picture, the picture will be drawn correctly. The clipping on the destination port when playing back the picture is irrelevant: once a picture is incorrectly recorded, it is too late.

#60: Drawing Characters into a Narrow GrafPort

See also: QuickDraw

Written by: Ginger Jernigan January 20, 1986
Updated: March 1, 1988

When you draw a character into a `GrafPort`, your program will die with an address error if the width of the `GrafPort` is smaller than the width of the character. If you check before drawing the character to see if the `GrafPort` is wide enough, you can avoid this unfortunate tragedy.

#61: GetItemStyle Bug

Written by: Jim Friedlander January 21, 1986
Updated: March 1, 1988

This note formerly described a bug (in `GetItemStyle`) which occurs only on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

Macintosh Technical Notes

#62: Don't Use Resource Header Application Bytes

See also: The Resource Manager

Written by: Bryan Stearns January 23, 1986
Updated: March 1, 1988

The section of the Resource Manager chapter of *Inside Macintosh* which describes the internal format of a resource file shows an area of the resource header labeled "available for application data." You **should not** use this area—it is used by the Resource Manager.

Macintosh Technical Notes

#63: WriteResource Bug Patch

Written by:	Rick Blair Jim Friedlander Bryan Stearns	January 15, 1986
Modified by :	Jim Friedlander	March 3, 1986
Updated:		March 1, 1988

This note formerly contained a patch to fix a bug in `WriteResource` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

Macintosh Technical Notes

#64: IAZNotify

Written by:	Jim Friedlander	January 15, 1986
Modified by:	Jim Friedlander	August 18, 1986
Updated:		March 1, 1988

Previous versions of this technical note recommended use of a low memory hook called `IAZNotify`. We no longer recommend use of `IAZNotify`, since the `IAZNotify` hook is never called under MultiFinder.

Macintosh Technical Notes

#65: Macintosh Plus Pinouts

See also: *Macintosh Hardware Reference Manual*

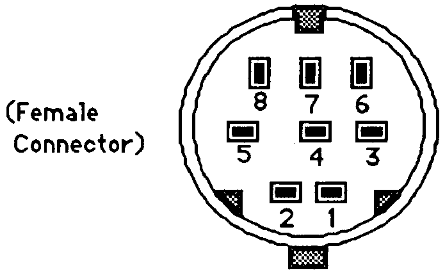
Written by:	Mark Baumwell	January 27, 1986
Modified by:	Mark Baumwell	March 20, 1986
Updated:		March 1, 1988

This note gives pinout descriptions for some of the Macintosh Plus ports and Macintosh Plus cables that are different than the Macintosh 128K and 512K.

Below are pinout descriptions for some Macintosh Plus ports and cables that are different than the Macintosh 128K and 512K. Note that any unconnected pins are omitted.

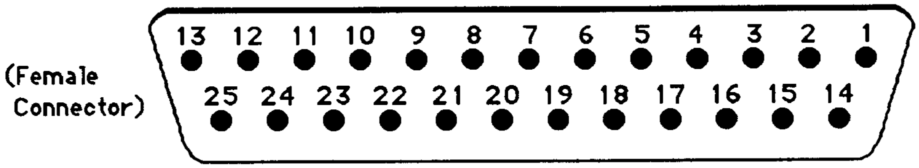
Macintosh Plus Port Pinouts

Macintosh Plus Serial Connectors (Mini DIN-8)



Pin	Name	Description/Notes
1	HSKo	Output Handshake (from Zilog 8530 DTR pin)
2	HSKi/External Clock	Input Handshake (CTS) or TRxC (depends on 8530 mode)
3	TxD–	Transmit Data line
4	Ground	
5	RxD–	Receive Data line
6	TxD+	Transmit Data line
7	Not connected	
8	RxD+	Receive Data line; ground this line to emulate RS232

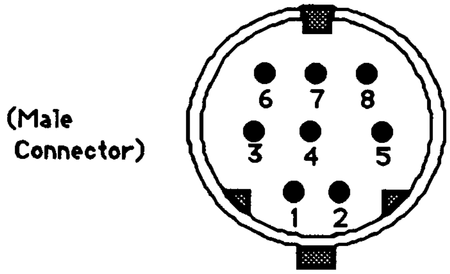
Macintosh Plus SCSI Connector (DB-25)



Pin	Name	Description/Notes
1	REQ–	
2	MSG–	
3	I/O–	
4	RST–	
5	ACK–	
6	BSY–	
7	Ground	
8	DB0–	
9	Ground	
10	DB3–	
11	DB5–	
12	DB6–	
13	DB7–	
14	Ground	
15	C/D–	
16	Ground	
17	ATN–	
18	Ground	
19	SEL–	
20	DBP–	
21	DB1–	
22	DB2–	
23	DB4–	
24	Ground	
25	TPWR	Not connected

Macintosh Plus Cable Pinouts

Apple System Peripheral-8 Cable (connects Macintosh Plus to ImageWriter II and Apple Personal Modem)
(Product part number: M0187)
(Cable assembly part number: 590-0340-A (stamped on cable itself).



(DIN-8)	(DIN-8)
1	2
2	1
3	5
4	4
5	3
6	8
7	7
8	6

Macintosh Plus Adapter Cable (connects Macintosh Plus DIN-8 to existing Macintosh DB-9 cables)
(Apple part number: M0189)
(Cable assembly part number: 590-0341-A (stamped on cable itself).

(DIN-8)	Name	(DB-9)	Notes
1	+12V	6	
2	HSK	7	
3	TxD–	5	
4	Ground	3	Jumpered to DB-9 pin 1 (in DB-9 connector)
5	RxD–	9	
6	TxD+	4	
7	no wire		
8	RxD+	8	
	Ground	1	Jumpered to DB-9 pin 3 (in DB-9 connector)

#66: Determining Which File System Is Active

Revised by: Robert Lenoil & Brian Bechtel August 1990
Written by: Jim Friedlander December 1985

This Technical Note discusses how to determine which file system a particular volume is running.
Changes since June 1990: Removed text about IDs $0001-$0016 being AppleShare volumes; other file systems use this range too.

Under certain circumstances it is necessary to determine which file system is currently running on a particular volume. For example, on a 64K ROM machine, your application (i.e., especially disk recovery utilities or disk editors, etc.) may need to check for MFS versus HFS. Note that this is usually not necessary, because all ROMs, except the original 64K ROMs, include HFS. If your application only runs on 128K ROMs or newer, you do not need to check for HFS versus MFS. You may need to check if a particular volume is in High Sierra, ISO 9660, or audio CD format.

Before performing these file system checks, be sure to call `_SysEnvirons`, to make sure the machine on which you are running has ROMs which know about the calls you need.

To check for HFS on 64K ROM machines, check the low-memory global `FSFCBLen` (at location `$3F6`). This global is one word in length (two bytes) and is equal to -1 if MFS is active and a positive number (currently `$5E`) if HFS is active. From Pascal, the following would perform the check:

```
CONST
  FSFCBLen = $3F6;     {address of the low-memory global}

VAR
  HFS: ^INTEGER;

  ...
  HFS:= POINTER(FSFCBLen);
  IF HFS^ > 0 THEN
      {we're running HFS}
  ELSE
      {we're running MFS}
  END;
```

If an application determines that it is running under HFS, it should not assume that all mounted volumes are HFS. To check individual volumes for HFS, call `_PBHGetVInfo` and check the directory signature (the `ioVSigWord` field of an `HParamBlockRec`). A directory signature of $D2D7 means the volume is an MFS volume, while a directory signature of $4244 means the volume is an HFS volume.

To find out if a volume uses a file system other than HFS or MFS, call `_PBHGetVInfo` and check the file system ID (the `ioVFSID` field of an `HParamBlockRec`). A file system ID of $0000 means the volume is either HFS or MFS. A file system ID of $4242 means the volume is a High Sierra volume, while a file system ID of $4147 is an ISO 9660 volume, and a file system ID of $4A48 is an audio CD volume. AppleShare and other file systems use a dynamic technique of obtaining the first unused file system ID; therefore, low-numbered IDs cannot be associated with any particular file system.

When dealing with High Sierra and ISO 9660 formats, do not assume that the volumes are CD-ROM discs. Support for these file systems is done with the External File System hook in the File Manager, so any block-based media could potentially be in these formats. It is possible to have a High Sierra formatted floppy disk, although it would be useless except for testing purposes.

Further Reference:

- *Inside Macintosh*, Volume IV, File Manager
- Technical Note #209, High Sierra & ISO 9660 CD-ROM Formats
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond

Developer Technical Support

#67: How to Bless a Folder to Be the System Folder

Rewritten by: Colleen K. Delgadillo May 1992
Updated by: Jim Friedlander March 1988
Written by: Jim Friedlander January 1986

This Technical Note describes how to determine which folder on an HFS volume is the blessed folder, that is, the folder that contains both the System file and the Finder.

Changes since January 1986: The information about how to find the "Blessed Folder" has been deleted from this technical note. The FindFolder function can now be used to find the "Blessed Folder" and is documented in Inside Macintosh Volume VI, pages 9-42 to 9-44. This note now includes information about how to bless a folder to the new system folder.

Note: The following information may be affected by future changes to system software. If you choose to use this information, you must do so at your own risk.

The way to bless a folder is by taking the longword which is the directory ID of the blessed folder and putting it into the Master Directory Block (MDB). This can be accomplished by using the HFS call `PBSetVInfo`. You should not attempt to change this block directly. First call `PBHGetVInfo` and set `ioVFnderInfo[1]` to the directory ID of the the new folder to be blessed. Then call `PBSetVInfo` to save this information. Once you have done this, you will find that the Finder takes a little while to realize that you have blessed the folder. Therefore, the icon will take a little while to change. Exactly how long you will have to wait to see the new icon is unknown.

Forcing the icon to change sooner is not a difficult task. The best way for you to see the icon change more quickly is to change the modification date of the directory into which you are copying the new System Folder. Doing this will cause the Finder to reexamine the window and its contents. When the Finder notices that the volume's modification date has changed, it begins scanning for changes in all the open folders. This scanning process takes place about once every 10 seconds. You can change the last modification date for that volume and the System Folder's directory ID for that volume using `PBSetVInfo`. Changing the file's `FndrInfo` or renaming the file does not change the modification date. When you call `PBSetVInfo` you will need to put the System Folder's directory ID in the longword at `ioVfndrInfo`. This longword will be the first four bytes of this directory ID. (As usual, whenever you make a change to a field of a structure you need to first do a `PBGetCatInfo`, change what you are going to change, and then do `PBSetCatInfo`. This ensures that you change only the portion of the volume that you intended, in this case a longword, and not the whole structure.)

Further Reference:

- Master Directory Block: Inside Macintosh Volume IV on page 166.

#68: Searching Volumes—Solutions and Problems

Revised by: Jim Luther — January 1992
Written by: Jim Friedlander and Rick Blair — December 1985 – October 1988

This Technical Note discusses the PBCatSearch function and tells why it should be used. It also provides simple algorithms for searching both MFS and HFS volumes and discusses the problems with indexed search routines.

Changes since October 1988: Includes information on PBCatSearch and notes the problems with indexed search routines. Source code examples have been added and revised. Thanks to John Norstad at Northwestern University for pointing out some of the shortcomings of the indexed search routines. Thanks to the System 7 engineering team for adding PBCatSearch.

It may be necessary to search the volume hierarchy for files or directories with specific characteristics. Generally speaking, your application should avoid searching entire volumes because searching can be a very time-consuming process on a large volume. Your application should rely instead on files being in specific directories (the same directory as the application, or in one of the system-related folders that can be found with FindFolder) or on having the user find files with Standard File.

Searching MFS Volumes

Under MFS, indexed calls to PBGetFInfo return information about all files on a given volume. Under HFS, the same technique returns information only about files in the current directory. Here's a short code snippet showing how to use PBGetFInfo to list all files on an MFS volume:

```
PROCEDURE EnumMFS (theVRefNum: Integer);
{ search the MFS volume specified by theVRefNum }
   VAR
      pb: ParamBlockRec;
      itemName: Str255;
      index: Integer;
      err: OSErr;
BEGIN
   WITH pb DO
      BEGIN
         ioNamePtr := @itemName;
         ioVRefNum := theVRefNum;
         ioFVersNum := 0;
      END;
   index := 1;
   REPEAT
      pb.ioFDirIndex := index;
      err := PBGetFInfoSync(@pb);
      IF err = noErr THEN
         BEGIN
            { do something useful with the file information in pb }
            END;
         index := index + 1;
      UNTIL err <> noErr;
   END;
```

As noted in Macintosh Technical Note #66, a directory signature of $D2D7 means a volume is an MFS volume, while a directory signature of $4244 means the volume is an HFS volume.

Searching HFS Volumes

Fast, Reliable Searches Using `PBCatSearch`

The fastest and most reliable way to search an HFS volume's catalog is with the File Manager's `PBCatSearch` function. `PBCatSearch` returns a list of `FSSpec` records to files or directories that match the search criteria specified by your application. However, `PBCatSearch` is not available on all volumes or under all versions of the File Manager. Volumes that support `PBCatSearch` can be identified using the `PBHGetVolParms` function. (See the following code.) Versions of the File Manager that support `PBCatSearch` can be identified with the `gestaltFSAttr` Gestalt selector and `gestaltFullExtFSDispatching` bit as shown in the following code:

```
FUNCTION HasCatSearch (vRefNum: Integer): Boolean;
{ See if volume specified by vRefNum supports PBCatSearch }
   VAR
      pb: HParamBlockRec;
      infoBuffer: GetVolParmsInfoBuffer;
      attrib: LongInt;

BEGIN
   HasCatSearch := FALSE; { default to no PBCatSearch support }
   IF GestaltAvailable THEN { See Inside Macintosh Volume VI, Chapter 3 }
      IF Gestalt(gestaltFSAttr, attrib) = noErr THEN
         IF BTst(attrib, gestaltFullExtFSDispatching) THEN
            BEGIN { this version of the File Manager can call PBCatSearch }
               WITH pb DO
                  BEGIN
                     ioNamePtr := NIL;
                     ioVRefNum := vRefNum;
                     ioBuffer := @infoBuffer;
                     ioReqCount := sizeof(infoBuffer);
                  END;
               IF PBHGetVolParmsSync(@pb) = noErr THEN
                  IF BTST(infoBuffer.vMAttrib, bHasCatSearch) THEN
                     HasCatSearch := TRUE; { volume supports PBCatSearch }
            END;
END;
```

Note: File servers that support the AppleTalk Filing Protocol (AFP) version 2.1 support `PBCatSearch`. That includes volumes and directories shared by System 7 File Sharing and by the AppleShare 3.0 file server. Although AFP version 2.1 supports `PBCatSearch`, the `fsSBNegate` bit is not supported in the `ioSearchBits` field. Using `PBCatSearch` to ask the file server to perform the search is usually faster than using the recursive indexed search described in the next section.

`PBCatSearch` should be used if it is available because it is usually *much* faster than a recursive search. For example, the search time for finding all files and directories on a recent Developer CD was around 18 seconds with PBCatSearch. It took 6 minutes and 36 seconds with a recursive indexed search. How long do you want the users of your application to wait?

PBCatSearch can be used to collect a list of FSSpec records to all items on a volume by setting ioSearchBits in the parameter block to 0.

Recursive Indexed Searches Using PBGetCatInfo

When PBCatSearch is not available, an application must resort to a recursive indexed search. There are a couple of potential problems with a recursive indexed search; a recursive indexed search can use up a lot of stack space and the volume directory structure can change in the multi-user/multiprocess Macintosh environment. The example code in this note addresses the stack space problem, but for reasons explained later, does not address problems caused by multiple users or processes changing the volume directory structure during a recursive search.

The default stack space on the Macintosh can be as small as 8K; therefore, the recursive indexed search example shown in this Note encloses the actual recursive routine in a shell that can hold most of the variables needed, which dramatically reduces the size of the stack frame. This example uses only 26 bytes of stack space each time the routine recurses. That is, it could search 100 levels deep (pretty unlikely) and use only 2600 bytes of stack space.

Please notice that when the routine comes back from recursing, it has to clear the nonlocal variable err to noErr, since the reason the routine came back from recursing is that PBGetCatInfo returned an error:

```
EnumerateCatalog(myCPB.ioDrDirID);
err := noErr; {clear error return on way back}
```

Please notice also that you must set myCPB.ioDrDirId each time you call PBGetCatInfo, because if PBGetCatInfo gets information about a file, it returns ioFlNum (the file number) in the same location that ioDrDirID previously occupied.

Be sure to check bit 4, the fifth least significant bit, when you check the file attributes bit to see if you've got a file or a folder. The following routine uses MPW Pascal's BTST function to check that bit. If you use the Toolbox bit manipulation routines (e.g., BitTst), remember to order the bits in reverse order from standard 68000 notation.

Here is the routine in MPW Pascal:

```
PROCEDURE EnumerShell (vRefNumToSearch: Integer; { the vRefNum to search}
                       dirIDToSearch: LongInt);  { the dirID to search }
   VAR
      itemName: Str63;
      myCPB: CInfoPBRec;
      err: OSErr;

   {-----}

   PROCEDURE EnumerateCatalog (dirIDToSearch: LongInt);
      CONST
         ioDirFlgBit = 4;
      VAR
         index: Integer;
   BEGIN { EnumerateCatalog }
      index := 1;
      REPEAT
         WITH myCBP DO
```

```
            BEGIN
               ioFDirIndex := index;
               ioDrDirID := dirIDToSearch; { we need to do this every }
                                           { time through }
               filler2 := 0; { Clear the ioACUser byte if search is  }
                             { interested in it. Nonserver volumes }
                             { won't clear it for you and the value  }
                             { returned is meaningless. }
            END;
         err := PBGetCatInfo(@myCPB, FALSE);
         IF err = noErr THEN
            IF BTST(myCPB.ioFlAttrib, ioDirFlgBit) THEN
               BEGIN { we have a directory }

                  { do something useful with the directory information }
                  { in myCPB }

                  EnumerateCatalog(myCPB.ioDrDirID);
                  err := noErr; {clear error return on way back}
               END
            ELSE
               BEGIN { we have a file }

                  { do something useful with the file information }
                  { in myCPB }

               END;
         index := index + 1;
      UNTIL (err <> noErr);
   END;  { EnumerateCatalog }

   {-----}

BEGIN { EnumerShell }
   WITH myCPB DO
      BEGIN
         ioNamePtr := @itemName;
         ioVRefNum := vRefNumToSearch;
      END;
   EnumerateCatalog(dirIDToSearch);
END; { EnumerShell }
```

In MPW C:

```
/* the following variables are globals */
HFileInfo      gMyCPB;           /* for the PBGetCatInfo call */
Str63          gItemName;        /* place to hold file name */
OSErr          gErr;             /* the usual */

/*--------------------------------------------------------------------*/

void EnumerateCatalog (long int dirIDToSearch)
{  /* EnumerateCatalog */

   short int          index=1;
   do
   {
      gMyCPB.ioFDirIndex= index;
      gMyCPB.ioDirID= dirIDToSearch; /* we need to do this every time    */
                                     /* through, since GetCatInfo        */
                                     /* returns ioFlNum in this field */
      gMyCPB.filler2= 0; /* Clear the ioACUser byte if search is         */
                         /* interested in it. Nonserver volumes won't  */
                         /* clear it for you and the value returned is */
                         /* meaningless. */
      gErr= PBGetCatInfo(&gMyCPB,false);
      if (gErr == noErr)
      {
         if ((gMyCPB.ioFlAttrib & ioDirMask) != 0)
         {  /* we have a directory */

            /* do something useful with the directory information */
            /* in gMyCPB */

            EnumerateCatalog(gMyCPB.ioDirID); /* recurse */
            gErr = noErr; /* clear error return on way back */
         }
         else
         {  /* we have a file */

            /* do something useful with the file information */
            /* in gMyCPB */

         }
      }
      ++index;
   } while (gErr == noErr);
}  /* EnumerateCatalog */

/*---------------------------------------------------------------------*/

EnumerShell(short int vRefNumToSearch, long int dirIDToSearch)

{  /* EnumerShell */
   gMyCPB.ioNamePtr = gItemName;
   gMyCPB.ioVRefNum = vRefNumToSearch;
   EnumerateCatalog(dirIDToSearch);
}  /* EnumerShell */
```

Please make sure that you are running under HFS before you use this routine (see Technical Note #66). You can search the entire volume by specifying a starting directory ID of `fsRtDirID`, the root directory constant. You can do partial searches of a volume by specifying a starting directory ID other than `fsRtDirID`.

Searching in a Multi-user/Multiprocess Environment

Volumes can be shared by multiple users accessing a file server or multiple processes running on a single Macintosh. Each user or process with access to such a shared volume may be able to make changes to the volume's catalog at any time. Changes in a volume's catalog in the middle of a search can cause two problems:

- Files and directories renamed or moved by another user or process can be entirely missed or found multiple times by a search routine.
- A search routine can easily lose track of its position within the hierarchical directory structure when files or directories are created, deleted, or renamed by another user or process.

A volume searched with a single call to `PBCatSearch` ensures that all parts of the volume are searched without another user or process changing the volume's catalog. However, a single call to `PBCatSearch` may not be possible or practical because of the number of matches you expect, or because you may want to set a time limit on the search so that the user can cancel a long search. `PBCatSearch` returns a `catChangedErr` (–1304) and no matches when the catalog of a volume is changed by another user or process in a way that might affect the current search. The search can be continued with the `CatPositionRec` returned with the `catChangedErr` error, but at the risk of missing catalog entries or finding duplicate catalog entries.

Things aren't so nice for search routines based on indexed File Manager calls. The File Manager won't notify you when a volume's catalog has changed. In fact, there are several ways the catalog can change that are very difficult to detect and correct for. Since methods that attempt to resynchronize an indexed search and find all catalog entries that might be missed or found multiple times when the catalog changes do not work for all cases, those methods are not discussed in this Technical Note. The following paragraphs describe why some changes are very difficult to detect.

There are three changes you can make to the contents of a directory that change the list of files and directories returned by an indexed search: creating, deleting, and renaming. Directories of an HFS volume are always sorted alphabetically, so when a file or subdirectory is deleted from a directory, any directory entries after it bubbles up to fill the vacated entry position; when a file or subdirectory is created, it is inserted into the list and all entries after it bubbles down one position. When a file or subdirectory is renamed, it is removed from its current position and moved into its alphabetically correct position. The first two changes, creating and deleting, can be detected only at the parent directory level. That's because a creation or deletion changes only the modification date of the parent directory but not the modification date of any of the parent directory's ancestors. Renaming a file or subdirectory does not change the modification date of the file or subdirectory renamed or the modification date its parent directory, but it does change the order of files and subdirectories found by an indexed search.

With this in mind, here are a couple of examples that are very difficult to detect.

The first example shows a file, Dashboard, moved (by another user or process) with `PBCatMove` from the CDevs subdirectory to the Control Panels subdirectory. (See figures 1 and 2.) At the time of the move, the search routine has just finished recursively looking through the Development directory and is ready to recursively search the Games directory. After the move, two directories, CDevs and Control Panels, have new modification dates but no change is seen at the root directory of My Disk. There is nothing to immediately tell the search routine something has changed (except for the volume modification date which may or may not mean the directory structure has changed), so the search will see Dashboard twice. If the move were in the opposite direction, from Control Panels to CDevs, Dashboard would be missed by the search routine.

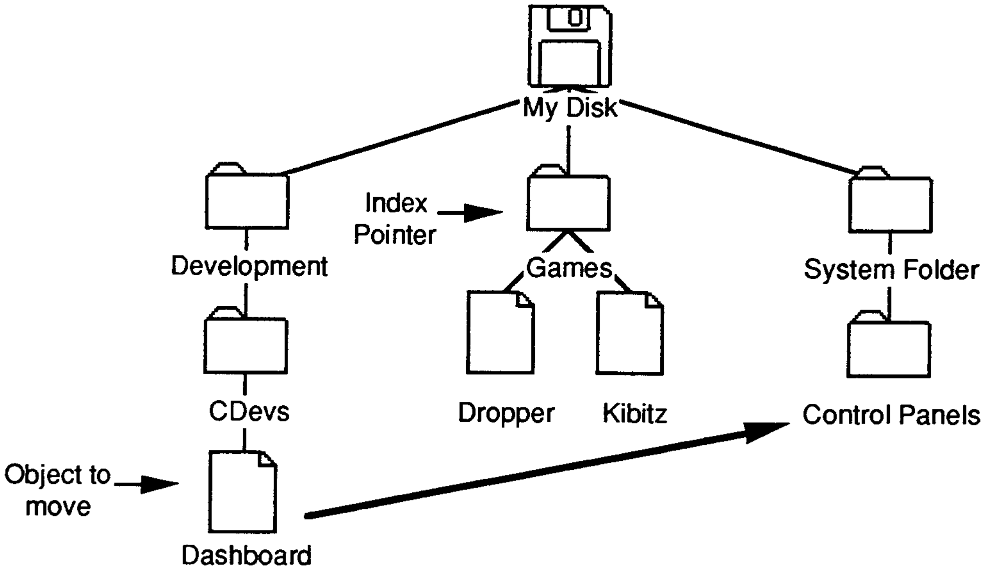


Figure 1 Before Dashboard Is Moved With `PBCatMove`

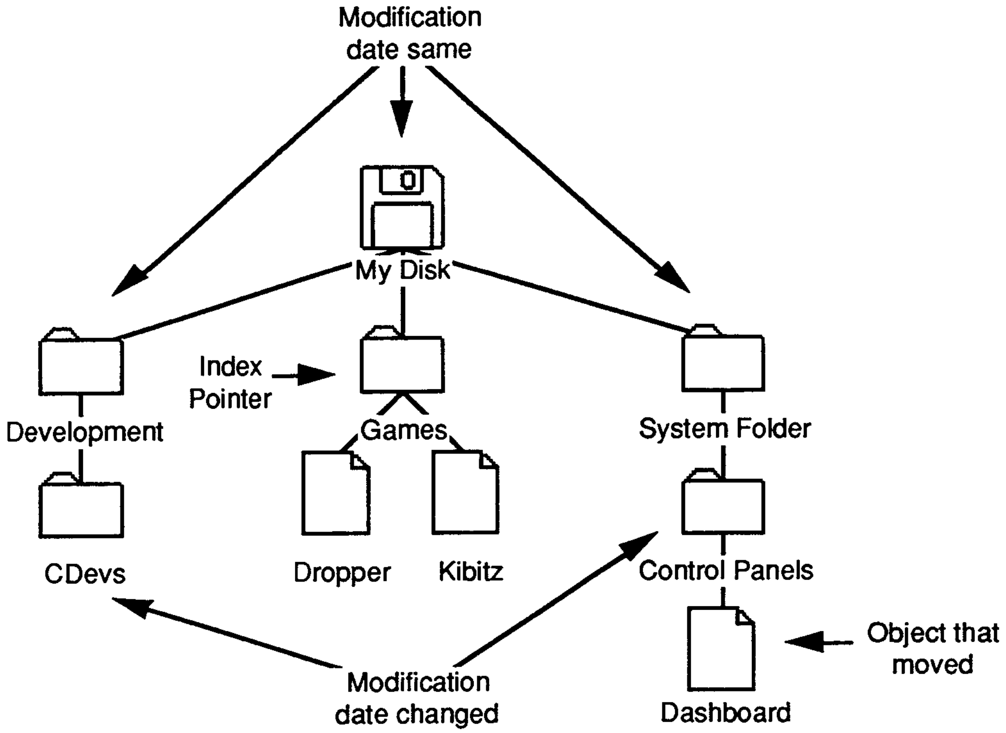


Figure 2 After Dashboard Is Moved With `PBCatMove`

The second example (see Figures 3 and 4) shows a directory, Toys, renamed (by another user or process) with `PBHRename` to Games. At the time of the move, the search routine has seen the files Aardvark and Letter and is looking at the third object in the directory, the file Résumé. After the move, the index pointer is still pointing at the third object but now the third object is the file Letter, a file that has already been seen by the search. This change cannot be detected by looking at the parent directory's modification date because `PBHRename` does not change any modification dates. However, this change can be detected by checking to see if the index pointer still points to the same file or directory. The search routine could re-index through the directory to find the Résumé file again and start searching from there, but what about the directory that was renamed? The search routine either must miss it (and its contents) or it must repeat the search of the entire directory to ensure nothing is missed.

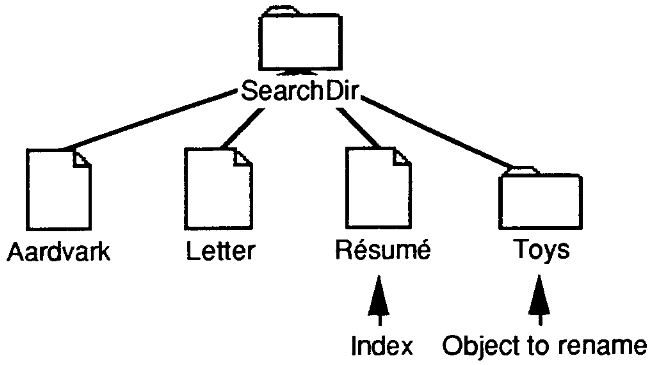


Figure 3 Before Toys Is Renamed With `PBHRename`

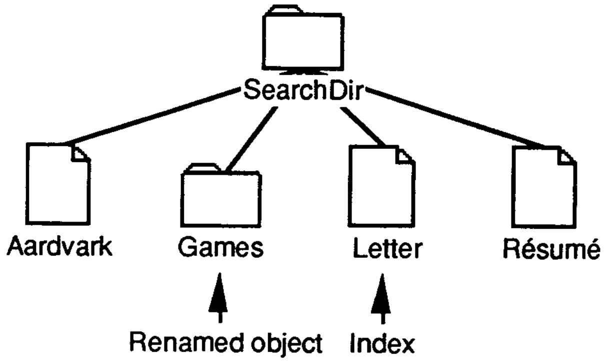


Figure 4 After Toys Is Renamed to Games With `PBHRename`

As these examples show, a change during a search of a hierarchical directory structure with indexed File Manager calls involves the risk of missing catalog entries or finding duplicate catalog entries. If your application depends on seeing all items on a volume at least once and only once, you should make the users of your application aware of the problems associated with indexed searches and suggest to them ways to make sure the volume's catalog is not changed during the indexed search. Here's a good suggestion you could make to the user: do not use other programs during the search. Other programs may create, delete, or rename files during the search.

Conclusion

You should always use `PBCatSearch` to search a volume if it is available. If `PBCatSearch` isn't available and you must use an indexed search, be aware that it is difficult to ensure that you do not miss some catalog entries or see some catalog entries multiple times during your search.

Further Reference:

- *Inside Macintosh,* Volume IV, The File Manager
- *Inside Macintosh,* Volume V, File Manager Extensions in a Shared Environment
- *Inside Macintosh,* Volume VI, The Finder Interface
- *Inside Macintosh,* Volume VI, The File Manager
- Technical Note #66, Determining Which File System Is Active
- Technical Note #305, PBShare, PBUnshare, and PBGetUGEntry

#69: Setting ioFDirIndex in PBGetCatInfo Calls

See also: The File Manager
Technical Note #24—Available Volumes and Files
Technical Note #67—Finding the Blessed Folder

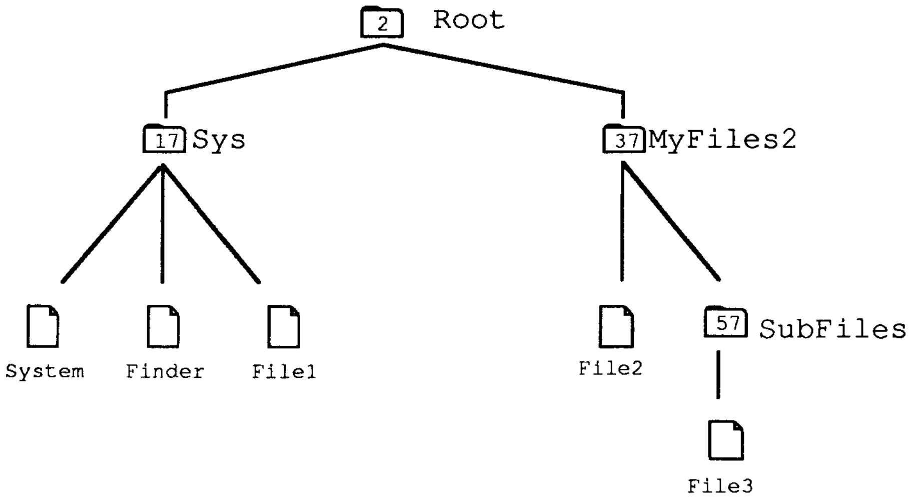
Written by: Jim Friedlander February 15, 1986
Updated: March 1, 1988

This technical note describes how to set `ioFDirIndex` for `PBGetCatInfo`.

The File Manager chapter of *Inside Macintosh* volume IV is not very specific in describing how to use `ioFDirIndex` when calling `PBGetCatInfo`. It correctly says that `ioFDirIndex` should be positive if you are making indexed calls to `PBGetCatInfo` (analogous to making indexed calls to `PBGetVInfo` as described in Technical Note #24). However, the statement "If `ioFDirIndex` is negative or 0, the File Manager returns information about the file having the name in `ioNamePtr`..." is not specific enough.

If `ioFDirIndex` is 0, you will get information about files or directories, depending on what is specified by `ioNamePtr^`.

If `ioFDirIndex` is –1, you will get information about directories only. The name in `ioNamePtr^` is ignored. For example, given the following tree structure (with sample `DirIDs` for the directories):



Calling PBGetCatInfo

We will now make calls to PBGetCatInfo of the form:

```
err:= PBGetCatInfo(@myCInfoPBRec,FALSE);
```

Note: We will assume that we just have a WDRefnum and a file name—the information that SFGetFile returns.

Setting up the parameter block

We will use the following fields in the parameter block. Before the call, ioCompletion will always be set to NIL, ioNamePtr will always point at a str255, ioVRefNum will always contain a WDRefNum that references the directory 'SubFiles', and offset 48 (dirID/flNum) will always contain a zero:

Offset in parameter block	Variable name(s)
12	ioCompletion
18	ioNamePtr
22	ioVRefNum
28	ioFDirIndex
48	ioDirID/ioFLNum/ioDrDirID
100	ioDrParID/ioFlParID

Sample calls to PBGetCatInfo

The first example will call PBGetCatInfo for the file 'File3'—we will get information about the file (ioFDirIndex = 0):

Before the call		**After the call**	
ioNamePtr^:	'File3'	ioNamePtr^:	'File3'
ioFDirIndex:	0	Offset 48(ioFLNum):	a file number
		Offset 100(parID):	57

Now we will get information about the directory that is specified by the iovRefNum (ioFDirIndex = –1). Notice that ioNamePtr^ is ignored:

Before the call		**After the call**	
ioNamePtr^:	ignored	ioNamePtr^:	'SubFiles'
ioFDirIndex:	–1	Offset 48(dirID):	57
		Offset 100(parID):	37

Notice that, since ioNamePtr^ is ignored, Offset 48 contains the dirID of the directory specified by the iovRefNum that we passed in and that Offset 100 contains the parent ID of that directory.
Notice that if we try to get information about the directory SubFiles by calling PBGetCatInfo with ioFDirIndex set to 0, we will get an error –43 (File not found error) back because there is neither a file nor a directory with the name 'SubFiles' in the directory that ioVRefNum refers to.

If you specify a **full** pathname in ioNamePtr^, then the call returns information about that path, whether it is a directory or a file. The ioVRefNum is ignored:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys'	ioNamePtr^:	'Root:Sys'
ioFDirIndex:	0	Offset 48 (dirID):	17
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	2

Or, if the full pathname specifies a file, the iovRefNum is overridden:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys:Finder'	ioNamePtr^:	'Root:Sys:Finder'
ioFDirIndex:	0	Offset 48 (flNum):	fileNumber
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	17

Or, given an ioVRefNum that refers to MyFiles2 and a partial pathname in ioNamePtr^, we'll get information about the directory 'SubFiles':

Before the call		**After the call**	
ioNamePtr^:	'SubFiles'	ioNamePtr^:	'SubFiles'
ioFDirIndex:	0	Offset 48 (dirID):	57
ioVRefNum:	refers to 'MyFiles2'	Offset 100 (parID):	37

PBGetCatInfo and The Poor Man's Search Path (PMSP)

If no ioDirID is specified (ioDirID is set to zero), calls to PBGetCatInfo will return information about a file in the specified directory, but, if no such file is found, will continue searching down the Poor Man's Search Path. **Note:** the PMSP is not used if ioFDirIndex is non-zero (either –1 or >0). The default PMSP includes the directory specified by ioVRefNum (or, if ioVRefNum is 0, the default directory) and the directory that contains the System File and the Finder—the blessed folder. So for example:

Before the call		**After the call**	
ioNamePtr^:	'System'	ioNamePtr^:	'System'
ioFDirIndex:	0	Offset 48 (ioFLNum):	a file number
		Offset 100 (parID):	17

You must be careful when using PBGetCatInfo in this way to make sure that the file you're getting information about is in the directory that you think it is, and not in a directory further down the Poor Man's Search Path. Of course, this does not present a problem if you are using the fName and the vRefNum that SFGetFile returns.

If you want to specifically look at a file in the blessed folder, please use the technique described in technical note #67 to get the `dirID` of the 'blessed folder' and then use that `dirID` as input in the `ioDirID` field of the parameter block (offset 48).

Summary (`DirID` = 0 in all the following):

If `ioFDirIndex` is set to 0:

1) Information will be returned about files.
2) Information will be returned about directories as follows:
    - A) If a partial pathname is specified by `ioNamePtr^` then the volume and directory will be taken from `ioVRefNum`.
    - B) If a full pathname is specified by `ioNamePtr^`. In this case, `ioVRefNum` is ignored.

If `ioFDirIndex` is set to –1:

1) Only information about directories will be returned.
2) The name pointed to by `ioNamePtr` is ignored.
3) If `DirID` and `ioVRefNum` are 0, you'll get information about the default directory.

Macintosh Technical Notes

#70: Forcing Disks to be Either 400K or 800K

See also: The Disk Driver
The Disk Initialization Package

Written by: Rick Blair February 13, 1986
Updated: March 1, 1988

This document explains how to initialize a disk as either single- or double-sided. It only applies to 800K drives, of course.

You can call the disk driver to initialize a disk and determine programmatically whether it should be initialized as single- (MFS) or double- (HFS) sided. All you have to do is call the `.Sony` driver directly to do the formatting then the Disk Initialization Package to write the directory information.

Note: This is not the way you should normally format disks within an application. If the user puts in an unformatted disk, you should let her or him decide whether it becomes single- or double-sided via the Disk Initialization dialog. This automatically happens when you call DIBadMount or the user inserts a disk while in Standard File. The intent of this technical note is to provide a means for specific applications to produce, say, 400K disks. An example might be a production disk copying program.

From MPW Pascal:

```
VAR
  error:      OSErr;
  IPtr:       ^INTEGER;
  paramBlock: ParamBlockRec; {needs OSIntf}
...
WITH paramBlock DO BEGIN
   ioRefNum := -5;                             {.Sony driver}
   ioVRefNum := 1;                             {drive number}
   csCode := 6;                                {format control code}
   IPtr:=@csParam;                             {pretend it's an INTEGER}
   IPtr^:=1;                                   {number of sides}
END;
error:=PBControl(@paramBlock, FALSE);          {do the call}
IF error=ControlErr THEN
{you are under MFS, which doesn't support control code 6, but it}
{would always get formatted single-sided anyway.}
{other errors are possible: ioErr, etc.}
END;
```

From MPW C:

```
OSErr             error;
CntrlParam        paramBlock;


paramBlock.ioCRefNum = -5;    /*.Sony driver*/
paramBlock.ioVRefNum = 1;     /*drive number*/
paramBlock.csCode = 6;        /*format control code*/
paramBlock.csParam[0]=1;      /*for single sided,2 for double-sided*/

error=PBControl(&paramBlock, false);/*do the call*/
if (error==controlErr) ;
/*you are under MFS, which doesn't support control code 6, but it*/
/*would always get formatted single-sided anyway.*/
/*other errors are possible: ioErr, etc.*/
```

You then call DIZero to write a standard (MFS or HFS) directory. It will produce MFS if you formatted it single-sided, and HFS if you formatted double-sided.

Macintosh Technical Notes

#71: Finding Drivers in the Unit Table

See also: The Device Manager

Written by: Rick Blair February 4, 1986
Updated: March 1, 1988

This note will explain how code can be written to determine the reference number of a previously installed driver when only the name is known. **Changes since 2/86:** Since the driver can be purged and the DCE still be allocated, the code now tests for dCtlDriver being NIL as well.

You should already be familiar with The Device Manager chapter of *Inside Macintosh* before reading this technical note.

The Pascal code at the end of this note demonstrates how to obtain the reference number of a driver that has been installed in the Unit Table. The reference number may then be used in subsequent calls to the Device Manager such as `Open`, `Control` and `Prime`.

One thing to note is that the `dRAMBased` bit really only tells you whether `dCtlDriver` is a pointer or a handle, not necessarily whether the driver is in ROM or RAM. SCSI drivers, for instance, are in RAM but not relocatable; their DCE entries contain pointers to them.

From MPW Pascal:

```
PROCEDURE GetDrvrRefNum(driverName: Str255; VAR drvrRefNum: INTEGER);

  TYPE
    WordPtr      = ^INTEGER;

  CONST
    UTableBase   = $11C;     {low memory globals}
    UnitNtryCnt  = $1D2;

    dRAMBased    = 6;        {bit in dCtlFlags that indicates ROM/RAM}
    drvrName     = $12;      {length byte and name of driver [string]}

  VAR
    negCount     : INTEGER;
    DCEH         : DCtlHandle;
    drivePtr     : Ptr;
    s            : Str255;
    BEGIN
      UprString(driverName, FALSE); {force same case for compare}

      negCount := - WordPtr(UnitNtryCnt)^; {get -(table size)}

      {Check to see that driver is installed, obtain refNum.}
      {Assumes that an Open was done previously -- probably by an INIT.}
      {Driver doesn't have to be open now, though.}

      drvrRefNum := - 12 + 1;  {we'll start with driver refnum = -12,
                                right after .ATP entry}

      {Look through unit table until we find the driver or reach the end.}

      REPEAT
        drvrRefNum := drvrRefNum - 1; {bump to next refnum}
        DCEH := GetDCtlEntry(drvrRefNum); {get handle to DCE}

        s := '';                 {no driver, no name}

        IF DCEH <> NIL THEN
          WITH DCEH^^ DO BEGIN {this is safe -- no chance of heap moving
                                 before dCtlFlags/dCtlDriver references}
            IF (dCtlDriver <> NIL) THEN BEGIN
              IF BTST(dCtlFlags, dRAMBased) THEN
                drivePtr := Handle(dCtlDriver)^ {zee deréference}
              ELSE
                drivePtr := Ptr(dCtlDriver);

              IF drivePtr <> NIL THEN BEGIN
                s := StringPtr(ORD4(drivePtr) + drvrName)^;
                UprString(s,FALSE); {force same case for compare}
              END;
            END;             {IF}
          END;               {WITH}
      UNTIL (s = driverName) OR (drvrRefNum = negCount);

      {Loop until we find it or we've just looked at the last slot.}

      IF s <> driverName THEN drvrRefNum := 0; {can't find driver}
    END;
```

From MPW C:

```
short       GetDrvrRefNum(driverName)
char        *driverName[256];

{  /* GetDrvrRefNum */

    #define           UnitNtryCnt 0x1d2

    /*bit in dCtlFlags that indicates ROM/RAM*/
    #define           dRAMBased     6
    /*length byte and name of driver [string]*/
    #define           drvrName      0x12
    short          negCount,dRef;
    DCtlHandle     DCEH;
    char           *drivePtr,*s;

    negCount = -*(short *)(UnitNtryCnt); /*get -(table size)*/

    /*Check to see that driver is installed, obtain refNum.*/
    /*Assumes that an Open was done previously -- probably by an INIT.*/
    /*Driver doesn't have to be open now, though.*/

    dRef = -12 + 1;  /*we'll start with driver refnum == -12,
                       right after .ATP entry*/

    /*Look through unit table until we find the driver or reach the
    end.*/

    do
    {
        dRef -= 1; /*bump to next refnum*/
        DCEH = GetDCtlEntry(dRef); /*get handle to DCE*/

        s = "";

        if ((DCEH != nil) && ( (**DCEH).dCtlDriver != nil) )
        {
            if (((**DCEH).dCtlFlags >> dRAMBased) & 1)
                                              /* test dRamBased bit */
                drivePtr = *(Handle) (**DCEH).dCtlDriver;
                                              /*zee deréference*/
            else
                drivePtr = (**DCEH).dCtlDriver;

            if (drivePtr != nil)
                s = drivePtr + drvrName;
        }
    } while (EqualString(s,driverName,0,0) && (dRef != negCount));
    /*Loop until we find it or we've just looked at the last slot.*/

    if (EqualString(s,driverName,0,0))
        return dRef;
    else
        return 0; /*can't find driver*/
}/* GetDrvrRefNum */
```

That's all there is to locating a driver and picking up the reference number.

#72: Optimizing For The LaserWriter—Techniques

Revised by: Pete "Luke" Alexander October 1990
Written by: Ginger Jernigan February 1986

This Technical Note discusses techniques for optimizing code for printing on the LaserWriter. **Changes since March 1988:** Updated the "Printable Paper Area" and "Memory Considerations" sections as well as the printer IDs, moved the error messages from the end of the Note to Technical Note #161, A Printing Loop That Cares..., and removed the "Spool-A-Page/Print-A-Page" section because Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, already thoroughly covers this topic.

Introduction

Although the Printing Manager was originally designed to allow application code to be printer independent, there are some things about the LaserWriter that, in some cases, have to be addressed in a printer dependent way. This Note describes what the LaserWriter can and cannot do, memory considerations, speed considerations, as well as other things you need to watch out for if you want to make your printing more efficient on the LaserWriter.

How To Determine The Currently Selected Printer

With the addition of new picture comments and the `PrGeneral` procedure, an application should never need to know the type of device to which it is connected. However, some developers feel their application should be able to take advantage of all of the features provided by a particular device, not just those provided by the Printing Manager, and in doing so, these developers produce device-dependent applications, which can produce unpredictable results third-party and new Apple printing devices. For this reason, Apple strongly recommends that you use only the features provided by the Printing Manager, and do not try to use unsupported device features.

Even though there is no supported method for determining a device's type, there is one method described in the original *Inside Macintosh* that still works for ImageWriter and LaserWriter printer drivers. This method is **not** supported, meaning that at some point in the future it will no longer work. If you use this method in your application, it is up to you to weigh the value of the feature against the compatibility risk. The following method works for all ImageWriter, ImageWriter II, and LaserWriter (original, Plus, IINT, IINTX) drivers. Since all new devices released from Apple and third-party developers have their own unique ID, it is up to you to decide what to do with an ID that your application does not recognize.

If you are using the high-level Printing Manager interface, first call `PrValidate` to make sure you have the correct print record. Look at the high byte of the `wdev` word in the `TPrStl` subrecord of the print record. Note that if you have your own driver and want to have your own number, please let DTS know, and DTS can register it.

Following is the current list of printer IDs:

Printer	wDev
ImageWriter I, ImageWriter II	1
LaserWriter, LaserWriter Plus, LaserWriter IINT, LaserWriter IINTX, and Personal LaserWriter NT	3
LaserWriter IISC, Personal LaserWriter SC	4
ImageWriter LQ	5

If you are using the low-level Printing Manager interface, there is no dependable way of getting the wDev information. You should **not** attempt to determine the device ID when using the low-level Printing Manager interface.

Using QuickDraw With the LaserWriter

When you print to the LaserWriter, all of the QuickDraw calls you make are translated (via QuickDraw bottlenecks) into PostScript®, which is in the LaserWriter ROM. Most of the operations available in QuickDraw are available in PostScript, with a few exceptions. The LaserWriter driver does not support the following:

- `XOR` and `NotXOR` transfer modes.
- The grafverb `invert`.
- `_SetOrigin` calls within `PrOpenPage` and `PrClosePage` calls. Use `_OffsetRect` instead. (This is fixed in version 3.0 and later of the driver.)
- Regions are ignored. You can simulate regions using polygons or bitmaps. Refer to Technical Note #41, Drawing Into An Off-Screen Bitmap, for how to create off-screen bitmaps.
- Clip regions should be limited to rectangles.
- There is a small difference in character widths between screen fonts and printer fonts. Only the end points of text strings are the same.

What You See Is Not Always What You Get

Unfortunately, what you see on the screen is not always what you get. If you are using standard graphic objects, like rectangles, circles, etc., the object is the same size on the LaserWriter as it is on the screen. There are, however, two types of objects where this is not the case: text and bitmaps.

The earlier noted difference between the widths of characters on the screen and the widths of characters on the printer is due to the difference in resolution. However, to maintain the integrity of line breaks, the driver changes the word and character spacing to maintain the end points of the lines as specified. What this all means is that you cannot count on the positions or the widths of printed characters being exactly the same as they are on the screen. This is why in the original MacDraw®, for example, if one carefully places text and a rectangle and prints it, the text sometimes extends beyond the bounds of the rectangle on the printed page. If an application does its own line layout (i.e., positions the words on the line itself), then it may want to disable the LaserWriter's line layout routines. To disable these routines, use the `LineLayoutOff` picture comment described in the LaserWriter Reference Manual and Technical Note #91, Optimizing for the LaserWriter—Picture Comments.

The sole exception to this rule is if an application is running on 128K ROMs or later. The 128K ROM Font Manager supports the specification of fractional pixel widths for screen fonts, increasing the screen to printer accuracy. This fractional width feature is disabled by default. To enable it, an application can use `_SetFractEnable`, after calling `_InitFonts`.

Applications can use picture comments to left-, right-, or center-justify text. Only the left, right, or center end points are accurate. If the text is fully justified, both end points are accurate. Technical Note #91, Optimizing for the LaserWriter—Picture Comments, discusses these picture comments.

Memory Considerations

To print to the LaserWriter, you need to make sure that you have enough memory available to load the driver's code. The best way to do this is to have all the code you need for printing in a separate segment and unload everything else. When you print to the LaserWriter you are only able to print in Draft mode. You are not able to spool (as the ImageWriter does in the standard or high-quality settings), and your print code, data, and the driver code have to be resident in memory.

In terms of memory requirements, there is not any magic number that always works with all printer drivers (including third-party printer drivers) that are available for the Macintosh. To make sure there is enough memory available during print time, you should make your printing code a separate segment and swap out all unwanted code and data before you call `_PrOpen`.

Printable Paper Area

On the LaserWriter there is a 0.45-inch border that surrounds the printable area of the paper (this is assuming an 8.5" x 11" paper). If you select the "Larger Print Area" option in the Page Setup dialog box, the border changes to 0.25 of an inch. This printable area is different than the available print area of the ImageWriter. An application cannot print a larger area because of the memory PostScript needs to image a page. PostScript takes the amount of memory available in the printer and centers it on the paper, and there is not enough RAM in the LaserWriter to image an entire sheet of paper.

Page Sizes

Many developers have expressed a desire to support page sizes other than those provided by the Apple printer drivers. Even though some devices can physically support other page sizes, there is no way for an application to tell the driver to use this size. With the ImageWriter driver, it is possible to modify certain fields in the print record and expand the printable area of the page. However, each of the Apple drivers implements the page sizes in a different way. No one method works for all drivers. Because of this difference, it is strongly recommended that applications do not attempt to change the page sizes provided in the "Style" dialog box. If your application currently supports page sizes other than those provided by the printer driver, you are taking a serious compatibility risk with future Apple and third-party printer drivers.

Speed Considerations

Although the LaserWriter is relatively fast, there are some techniques an application can use to ensure its maximum performance.

- Try to avoid using the QuickDraw Erase calls (e.g., _EraseRect, _EraseOval, etc.). It takes a lot of time to handle the erase function because every bit (90,000 bits per square inch) has to be cleared. Erasing is unnecessary because the paper does not need to be erased the way the screen does.

- Printing patterns takes time, since the bitmap for the pattern has to be built. The patterns black, white, and all the gray patterns have been optimized to use the PostScript gray scales. If you use a different pattern it works, but it just takes longer than usual. In addition, the patterns in driver version 3.0 are rotated; they are not rotated in version 1.0.

- Try to avoid frequently changing fonts. PostScript has to build each character it needs either by using the drawing commands for the built-in LaserWriter fonts or by resizing bitmaps downloaded from screen fonts on the Macintosh. As each character is built, it is cached (if there's room), so if that character is needed again PostScript gets if from the cache. When the font changes, the characters have to be built from scratch in the new font, which takes time. If the font is not in the LaserWriter, it takes time to download it from the Macintosh. If the user has the option of choosing fonts, you have no control over this variable; however, if you control which fonts to use, keep this in mind.

- Avoid using _TextBox. It makes calls to _EraseRect, which slows the printer, for every line of text it draws. You might want to use a different method of displaying text (e.g., _DrawString or _DrawText) or write your own version of _TextBox. If an application is currently calling _TextBox, changing to another method of displaying text can improve speed on the order of five to one.

- Because of the way rectangle intersections are determined, if your clip region falls outside of the rPage rectangle, you slow down the printer substantially. By making sure your clip region is entirely within the rPage rectangle, you can get a speed improvement of approximately four to one.

- Do not use spool-a-page/print-a-page as some applications do when printing on the ImageWriter. It slows things down considerably because of all of the preparation that has to be done when a job is initiated. Refer to Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, for more information.

- Using _DrawChar to place every character to print can take a lot of time. One reason, of course, is because it has to go through the bottlenecks for every character that is drawn. The other is that the printer driver does its best to do line layout, making the character spacing just right. If you are trying to position characters and the driver is trying to position characters too, there is conflict, and printing takes much longer than necessary. In version 3.0 of the driver, there are picture comments that turn off the line layout optimization, alleviating some of the problem. Refer to Technical Note #91, Optimizing for the LaserWriter—Picture Comments, for more information.

Clipping Within Text Strings

When clipping characters out of a string, make sure that the clipping rectangle or region is greater than the bounding box of the text you want to clip. The reason is that if you clip part of a character (e.g., a descender), the clipped character has to be rebuilt, which takes time. In addition, because of the difference between screen fonts and printer fonts, chances are that you cannot accurately clip the right characters unless you are running on the 128K ROMs and have fractional pixel widths enabled.

When to Validate the Print Record

To validate the print record, call `PrValidate`. You need validation to check to see if all of the fields are accurate according to the current printer selected and the current version of the driver. You should call `PrValidate` when you have allocated a new print record or whenever you need to access information from the print record (i.e., when you get `rPage`). The routines `PrStlDialog` and `PrJobDialog` call `PrValidate` when they are called, so you do not have to worry about it if you use these calls.

Empty QuickDraw Objects

QuickDraw objects that are empty (i.e., they have no pixels in them) and are filled but not framed, do not print on the ImageWriter and do not show up on the screen; however, on the LaserWriter they are real objects and do print.

Further Reference:

- *Inside Macintosh*, Volume I, QuickDraw
- *Inside Macintosh*, Volume II, The Printing Manager
- *LaserWriter Reference Manual*
- Technical Note #41, Drawing Into An Off-Screen Bitmap
- Technical Note #91, Optimizing for the LaserWriter—Picture Comments
- Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers
- Technical Note #161, A Printing Loop That Cares…
- PostScript Language Reference, Adobe Systems, Incorporated
- PostScript Language Tutorial and Cookbook, Adobe Systems, Incorporated

MacDraw is a registered trademark of Claris Corporation.
PostScript is a registered trademark of Adobe Systems, Incorporated.

#73: Color Printing

See also: QuickDraw
The Printing Manager
PostScript Language Reference Manual,
Adobe Systems

Written by: Ginger Jernigan February 3, 1986
Modified by: Scott "ZZ" Zimmerman January 1, 1988
Updated: March 1, 1988

This discusses color printing in a Macintosh application.

Whereas the original eight-color model of QuickDraw was sufficient for printing in color on the ImageWriter II, the introduction of Color QuickDraw has created the need for more sophisticated printing methods.

The first section describes using the eight-color QuickDraw model with the ImageWriter II and ImageWriter LQ drivers. Since the current Print Manager does not support Color GrafPorts, the eight-color model is the only method available for the ImageWriters.

The next section describes a technique that can be used for printing halftone images using PostScript (when it is available). Also described is a device independent technique for sending the PostScript data. This technique can be used on any LaserWriter driver 3.0 or later. It will work with all LaserWriters except the the LaserWriter IISC.

It is very likely that better color support will be added to the Print Manager in the future. Until then, these are the best methods available.

Part 1, ImageWriters

The ImageWriter drivers are capable of generating each of the eight standard colors defined in QuickDraw by the following constants:

```
whiteColor
blackColor
redColor
greenColor
blueColor
cyanColor
magentaColor
yellowColor
```

To generate color all you need to do is set the foreground and background colors before you begin drawing (initially they are set to blackColor foreground and whiteColor background). To do this you call the QuickDraw routines `ForeColor` and `BackColor` as described in *Inside Macintosh*. If you are using QuickDraw pictures, make sure you set the foreground and background colors before you call `ClosePicture` so that they are recorded in the picture. Setting the colors before calling `DrawPicture` doesn't work.

The drivers also recognize two of the transfer modes: `srcCopy` and `srcOr`. The effect of the other transfer modes is not well defined and has not been tested. It may be best to stay away from them.

Caveats

When printing a large area of more than one color you will encounter a problem with the ribbon. When you print a large area of one color, the printer's pins pick up the color from the back of the ribbon. When another large area of color is printed, the pins deposit the previous color onto the back of the ribbon. Eventually the first color will come through to the front of the ribbon, contaminating the second color. You can get the same kind of effect if you set, for example, a foreground color of yellow and a background color of blue. The ribbon will pick up the blue as it tries to print yellow on top of it. This problem is partially alleviated in the 2.3 version of the ImageWriter driver by using a different printing technique.

The ribbon goes through the printer rather quickly when printing large areas. When the ribbon comes through the second time the colors don't look too great.

Part 2, LaserWriters

Using the PostScript 'image' Operator to Print Halftones

About 'image'

The PostScript image operator is used to send Bitmaps or Pixmaps to the LaserWriter. The image operator can handle depths from 1 to 8 bits per pixel. Our current LaserWriters can only image about twenty shades of gray, but the printed page will look like there's more. Being that the image operator is still a PostScript operator, it expects its data in the form of hexidecimal bytes. The bytes are represented by two ASCII characters(0-9,A-F). The image operator takes these parameters:

```
width  height  depth  matrix  image-data
```

The first three are the width, height, and depth of the image, and the matrix is the transformation matrix to be applied to the current matrix. See the *PostScript Language Reference Manual* for more information. The image data is where the actual hex data should go. Instead of inserting the data between the first parameters and the image operator itself, it is better to use a small, PostScript procedure to read the data starting from right after the image operator. For example:

```
640 480 8 [640 0 0 480 0 0]
{currentfile picstr readhexstring pop}
image
FF 00 FF 00 FF 00 FF 00 ...
```

In the above example, the width of the image is 640, the height is 480, and the depth is 8. The matrix (enclosed in brackets) is setup to draw the image starting at QuickDraw's 0,0 (top left of page), and with no scaling. The PostScript code (enclosed in braces) is not executed. Instead, it is passed to the image operator, and the image operator will call it repeatedly until it has enough data to draw the image. In this case, it will be expecting 640*480 bytes. When the image operator calls the procedure, it does the following:

1. Pushes the current file which in this case is the stream of data coming to the LaserWriter over AppleTalk. This is the first parameter to readhexstring.

2. Next picstr is pushed. picstr is a string variable defined to hold one row of hex data. The PostScript to create the picstr is:

   ```
   /picstr 640 def
   ```

3. Now readhexstring is called to fill picstr with data from the current file. It begins reading bytes which are the characters following the image operator.

4. Since readhexstring leaves both the string we want, and a boolean that we don't want on the stack, we do one pop to kill of the boolean. Now the string is left behind for the image operator to use.

So using the above PostScript code you can easily print an image. Just fill in the width height and depth, and send the hex data immediately following the PostScript code.

Setting Up for 'image'

Most of the users of this technique are going to want to print a Color QuickDraw PixMap. Although the image command does a lot of the work for you, there are still a couple of tricks that are recommended for performance.

Assume the Maximum Depth

Since the current version of the image operator has a maximum depth of 8 bits/pixel, it is wise to convert the source image to the same depth before imaging. This can be done very simply by using an offscreen GrafPort that is set to 8 bits/pixel, and then using CopyBits to do the depth conversion for you. This will do a nice job of converting lower resolution images to 8 bits/pixel.

Build a Color Table

An 8 bit deep image can only use 256 colors. Since the image that you are starting with is probably color, and the image you get will be grayscale, you need to convert the colors in the source color table into PostScript grayscale values. This is actually easy to do using the Color Manager. First create a table that can hold 512 bytes. This is 2 bytes for each color value from 0 to 255. Since PostScript wants the values in ASCII, you need two characters for each pixel. Now loop through the colors in the color table. Call Index2Color to get the real RGB color for that index, and then call RGB2HSL to convert the RGB color into a luminance value. This value will be expressed as a SmallFract which can then be scaled into a value from 0 to 255. This value should then be converted to ASCII, and stored at the appropriate location in the table. When you are done, you should be able to use a pixel value as an index into your table of PostScript color values. For each pixel in the image, send two characters to the LaserWriter.

Sending the Data

Once you have set up the color table, all that left to do is to loop through all of the pixels, and send their PostScript representation to the LaserWriter. There are a couple of ways to do this. First is to use the low-level Print Manager interface and stream the PostScript using the stdBuf PrCtlCall. Although this seems like it would be the fastest way, the latest version of the LaserWriter driver (5.0) converts all low-level calls to their high level equivalent before executing them. Because of this, the low-level interface is no longer faster than the high level. In an FKEY I have written, I use the high-level Print Manager interface, and send the data via the PostScriptHandle PicComment. This way, I can buffer a large amount of data, before actually sending it. Using this technique, I have been able to image a Mac II screen in about 5 minutes on a LaserWriter Plus, and about 1.5 minutes on a LaserWriter II NTX.

#74: Don't Use the Resource Fork for Data

See also: The Resource Manager
Technical Note #62—Resource Header Application Bytes

Written by: Bryan Stearns March 13, 1986
Updated: March 1, 1988

Don't use the resource fork of a file for non-resource data. Parts of the system (including the File Manager and the Finder) assume that if this fork exists, it will contain valid Resource Manager information.

`PBOpenRF` was provided to allow copying of the resource fork of a file in its entirety, without Resource Manager interpretation. Do not use it to open "another data fork."

The File Manager assumes that the first block of the resource fork of a file will be part of the resource header, and puts information there to aid in scavenging. Note that this means that if you copy a resource file (opened with `PBOpenRF`), the duplicate may not be exactly like the original.

#75: Apple's Multidisk Installer

Revised by: Rich Kubota — January 1992
Written by: scott douglass — March 1986

This Technical Note documents Apple's Multidisk Installer, and it is in addition to separate Installer documentation which provides the details of writing scripts.
Changes since September 1991: Revised information on the use of Installer version 3.1 to version 3.2. Revised information on the use of ScriptCheck version 3.2.1 with Installer version 3.2. Added Common Questions and Answers relating to the use of the Installer.

Apple's Multidisk Installer is intended to make it easy for Macintosh users to add or update software. It is a very useful tool for adding third-party software, and Apple recommends that you use the Installer unless your software installation is simple. Apple also recommends that you use version 3.2 of the Installer.

The Multidisk Installer has the following features, as of version 3.2:

- "Easy Install" mode where the Installer script writer can determine the appropriate installation based upon examination of the target environment and provide the user with "One-Button Installation"
- An optional "Custom Install" mode where power users can customize their installation
- "Live" installation to the currently booted and active system; thus it is no longer necessary to ship the Installer on a bootable disk with a System Folder
- Ability to install from an AppleShare server ("Network Install")
- Ability to install from multiple source disks
- Installation of software to folders other than the System Folder as well as creation of new folders as necessary
- Runs under System 4.2 and later versions
- "User Function" support; this feature provides linkage to developer-defined code segments during Easy Install, so script writers can customize the process of determining what software to install and how to install it
- "Action Atom" support; this feature provides linkage to developer-defined code segments that are called before or after the installation takes place; script writers can use this feature to extend the capabilities of the Installer
- Audit Records; this feature provides the script writer with the ability to record details about an installation so that future installations can be more intelligent

The `'indm'` (default map) resource of Installer 3.0.1 is no longer supported in Installer 3.1 and later versions. This was used by script writers to implement Easy Install. It is replaced by `'infr'` (framework) and `'inrl'` (rule) resources.

Note: If the user opens the Installer document rather than the Installer, the wrong Installer may be launched (depending upon the contents of their mounted volumes). This is only a problem between versions 3.1 and 3.0.x. If you are developing a 3.1 script, you may want to add an 'indm' resource that puts up a warning dialog box. If you are developing a 3.0.x script, you may want to add an 'infr' and 'inrl' resource that puts up a reportSysError dialog box. This problem is resolved in Installer 3.2. With version 3.2, the file type and creator are both 'bjbc' as opposed to the use of 'cfbj' with versions 3.0.1 and 3.1.

Installer version 3.2 is available as a complete reference suite which includes the following:

- *Installer 3.2 Scripting Guide* (dated December 1, 1991, on the cover)
- *Installer ScriptCheck 3.2b7 User's Manual*
- Installer 3.2 application
- ScriptCheck 3.2.1 (MPW Tool)
- InstallerTypes.r (MPW Rez interface file)
- ActionAtomIntf.a, .h, .p (Action Atom interface files for Assembler, C, and Pascal)

The reference suite for Multidisk Installer 3.2 is available on the latest Developer CD and on AppleLink in the Developer Services Bulletin Board. The Multidisk Installer was also provided on the System 7 Golden Master CD-ROM: however, that package included the b7 release of the MPW ScriptCheck tool.

Multidisk Installer version 3.2 contains a few minor improvements that will make it easier to write scripts that work on both System 6.0.x and 7.0. Installer 3.1 had minimal testing with System 7.0. If you are expecting to install software onto machines running System 7.0, you should consider upgrading. Script changes should be minimal.

Common Question and Answers

Q How can I check for a minimum system version?

A Use the checkFileVersion clause as part of the 'inrl' Rules Framework resource. The format of the minimal-version parameter is shown in the InstallerTypes.r file as '#define Version'. The most common difficulties are in remembering that BCD values are required and in dealing with two-digit version numbers. Some samples follow.

Assuming that the target-filespec resource, 'infs', for the System file is 1000, use the following clause to check for System version 6.0.5:

```
checkFileVersion{1000, 6, 5, release, 0};
```

Assuming that the target-filespec resource, 'infs', for the Finder file is 1001, use the following clause to check for Finder version 6.1.5:

```
checkFileVersion{1001, 6, 0x15, release, 0};
```

Assuming that the target-filespec resource, 'infs', for the AppleTalk resource file is 1002, use the following clause to check for AppleTalk version 53:

```
checkFileVersion{1002, 0x53, 0, release, 0};
```

Q My Installer script installs a desk accessory. Under System 6, each time I run the script, a new copy of the DA appears as a DRVR resource in the System file. Why?

A Unfortunately, this is a symptom when the 'deleteWhenInstalling' flag is used in conjunction with the 'updateExisting' flag. The *Installer 3.1 & 3.2 Scripting Guide* indicates that resources marked with the 'dontDeleteWhenInstalling' flag can be replaced with a new resource. The guide also indicates that the Installer will overwrite a preexisting resource in the target file if the `'updateExisting'` flag is set. Given these two flag settings, and the use of the replace `'byName'` (noByID) flag, the Installer does not delete the DA. Instead a new DRVR resource is created with the same name but a new resource ID.

The correct Installer action is accomplished by setting the `'deleteWhenInstalling'` flag in conjunction with the `'updateExisting'` flag. Alternatively, use the `'dontDeleteWhenInstalling'` flag with the `'keepExisting'` flag.

Q How can I include the current volume name in a `reportVolError` alert as many of the installation scripts from Apple do?

A The volume name can be included by inserting "^0" in the desired location of the Pascal string passed to the `reportVolError` error reporting clause.

Q. I set the `searchForFile` flag in my '`infs`' resource, however, the Installer acts as if it's unable to find the file. Why?

A. The likely reason for this problem is that the desired file is within a folder by the same name. When the `searchForFile` flag is set, the Installer will also find a match on a folder. The Installer will not replace a folder with a file, nor will it add a resource to a folder. The Installer continues as if the search failed.

Q What is the `'incd'` resource about?

A When the MPW ScriptCheck tool is used, it reads the script's file creation date/time stamp and converts it into a long word with the `Date2Secs` procedure. ScriptCheck stores this long word in the `'incd'` resource for use with verifying files when a network installation is performed. See the following questions for a discussion of this resource.

Q What checks are made by the Installer when preflighting an installation? Occasionally the alert "Could not find a required file . . ." occurs and the installation is aborted.

A The Installer compiles a list of the source file specifications from each of the resource `'inra'` and file `'infa'` atoms specified among the package `'inpk'` atoms included for installation. Each source file specification includes a complete path name. As each source file is accessed, a check is made of the file's creation date/time stamp with the date/time stamp recorded in the corresponding `'infs'` resource. If the date/time stamps do not match, the alert results and the installation is aborted. The creation date/time stamp in the `'infs'` resource can be

- entered manually into the script file so long as the value is not 1 or 0,
- filled in by ScriptCheck automatically, if a value of 1 is entered in the date field,
- forced to be updated, if the -d switch is used with ScriptCheck.

Q What are some of the considerations when configuring a network installation setup?

A Under Installer 3.1/3.2, network software installations are made possible by setting up an installation folder on the server volume. This folder will contain the Installer application, the Script file, and a folder(s) matching the names of the required disk(s). Within the disk folder(s) are the corresponding contents of the disk(s).

A problem can occur when a workstation is used to create the server installation folder and the system date and time differ significantly between the two systems. Under such condition, files copied from the workstation to the server may have their creation and modification date time/stamps altered. If a modification is made, the “delta” is constant for both the creation and modification date/time stamp and for all files copied at that time.

As indicated in the previous question, the Installer preflights a file by comparing its creation date/time stamp with the value stored in the corresponding 'infs' resource in the script file. To compensate for the fact that a server may alter a file’s creation date/time stamp, the Installer implements the `'incd'` resource. After the user selects the Install button, the Installer reads the 'incd' resource and compares it with the script file’s creation date/time stamp. The difference is stored as the “delta.” On a normal disk installation, the “delta” is always zero. As the Installer finds each required source file, the file’s creation date/time stamp is converted to a long word and adjusted by the “delta.” The modified date/time stamp is then compared with that stored in the script file. If the values match, the file is considered found and the installation proceeds. On network installations, the delta may be nonzero. If so, it indicates that the file’s creation date/time stamps were modified when copied to the server. Thus the 'incd' resource gives the Installer a way to maintain file verification even though the date/time stamp may be altered.

A specific problem can occur when an installation is set up on some systems running older versions of Novell Server software. Under specific conditions, files copied to some Novell servers have their creation time stamp altered to 12:00 A.M. regardless of the original time stamp. This includes the creation time stamp of the script file. This condition wreaks havoc with the Installer's preflight mechanism. The “delta” determined between the `'incd'` resource and the Script file’s creation date/time stamp may not be consistent with the creation date/time stamp stored in the infs resource and the corresponding file’s time stamp now at 12:00 A.M.

A work-around solution for this problem is to set the Creation time stamp for all files on the installation disk to 12:00 A.M. , BEFORE running the ScriptCheck tool. Use the MPW tool SetFile to perform this function. Here's a sample MPW script for performing this function:

```
SetFile   -d "1/1/92 12:00AM" `files -r -s -f ≈`
```

This script assumes that the current directory is set to the root of the Installation disk. For multiple disks, run this script on each disk. Use the '-f' switch with ScriptCheck to ensure that the date/time stamps are updated on scripts previously checked.

Installation of software is a nontrivial process. Apple recommends that you reserve time for development and testing to ensure that the installation process proceeds smoothly on all target machine configurations.

To ship the Installer with your product you must contact Apple’s Software Licensing Department (AppleLink: SW.LICENSE) and license the Installer alone or with the system software package that includes the version of the Installer you intend to use. Software Licensing also supplies you with a copy of the Installer that you may ship.

Further Reference:

- Installer 3.2 Reference Suite

#76: The Macintosh Plus Update Installation Script

Written by: scott douglass February 24, 1986
Updated: March 1, 1988

Earlier versions of this note described the Macintosh Plus Update installation script, because it was the first script created for the Installer. Since then, many versions of this script have been created which no longer match what was described here. In addition, many other scripts now exist.

#77: HFS Ruminations

See also: The File Manager
Technical Note #66—
Determining Which File System is Active
Technical Note #67—Finding the "Blessed Folder"
Technical Note #68—
Searching All Directories on an HFS Volume

Written by: Jim Friedlander June 7, 1986
Updated: March 1, 1988

This technical note contains some thoughts concerning HFS.

HFS numbers

A drive number is a small positive word (e.g. 3).

A VRefNum (as opposed to a WDRefNum) is a small negative word (e.g. $FFFE).

A WDRefNum is a large negative word (e.g. $8033).

A DirID is a long word (e.g. 38). The root directory of an HFS volume always has a dirID of 2.

Working Directories

Normally an application doesn't need to open working directories (henceforth WDs) using PBOpenWD, since SFGetFile returns a WDRefnum if the selected file is in a directory on a hierarchical volume and you are running HFS. There are times, however, when opening a WD is desirable (see the discussion about BootDrive below).

If you do open a WD, it should be created with an ioWDProcID of 'ERIK' ($4552494B) and it will be deallocated by the Finder. Note that under MultiFinder, ioWDProcID will be ignored, so you should only use 'ERIK'.

SFGetFile also creates WDs with an ioWDProcID of 'ERIK'. If SFGetFile opens two files from the same directory (during the same application), it will only create one working directory.

There are no WDRefnums that refer to the root—the root directory of a volume is always referred to by a vRefNum.

When you can use HFS calls

All of the HFS 'H' calls, except for PBHSetVInfo, can be made without regard to file system as long as you pass in a pointer to an HFS parameter block. PBHGetVol, PBHSetVol (see the warnings in the File Manager chapter of *Inside Macintosh*), PBHOpen, PBHOpenRF, PBHCreate, PBHDelete, PBHGetFInfo, PBHSetFInfo, PBHSetFLock, PBHRstFLock and PBHRename differ from their MFS counterparts only in that a dirID can be passed in at offset $30.

The only difference between, for example, PBOpen and PBHOpen is that bit 9 of the trap word is set, which tells HFS to use a larger parameter block. MFS ignores this bit, so it will use the smaller parameter block (not including the dirID). Remember that all of these calls will accept a WDRefNum in the ioVRefNum field of the parameter block.

PBHGetVInfo returns more information than PBGetVInfo, so, if you're counting on getting information that **is** returned in the HFS parameter block, but not in the MFS parameter block, you should check to see which file system is active.

HFS-specific calls can only be made if HFS is active. These calls are: PBGetCatInfo, PBSetCatInfo, PBOpenWD, PBCloseWD, PBGetFCBInfo,PBGetWDInfo, PBCatMove and PBDirCreate. PBHSetVInfo has no MFS equivalent. If any of these calls are made when MFS is running, a system error will be generated. If PBCatMove or PBDirCreate are called for an MFS volume, the function will return the error code –123 (wrong volume type). If PBGetCatInfo or PBSetCatInfo are called on MFS volumes, it's just as if PBGetFInfo and PBSetFInfo were called.

Default volume

If HFS is running, a call to GetVol (before you've made any SetVol calls) will return the WDRefNum of your application's parent directory in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the WDRefNums of those documents' parent directories are available in the vRefNum field of the AppFile record returned from GetAppFiles.

If MFS is running, a call to GetVol (before you've made any SetVol calls) will return the vRefNum of the volume your application is on in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the vRefNum of those documents' volume are available in the vRefNum field of the AppFile record returned from GetAppFiles.

BootDrive

If your application or desk accessory needs to get the WDRefNum of the “blessed folder” of the boot drive (for example, you might want to store a configuration file there), it can not rely on the low-memory global BootDrive (a word at $210) to contain the correct value. If your application is the startup application, BootDrive will contain the WDRefNum of the directory/volume that your application is in (not the WDRefNum of the “blessed folder”); Your application could have been _Launched from an application that has modified BootDrive; if you are a desk accessory, you might find that some applications alter BootDrive.

To get the “real” WDRefNum of the “blessed folder” that contains the currently open System file, you should call SysEnvirons (discussed in Technical Note #129). If that is impossible, you can do something like this (**Note:** if you are running under MFS, BootDrive always contains the vRefNum of the volume on which the currently open System file is located):

```
...
CONST
        SysWDProcID   = $4552494B; {"ERIK"}
        BootDrive     = $210;      {address of Low-Mem global BootDrive}
        FSFCBLen      = $3F6;      {address of Low-Mem global to
                                    distinguish file systems }
        SysMap        = $A58;      {address of Low-Mem global that contains
                                    system map reference number}

TYPE
        WordPtr = ^Integer;                 {Pointer to a word(2 bytes)}
...

FUNCTION HFSExists: BOOLEAN;

Begin {HFSExists}
        HFSExists := WordPtr(FSFCBLen)^ > 0;
End;   {HFSExists}


FUNCTION GetRealBootDrive: INTEGER;

VAR
        MyHPB          : HParamBlockRec;
        MyWDPB         : WDPBRec;
        err            : OSErr;
        sysVRef        : integer; {will be the vRefNum of open system's vol}

Begin {GetRealBootDrive}
        if HFSExists then Begin    {If we're running under HFS... }

                {get the VRefNum of the volume that }
                {contains the open System File      }
                err:= GetVRefNum(WordPtr(SysMap)^,sysVRef);
```

```
              with MyHPB do Begin
              {Get the “System” vRefNum and “Blessed” dirID}
                     ioNamePtr    := NIL;
                     ioVRefNum    := sysVRef; {from the GetVrefNum call}
                     ioVolIndex   := 0;
              End; {with}
              err := PBHGetVInfo(@MyHPB, FALSE);

              with myWDPB do Begin      {Open a working directory there}
                     ioNamePtr   := NIL;
                     ioVRefNum   := sysVRef;
                     ioWDProcID  := SysWDProcID; {Using the system proc ID}
                     ioWDDirID  := myHPB.ioVFndrInfo[1];{ see TechNote 67}
              End; {with}
              err := PBOpenWD(@myWDPB, FALSE);

              GetRealBootDrive := myWDPB.ioVRefNum;
              {We've got the real WD}
       End Else {we're running MFS}
              GetRealBootDrive := WordPtr(BootDrive)^;
              {BootDrive is valid under MFS}
End;  {GetRealBootDrive}
```

From MPW C:

```
/*"ERIK"*/
#define     SysWDProcID 0x4552494B
#define     BootDrive   0x210
/*address of Low-Mem global that contains system map reference number*/
#define     SysMap      0xA58
#define     FSFCBLen    0x3F6
#define     HFSIsRunning ((*(short int *)(FSFCBLen)) > 0)

OSErr GetRealBootDrive(BDrive)
short int *BDrive;
{ /*GetRealBootDrive*/

      /*three different parameter blocks are used here for clarity*/
      HVolumeParam      myHPB;
      FCBPBRec          myFCBRec;
      WDPBRec           myWDPB;
      OSErr             err;
      short int         sysVRef; /*will be the vRefNum of open system's
                                    vol*/

      if (HFSIsRunning)

      { /*if we're running under HFS... */

      /*get the vRefNum of the volume that contains the open System File*/
            myFCBRec.ioNamePtr= nil;
            myFCBRec.ioVRefNum = 0;
            myFCBRec.ioRefNum = *(short int *)(SysMap);
            myFCBRec.ioFCBIndx = 0;

            err = PBGetFCBInfo(&myFCBRec,false);
            if (err != noErr) return(err);
      /*now we need the dirID of the "Blessed Folder" on this volume*/
```

```
                myHPB.ioNamePtr = nil;
                myHPB.ioVRefNum = myFCBRec.ioFCBVRefNum;
                myHPB.ioVolIndex = 0;

                err = PBHGetVInfo(&myHPB,false);
                if (err != noErr) return(err);

          /*we can now open a WD for the directory that contains the open
          system file one will most likely already be open, so PBOpenWD will
          just return that WDRefNum*/
                myWDPB.ioNamePtr = nil;
                myWDPB.ioVRefNum = myHPB.ioVRefNum;
                myWDPB.ioWDProcID = SysWDProcID; /*'ERIK'*/
                myWDPB.ioWDDirID = myHPB.ioVFndrInfo[0]; /* see Technote # 67
                                                     [c has 0-based arrays]*/

                err = PBOpenWD(&myWDPB,false);
                if (err != noErr) return err;

                *BDrive = myWDPB.ioVRefNum; /*that's all!*/
          } /* if (HFSIsRunning) */
          else
                *BDrive = *(short int *)(BootDrive);
                /*BootDrive is valid under MFS*/

          return noErr;
    }                                        /*GetRealBootDrive*/
```

The Poor Man's Search Path (PMSP)

If HFS is running, the PMSP is used for any file system call that can return a file-not-found error, such as PBOpen, PBClose, PBDelete, PBGetCatInfo, etc. It is **not** used for indexed calls (that is, where ioFDirIndex is positive) or when a file is created (PBCreate) or when a file is being moved between directories (PBCatMove). The PMSP is also **not** used when a non-zero dirID is specified.

Here's a brief description of how the default PMSP works.

1) The directory that you specify (specified by WDRefNum or pathname) is searched; if the specified file is not found, then

2) the volume/directory specified by BootDrive (low-memory global at $210) is searched IF it is on the same volume as the directory you specified (see #1 above); if the specified file is not found, or the directory specified by BootDrive is not on the same volume as the directory that you specified, then

3) if there is a "blessed folder" on the same volume as the directory you specified (see #1 above), it is searched. Please note that if #2 above specifies the same directory as #3, then that directory is **not** searched twice. If no file is found, then

4) fnfErr is returned.

ioDirId and ioFlNum

Two fields of the HParamBlockRec record share the same location. ioDirID and ioFlNum are both at offset $30 from the start of the parameter block. This causes a problem, since, in some calls (e.g. PBGetCatInfo), a dirID is passed in and a file number is returned in the same field.

Future versions of Apple's HFS interfaces will omit the ioFlNum designator, so, if you need to get the file number of a file, it will be in the ioDirID of the parameter block **after** you have made the call. If you are making successive calls that depend on ioDirID being set correctly, you must "reset" the ioDirID field before each call. The program fragment in Technical Note #68 does this.

PBHGetVInfo

Normally, PBHGetVInfo will be called specifying a vRefNum. There are times, however, when you may make the call and only specify a volume name. If this is so, there are a couple of things to look out for.

Let's say that we have two volumes mounted: "Vol1:" (the default volume) and "Vol2:". We also have a variable of type HParamBlockRec called MyHPB. We want to get information about Vol2:, so we put a pointer to a string (let's call it fName) in MyHPB.ioNamePtr. The string fName is equal to "Vol2" (Please note the missing colon). We also initialize MyHPB.ioVRefNum to 0. Then we make the call. We are very surprised to find out that we are returned an error of 0 (noErr) and that the ioVRefNum that we get back is **not** the vRefNum of Vol2:, but rather that of Vol1:.

Here's what's happening: PBHGetVInfo looks at the volume name, and sees that it is improper (it is missing a colon). So, being a forgiving sort of call, it goes on to look at the ioVRefNum field that you passed it (see pp. 99 of *Inside Macintosh*, vol. II). It sees a 0 there, so it returns information about the default volume.

If you want to get information about a volume, and you just have its name and you are not sure that the name is a proper one, you should set MyHPB.ioVRefNum to –32768 ($8000). No vRefNum or WDRefNum can be equal to $8000. By doing this, you are forcing PBHGetVInfo to use the volume name and, if that name is invalid, to return a –35 error (nsvErr), "No such volume."

PBGetWDInfo and Register D1

There was a problem with PBGetWDInfo that sometimes caused the call to inaccurately report the dirID of a directory. It is fixed in System 3.2 and later. To be absolutely sure that you won't get stung by this, clear register D1 (CLR.L D1) before a call to PBGetWDInfo. You can do this either with an INLINE (Lisa Pascal and most C's) or with a short assembly-language routine before the call to PBGetWDInfo.

Macintosh Technical Notes

#78: Resource Manager Tips

See also: The Resource Manager
The Memory Manager
The Menu Manager
Technical Note #129—SysEnvirons

Written by: Jim Friedlander — June 8, 1986
Updated: March 1, 1988

This note discusses some problems with the Resource Manager and how to work around them.

OpenResFile Bug

This section of the note formerly described a bug in `OpenResFile` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

GetMenu and ResErrProc

If your application makes use of `ResErrProc` (a pointer to a procedure stored in low-memory global `$AF2`) to detect resource errors, you will get unexpected calls to your `ResErrProc` procedure when calling `GetMenu` on 128K ROMs. The Menu Manager call `GetMenu` makes a call to `GetResInfo`, requesting resource information about MDEF 0. Unfortunately, `ROMMapInsert` is set to `FALSE`, so this call fails, setting `ResErr` to –192 (`resNotFound`). This in turn will cause a call to your `ResErrProc`, procedure even though the `GetMenu` call has worked correctly. This is **only** a problem if you are using `ResErrProc`.

The workaround is to:

1) save the address of your `ResErrProc` procedure
2) clear `ResErrProc`
3) do a `GetResource` call on the MENU resource you want to get
4) check to see if you get a nil handle back, if you do, you can handle the error in whatever way is appropriate for your application
5) call `GetMenu`, and
6) when you are done calling `GetMenu`, restore `ResErrProc`

SetResAttrs on read-only resource maps

SetResAttrs does not return an error if you are setting the resource attributes of a resource in a resource file that has a read-only resource map. The workaround is to check to see if the map is read-only and proceed from there:

```
CONST
    MapROBit = 8; {Toolbox bit ordering for bit 7 of low-order byte}

BEGIN
    ...
    attrs:= GetResFileAttrs(refNum);
    IF BitTst(@attrs,MapROBit) THEN ... {write-protected map}
```

Macintosh
Technical Notes

Developer Technical Support

#79: _ZoomWindow

Revised by: Craig Prouse April 1990
Written by: Jim Friedlander June 1986

This Technical Note contains some hints about using _ZoomWindow.
Changes since February 1990: Fixed a bug in DoWZoom which caused crashes if the content of a window did not intersect with any device's gdRect. Also made DoWZoom more robust by making savePort a local variable and checking for off-screen and inactive GDevice records. (One variable name has changed.) Additional minor changes: Corrected original sample code to use _EraseRect before zooming and added references to Human Interface Note #7, Who's Zooming Whom? for more subtle and application-specific considerations.

Basics

_ZoomWindow allows a window to be toggled between two states (where "state" means size and location): a default state and a user-selectable state. The default state stays the same unless the application changes it, while the user-selectable state is altered when the user changes the size or location of a zoomable window. The code to handle zoomable windows in a main event loop would look something like the examples which follow.

Note: _ZoomWindow assumes that the window that you are zooming is the current GrafPort. If thePort is not set to the window that is being zoomed, an address error is generated.

MPW Pascal

```
CASE myEvent.what OF
  mouseDown: BEGIN
    partCode:= FindWindow(myEvent.where, whichWindow);
      CASE partCode OF
        inZoomIn, InZoomOut:
          IF TrackBox(whichWindow, myEvent.where, partCode) THEN
            BEGIN
              GetPort(oldPort);
              SetPort(whichWindow);
              EraseRect(whichWindow^.portRect);
              ZoomWindow(whichWindow, partCode, TRUE);
              SetPort(oldPort);
            END; {IF}
        ...{and so on}
      END; {CASE}
    END; {mouseDown}
  ...{and so on}
END; {CASE}
```

MPW C

```
switch (myEvent.what) {
  case mouseDown:
    partCode = FindWindow(myEvent.where, &whichWindow);
    switch (partCode) {
      case inZoomIn:
      case inZoomOut:
        if (TrackBox(whichWindow, myEvent.where, partCode)) {
          GetPort(&oldPort);
          SetPort(whichWindow);
          EraseRect(whichWindow->portRect);
          ZoomWindow(whichWindow, partCode, true);
          SetPort(oldPort);
          } /* if */
        break;
      ... /* and so on */
    } /* switch */
  ... /* and so on */
} /* switch */
```

If a window is zoomable, that is, if it has a window definition ID = 8 (using the standard 'WDEF'), WindowRecord.dataHandle points to a structure that consists of two rectangles. The user-selectable state is stored in the first rectangle, and the default state is stored in the second rectangle. An application can modify either of these states, though modifying the user-selectable state might present a surprise to the user when the window is zoomed from the default state. An application should also be careful to not change either rectangle so that the title bar of the window is hidden by the menu bar.

Before modifying these rectangles, an application must make sure that DataHandle is not NIL. If it is NIL for a window with window definition ID = 8, that means that the program is not executing on a system or machine that supports zooming windows.

One need not be concerned about the use of a window with window definition ID = 8 making an application machine-specific—if the system or machine that the application is running on doesn't support zooming windows, _FindWindow never returns inZoomIn or inZoomOut, so neither _TrackBox nor _ZoomWindow are called.

If DataHandle is not NIL, an application can set the coordinates of either rectangle. For example, the Finder sets the second rectangle (default state) so that a zoomed-out window does not cover the disk and trash icons.

For the More Adventurous (or Seeing Double)

Developers should long have been aware that they should make no assumptions about the screen size and use screenBits.bounds to avoid limiting utilization of large video displays. Modular Macintoshes and Color QuickDraw support multiple display devices, which invalidates the use of screenBits.bounds unless the boundary of only the primary display (the one with the menu bar) is desired. When dragging and growing windows in a **multi-screen environment**, developers are now urged to use the bounding rectangle of the GrayRgn. In most cases, this is not a major modification and does not add a significant amount of code. Simply define a variable

```
desktopExtent := GetGrayRgn^^.rgnBBox;
```

and use this in place of screenBits.bounds. When zooming a document window, however, additional work is required to implement a window-zooming strategy which fully conforms with Apple's Human Interface Guidelines.

One difficulty is that when a new window is created with _NewWindow or _GetNewWindow, its default stdState rectangle (the rectangle determining the size and position of the zoomed window) is set by the Window Manager to be the gray region of the main display device inset by three pixels on each side. If a window has been moved to reflect a position on a secondary display, that window still zooms onto the main device, requiring the user to pan across the desktop to follow the window. The preferred behavior is to zoom the window onto the device containing the **largest portion** of the unzoomed window. This is a perfect example of a case where it is necessary for the application to modify the default state rectangle before zooming.

DoWZoom is a Pascal procedure which implements this functionality. It is a good example of how to manipulate both a WStateData record and the Color QuickDraw device list. On machines without Color QuickDraw (e.g., Macintosh Plus, Macintosh SE, Macintosh Portable) the stdState rectangle is left unmodified and the procedure reduces to five instructions, just like it is illustrated under "Basics." If Color QuickDraw is present, a sequence of calculations determines which display device contains most of the window prior to zooming. That device is considered dominant and is the device onto which the window is zoomed. A new stdState rectangle is computed based on the gdRect of the dominant GDevice. Allowances are made for the window's title bar, the menu bar if necessary, and for the standard three-pixel margin. (Please note that DoWZoom only mimics the behavior of the default _ZoomWindow trap as if it were implemented to support multiple displays. It does not account for the "natural size" of a window for a particular purpose. See Human Interface Note #7, Who's Zooming Whom?, for details on what constitutes the natural size of a window.) It is not necessary to set stdState prior to calling _ZoomWindow when zooming back to userState, so the extra code is not executed in this case.

DoWZoom is too complex to execute within the main event loop as shown in "Basics," but if an application is already using a similar scheme, it can simply add the DoWZoom procedure and replace the conditional block of code following

```
        IF TrackBox...
```

with

```
        DoWZoom(whichWindow, partCode);.
```

Happy Zooming.

```
PROCEDURE DoWZoom (theWindow: WindowPtr; zoomDir: INTEGER);
VAR
  windRect, theSect, zoomRect : Rect;
  nthDevice, dominantGDevice : GDHandle;
  sectArea, greatestArea : LONGINT;
  bias : INTEGER;
  sectFlag : BOOLEAN;
  savePort : GrafPtr;
BEGIN
  { theEvent is a global EventRecord from the main event loop }
  IF TrackBox(theWindow,theEvent.where,zoomDir) THEN
    BEGIN
      GetPort(savePort);
      SetPort(theWindow);
      EraseRect(theWindow^.portRect);    {recommended for cosmetic reasons}

      { If there is the possibility of multiple gDevices, then we  }
      { must check them to make sure we are zooming onto the right }
      { display device when zooming out. }
      { sysConfig is a global SysEnvRec set up during initialization  }
      IF (zoomDir = inZoomOut) AND sysConfig.hasColorQD THEN
        BEGIN
          { window's portRect must be converted to global coordinates }
          windRect := theWindow^.portRect;
          LocalToGlobal(windRect.topLeft);
          LocalToGlobal(windRect.botRight);
          { must calculate height of window's title bar }
          bias :=   windRect.top - 1
              -  WindowPeek(theWindow)^.strucRgn^^.rgnBBox.top;
          windRect.top := windRect.top - bias; {Thanks, Wayne!}
          nthDevice := GetDeviceList;
          greatestArea := 0;
          { This loop checks the window against all the gdRects in the   }
          { gDevice list and remembers which gdRect contains the largest }
          { portion of the window being zoomed. }
          WHILE nthDevice <> NIL DO
            IF TestDeviceAttribute(nthDevice,screenDevice) THEN
              IF TestDeviceAttribute(nthDevice,screenActive) THEN
                BEGIN
                  sectFlag := SectRect(windRect,nthDevice^^.gdRect,theSect);
                  WITH theSect DO
                    sectArea := LONGINT(right - left) * (bottom - top);
                  IF sectArea > greatestArea THEN
                    BEGIN
                      greatestArea := sectArea;
                      dominantGDevice := nthDevice;
                    END;
                  nthDevice := GetNextDevice(nthDevice);
                END; {of WHILE}
          { We must create a zoom rectangle manually in this case. }
          { account for menu bar height as well, if on main device }
          IF dominantGDevice = GetMainDevice THEN
            bias := bias + GetMBarHeight;
          WITH dominantGDevice^^.gdRect DO
            SetRect(zoomRect,left+3,top+bias+3,right-3,bottom-3);
          { Set up the WStateData record for this window. }
          WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
        END; {of Color QuickDraw conditional stuff}

      ZoomWindow(theWindow,zoomDir,TRUE);
      SetPort(savePort);
    END;
END;
```

In an attempt to avoid declaring additional variables, the original version of this document was flawed. In addition, the assignment statement responsible for setting the `stdState` rectangle is relatively complex and involves two type-casts. The following may look like C, but it really is Pascal. Trust me.

```
WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
```

It could be expanded into a more readable form such as:

```
VAR
   theWRec : WindowPeek;
   zbRec   : WStateDataHandle;

theWRec := WindowPeek(theWindow);
zbRec := WStateDataHandle(theWRec^.dataHandle);
zbRec^^.stdState := zoomRect;
```

Further Reference:

- *Inside Macintosh*, Volume IV, The Window Manager (pp. 49–52)
- *Inside Macintosh*, Volume V, Graphics Devices (p. 124), The Window Manager (p. 210)
- Human Interface Note #7, Who's Zooming Whom?

Macintosh Technical Notes

#80: Standard File Tips

See also: The Standard File Package

Written by: Jim Friedlander June 7, 1986
Updated: March 1, 1988

SFSaveDisk and CurDirStore

Low-memory location $214 (SFSaveDisk—a word) contains –1* the vRefNum of the volume that SF is displaying (MFS and HFS). It never contains –1* a WDRefNum.

Low-memory location $398 (CurDirStore—a long word) contains the dirID of the directory that SF is displaying (HFS only).

This information can be particularly useful at hook time, when the vRefNum field of the reply record has not yet been filled in. **Note:** reply.fName is filled in correctly at hook time if a file has been selected. If a directory has been selected, reply.fType is non-zero (it contains the dirID of the selected directory). If neither a file nor a directory is selected, both reply.fName[0] and reply.fType are 0.

Setting Standard File's default volume and directory

If you want SFGetFile or SFPutFile to display a certain volume when it draws its dialog, you can put –1 * the vRefNum of the volume you wish it to display into the low-memory global SFSaveDisk (a word at $214).

In Pascal, you would use something like:

```
...
TYPE
   WordPtr = ^INTEGER;                 {pointer to a two-byte location}
CONST
   SFSaveDisk = $214;                  {location of low-memory global}
VAR
   SFSaveVRef : WordPtr;
   myVRef     : INTEGER;
BEGIN
...
{myVRef gets assigned here}
...
SFSaveVRef := WordPtr(SFSaveDisk);     {point to SFSaveDisk}
SFSaveVRef^:= -1 * myVRef;             {"stuff" the value in}
SFGetFile(...
```

In C you would use something like this (where a variable of type "short" occupies 2 bytes):

```
#define SFSaveDisk (*(short *)0x214)

short myVRef;
...
/* myVRef gets assigned here */
...
SFSaveDisk = -1 * myVRef; /* "stuff" the value in */
SFGetFile(...
```

If you are running HFS and would like to have Standard File display a particular directory as well as a particular volume, you can't just put a WDRefNum into SFSaveDisk. If you do put a WDRefNum into SFSaveDisk, Standard File will display the root directory of the default volume. Instead, you must put –1 * the vRefNum into SFSaveDisk (see above) and put the dirID of the directory that you wish to have displayed in CurDirStore. If you put an invalid dirID into CurDirStore, Standard File will display the root level of the volume referred to by SFSaveDisk. To change CurDirStore you can use a technique similar to the above, but remember that CurDirStore is a four-byte value. If your application is running under MFS, Standard File ignores CurDirStore, so you can use the same code regardless of file system.

#81: Caching

See also: The File Manager
The Device Manager
Technical Note #14—The INIT 31 Mechanism

Written by: Rick Blair June 17, 1986
Updated: March 1, 1988

This technical note describes disk and File System caching on the Macintosh, with particular emphasis on the high-level File System cache. Of the three caches used for file I/O, this is the one which could have the most impact on your program. **Note**: This big File System cache is not available on 64K ROM machines.

A term

In this note I will use the term "HFS" to mean the Hierarchical File System **and** the Sony driver which can access the 800K drives. Both RAM-based HFS (Hard Disk 20 file) and the 128K ROM version include the second-generation Sony driver.

There's always a cache (type 1)

The first type of cache used by the File System has been around since the days of the Macintosh File System. Under MFS, each volume has a one-block buffer for all file/volume data. This prevents a read of two bytes followed by a read (at the next file position) of 4 bytes from causing actual disk I/O. The volume allocation map also gets saved in the system heap but it's not really part of the cache.

This type of caching is still used by HFS, which includes MFS-format volumes which may be mounted while running HFS. With HFS, the cache is a little bigger: each volume gets 1 block of buffering for the bitmap, 2 blocks for volume (including file) data, and 16 blocks for HFS B*-tree control buffering.

This cache lives in the system heap (unless HFS is using the new File System caching mechanism, in which case things become more complicated. See "type 3" below).

Cache track fever (type 2)

The track cache, only present with the enhanced Sony driver, will cache the current track (up to twelve blocks) so that subsequent reads to that track may use the cache. The track cache is "write through"; all writes go to both the cache and the Sony disk so flushing is never required.

Track caching only takes place for synchronous I/O calls; when an application makes asynchronous calls it expects to use the time while the disk is seeking, etc. to execute other code.

The track cache gets its storage space from the system heap.

Cache me if you can (type 3)

The last type of cache to be discussed is only available under the 128K and greater ROMs. This user-controlled cache is **not** "write-through".

Based on how much space the user has allocated via the control panel, the File System will set up a cache which can accommodate a certain number of blocks. This storage will come from the application heap in the space above `BufPtr` (see technical note #14 and below). This is really the space above the jump table and the "`A5` world", not technically part of the application heap. However, moving `BufPtr` down will cause a corresponding reduction in the space available to the application heap.

The installation code will also grab the space used by the old File System cache (type 1) since all types of disk blocks can be accommodated by this new cache.

The bulk of the caching **code** used for this RAM cache is also loaded above `BufPtr` at application launch time. This is accomplished by the INIT 35 resource which is installed in the system heap and initialized at boot time. At application launch time, INIT 35 checks the amount of cache allocated via the control panel and moves `BufPtr` down accordingly before bringing in the balance of the caching code. The RAM caching code is in the 'CACH' 1 resource in the System File.

The caching code always makes sure there is room for 128K of application heap and 32K of cache. If the user-requested amount would reduce the heap/cache below these values then the cache space is readjusted accordingly.

Up to 36 separate files may be buffered by the cache. Each queue is a list of blocks cached for that file. Information is kept about the "age" of each block and the blocks are also kept in a list in the order in which they occur in the file. The aging information tells which blocks were least recently used; these are the first to be released when new blocks become eligible for caching. The file order information is useful for flushing the cache to the disk in an efficient manner, i.e. the file order approximates disk order.

Assuming this cache has been enabled by the user, all files which are read from *or* written to by File System (HFS) calls are subject to caching under the current implementation. The cache is not "write through" like the track cache. When a File System write (`PBWrite`, `WriteResource`, etc.) is done, the block is buffered until the block is released (age discrimination), a volume flush is done or the application terminates.

It may be useful to an application to prevent this process of reading and writing "in place". The Finder disables caching of newly read/written blocks while doing file copies since it would be silly to cache files that the Finder was reading into memory anyway. Copy protection schemes may also need this capability. Disabling reading and writing in place is accomplished by setting a bit in a low memory flag byte, `CacheCom` (see below). When you set this flag, no new candidates for caching will be accepted. Blocks already saved may still be read from the cache, of course.

`CacheCom` is at `$39C`. Bit 7 is the bit to set to disable subsequent caching, as follows:

```
        MOVE.B CacheCom,saveTemp ;save away the old value
        BSET.B #7,CacheCom       ;tell caching code to stop R/W I.P.
        ...
        BTST.B #7,saveTemp       ;check saved value
        BNE.S  @69
        BCLR.B #7,CacheCom       ;clear it if it was cleared before
   @69
```

Bit 6 contains another flag which can force **all** I/O to go to the disk. If that flag is set then every time even one byte is requested from the File System the disk will be hit. I can think of no good reason to use this except to test the system code itself. The other bits should likewise be left alone.

Please don't use this feature unnecessarily; the user should retain control over caching. **Important:** if your program doesn't have enough space to run due to caching you should ask the user to disable (or reduce) it with the control panel and then relaunch your application. This may be the subject of a future technical note.

BufPtr

The RAM-resident caching software arbitrates `BufPtr` in the friendliest manner possible. It saves the old value away before changing it, and then when it is time to release its space it looks at it again. If `BufPtr` has been moved again, it knows that it can't restore the old value it saved until `BufPtr` is put back to where it left it. In this manner any subsequent code or data put up under `BufPtr` is assured of not being obliterated by the caching routines.

A final note

To avoid problems with data in the cache not getting written out to disk, call `FlushVol` after each time you write a file to disk. This ensures that the cache is written, in case a crash occurs soon thereafter.

#82: TextEdit: Advice & Descent

See also: TextEdit
Technical Note #22—TEScroll Bug
Technical Note #127—TextEdit EOL Ambiguity
Technical Note #131—TextEdit Bugs

Written by: Rick Blair June 21, 1986
Updated: March 1, 1988

This technical note will point out some bugs (and possible workarounds), and other items of interest for the TextEdit programmer.

TESelRect

Multiple line selections are often more complex shapes than simple rectangles. If this is the case, the `teSelRect` field of the `TERec` is set to the **last** (bottommost) rectangle in the selection. The `teHiHook` is called to invert each line of the selection.

The ROM limits the selection range (i.e. the lines that get set into `teSelRect`) to only those lines which will fit into the `viewRect`. This means that `teSelRect` will be left at the last **visible** line. (The old 64K ROMs made all the calls for the complete selection and just let clipping take care of the rest.)

TEDoText

The parameters of this special hook into TextEdit need a little additional explanation. `D3` and `D4` are described on page 391 of *Inside Macintosh Volume I* as being the first and last characters to be redrawn. This is true but specific to the –1 "`DoDraw`" case. In fact, all the calls to `TEDoText` are interested in these first and last character positions. They determine the selection for a (1) highlight call, the caret position for a (–2) `DoCaret` call (where `D4` is ignored as it's assumed to equal `D3`), etc.

Note that the `DoCaret` (–2) call behaves differently than described in *Inside Macintosh*, as well. Good old page 391 says it sets up the pen position for caret drawing. Since an `InvertRect` call is used to draw the caret if you use the default `teCarHook`, the ROMs just set up `teSelRect`, they don't bother with the QuickDraw pen.

TEScrpLength

Inside Macintosh describes `TEScrpLength` as a long integer; indeed, four bytes are reserved for this value with the intent of someday using that range of values. However, the ROMs use word operations in their accesses to `TEScrpLength` and make word calculations with it. This means that the high word of `TEScrpLength` is used for calculations. This is something to watch out for.

CharWidth

Inside Macintosh says that `CharWidth` takes stylistic variations into account when determining the width of a character. In fact, for *italic* and outlined styles the extra width is not taken into account. TextEdit relies on `CharWidth` for positioning of the caret, etc. If you have chosen to use, for instance, italic style in your TE record you will find that as you type the caret actually overlaps the character to the left and so when the caret is erased some of that character will get erased, too. This is somewhat disconcerting to the user but the program will still function correctly.

Clikloops

If you add your own click loop and try to do something like update scroll bars you may run into trouble. Before your routine gets called, TextEdit will have set clipping down to just the `viewRect`. You will have to save away the old clipping region, set it out to sufficient size (–32767, –32767, 32767, 32767 is probably OK), do your drawing, then restore TextEdit's clipping area so that it can function properly.

Macintosh Technical Notes

#83: System Heap Size Warning

See also: The Memory Manager

Written by: Jim Friedlander June 21, 1986
Updated: March 1, 1988

Earlier versions of this note pointed out that, due to varying system heap sizes, the application heap does not always start at `$CB00`. The start of the application heap has not been fixed for some time now; programs that depend on it never work on the Macintosh SE or the Macintosh II.

#84: Edit File Format

Written by: Harvey Alcabes April 11, 1985
Modified by: Bryan Johnson August 15, 1986
Updated: March 1, 1988

This technical note describes the format of the files created by Edit. It has been verified for versions 1.x and 2.0.

Edit, a text editor licensed by Apple and included in the Consulair 68000 Development System, can read any text-only file whose file type is TEXT. Files created by Edit have a creator ID of EDIT. Edit is a disk-based editor so the file length is not limited by available memory. Files created or modified by Edit, have the format described below; if they are not too long they can be read by any application which can read TEXT files (eg: MacWrite, Microsoft Word, or the APDA example program File).

> The data fork contains text (ASCII characters). Carriage return characters indicate line breaks; tab characters are displayed as described below. No other characters have special significance.
>
> The resource fork contains resources of type ETAB and EFNT. If Edit opens a text-only file that does not have these resources it will add them.
>
> The ETAB (Editor TAB) resource, resource ID 1004, contains two integers. The first is the number of pixels to display for each space within a tab (not necessarily the same as for the space character). The second integer is the number of these spaces which will be displayed for each tab character.
>
> The EFNT (Editor FoNT) resource, resource ID 1003, contains an integer followed by a Pascal string (length byte followed by characters). The integer is the point size of the document's font. The string contains the font name. If the string size (including the length byte) is odd, an extra byte is added so that the resource size is even.

For more information about Edit, contact:

Consulair Corp.
140 Campo Drive
Portola Valley, CA 94025
(415) 851-3272

#85: GetNextEvent; Blinking Apple Menu

See also: The Menu Manager
The Toolbox Event Manager
The Desk Manager

Written by: Rick Blair August 14, 1986
Updated: March 1, 1988

Wherein arcane mysteries are unraveled so you can make the Alarm Clock (or a similar desk accessory) blink the Apple menu at the appointed second. Also, why `GetNextEvent` is a good thing.

The obvious

Don't disable interrupts within an application! There will almost certainly come a time (or Macintosh) where you won't be able to change the interrupt mask because the processor is running in user mode. The one-second interrupt is used to blink the apple.

The not-so-obvious

You must call `GetNextEvent` periodically. `GetNextEvent` uses a filter (GNE filter) which allows for a routine to be installed which overrides (or augments) the behavior of the system. The GNE filter is installed by pointing the low-memory global `jGNEFilter` (a long word at `$29A`) to the routine. After all other GNE processing is complete, the `routine` will be called with `A1` pointing to the event record and `D0` containing the boolean result. The filter may then modify the event record or change the function result by altering the word on the stack at `4(A7)`. This word will match `D0` initially, of course.

A GNE filter is used to do the blinking when the interrupt handler has announced that the moment is at hand. `GetOSEvent` won't do. If you don't have a standard main event loop, it is generally a good idea to give `GetNextEvent` (and `SystemTask`, too) a call whenever you have any idle time. `GetNextEvent` "extra" services include, but aren't limited to, the following:

1. Calling the GNE filter.
2. Removing lingering disk-switched windows (uncommon unless memory is tight).
3. Making Window Manager activate, deactivate and update events happen.
4. Getting various events from a journaling driver when one is playing.
5. Giving `SystemEvent` a chance at each event.
6. Running command-shift function key routines (e.g. command-shift-4 to print the screen to an ImageWriter).

The more subtle

When the (default) GNE filter sees that the interrupt handler has set the "time to blink" flag, it looks at the first menu in `MenuList`. The title of that menu must consist solely of the "apple" character or no blinking will occur. It really just looks at the first word of the string to see if it is `$0114`. This is a Pascal string which has only the `$14` "apple" character in it. So you musn't have any spaces or any other characters in the title of your first menu or you'll get no blinkin' results.

Macintosh Technical Notes

Developer Technical Support

#86: MacPaint Document Format

Revised by: Jim Reekes
Written by: Bill Atkinson

June 1989
1983

This Technical Note describes the internal format of a MacPaint® document, which is a standard used by many other programs. This description is the same as that found in the "Macintosh Miscellaneous" section of early *Inside Macintosh* versions.
Changes since October 1988: Fixed bugs in the example code.

MacPaint documents are easy to read and write, and they have become a standard interchange format for full–page images on the Macintosh. This Note describes the MacPaint internal document format to help developers generate and interpret files in this format.

MacPaint documents have a file type of "PNTG," and since they use only the data fork, you can ignore the resource fork. The data fork contains a 512–byte header followed by compressed data which represents a single bitmap (576 pixels wide by 720 pixels tall). At a resolution of 72 pixels per inch, this bitmap occupies the full 8 inch by 10 inch printable area of a standard ImageWriter printer page.

Header

The first 512 bytes of the document form a header of the following format:

- 4–byte version number (default = 2)
- 38*8 = 304 bytes of patterns
- 204 unused bytes (reserved for future expansion)

As a Pascal record, the document format could look like the following:

```
MPHeader = RECORD
        Version:        LONGINT;
        PatArray:       ARRAY [1..38] of Pattern;
        Future:         PACKED ARRAY [1..204] of SignedByte;
END;
```

If the version number is zero, the document uses default patterns, so you can ignore the rest of the header block, and if your program generates MacPaint documents, you can write 512 bytes of zero for the document header. Most programs which read MacPaint documents can skip the header when reading.

Bitmap

Following the header are 720 compressed scan lines of data which form the 576 pixel wide by 720 pixel tall bitmap. Without compression, this bitmap would occupy 51,840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10K using the `_PackBits` procedure to compress runs of equal bytes within each scan line. The bitmap part of a MacPaint document is simply the output of _PackBits called 720 times, with 72 bytes of input each time.

To determine the maximum size of a MacPaint file, it is worth noting what *Inside Macintosh* says about _PackBits:

> *"The worst case would be when _PackBits adds one byte to the row of bytes when packing."*

If we include an extra 512 bytes for the file header information to the size of an uncompressed bitmap (51,840), then the total number of bytes would be 52,352. If we take into account the extra 720 "potential" bytes (one for each row) to the previous total, the maximum size of a MacPaint file becomes 53,072 bytes.

Reading Sample

```
PROCEDURE ReadMPFile;
{ This is a small example procedure written in Pascal that demonstrates
  how to read MacPaint files. As a final step, it takes the data that
  was read and displays it on the screen to show that it worked.
  Caveat: This is not intended to be an example of good programming
  practice, in that the possible errors merely cause the program to exit.
  This is VERY uninformative, and there should be some sort of error handler
  to explain what happened. For simplicity, and thus clarity, those types
  of things were deliberately not included. This example will not work
  on a 128K Macintosh, since memory allocation is done too simplistically.
}

CONST
       DefaultVolume = 0;
       HeaderSize = 512;                   { size of MacPaint header in bytes }
       MaxUnPackedSize = 51840;            { maximum MacPaint size in bytes }
                                           { 720 lines * 72 bytes/line }

VAR
       srcPtr:         Ptr;
       dstPtr:         Ptr;
       saveDstPtr:     Ptr;
       lastDestPtr:    Ptr;
       srcFile:        INTEGER;
       srcSize:        LONGINT;
       errCode:        INTEGER;
       scanLine:       INTEGER;
       aPort:          GrafPort;
       theBitMap:      BitMap;

BEGIN
       errCode := FSOpen('MP TestFile', DefaultVolume, srcFile); { Open the file. }
       IF errCode <> noErr THEN ExitToShell;

       errcode := SetFPos(srcFile, fsFromStart, HeaderSize);     { Skip the header. }
       IF errCode <> noErr THEN ExitToShell;

       errCode := GetEOF(srcFile, srcSize);       { Find out how big the file is, }
       IF errCode <> noErr THEN ExitToShell;      { and figure out source size. }

       srcSize := srcSize - HeaderSize ;          { Remove the header from count. }
       srcPtr := NewPtr(srcSize);                 { Make buffer just the right size. }
       IF srcPtr = NIL THEN ExitToShell;

       errCode := FSRead(srcFile, srcSize, srcPtr); { Read the data into the buffer. }
       IF errCode <> noErr THEN ExitToShell;      { File marker is past header. }
```

```
         errCode := FSClose(srcFile);           { Close the file we just read. }
         IF errCode <> noErr THEN ExitToShell;

         { Create a buffer that will be used for the Destination BitMap. }
         dstPtr := NewPtrClear(MaxUnPackedSize);    {MPW library routine, see TN 219}
         IF dstPtr = NIL THEN ExitToShell;
         saveDstPtr := dstPtr;

         { Unpack each scan line into the buffer. Note that 720 scan lines are
           guaranteed to be in the file. (They may be blank lines.) In the
           UnPackBits call, the 72 is the count of bytes done when the file was
           created.  MacPaint does one scan line at a time when creating the file.
           The destination pointer is tested each time through the scan loop.
           UnPackBits should increment this pointer by 72, but in the case where
           the packed file is corrupted UnPackBits may end up sending bits into
           uncharted territory.  A temporary pointer "lastDstPtr" is used for testing
           the result.}

         FOR scanLine := 1 TO 720 DO BEGIN
              lastDstPtr := dstPtr;
              UnPackBits(srcPtr, dstPtr, 72);        { bumps both pointers }
              IF ORD4(lastDstPtr) + 72 <> ORD4(dstPtr) THEN ExitToShell;
         END;

         { The buffer has been fully unpacked. Create a port that we can draw into.
           You should save and restore the current port.  }
         OpenPort(@aPort);

         { Create a BitMap out of our saveDstPtr that can be copied to the screen. }
         theBitMap.baseAddr := saveDstPtr;
         theBitMap.rowBytes := 72;                { width of MacPaint picture }
         SetPt(theBitMap.bounds.topLeft, 0, 0);
         SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

         { Now use that BitMap and draw the piece of it to the screen.
           Only draw the piece that is full screen size (portRect). }
         CopyBits(theBitMap, aPort.portBits, aPort.portRect,
                        aPort.portRect, srcCopy, NIL);

         { We need to dispose of the memory we've allocated.  You would not
           dispose of the destPtr if you wish to edit the data.  }
         DisposPtr(srcPtr);                   { dispose of the source buffer }
         DisposPtr(dstPtr);                   { dispose of the destination buffer }
END;
```

Writing Sample

```
PROCEDURE WriteMPFile;
{ This is a small example procedure written in Pascal that demonstrates how
  to write MacPaint files. It will use the screen as a handy BitMap to be
  written to a file.
}

CONST
       DefaultVolume = 0;
       HeaderSize = 512;                    { size of MacPaint header in bytes }
       MaxFileSize = 53072;                 { maximum MacPaint file size. }

VAR
       srcPtr:        Ptr;
       dstPtr:        Ptr;
       dstFile:       INTEGER;
       dstSize:       LONGINT;
       errCode:       INTEGER;
       scanLine:      INTEGER;
       aPort:         GrafPort;
       dstBuffer:     PACKED ARRAY[1..HeaderSize] OF BYTE;
       I:             LONGINT;
       picturePtr:    Ptr;
       tempPtr:       BigPtr;
       theBitMap:     BitMap;

BEGIN
       { Make an empty buffer that is the picture size. }
       picturePtr := NewPtrClear(MaxFileSize);    {MPW library routine, see TN 219}
       IF picturePtr = NIL THEN ExitToShell;

       { Open a port so we can get to the screen's BitMap easily.  You should save
         and restore the current port. }
       OpenPort(@aPort);

       { Create a BitMap out of our dstPtr that can be copied to the screen. }
       theBitMap.baseAddr := picturePtr;
       theBitMap.rowBytes := 72;                   { width of MacPaint picture }
       SetPt(theBitMap.bounds.topLeft, 0, 0);
       SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

       { Draw the screen over into our picture buffer. }
       CopyBits(aPort.portBits, theBitMap, aPort.portRect,
                      aPort.portRect, srcCopy, NIL);

       { Create the file, giving it the right Creator and File type.}
       errCode := Create('MP TestFile', DefaultVolume, 'MPNT', 'PNTG');
       IF errCode <> noErr THEN ExitToShell;

       { Open the data file to be written. }
       errCode := FSOpen(dstFileName, DefaultVolume, dstFile);
       IF errCode <> noErr THEN ExitToShell;

       FOR I := 1 to HeaderSize DO                 { Write the header as all zeros. }
            dstBuffer[I] := 0;
       errCode := FSWrite(dstFile, HeaderSize, @dstBuffer);
       IF errCode <> noErr THEN ExitToShell;
```

```
        { Now go into a loop where we pack each line of data into the buffer,
          then write that data to the file. We are using the line count of 72
          in order to make the file readable by MacPaint. Note that the
          Pack/UnPackBits can be used for other purposes. }
        srcPtr := theBitMap.baseAddr;                 { point at our picture BitMap }
        FOR scanLine := 1 to 720 DO
             BEGIN
                dstPtr := @dstBuffer;                 { reset the pointer to bottom }
                PackBits(srcPtr, dstPtr, 72);                { bumps both ptrs }
                dstSize := ORD(dstPtr)-ORD(@dstBuffer);      { calc packed size }
                errCode := FSWrite(dstFile, dstSize, @dstBuffer);
                IF errCode <> noErr THEN ExitToShell;
             END;

        errCode := FSClose(dstFile);                  { Close the file we just wrote. }
        IF errCode <> noErr THEN ExitToShell;
END;
```

Further Reference:

- *Inside Macintosh*, Volume I-135, QuickDraw
- *Inside Macintosh*, Volume I-465, Toolbox Utilities
- *Inside Macintosh*, Volume II-77, The File Manager
- Technical Note #219, New Memory Manager Glue Routines

MacPaint is a registered trademark of Claris Corporation.

Macintosh Technical Notes

#87: Error in FCBPBRec

See also: The File Manager

Written by: Jim Friedlander August 18, 1986
Updated: March 1, 1988

The declaration of a FCBPBRec is wrong in *Inside Macintosh Volume IV* and early versions of MPW. This has been fixed in MPW 1.0 and newer.

An error was made in the declaration of an FCBPBRec parameter block that is used in PBGetFCBInfo calls. The field ioFCBIndx was incorrectly listed as a LONGINT. The following declaration (found in *Inside Macintosh*):

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      LONGINT;
ioFCBFlNm:      LONGINT;
...
```

should be changed to:

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      INTEGER;
ioFCBFiller1:   INTEGER;
ioFCBFlNm:      LONGINT;
...
```

#88: Signals

See also: Using Assembly Language (Mixing Pascal & Assembly)

Written by: Rick Blair

August 1, 1986
Updated: March 1, 1988

Signals are a form of intra-program interrupt which can greatly aid clean, inexpensive error trapping in stack frame intensive languages. A program may invoke the `Signal` procedure and immediately return to the last invocation of `CatchSignal`, including the complete stack frame state at that point.

Signals allow a program to leave off execution at one point and return control to a convenient error trap location, regardless of how many levels of procedure nesting are in between.

The example is provided with a Pascal interface, but it is easily adapted to other languages. The only qualification is that the language **must** bracket its procedures (or functions) with `LINK` and `UNLK` instructions. This will allow the signal code to clean up at procedure exit time by removing `CatchSignal` entries from its internal queue. **Note:** only procedures and/or functions that call `CatchSignal` need to be bracketed with `LINK` and `UNLK` instructions.

Important: `InitSignals` must be called from the main program so that `A6` can be set up properly.

Note that there is no limit to the number of local `CatchSignals` which may occur within a single routine. Only the last one executed will apply, of course, unless you call `FreeSignal`. `FreeSignal` will "pop" off the last `CatchSignal`. If you attempt to `Signal` with no `CatchSignals` pending, `Signal` will halt the program with a debugger trap.

`InitSignals` creates a small relocatable block in the application heap to hold the signal queue. If `CatchSignal` is unable to expand this block (which it does 5 elements at a time), then it will signal back to the last successful `CatchSignal` with `code` = 200. A `Signal(0)` acts as a `NOP`, so you may pass `OSErrs`, for instance, after making File System type calls, and, if the `OSErr` is equal to `NoErr`, nothing will happen.

CatchSignal may not be used in an expression if the stack is used to evaluate that expression. For example, you can't write:

```
    c:= 3*CatchSignal;
```

"Gotcha" summary

1. Routines which call CatchSignal must have stack frames.
2. InitSignals must be called from the outermost (main) level.
3. Don't put the CatchSignal function in an expression. Assign the result to an INTEGER variable; i.e. i:=CatchSignal.
4. It's safest to call a procedure to do the processing after CatchSignal returns. See the Pascal example TestSignals below. This will prevent the use of a variable which may be held in a register.

Below are three separate source files. First is the Pascal interface to the signaling unit, then the assembly language which implements it in MPW Assembler format. Finally, there is an example program which demonstrates the use of the routines in the unit.

```
{File ErrSignal.p}
UNIT ErrSignal;

INTERFACE

{Call this right after your other initializations (InitGraf, etc.)--in other
words as early as you can in the application}
PROCEDURE InitSignals;

{Until the procedure which encloses this call returns, it will catch
subsequent Signal calls, returning the code passed to Signal. When
CatchSignal is encountered initially, it returns a code of zero. These calls
may "nest"; i.e. you may have multiple CatchSignals in one procedure.
Each nested CatchSignal call uses 12 bytes of heap space }
FUNCTION CatchSignal:INTEGER;

{This undoes the effect of the last CatchSignal. A Signal will then invoke
the CatchSignal prior to the last one.}
PROCEDURE FreeSignal;

{Returns control to the point of the last CatchSignal. The program will then
behave as though that CatchSignal had returned with the code parameter
supplied to Signal.}
PROCEDURE Signal(code:INTEGER);

END.
{End of ErrSignal.p}
```

Here's the assembly source for the routines themselves:

```
    ; ErrSignal code w. InitSignal, CatchSignal,FreeSignal, Signal
    ; defined
    ;
    ;              Version 1.0 by Rick Blair

                PRINT     OFF
                INCLUDE   'Traps.a'
                INCLUDE   'ToolEqu.a'
                INCLUDE   'QuickEqu.a'
                INCLUDE   'SysEqu.a'
                PRINT     ON


    CatchSigErr EQU       200             ;"insufficient heap" message
    SigChunks   EQU       5               ;number of elements to expand by
    FrameRet    EQU       4               ;return addr. for frame (off A6)
    SigBigA6    EQU       $FFFFFFFF       ;maximum positive A6 value


 ; A template in MPW Assembler describes the layout of a collection of data
 ; without actually allocating any memory space. A template definition starts ;
 with a RECORD directive and ends with an ENDR directive.

 ; To illustrate how the template type feature works, the following template
 ; is declared and used. By using this, the assembler source appromixates very
 ; closely Pascal source for referencing the corresponding information.

 ;template for our table elements
   SigElement  RECORD    0     ;the zero is the template origin
   SigSP       DS.L      1     ;the SP at the CatchSignal-(DS.L just like EQU)
   SigRetAddr  DS.L      1     ;the address where the CatchSignal returned
   SigFRet     DS.L      1     ;return addr. for encl. procedure
   SigElSize   EQU       *     ;just like EQU 12
               ENDR


 ; The global data used by these routines follows. It is in the form of a
 ; RECORD, but, unlike above, no origin is specified, which means that memory
 ; space *will* be allocated.
 ; This data is referenced through a WITH statement at the beginning of the
 ; procs that need to get at this data. Since the Assembler knows when it is
 ; referencing data in a data module (since they must be declared before they
 ; are accessed), and since such data can only be accessed based on A5, there
 ; is no need to explicitly specify A5 in any code which references the data
 ; (unless indexing is used). Thus, in this program we have omitted all A5
 ; references when referencing the data.

   SigGlobals RECORD                  ;no origin means this is a data record
                                      ;not a template(as above)
   SigEnd     DS.L      1             ;current end of table
   SigNow     DS.L      1             ;the MRU element
   SigHandle  DC.L      0             ;handle to the table
              ENDR
```

```
InitSignals PROC     EXPORT           ;PROCEDURE InitSignals;

            IMPORT   CatchSignal
            WITH     SigElement,SigGlobals

;the above statement makes the template SigElement and the global data
;record SigGlobals available to this procedure

            MOVE.L   #SigChunks*SigElSize,D0
            _NewHandle                ;try to get a table
            BNE.S    forgetit         ;we couldn't get that!?

            MOVE.L   A0,SigHandle     ;save it
            MOVE.L   #-SigElSize,SigNow ;point "now" before start
            MOVE.L   #SigChunks*SigElSize,SigEnd ;save the end
            MOVE.L   #SigBigA6,A6     ;make A6 valid for Signal
forgetit    RTS
            ENDP

CatchSignal PROC     EXPORT           ;FUNCTION CatchSignal:INTEGER;
            IMPORT   SiggySetup,Signal,SigDeath
            WITH     SigElement,SigGlobals

            MOVE.L   (SP)+,A1         ;grab return address
            MOVE.L   SigHandle,D0     ;handle to table
            BEQ      SigDeath         ;if NIL then croak
            MOVE.L   D0,A0            ;put handle in A-register
            MOVE.L   SigNow,D0
            ADD.L    #SigElSize,D0
            MOVE.L   D0,SigNow        ;save new position
            CMP.L    SigEnd,D0        ;have we reached the end?
            BNE.S    catchit          ;no, proceed

            ADD.L    #SigChunks*SigElSize,D0 ;we'll try to expand
            MOVE.L   D0,SigEnd        ;save new (potential) end
            _SetHandleSize
            BEQ.S    @0               ;jump around if it worked!

;signals, we use 'em ourselves
            MOVE.L   SigNow,SigEnd    ;restore old ending offset
            MOVE.L   #SigElSize,D0
            SUB.L    D0,SigNow        ;ditto for current position
            MOVE.W   #catchSigErr,(SP);we'll signal a "couldn't
                                      ;                catch" error
            JSR      Signal           ;never returns of course



@0          MOVE.L   SigNow,D0

catchit     MOVE.L   (A0),A0          ;deref.
            ADD.L    D0,A0            ;point to new entry
            MOVE.L   SP,SigSP(A0)     ;save SP in entry
            MOVE.L   A1,SigRetAddr(A0) ;save return address there
            CMP.L    #SigBigA6,A6     ;are we at the outer level?
            BEQ.S    @0               ;yes, no frame or cleanup needed
            MOVE.L   FrameRet(A6),SigFRet(A0);save old frame return
                                              ;               address
```

```
              LEA         SiggyPop,A0
              MOVE.L      A0,FrameRet(A6) ;set cleanup code address
  @0          CLR.W       (SP)            ;no error code (before its time)
              JMP         (A1)            ;done setting the trap

  SiggyPop    JSR         SiggySetup      ;get pointer to element
              MOVE.L      SigFRet(A0),A0 ;get proc's real return address
              SUB.L       #SigElSize,D0
              MOVE.L      D0,SigNow       ;"pop" the entry
              JMP         (A0)            ;gone
              ENDP

  FreeSignal  PROC        EXPORT          ;PROCEDURE FreeSignal;
              IMPORT      SiggySetup
              WITH        SigElement,SigGlobals
              JSR         SiggySetup      ;get pointer to current entry
              MOVE.L      SigFRet(A0),FrameRet(A6) ;"pop" cleanup code
              SUB.L       #SigElSize,D0
              MOVE.L      D0,SigNow       ;"pop" the entry
              RTS
              ENDP

  Signal      PROC        EXPORT          ;PROCEDURE Signal(code:INTEGER);
              EXPORT      SiggySetup,SigDeath
              WITH        SigElement,SigGlobals
              MOVE.W      4(SP),D1        ;get code
              BNE.S       @0              ;process the signal if code is non-zero
              MOVE.L      (SP),A0         ;save return address
              ADDQ.L      #6,SP           ;adjust stack pointer
              JMP         (A0)            ;return to caller(code was 0)

  @0          JSR         SiggySetup      ;get pointer to entry
              BRA.S       SigLoop1

  SigLoop     UNLK        A6              ;unlink stack by one frame
  SigLoop1    CMP.L       SigSP(A0),A6    ;is A6 beyond the saved stack?
              BLO.S       SigLoop         ;yes, keep unlinking
              MOVE.L      SigSP(A0),SP    ;bring back our SP
              MOVE.L      SigRetAddr(A0),A0 ;get return address
              MOVE.W      D1,(SP)         ;return code to CatchSignal
              JMP         (A0)            ;Houston, boost the Signal!
                          ;(or Hooston if you're from the Negative Zone)

  SiggySetup  MOVE.L      SigHandle,A0
              MOVE.L      (A0),A0         ;deref.
              MOVE.L      A0,D0           ;to set CCR
              BEQ.S       SigDeath        ;nil handle means trouble
              MOVE.L      SigNow,D0       ;grab table offset to entry
              BMI.S       SigDeath        ;if no entries then give up
              ADD.L       D0,A0           ;point to current element
              RTS

  SigDeath    _Debugger                   ;a signal sans catch is bad news

              ENDP
              END
```

Now for the example Pascal program:

```
PROGRAM TestSignals;
USES ErrSignal;

VAR i:INTEGER;

PROCEDURE DoCatch(s:STR255; code:INTEGER);
BEGIN
  IF code<>0 THEN BEGIN
    Writeln(s,code);
    Exit(TestSignals);
  END;
END; {DoCatch}

PROCEDURE Easy;
  PROCEDURE Never;
    PROCEDURE DoCatch(s:STR255; code:INTEGER);
    BEGIN
      IF code<>0 THEN BEGIN
        Writeln(s,code);
        Exit(Never);
      END;
    END; {DoCatch}

  BEGIN {Never}
  i:=CatchSignal;
  DoCatch('Signal caught from Never, code = ', i );

  i:=CatchSignal;
  IF i<>0 THEN DoCatch('Should never get here!',i);

  FreeSignal; {"free" the last CatchSignal}
  Signal(7); {Signal a 7 to the last CatchSignal}
  END;{Never}
BEGIN {Easy}
Never;
Signal(69);       {this won't be caught in Never}
END;{Easy}        {all local CatchSignals are freed when a procedure exits.}

BEGIN {PROGRAM}
InitSignals; {You must call this early on!}

{catch Signals not otherwise caught by the program}
i:=CatchSignal;
IF i<>0 THEN
 DoCatch('Signal caught from main, code = ',i);

Easy;
END.
```

The example program produces the following two lines of output:

```
Signal caught from Never, code = 7
Signal caught from main, code = 69
```

#89: DrawPicture Bug

Written by: Ginger Jernigan August 16, 1986
Updated: March 1, 1988

Earlier versions of this note described a bug in `DrawPicture`. This bug never occurred on 64K ROM machines, and has been fixed in System 3.2 and newer. Use of Systems older than 3.2 on non-64K ROM machines is no longer recommended.

#90: SANE Incompatibilities

Written by: Mark Baumwell August 14, 1986
Updated: March 1, 1988

Earlier versions of this note described a problem with SANE and System 2.0. Use of System 2.0 is only recommended for Macintosh 128 machines, which contain the 64K ROMs. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#91: Optimizing for the LaserWriter—Picture Comments

See also:
The Print Manager
QuickDraw
Technical Note #72—
Optimizing for the LaserWriter—Techniques
Technical Note #27—MacDraw Picture Comments
PostScript Language Reference Manual, Adobe Systems
PostScript Language Tutorial and Cookbook,
Adobe Systems
LaserWriter Reference Manual

Written by:	Ginger Jernigan	November 15, 1986
Modified by:	Ginger Jernigan	March 2, 1987
Updated:		March 1, 1988

This technical note is a continuation of Technical Note #72. This technical note discusses the picture comments that the LaserWriter driver recognizes.

This technical note has been modified to include corrected descriptions of the `SetLineWidth`, `PostScriptFile` and `ResourcePS` comments and to include some additional warnings.

The implementation of QuickDraw's `picComment` facility by the LaserWriter driver allows you to take advantage of features (like rotated text) which are available in PostScript but may not be available in QuickDraw.

Warning: Using PostScript-specific comments will make your code printer-dependent and may cause compatibility problems with non-PostScript devices, so don't use them unless you absolutely have to.

Some of the picture comments below are designed to be issued along with QuickDraw commands that simulate the commented commands on the Macintosh screen. When the comments are used, the accompanying QuickDraw comments are ignored. If you are designing a picture to be printed by the LaserWriter, the structure and use of these comments must be precise, otherwise nothing will print. If another printer driver (like the ImageWriter I/II driver) has not implemented these comments, the comments are ignored and the accompanying QuickDraw commands are used.

Below are the picture comments that the LaserWriter driver recognizes:

Type	Kind	Data Size	Data	Description
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NIL	End text function
StringBegin	152	0	NIL	Begin pieces of original string
StringEnd	153	0	NIL	End pieces of original string
TextCenter	154	8	TTxtCenter	Offset to center of rotation
* LineLayoutOff	155	0	NIL	Turns LaserWriter line layout off
* LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following poly data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
picPlyClo	165	0	NIL	Close the poly
* DashedLine	180	-	TDashedLine	Draw following lines as dashed
* DashedStop	181	0	NIL	End dashed lines
* SetLineWidth	182	4	Point	Set fractional line widths
* PostScriptBegin	190	0	NIL	Set driver state to PostScript
* PostScriptEnd	191	0	NIL	Restore QuickDraw state
* PostScriptHandle	192	-	PSData	PostScript data in handle
*† PostScriptFile	193	-	FileName	FileName in data handle
* TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
*† ResourcePS	195	8	Type/ID/Index	PostScript data in a resource file
**RotateBegin	200	4	TRotation	Begin rotated port
**RotateEnd	201	0	NIL	End rotation
**RotateCenter	202	8	Center	Offset to center of rotation
**FormsPrinting	210	0	NIL	Don't clear print buffer after each page
**EndFormsPrinting	211	0	NIL	End forms printing after PrClosePage

* These comments are only implemented in LaserWriter driver 3.0 or later.
** These comments are only implemented in LaserWriter driver 3.1 or later.
† These comments are not available when background printing is enabled.

Each of these comments are discussed below in six groups: Text, Polygons, Lines, PostScript, Rotation, and Forms. Code examples are given where appropriate. For other examples of how to use picture comments for printing please see the Print example program in the Software Supplement (currently available through APDA as "Macintosh Example Applications and Sources 1.0").

Note: The examples used in the *LaserWriter Reference Manual* are incorrect. Please use the examples presented here instead.

Text

In order to support the What-You-See-Is-What-You-Get paradigm, the LaserWriter driver uses a line layout algorithm to assure that the placement of the line on the printer closely approximates the placement of the line on the screen. This means that the printer driver gets the width of the line from QuickDraw, then tells PostScript to place the text in exactly the same place with the same width.

The TextBegin comment allows the application to specify the layout and the orientation of the text that follows it by specifying the following information:

```
TTxtPicRec = PACKED RECORD
  tJus:  Byte;     {0,1,2,3,4 or greater => none, left, center, right, full
                   justification }
  tFlip: Byte;     {0,1,2 => none, horizontal, vertical coordinate flip }
  tRot:  INTEGER;  {0..360 => clockwise rotation in degrees }
  tLine: Byte;     {1,2,3.. => single, 1-1/2, double.. spacing }
  tCmnt: Byte;     {Reserved }
END; { TTxtPicRec }
```

Left, right or center justification, specified by tJust, tells the driver to maintain only the left, right or center point, without recalculating the interword spacing. Full justification specifies that both endpoints be maintained and interword spacing be recalculated. This means that the driver makes sure that the specified points are maintained on the printer without caring whether the overall width has changed. Full justification means that the overall width of the line has been maintained. tFlip and tRot specify the orientation of the text, allowing the application to take advantage of the rotation features of PostScript. tLine specifies the interline spacing. When no TextBegin comment is used, the defaults are full justification, no rotation and single-spaced lines.

String Reconstruction

The StringBegin and StringEnd comments are used to bracket short strings of text that are actually sections of an original long string. MacDraw, for instance, breaks long strings into shorter pieces to avoid stack overflow problems with QuickDraw in the 64K ROM. When these smaller strings are bracketed by StringBegin and StringEnd, the LaserWriter driver assumes that the enclosed strings are parts of one long string and will perform its line layout accordingly. Erasing or filling of background rectangles should take place before the StringBegin comment to avoid confusing the process of putting the smaller strings back together.

Text Rotation

In order to rotate a text object, PostScript needs to have information concerning the center of rotation. The TextCenter comment provides this information when a rotation is specified in the TextBegin comment. This comment contains the offset from the present pen location to the center of rotation. The offset is given as the y-component, then the x-component, which are declared as fixed-point numbers. This allows the center to be in the middle of a pixel. This comment should appear after the TextBegin comment and before the first following StringBegin comment.

The associated comment data looks like this:

```
TTxtCenter = RECORD
   y,x: Fixed;        {offset from current pen location to center of rotation}
END; { TTxtCenter }
```

Right after a `TextBegin` comment, the LaserWriter driver expects to see a `TextCenter` comment specifying the center of rotation for any text enclosed within the text comment calls. It will ignore all further `CopyBits` calls, and print all standard text calls in the rotation specified by the information in `TTxtPicRec`. The center of rotation is the offset from the beginning position of the first string following the `TextCenter` comment. The printer driver also expects the string locations to be in the coordinate system of the current QuickDraw port. The printer driver rotates the entire port to draw the text so it can draw several strings with one rotation comment and one center comment. It is good practice to enclose an entire paragraph or paragraphs of text in a single rotation comment so that the driver makes the fewest number of rotations.

The printer driver can draw non-textual objects within the bounds of the text rotation comments but it must unrotate to draw the object, then re-rotate to draw the next string of text. To do this the printer driver must receive another `TextCenter` comment before each new rotation. So, rotated text and unrotated objects can be drawn inter-mixed within one `TextBegin/TextEnd` comment pair, but performance is slowed.

Note that all bit maps and all clip regions are ignored during text rotation so that clip regions can be used to clip out the strings on printers that can't take advantage of these comments. This has the unfortunate side effect of not allowing rotated text to be clipped.

Rotated text comments are not associated with landscape and portrait orientation of the printer paper as selected by the Page Setup dialog. These are rotations with reference to the current QuickDraw port only.

All of the above text comments are terminated by a `TextEnd` comment.

Turning Off Line Layout

If your application is using its own line layout algorithm (it uses its own character widths or does its own character or word placement), the printer driver doesn't need to do it too. To turn off line layout, you can use the `LineLayoutOff` comment. `LineLayoutOn` turns it on again.

Turning on `FractEnable` for the 128K ROMs has the same effect as `LineLayoutOff`. When the driver detects that `FractEnable` has been turned on, line layout is not performed. The driver assumes that all text being printed is already spaced correctly for the LaserWriter and just sends it as is.

Polygons

The polygon comments are recognized by the LaserWriter driver because they are used by MacDraw as an alternate method of defining polygons.

The PolyBegin and PolyEnd comments bracket polygon line segments, giving an alternate way to specify a polygon. All StdLine calls between these two comments are part of the polygon. The endpoints of the lines are the vertices of the polygon.

The picPlyClo comment specifies that the current polygon should be closed. This comes immediately after PolyBegin, if at all. It is not sufficient to simply check for begPt = endPt, since MacDraw allows you to create a "closed" polygon that isn't really closed. This comment is especially critical for smooth curves because it can make the difference between having a sharp corner or not in the curve.

These comments also work with the StdPoly call. If a FillRgn is encountered before the PolyEnd comment, then the polygon is filled. Unlike QuickDraw polygons, comment polygons do not require an initial MoveTo call within the scope of the polygon comment. The polygon will be drawn using the current pen location at the time the polygon comment is received. The pen must be set before the polygon comment is called.

Splines

A spline is a method used to determine the smallest number of points that define a curve. In MacDraw, splines are used as a method for smoothing polygons. The vertices of the underlying unsmoothed polygon are the control nodes for the quadratic B-spline curve which is drawn. PostScript has a direct facility for cubic B-splines and the LaserWriter translates the quadratic B-spline nodes it gets into the appropriate nodes for a cubic B-spline that will exactly emulate the original quadratic B-spline.

The PolySmooth comment specifies that the current polygon should be smoothed. This comment also contains data that provides a means of specifying which verbs to use on the smoothed polygon (bits 7 through 3 are not currently assigned):

```
TPolyVerb = PACKED RECORD
   f7, f6, f5, f4, f3, fPolyClose, fPolyFill, fPolyframe : Boolean;
END; { TPolyVerb }
```

Although the closing information is redundant with the picPlyClo comment, it is included for the convenience of the LaserWriter.

The LaserWriter uses the pen size at the time the PolyBegin comment is received to frame the smoothed polygon if framing is called for by the TPolyVerb information. When the PolyIgnore comment is received by the LaserWriter driver, all further StdLine calls are ignored until the PolyEnd comment is encountered. For polygons that are to be smoothed, set the initial pen width to zero after the PolyBegin comment so that the unsmoothed polygon will not be drawn by other printers not equipped to handle polygon comments. To fill the polygon, call StdRgn with the fill verb and the appropriate pattern set, as well as specifying fill in the PolySmooth comment.

Lines

The DashedLine and DashedLineStop comments are used to communicate PostScript information for drawing dashed lines.

The DashedLine comment contains the following additional data:

```
  TDashedLine = PACKED RECORD
    offset:   SignedByte;                 {Offset as specified by PostScript}
    centered: SignedByte;                 {Whether dashed line should be
                                           centered to begin and end points}
    dashed:   Array[0..1] of SignedByte;  {1st byte is # bytes following}
  END; { TDashedLine }
```

The printer driver sets up the PostScript dashed line command, as defined on page 214 of Adobe's *PostScript Language Reference Manual*, using the parameters specified in the comment. You can specify that the dashed line be centered between the begin and end points of the lines by making the centered field nonzero.

The SetLineWidth comment allows you to set the pen width of all subsequent objects drawn. The additional data is a point. The vertical portion of the point is the numerator and the horizontal portion is the denominator of the scaling factor that the horizontal and vertical components of the pen are then multiplied by to obtain the new pen width. For example, if you have a pen size of 1,2 and in your line width comment you use 2 for the horizontal of the point and 7 for the vertical, the pen size will then be (7/2)*1 pixels wide and (7/2)*2 pixels high.

Below is an example of how to use the line comments:

```
PROCEDURE LineTest;
{This procedure shows how to do dashed lines and how to change the line width}
CONST
  DashedLine = 180;
  DashedStop = 181;
  SetLineWidth = 182;

TYPE
  DashedHdl = ^DashedPtr;
  DashedPtr = ^TDashedLine;
  TDashedLine = PACKED RECORD
    offset: SignedByte;
    Centered: SignedByte;
    dashed: Array[0..1] of SignedByte;    { the 0th element is the length }
  END; { TDashedLine }
  widhdl = ^widptr;
  widptr = ^widpt;
  widpt = Point;


VAR
  arect    : rect;
  Width    : Widhdl;
  dashedln : DashedHdl;
```

```
 BEGIN {LineTest}
   Dashedln := dashedhdl(NewHandle(sizeof(tdashedline)));
   Dashedln^^.offset := 0;         { No offset}
   Dashedln^^.centered := 0;       { don't center}
   Dashedln^^.dashed[0] := 1;      { this is the length }
   Dashedln^^.dashed[1] := 8;      { this means 8 points on, 8 points off }

   Width := widhdl(NewHandle(sizeof(widpt)));
   Width^^.h := 2;                 { denominator is 2}
   Width^^.v := 7;                 { numerator is 7}

   myPic := OpenPicture(theWorld);
     SetPen(1,2);                  { Set the pen size to 1 wide x 2 high }
     ClipRect(theWorld);
     MoveTo(20,20);
     DrawString('Do line test');
     PicComment(DashedLine,GetHandleSize(Handle(dashedln)),Handle(dashedln));
     PicComment(SetLineWidth,4,Handle(width));   {SetLineWidth}
     SetRect(arect,100,100,500,500);
     FrameRect(aRect);
     MoveTo(500,500);
     Lineto(100,100);
     PicComment(DashedStop,0,nil);               {DashedStop}
  ClosePicture;
  DisposHandle(handle(width));                   {Clean up}
  DisposHandle(handle(dashedln));
  PrintThePicture;                               {print it please}
  KillPicture(MyPic);
END; {LineTest}
```

PostScript

The PostScript comments tell the printer driver that the application is going to be communicating with the LaserWriter directly using PostScript commands instead of QuickDraw. The driver sends the accompanying PostScript to the printer with no preprocessing and no error checking. The application can specify data in the comment handle itself or point to another file which contains text to send to the printer. When the application is finished sending PostScript, the `PostScriptEnd` comment tells the printer driver to resume normal QuickDraw mode.

Any Quickdraw drawing commands made by the application between the `PostScriptBegin` and `PostScriptEnd` comments will be ignored by PostScript printers. In order to use PostScript in a device independent way, you should always include two representations of your document. The first representation should be a series of Quickdraw drawing commands. The second representation of your document should be a series of PostScript commands, sent to the Printing Manager via picture comments. This way, when you are printing to a PostScript device, the picture comments will be executed, and the Quickdraw commands ignored. When printing to a non-PostScript device, the picture comments will be ignored, and the Quickdraw commands will be executed. This method allows you to use PostScript, without having to ask the device if it supports it. This allows your application to get the best results with any printer, without being device dependent.

Here are some guidelines you need to remember:

- The graphic state set up during QuickDraw calls is maintained and is not affected by PostScript calls made with these comments.

- The header has changed a number of parameters so sometimes you won't get the results you expect. You may want to take a look at the header listed in *The LaserWriter Reference Manual* available through APDA.

- The header changes the PostScript coordinate system so that the origin is at the top-left corner of the page instead of at the bottom-left corner. This is done so that the QuickDraw coordinates that are used don't have to be remapped into the standard PostScript coordinate system. If you don't allow for this, all drawing is printed upside down. Please see the *PostScript Language Reference Manual* for details about transformation matrices.

- Don't call `showpage`. This is done for you by the driver. If you do, you won't be able to switch back to QuickDraw mode and an additional page will be printed when you call `PrClosePage`.
- Don't call `exitserver`. You may get very strange results.
- Don't call `initgraphics`. Graphics states are already set up by the header.

- Don't do anything that you expect to live across jobs.

- You won't be able to interrogate the printer to get information back through the driver.

The PostScriptBegin comment sets the driver state to prepare for the generation of PostScript by the application by calling gsave to save the current state. PostScript is then sent to the printer by using comments 192 through 195. The QuickDraw state of the driver is then restored by the PostScriptEnd comment. All QuickDraw operations that occur outside of these comments are performed; no clipping occurs as with the text rotation comments.

PostScript From a Text Handle

When the PostScriptHandle comment is used, the handle PSData points to the PostScript commands which are sent. PSData is a generic handle that points to text, without a length byte. The text is terminated by a carriage return. This comment is terminated by a PostScriptEnd comment.

Note: Due to a bug in the 3.1 LaserWriter driver, PostScriptEnd will not restore the QuickDraw state after the use of a PostScriptHandle comment. The workaround is to only use this comment at the end of your drawing, after you have made all the QuickDraw calls you need. This problem is fixed in more recent versions of the driver.

Here's an example of how to use this comment:

```
PROCEDURE PostHdl;
{this procedure shows how to use PostScript from a text Handle}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  PostScriptHandle = 192;

VAR
  MyString   : Str255;
  tempstr    : String[1];
  MyHandle   : Handle;
  err        : OSErr;

BEGIN { PostHdl }
  MyString := '/Times-Roman findfont 12 scalefont setfont 230 600 moveto
               (Hello World) show';
  tempstr:=' ';
  tempstr[1] := chr(13); {has to be terminated by a carriage return }
  MyString := Concat(MyString, tempstr); { in order for it to execute}
  err := PtrToHand (Pointer(ord(@myString)+1), MyHandle, length(MyString));
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('PostScript from a Handle');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(PostScriptHandle,length(mystring),MyHandle);
    PicComment(PostScriptEnd,0,nil);        {PostScript End}
  ClosePicture;
  DisposHandle(MyHandle);                   {Clean up}
  PrintThePicture;                          {print it please}
  KillPicture(MyPic);
END; { PostHdl }
```

Defining PostScript as QuickDraw Text

All QuickDraw text following the `TextIsPostScript` comment is sent as PostScript. No error checking is performed. This comment is terminated by a `PostScriptEnd` comment.

Here is an example:

```
PROCEDURE PostText;
{Shows how to use PostScript in strings in a QuickDraw picture}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  TextIsPostScript = 194;

BEGIN { PostTest }
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('TextIsPostScript Comment');
    PicComment(PostScriptBegin,0,nil);     {Begin PostScript}
    PicComment(TextIsPostScript,0,nil);    {following text is PostScript}
      DrawString('0 728 translate');       {move the origin and rotate the}
      DrawString('1 -1 scale');            {coordinate system}

      DrawString('newpath');
      DrawString('100 470 moveto');
      DrawString('500 470 lineto');
      DrawString('100 330 moveto');
      DrawString('500 330 lineto');
      DrawString('230 600 moveto');
      DrawString('230 200 lineto');
      DrawString('370 600 moveto');
      DrawString('370 200 lineto');
      DrawString('10 setlinewidth');
      DrawString('stroke');
      DrawString('/Times-Roman findfont 12 scalefont setfont');
      DrawString('230 600 moveto');
      DrawString('(Hello World) show');
    PicComment(PostScriptEnd,0,nil);       {PostScriptEnd}
  ClosePicture;
  PrintThePicture;                                     {print it please}
  KillPicture(MyPic);
END; { PostText }
```

PostScript From a File

The `PostScriptFile` and `ResourcePS` comments allow you to send PostScript to the printer from a resource file. Before these comments are described there are some restrictions you need to follow:

- Don't ever copy a picture containing these comments to the clipboard. If it is pasted into another application and the specified file or resource is not available, printing will be aborted and the user won't know what went wrong. This could be very confusing to a user. If you want the PostScript information to be available when printed from another application, use one of the other comments and include the information in the picture.
- Don't keep the PostScript in a separate file from the actual data file. If the data file ever gets moved without the PostScript file, when the picture is printed the data file may not be found and the print job will be aborted, again without the user knowing what went wrong. Keeping the data and PostScript in the same file will forestall many headaches for you and the user.

Now, a description of the comments:

The `PostScriptFile` comment tells the driver to use the POST type resources contained in the file `FileNameString`. `FileNameString` is declared as a `Str255`.

When this comment is encountered, the driver calls `OpenResFile` using the file name specified in `FileNameString`. It then calls `GetResource('POST',theID);` repeatedly, where `theID` begins at 501 and is incremented by one for each `GetResource` call. If the driver gets a `ResNotFound` error, it closes the specified resource file. If the first byte of the resource is a 3, 4, or 5 then the remaining data is sent and the file is closed.

The format of the POST resource is as follows: The IDs of the resources start at 501 and are incremented by one for each resource. Each resource begins with a 2 byte data field containing the data type in the first byte and a zero in the second. The possible values for the first byte are:

0 ignore the rest of this resource (a comment)
1 data is ASCII text
2 data is binary and is first converted to ASCII before being sent
3 AppleTalk end of file. The rest of the data, if there is any, is interpreted as ASCII text and will be sent after the EOF.
4 open the data fork of the current resource file and send the ASCII text there
5 end of the resource file

The second byte of the field must always be zero. Resources should be kept small, around 2K. Text and binary should not be mixed in the same resource. Make sure you include either a space or a return at the end of each PostScript string to separate it from the following command.

Here's an example:

```
PROCEDURE PostFile;
{This procedure shows how to use PostScript from a specified FILE}
CONST
  PostScriptBegin = 190;
  PostScriptFile = 193;
  PostScriptEnd = 191;

VAR
  MyString          : Str255;
  MyHandle          : Handle;
  err               : OSErr;

BEGIN   { PostFile }
  {You should never do this in a real program. This is only a test.}
  MyString := 'HardDisk:MPW:Print Examples:PSTestDoc';
  err := PtrToHand(pointer(MyString),MyHandle,length(MyString) + 1);
  MyPic := OpenPicture(theWorld);
  ClipRect(theWorld);
  MoveTo(20,20);
  DrawString('PostScriptFile Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(PostScriptFile,GetHandleSize(MyHandle),MyHandle);
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  MoveTo(50,50);
  DrawString('PostScriptEnd has terminated');
  ClosePicture;
  DisposHandle(MyHandle); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;    { PostFile }
```

Here are the resources:

```
type 'POST' {
  switch {
    case Comment:             /* this is a comment */
         key bitstring[8] = 0;
         fill byte;
         string;

    case ASCII:               /* this is just ASCII text */
         key bitstring[8] = 1;
         fill byte;
         string;

    case Bin:                 /* this is binary */
         key bitstring[8] = 2;
         fill byte;
         string;

    case ATEOF:               /* this is an AppleTalk EOF */
         key bitstring[8] = 3;
         fill byte;
         string;
```

```
          case DataFork:          /* send the text in the data fork */
               key bitstring[8] = 4;
               fill byte;

          case EOF:               /* no more */
               key bitstring[8] = 5;
               fill byte;
          };
     };

     resource 'POST' (501) {
     ASCII{"0 728 translate "}};

     resource 'POST' (502) {
     ASCII{"1 -1 scale "}};

     resource 'POST' (503) {
     ASCII{"newpath "}};

     resource 'POST' (504) {
     ASCII{"100 470 moveto "}};

     resource 'POST' (505) {
     ASCII{"500 470 lineto "}};

     resource 'POST' (506) {
     ASCII{"100 330 moveto "}};

     resource 'POST' (507) {
     ASCII{"500 330 lineto "}};

     resource 'POST' (508) {
     ASCII{"230 600 moveto "}};

     resource 'POST' (509) {
     ASCII{"230 200 lineto "}};

     resource 'POST' (510) {
     ASCII{"370 600 moveto "}};

     resource 'POST' (511) {
     ASCII{"370 200 lineto "}};

     resource 'POST' (512) {
     ASCII{"10 setlinewidth "}};

     resource 'POST' (513) {
     ASCII{"stroke "}};

     resource 'POST' (514) {
     ASCII{"/Times-Roman findfont 12 scalefont setfont "}};

     resource 'POST' (515) {
     ASCII{"230 600 moveto "}};

     resource 'POST' (516) {
     ASCII{"(Hello World) show "}};
```

```
/* It will stop reading and close the file after 517 */
 resource 'POST' (517) {
 EOF
 {}};

/* it never gets here */
 resource 'POST' (518) {
 DataFork
 {}};
```

When the `ResourcePS` comment is encountered, the LaserWriter driver sends the text contained in the specified resource as PostScript to the printer. The additional data is defined as

```
PSRsrc = RECORD
           PSType : ResType;
           PSID   : INTEGER;
           PSIndex: INTEGER;
         END;
```

The resource can be of type STR or STR#. If the Type is STR then the index should be 0. Otherwise an index should be given.

This comment is essentially the same as the PrintF control call to the driver. The imbedded command string it uses is `'^r^n'`, which basically tells the driver to send the string specified by the additional data, then send a newline. For more information about printer control calls see the *LaserWriter Reference Manual*.

Here's an example:

```
PROCEDURE PostRSRC;
{This procedure shows how to get PostScript from a resource FILE}
  CONST
    PostScriptBegin = 190;
    PostScriptEnd = 191;
    ResourcePS = 195;

  TYPE
    theRSRChdl = ^theRSRCptr;
    theRSRCptr = ^theRSRC;
    theRSRC = RECORD
       theType: ResType;
       theID: INTEGER;
       Index: INTEGER;
    END;

  VAR
    temp          : Rect;
    TheResource   : theRSRChdl;
    i,j           : INTEGER;
    myport        : GrafPtr;
    err           : INTEGER;
    atemp         : Boolean;
  BEGIN            { PostRSRC }
    TheResource := theRSRChdl(NewHandle(SizeOf(theRSRC)));
    TheResource^^.theID := 500;
    TheResource^^.Index := 0;
    TheResource^^.theType := 'STR ';
    HLock(Handle(TheResource));
    MyPic := OpenPicture(theWorld);
    DrawString('ResourcePS Comment');
    PicComment(PostScriptBegin,0,nil); {Begin PostScript}
    PicComment(ResourcePS,8,Handle(TheResource)); {Send postscript}
    PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
    ClosePicture;
    DisposHandle(Handle(TheResource)); {Clean up}
    PrintthePicture; {print it please}
    KillPicture(MyPic);
  END;             { PostRSRC }
```

Here's the resource:

```
resource 'STR ' (500)
{"0 728 translate 1 -1 scale newpath 100 470 moveto 500 470 lineto 100 330
moveto 500 330 lineto 230 600 moveto 230 200 lineto 370 600 moveto 370 200
lineto 10 setlinewidth stroke /Times-Roman findfont 12 scalefont setfont 230
600 moveto (Hello World) show"
};
```

Rotation

The concept of rotation doesn't apply to text alone. PostScript can rotate any object. The rotation comments work exactly like text rotation except that all objects drawn between the two comments are drawn in the rotated coordinate system specified by the center of rotation comment, not just text. Also, no clipping of CopyBits calls occurs. These comments only work on the 3.1 and newer LaserWriter drivers.

The RotateBegin comment tells the driver that the following objects will be drawn in a rotated plane. This comment contains the following data structure:

```
Rotation = RECORD
  Flip:  INTEGER; {0,1,2 => none, horizontal, vertical coordinate flip }
  Angle: INTEGER; {0..360 => clockwise rotation in degrees }
END; { Rotation }
```

When you are finished, the RotateEnd comment returns the coordinate system to normal, terminating the rotation.

The relative center of rotation is specified by the RotateCenter comment in exactly the same manner as the TextCenter comments. The difference, however, is that this comment **must** appear **before** the RotateBegin comment. The data structure of the accompanying handle is exactly like that for the TextCenter comment.

Here's an example of how to use rotation comments:

```
PROCEDURE Test;
{This procedure shows how to do rotations}
CONST
  RotateBegin = 200;
  RotateEnd = 201;
  RotateCenter = 202;

TYPE
    rothdl = ^rotptr;
    rotptr = ^trot;
    trot = RECORD
      flip : INTEGER;
      Angle : INTEGER;
    END; { trot }
    centhdl = ^centptr;
    centptr = ^cent;
    Cent = PACKED RECORD
             yInt: INTEGER;
             yFrac: INTEGER;
             xInt: INTEGER;
             xFrac: INTEGER;
    END; { Cent }

 VAR
   arect    : Rect;
   rotation : rothdl;
   center   : centhdl;
  BEGIN { Test }
    rotation := rothdl(NewHandle(sizeof(trot)));
    rotation^^.flip := 0;                                    {no flip}
    rotation^^.angle := 15;        {15 degree rotation}

    center := centhdl(NewHandle(sizeof(cent)));
    center^^.xInt := 50;                                     {center at 50,50}
    center^^.yInt := 50;
    center^^.xFrac := 0;                                     {no fractional part}
    center^^.yFrac := 0;

   myPic := OpenPicture(theWorld);
     ClipRect(theWorld);
     MoveTo(20,20);
     DrawString('Begin Rotation');

     {set the center of Rotation}
     PicComment(RotateCenter,GetHandleSize(Handle(center)),Handle(center));
     {Begin Rotation}
    PicComment(RotateBegin,GetHandleSize(Handle(rotation)),Handle(rotation));
     SetRect(arect,100,100,500,500);
     FrameRect(aRect);
     MoveTo(500,500);
     Lineto(100,100);
     PicComment(RotateEnd,0,nil);                  {RotateEnd}
   ClosePicture;
   DisposHandle(handle(rotation));                 {Clean up}
   DisposHandle(handle(center));
   PrintThePicture;                                                          {print
it please}
  KillPicture(MyPic);
END; { Test }
```

Forms

The two form printing comments allow you to prepare a template to use for printing. When the `FormsBegin` comment is used, the LaserWriter's buffer is not cleared after `PrClosePage`. This allows you to download a form then change it for each subsequent page, inserting the information you want. `FormsEnd` allows the buffer to be cleared at the next `PrClosePage`.

#92: The Appearance of Text

See also: The Printing Manager
The Font Manager
Technical Note #91—
Optimizing for the LaserWriter—Picture Comments

Written by: Ginger Jernigan

November 15, 1986
Updated: March 1, 1988

This technical note describes why text doesn't always look the way you expect depending on the environment you are in.

There are a number of Macintosh text editing applications where layout is critical. Unfortunately, text on a newer machine sometimes prints differently than text on a 64K ROM Macintosh. Let's examine some differences you should expect and why.

The differences we will consider here are only differences in the layout of text lines (line layout), not differences in the appearance of fonts or the differences between different printers. Differences in line layout may affect the position of line, paragraph and page breaks. The four variables that can affect line layout are fonts, the printer driver, the font manager mode, and ROMs.

Fonts

Every font on a Macintosh contains its own table of widths which tells QuickDraw how wide characters are on the screen. For every style point size there is a separate table which may contain widths that vary from face to face and from point size to point size. Character widths can vary between point sizes of characters even in the same face. In other words, fonts on the screen are not necessarily linearly scalable.

Non-linearity is not normally a problem since most fonts are designed to be as close to linear as possible. A font face in 6 point has very nearly the same scaled widths of the same font face in 24 point (plus or minus round-off or truncation differences). QuickDraw, however, requires only one face of any particular font to be in the System file to use it in any point size. If only a 10 point face actually exists, QuickDraw may scale that face to 9, 18, 24 (or whatever point size) by performing a linear scale of the 10 point face.

This can cause problems. Suppose a document is created on one Macintosh containing a font that only exists in that System file in one point size, say 9 point. The document is then taken to another Macintosh with a System file containing that same font but only in 24 point. The document may, in fact, appear differently on the two screens, and when it is printed, will have line breaks (and thus paragraph and page breaks) occurring in different places simply because of the differences in character widths that exist between the 9 point and 24 point faces.

The Printer Driver

Even when the printer you are using has a much higher resolution than what the screen can show, printer drivers perform line layout to match the screen layout as closely as possible.

The line layout performed by printer drivers is limited to single lines of text and does not change line break positions within multiple lines. The driver supplies metric information to the application about the page size and printable area to allow the application to determine the best place to make line and page breaks.

Printer driver line layout does affect word spacing, character spacing and even word positioning within a line. This may affect the overall appearance of text, particularly when font substitutions are made or various forms of page or text scaling are involved. But print drivers NEVER change line, paragraph or page break positions from what the application or screen specified. This means that where line breaks appear on the screen, they will always appear in the same place on the printer regardless of how the line layout may affect the appearance within the line.

Operating System and ROMs

In this context, operating system refers to the ROM trap routines which handle fonts and QuickDraw. Changes have occurred between the ROMs in the handling of fonts. Fonts in the 64K ROMs contain width tables (as described above) which are limited to integer values. Several new tables, however, have been added to fonts for the newer ROMs. The newer ROMs add an optional global width table containing fractional or fixed point decimal values. In addition, there is another optional table containing fractional values which can be scaled for the entire range of point sizes for any one face. There is also an optional table which provides for the addition (or removal) of width to a font when its style is changed to another value such as bold, outline or condensed. It is also possible, under the 128K ROMs, to add fonts to the system with inherent style properties containing their own width tables that produce different character widths from derived style widths.

One or all of the above tables may or may not be invoked depending on, first, their presence, and second, the mode of the operating system. The Font Manager in the newer ROMs allows the application to arbitrarily operate in either the fractional mode or integer mode (determined, in most cases, by the setting of `FractEnable`) as it chooses, with the default being integer. There is one case where fractional widths will be used if they exist even though fractional mode is disabled. When `FScaleDisable` is used fractional widths are always used if they exist regardless of the setting of `FractEnable`.

Differences in line layout (and thus line breaks) may be affected by any combination of the presence or absence of the optional tables, and the operating mode, either fractional or integer, of the application. Any of the combinations can produce different results from the original ROMs (and from each other).

The integer mode on the newer ROMs is very similar to, but not exactly the same as, the original 64K ROMs. When fonts with the optional tables present are used on Macintoshes with 64K ROMs, they continue to work in the old way with the integer widths. However, on newer ROMs, even in the integer mode, there may be variations in line width from what is seen on the old ROMs. In the plain text style there is very little if any difference (except if the global width table is present), but as various type styles are selected, line widths may vary more between ROMs.

Variations in the above options, by far, account for the greatest variation in the appearance of lines when a document is transported between one Macintosh and another. Line breaks may change position when documents created on one system (say a Macintosh) are moved to another system (like a Macintosh Plus). Variations are more pronounced as the number and sizes of various type styles increase within a document.

In all cases, however, a printer driver will produce exactly the same line breaks as appear on the screen with any given system combination.

#93: MPW: {$LOAD}; _DataInit;%_MethTables

See also: MPW Reference Manuals

Written by: Jim Friedlander November 15, 1986
Modified by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note discusses the Pascal `{$LOAD}` directive as well as how to unload the `_DataInit` and `%_MethTables` segments.

{$LOAD}

MPW Pascal has a `{$LOAD}` directive that can dramatically speed up compiles.

```
{$LOAD HD:MPW:PLibraries:PasSymDump}
```

will combine symbol tables of all units following this directive (until another `{$LOAD}` directive is encountered), and dump them out to `HD:MPW:PLibraries:PasSymDump`. In order to avoid using fully specified pathnames, you can use `{$LOAD}` in conjunction with the `-k` option for Pascal:

```
Pascal -k "{PLibraries}" myfile
```

combined with the following lines in `myfile`:

```
USES
    {$LOAD PasSymDump}
        MemTypes,QuickDraw, OSIntf, ToolIntf, PackIntf,
    {$LOAD} {This "turns off" $LOAD for the next unit}
        NonOptimized,
    {$LOAD MyLibDump}
        MyLib;
```

will do the following: the first time a program containing these lines is compiled, two symbol table dump files (in this case `PasSymDump` and `MyLibDump`) will be created in the directory specified by the `-k` option (in this case `{PLibraries}`). No dump file will be generated for the unit `NonOptimized`. The compiler will compile `MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf` (quite time consuming) and dump those units' symbols to `PasSymDump` and it will compile the interface to `MyLib` and dump its symbols to `MyLib`. For subsequent compiles of this program (or any program that uses the same dump file(s)), the interface files won't be recompiled, the compiler will simply read in the symbol table.

Compiling a sample five line program on a Macintosh Plus/HD20SC takes 62 seconds without using the `{$LOAD}` directive. The same program takes 10 seconds to compile using the `{$LOAD}` directive (once the dump file exists). For further details about this topic, please see the *MPW Pascal Reference Manual*.

Note: If any of the units that are dumped into a dump file change, you need to make sure that the dump file is deleted, so that it can be regenerated by the Pascal compiler with the correct information. The best way to do this is to use a makefile to check the dump file against the files it depends on, and delete the dump file if it is out of date with respect to any of the units that it contains. An excellent (and well commented) example of doing this is in the *MPW Workshop Manual.*

The _DataInit Segment

The Linker will generate a segment whose resource name is `%A5Init` for any program compiled by the C or Pascal compilers. This segment is called by a program's main segment. This segment is loaded into the application heap and locked in place. It is up to your program to unload this segment (otherwise, it will remain locked in memory, possibly causing heap fragmentation). To do this from Pascal, use the following lines:

```
PROCEDURE _DataInit;EXTERNAL;
...

BEGIN          {main PROGRAM}
UnloadSeg(@_DataInit);
{remove data initialization code before any allocations}
...
```

From C, use the following lines:

```
extern _DataInit();
...
{ /* main */
                UnloadSeg(_DataInit);
                /*remove data initialization code before any allocations*/
...
```

For further details about Data Initialization, see the *MPW Reference Manual.*

%_MethTables and %_SelProcs

Object use in Pascal produces two segments which can cause heap problems. These are `%_MethTables` and `%_SelProcs` which are used when method calls are made. MacApp deals with them correctly, so this only applies to Object Pascal programs that don't use MacApp. You can make the segments locked and preloaded (probably the easiest route), so they will be loaded low in the heap, or you can unload them temporarily while you are doing heap initialization. In the latter case, make sure there are no method calls while they are unloaded. To reload `%_MethTables` and `%_SelProcs`, call the dummy procedure `%_InitObj`. `%_InitObj` loads `%MethTables` —calling any method will then load `%_SelProcs`.

Reminder: The linker is case sensitive when dealing with module names. Pascal converts all module names to upper-case (unless a routine is declared to be a C routine). The Assembler default is the same as the Pascal default, though it can be changed with the CASE directive. C preserves the case of module names (unless a routine is declared to be `pascal`, in which case the module name is converted to upper-case letters).

Make sure that any external routines that you reference are capitalized the same in both the external routine and the external declaration (especially in C). If the capitalization differs, you will get the following link error (library routine = `findme`, program declaration = `extern FindMe();`):

```
### Link: Error  Undefined entry, name: FindMe
```

Macintosh Technical Notes

#94: Tags

See also: The File Manager

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

Apple has decided to eliminate support for file-system tags on its future products; this technical note explains this decision.

Some of Apple's disk products (and some third-party products) have the ability to store 532 bytes per sector, instead of the normal 512. Twelve of the extra bytes are used to store redundant file system information, known as "tags", to be used by a scavenging utility to reconstruct damaged disks.

Apple has decided to eliminate support for these tags on its products; this was decided for several reasons:

1) Tags were implemented back when we had to deal with "Twiggy" drives on Lisa. These drives were less reliable than current drives, and it was expected that tags would be needed for data integrity.

2) We're working on a scavenging utility (Disk First Aid), and we've found that tags don't help us in reconstructing damaged disks (ie, if we can't fix it without using tags, tags wouldn't help us fix it). So, at least the first two versions of our scavenging utility will not use tags, and a third version (which we've planned for, but will probably never implement) can probably work without them.

3) 532-byte-per-sector drives and controllers tend to cost more, even at Apple's volumes. Thus, the demise of tags saves us (and our customers) money. The Apple Hard Disk 20SC currently supports tags; this may not always be the case, however; we'll probably drop the large sectors when we run out of our current stock of drives.

The Hierarchical File System (HFS) documentation didn't talk about tags because the writer had no information available about how they worked under HFS. Because of this decision, it is unlikely that we'll ever have documentation on how to correctly implement them under HFS.

#95: How To Add Items to the Print Dialogs

See also: The Printing Manager
The Dialog Manager

Written by: Ginger Jernigan
Lew Rollins

November 15, 1986

Updated: March 1, 1988

This technical note discusses how to add your own items to the Printing Manager's dialogs.

When the Printing Manager was initially designed, great care was taken to make the interface to the printer drivers as generic as possible in order to allow applications to print without being device-specific. There are times, however, when this type of non-specific interface interferes with the flexibility of an application. An application may require additional information before printing which is not part of the general Printing Manager interface. This technical note describes a method that an application can use to add its own items to the existing style and job dialogs.

Before continuing, you need to be aware of some guidelines that will increase your chances of being compatible with the printing architecture in the future:

- Only add items to the dialogs as described in this technical note. Any other methods will decrease your chances of survival in the future.

- Do not change the position of any item in the current dialogs. This means don't delete items from the existing item list or add items in the middle. Add items **only at the end** of the list.

- Don't count on an item retaining its current position in the list. If you depend on the Draft button being a particular number in the ImageWriter's style dialog item list, and we change the Draft button's item number for some reason, your program may no longer function correctly.

- Don't use more than half the screen height for your items. Apple reserves the right to expand the items in the standard print dialogs to fill the top half of the screen.

- If you are adding lots of items to the dialogs (which may confuse users), you should consider having your own separate dialog in addition to the existing Printing Manager dialogs.

The Heart

Before we talk about how the dialogs work, you need to know this: at the heart of the printer dialogs is a little-known data structure partially documented in the MacPrint interface file. It's a record called TPrDlg and it looks like this:

```
TPrDlg = RECORD   {Print Dialog: The Dialog Stream object.}
   dlg         : DialogRecord;    {dialog window}
   pFltrProc   : ProcPtr;         {filter proc.}
   pItemProc   : ProcPtr;         {item evaluating proc.}
   hPrintUsr   : THPrint;         {user's print record.}
   fDoIt       : BOOLEAN;
   fDone       : BOOLEAN;
   lUser1      : LONGINT;         {four longs reserved by Apple}
   lUser2      : LONGINT;
   lUser3      : LONGINT;
   lUser4      : LONGINT;
   iNumFst     : INTEGER;         {numeric edit items for std filter}
   iNumLst     : INTEGER;
{... plus more stuff needed by the particular printing dialog.}
END;
TPPrDlg = ^TPrDlg;                {== a dialog ptr}
```

All of the information pertaining to a print dialog is kept in the TPrDlg record. This record will be referred to frequently in the discussion below.

How the Dialogs Work

When your application calls PrStlDialog and PrJobDialog, the printer driver actually calls a routine called PrDlgMain. This function is declared as follows:

```
FUNCTION PrDlgMain (hprint: THPrint; pDlgInit: ProcPtr): BOOLEAN;
```

PrDlgMain first calls the pDlgInit routine to set up the appropriate dialog (in Dlg), dialog hook (pItemProc) and dialog event filter (pFilterProc) in the TPrDlg record (shown above). For the job dialog, the address of PrJobInit is passed to PrDlgMain. For the style dialog, the address of PrStlInit is passed. These routines are declared as follows:

```
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
```

After the initialization routine sets up the TPrDlg record, PrDlgMain calls ShowWindow (the window is initially invisible), then it calls ModalDialog, using the dialog event filter pointed to by the pFltrProc field. When an item is hit, the routine pointed to by the pItemProc field is called and the items are handled appropriately. When the OK button is hit (this includes pressing Return or Enter) the print record is validated. The print record is not validated if the Cancel button is hit.

How to Add Your Own Items

To modify the print dialogs, you need to change the `TPrDlg` record before the dialog is drawn on the screen. You can add your own items to the item list, replace the addresses of the standard dialog hook and event filter with the addresses of your own routines and then let the dialog code continue on its merry way.

For example, to modify the job dialog, first call `PrJobInit`. `PrJobInit` will fill in the `TPrDlg` record for you and return a pointer to that record. Then call `PrDlgMain` directly, passing in the address of your own initialization function. The example code's initialization function adds items to the dialog item list, saves the address of the standard dialog hook (in our global variable `prPItemProc`) and puts the address of our dialog hook into the `pItemProc` field of the `TPrDlg` record. Please note that your dialog hook **must** call the standard dialog hook to handle all of the standard dialog's items.

Note: If you wish to have an event filter, handle it the same way that you do a dialog hook.

Now, here is an example (written in MPW Pascal) that modifies the job dialog. The same code works for the style dialog if you globally replace 'Job' with 'Stl'. Also included is a function (`AppendDITL`) provided by Lew Rollins (originally written in C, translated for this technical note to MPW Pascal) which demonstrates a method of adding items to the item list, placing them in an appropriate place, and expanding the dialog window's rectangle.

The MPW Pascal Example Program

```
PROGRAM ModifyDialogs;

 USES
  {$LOAD PasDump.dump}
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint;

 CONST
  MyDITL     = 256;
  MyDFirstBox  = 1;     {Item number of first box in my DITL}
  MyDSecondBox = 2;

 VAR
  PrtJobDialog: TPPrDlg;  { pointer to job dialog }
  hPrintRec  : THPrint;  { Handle to print record }
  FirstBoxValue,       { value of our first additional box }
  SecondBoxValue: Integer; { value of our second addtl. box }
  prFirstItem,        { save our first item here }
  prPItemProc : LongInt;  { we need to store the old itemProc here }
  itemType   : Integer;  { needed for GetDItem/SetDItem calls }
  itemH    : Handle;
  itemBox   : Rect;
  err     : OSErr;
  {-----------------------------------------------------------------------}

PROCEDURE _DataInit;
 EXTERNAL;
 {-----------------------------------------------------------------------}

 PROCEDURE CallItemHandler(theDialog: DialogPtr; theItem: Integer; theProc:
LongInt);
   INLINE $205F,$4E90;     { MOVE.L (A7)+,A0
                             JSR (A0)          }

{ this code pops off theProc and then does a JSR to it, which puts the
 real return address on the stack. }

 {-----------------------------------------------------------------------}

 FUNCTION AppendDITL(theDialog: DialogPtr; theDITLID: Integer): Integer;
 { version 0.1 9/11/86 Lew Rollins of Human-Systems Interface Group}
 { this routine still needs some error checking }

{ This routine appends all of the items of a specified DITL
onto the end of a specified DLOG - We don't even need to know the format
of the DLOG }

{ this will be done in 3 steps:
 1. append the items of the specified DITL onto the existing DLOG
 2. expand the original dialog window as required
 3. return the adjusted number of the first new user item
}
   TYPE
    DITLItem    = RECORD { First, a single item }
            itmHndl: Handle; { Handle or procedure pointer for this item }
            itmRect: Rect; { Display rectangle for this item }
            itmType: SignedByte; { Item type for this item - 1 byte }
            itmData: ARRAY [0..0] OF SignedByte; { Length byte of data }
          END;  {DITLItem}

    pDITLItem   = ^DITLItem;
    hDITLItem   = ^pDITLItem;

    ItemList    = RECORD { Then, the list of items }
            dlgMaxIndex: Integer; { Number of items minus 1 }
            DITLItems: ARRAY [0..0] OF DITLItem; { Array of items }
          END;  {ItemList}

    pItemList   = ^ItemList;
    hItemList   = ^pItemList;

    IntPtr    = ^Integer;

   VAR
    offset   : Point;  { Used to offset rectangles of items being appended }
    maxRect   : Rect;  { Used to track increases in window size }
    hDITL    : hItemList; { Handle to DITL being appended }
    pItem    : pDITLItem; { Pointer to current item being appended }
    hItems   : hItemList; { Handle to DLOG's item list }
    firstItem : Integer; { Number of where first item is to be appended }
    newItems,       { Count of new items }
    dataSize,       { Size of data for current item }
    i      : Integer; { Working index }
    USB     : RECORD  {we need this because itmData[0] is unsigned}
            CASE Integer OF
```

```
               1:
                 (SBArray: ARRAY [0..1] OF SignedByte);
               2:
                 (Int: Integer);
            END;   {USB}

   BEGIN          {AppendDITL}
 {
  Using the original DLOG

  1. Remember the original window Size.
  2. Set the offset Point to be the bottom of the original window.
  3. Subtract 5 pixels from bottom and right, to be added
     back later after we have possibly expanded window.
  4. Get working Handle to original item list.
  5. Calculate our first item number to be returned to caller.
  6. Get locked Handle to DITL to be appended.
  7. Calculate count of new items.
 }
    maxRect := DialogPeek(theDialog)^.window.port.portRect;
    offset.v := maxRect.bottom;
    offset.h := 0;
    maxRect.bottom := maxRect.bottom - 5;
    maxRect.right := maxRect.right - 5;
    hItems := hItemList(DialogPeek(theDialog)^.items);
    firstItem := hItems^^.dlgMaxIndex + 2;
    hDITL := hItemList(GetResource('DITL',theDITLID));
    HLock(Handle(hDITL));
    newItems := hDITL^^.dlgMaxIndex + 1;
{
 For each item,
  1. Offset the rectangle to follow the original window.
  2. Make the original window larger if necessary.
  3. fill in item Handle according to type.
}

    pItem := @hDITL^^.DITLItems;
    FOR i := 1 TO newItems DO BEGIN
     OffsetRect(pItem^.itmRect,offset.h,offset.v);
     UnionRect(pItem^.itmRect,maxRect,maxRect);

     USB.Int := 0;    {zero things out}
     USB.SBArray[1] := pItem^.itmData[0];

     { Strip enable bit since it doesn't matter here. }
     WITH pItem^ DO
      CASE BAND(itmType,$7F) OF
       userItem:  { Can't do anything meaningful with user items. }
        itmHndl := NIL;
       ctrlItem + btnCtrl,ctrlItem + chkCtrl,ctrlItem + radCtrl:{build Control }
        itmHndl := Handle(NewControl(theDialog, { theWindow }
                    itmRect, { boundsRect }
                    StringPtr(@itmData[0])^, { title }
                    true, { visible }
                    0,0,1, { value, min, max }
                    BAND(itmType,$03), { procID }
                    0)); { refCon }
       ctrlItem + resCtrl: BEGIN { Get resource based Control }
      itmHndl := Handle(GetNewControl(IntPtr(@itmData[1])^, { controlID }
                       theDialog)); { theWindow }
      ControlHandle(itmHndl)^^.contrlRect := itmRect; {give it the right
                               rectangle}
      {An actionProc for a Control should be installed here}
     END;      {Case ctrlItem + resCtrl}
     statText,editText: { Both need Handle to a copy of their text. }
      err := PtrToHand(@itmData[1], { Start of data }
                 itmHndl, { Address of new Handle }
                 USB.Int); { Length of text }
     iconItem:  { Icon needs resource Handle. }
      pItem^.itmHndl := GetIcon(IntPtr(@itmData[1])^); { ICON resID }
     picItem:   { Picture needs resource Handle. }
      pItem^.itmHndl := Handle(GetPicture(IntPtr(@itmData[1])^));{PICT resID}
     OTHERWISE
      itmHndl := NIL;
    END;      {Case}

   dataSize := BAND(USB.Int + 1,$FFFE);
   {now advance to next item}
   pItem := pDITLItem(Ptr(ord4(@pItem^) + dataSize + sizeof(DITLItem)));
  END;         {for}
  err := PtrAndHand
   (@hDITL^^.DITLItems,Handle(hItems),GetHandleSize(Handle(hDITL)));
  hItems^^.dlgMaxIndex := hItems^^.dlgMaxIndex + newItems;
  HUnlock(Handle(hDITL));
  ReleaseResource(Handle(hDITL));
  maxRect.bottom := maxRect.bottom + 5;
  maxRect.right := maxRect.right + 5;
  SizeWindow(theDialog,maxRect.right,maxRect.bottom,true);
  AppendDITL := firstItem;
 END;          {AppendDITL}

{-------------------------------------------------------------------}

PROCEDURE MyJobItems(theDialog: DialogPtr; itemNo: Integer);
{
This routine replaces the routine in the pItemProc field in the
TPPrDlg record.  The steps it takes are:
1. Check to see if the item hit was one of ours. This is done by "localizing"
   the number, assuming that our items are numbered from 0..n
2. If it's one of ours  then case it and Handle appropriately
3. If it isn't one of ours then call the old item handler
}

 VAR
  MyItem,firstItem: Integer;
  thePt    : Point;
  thePart   : Integer;
  theValue   : Integer;
  debugPart  : Integer;

 BEGIN          {MyJobItems}
  firstItem := prFirstItem; { remember, we saved this in myJobDlgInit }
  MyItem := itemNo - firstItem + 1; { "localize" current item No }
  IF MyItem > 0 THEN BEGIN { if localized item > 0, it's one of ours }
   { find out which of our items was hit }
   GetDItem(theDialog,itemNo,itemType,itemH,itemBox);
```

```
     CASE MyItem OF
      MyDFirstBox: BEGIN
       { invert value of FirstBoxValue and redraw it }
       FirstBoxValue := 1 - FirstBoxValue;
       SetCtlValue(ControlHandle(itemH),FirstBoxValue);
      END;        {case MyDFirstBox}
      MyDSecondBox: BEGIN
       { invert value of SecondBoxValue and redraw it }
       SecondBoxValue := 1 - SecondBoxValue;
       SetCtlValue(ControlHandle(itemH),SecondBoxValue);
      END;        {case MyDSecondBox}
      OTHERWISE
       Debug;     { OH OH — We got an item we didn't expect }
     END;         {Case}
    END          { if MyItem > 0 }
    ELSE        { chain to standard item handler, whose address is saved
                 in prPItemProc }
     CallItemHandler(theDialog,itemNo,prPItemProc);
   END;           { MyJobItems }

  {----------------------------------------------------------------------}

  FUNCTION MyJobDlgInit(hPrint: THPrint): TPPrDlg;
 {
 This routine appends items to the standard job dialog and sets up the
 user fields of the printing dialog record TPRDlg
 This routine will be called by PrDlgMain
 This is what it does:
 1. First call PrJobInit to fill in the TPPrDlg record.
 2. Append our items onto the old DITL. Set them up appropriately.
 3. Save the address of the old item handler and replace it with ours.
 4. Return the Fixed dialog to PrDlgMain.
 }

  VAR
   firstItem : Integer; { first new item number }

  BEGIN          {MyJobDlgInit}
   firstItem := AppendDITL(DialogPtr(PrtJobDialog),MyDITL);

   prFirstItem := firstItem; { save this so MyJobItems can find it }

   { now we'll set up our DITL items — The "First Box" }
   GetDItem(DialogPtr(PrtJobDialog),firstItem,itemType,itemH,itemBox);
   SetCtlValue(ControlHandle(itemH),FirstBoxValue);

   { now we'll set up the second of our DITL items — The "Second Box" }
   GetDItem(DialogPtr(PrtJobDialog),firstItem + 1,itemType,itemH,itemBox);
   SetCtlValue(ControlHandle(itemH),SecondBoxValue);

{ Now comes the part where we patch in our item handler.  We have to save
 the old item handler address, so we can call it if one of the standard
 items is hit, and put our item handler's address
 in pItemProc field of the TPrDlg struct}

   prPItemProc := LongInt(PrtJobDialog^.pItemProc);

   { Now we'll tell the modal item handler where our routine is }
```

```
    PrtJobDialog^.pItemProc := ProcPtr(@MyJobItems);

    { PrDlgMain expects a pointer to the modified dialog to be returned.... }
    MyJobDlgInit := PrtJobDialog;

   END;          {myJobDlgInit}

 {-----------------------------------------------------------------------}

 FUNCTION Print: OSErr;

   VAR
    bool    : BOOLEAN;

   BEGIN          {Print}
    hPrintRec := THPrint(NewHandle(sizeof(TPrint)));
    PrintDefault(hPrintRec);
    bool := PrValidate(hPrintRec);
    IF (PrError <> noErr) THEN BEGIN
     Print := PrError;
     Exit(Print);
    END;          {If}

    { call PrJobInit to get pointer to the invisible job dialog }
    PrtJobDialog := PrJobInit(hPrintRec);
    IF (PrError <> noErr) THEN BEGIN
     Print := PrError;
     Exit(Print);
    END;          {If}

 {Here's the line that does it all!}
    IF NOT (PrDlgMain(hPrintRec,@MyJobDlgInit)) THEN BEGIN
     Print := cancel;
     Exit(Print);
    END;          {If}

    IF PrError <> noErr THEN Print := PrError;

    { that's all for now }

   END;          { Print }

 {-----------------------------------------------------------------------}

  BEGIN          {PROGRAM}

   UnloadSeg(@_DataInit);  {remove data initialization code before any
allocations}
   InitGraf(@thePort);
   InitFonts;
   FlushEvents(everyEvent,0);
   InitWindows;
   InitMenus;
   TEInit;
   InitDialogs(NIL);
   InitCursor;

   { call the routine that does printing }
   FirstBoxValue := 0;    { value of our first additional box }
   SecondBoxValue := 0;    { value of our second addtl. box }
   PrOpen;    { Open the Print Manager }
   IF PrError = noErr THEN
    err := Print     { This actually brings up the modified Job dialog }
   ELSE BEGIN
    {tell the user that PrOpen failed}
   END;

   PrClose;    { Close the Print Manager and leave }
  END.
```

The Lightspeed C Example Program

```
/* NOTE: Apple reserves the top half of the screen (where the current DITL
      items are located). Applications may use the bottom half of the
      screen to add items, but should not change any items in the top half
      of the screen.  An application should expand the print dialogs only
      as much as is absolutely necessary.
*/

/* Note: A global search and replace of 'Job' with 'Stl' will produce
      code that modifies the style dialogs */
#include <DialogMgr.h>
#include <MacTypes.h>
#include <Quickdraw.h>
#include <ResourceMgr.h>
#include <WindowMgr.h>
#include <pascal.h>
#include <printmgr.h>
#define nil 0L


static TPPrDlg PrtJobDialog;        /* pointer to job dialog */

/*    This points to the following structure:

         struct {
               DialogRecord    Dlg;          (The Dialog window)
               ProcPtr         pFltrProc;    (The Filter Proc.)
               ProcPtr         pItemProc;    (The Item evaluating proc. --
                                             we'll change this)
               THPrint         hPrintUsr;    (The user's print record.)
               Boolean         fDoIt;
               Boolean         fDone;
                   (Four longs -- reserved by Apple Computer)
               long                   lUser1;
               long                   lUser2;
               long                   lUser3;
               long                   lUser4;
    } TPrDlg; *TPPrDlg;
*/



/*    Declare 'pascal' functions and procedures */
pascal Boolean PrDlgMain();          /* Print manager's dialog handler */
pascal TPPrDlg PrJobInit();          /* Gets standard print job dialog. */
pascal TPPrDlg MyJobDlgInit();       /* Our extention to PrJobInit */
pascal void MyJobItems();            /* Our modal item handler */

#define MyDITL 256                   /* resource ID of my DITL to be spliced
                                         on to job dialog */


THPrint hPrintRec;                   /* handle to print record */
short FirstBoxValue = 0;             /* value of our first additional box */
short SecondBoxValue = 0;            /* value of our second addtl. box */
long prFirstItem;                    /* save our first item here */
long prPItemProc;                /* we need to store the old itemProc here */
```

```
/*-----------------------------------------------------------------------*/
      WindowPtr   MyWindow;
      OSErr       err;
      Str255      myStr;
main()
{
      Rect        myWRect;

      InitGraf(&thePort);
      InitFonts();
      InitWindows();
      InitMenus();
      InitDialogs(nil);
      InitCursor();
      SetRect(&myWRect,50,260,350,340);

      /* call the routine that does printing */
      PrOpen();
      err = Print();

      PrClose();
} /* main */

/*-----------------------------------------------------------------------*
/

OSErr Print()

{
      /* call PrJobInit to get pointer to the invisible job dialog */
      hPrintRec = (THPrint)(NewHandle(sizeof(TPrint)));
      PrintDefault(hPrintRec);
      PrValidate(hPrintRec);
      if (PrError() != noErr)
            return PrError();

      PrtJobDialog = PrJobInit(hPrintRec);
      if (PrError() != noErr)
            return PrError();


      if (!PrDlgMain(hPrintRec, &MyJobDlgInit)) /* this line does all the
                                                  stuff */
            return Cancel;

      if (PrError() != noErr)
            return PrError();

/* that's all for now */

} /* Print */

/*-----------------------------------------------------------------------*
/

pascal TPPrDlg MyJobDlgInit (hPrint)
THPrint hPrint;
```

```
/* this routine appends items to the standard job dialog and sets up the
     user fields of the printing dialog record TPRDlg
     This routine will be called by PrDlgMain */
{
     short          firstItem;          /* first new item number */

     short          itemType;           /* needed for GetDItem/SetDItem call */
     Handle         itemH;
     Rect           itemBox;

     firstItem = AppendDITL (PrtJobDialog, MyDITL); /*call routine to do
                                                        this */

     prFirstItem = firstItem; /* save this so MyJobItems can find it */

/* now we'll set up our DITL items -- The "First Box" */
     GetDItem(PrtJobDialog,firstItem,&itemType,&itemH,&itemBox);
     SetCtlValue(itemH,FirstBoxValue);

/* now we'll set up the second of our DITL items  -- The "Second Box" */
     GetDItem(PrtJobDialog,firstItem+1,&itemType,&itemH,&itemBox);
     SetCtlValue(itemH,SecondBoxValue);

/* Now comes the part where we patch in our item handler.  We have to save
     the old item handler address, so we can call it if one of the
     standard items is hit, and put our item handler's address
     in pItemProc field of the TPrDlg struct
*/

     prPItemProc = (long)PrtJobDialog->pItemProc;

/* Now we'll tell the modal item handler where our routine is */
     PrtJobDialog->pItemProc = (ProcPtr)&MyJobItems;

/* PrDlgMain expects a pointer to the modified dialog to be returned.... */
     return PrtJobDialog;

} /*myJobDlgInit*/


/*-----------------------------------------------------------------------*/

/* here's the analogue to the SF dialog hook */

pascal void MyJobItems(theDialog,itemNo)
TPPrDlg     theDialog;
short       itemNo;

{ /* MyJobItems */
     short          myItem;
     short          firstItem;

     short          itemType;           /* needed for GetDItem/SetDItem call */
     Handle         itemH;
     Rect           itemBox;

     firstItem = prFirstItem; /* remember, we saved this in myJobDlgInit
*/
```

```
    myItem = itemNo-firstItem+1;  /* "localize" current item No */
    if (myItem > 0)   /* if localized item > 0, it's one of ours */
    {
        /* find out which of our items was hit */
        GetDItem(theDialog,itemNo,&itemType,&itemH,&itemBox);
        switch (myItem)
        {
            case 1:
                /* invert value of FirstBoxValue and redraw it */
                FirstBoxValue ^= 1;
                SetCtlValue(itemH,FirstBoxValue);
                break;

            case 2:
                /* invert value of SecondBoxValue and redraw it */
                SecondBoxValue ^= 1;
                SetCtlValue(itemH,SecondBoxValue);
                break;
            default: Debugger(); /* OH OH */
        } /* switch */
    } /* if (myItem > 0) */
    else /* chain to standard item handler, whose address is saved in
            prPItemProc */
    {
        CallPascal(theDialog,itemNo,prPItemProc);
    }
} /* MyJobItems */
```

The Rez Source

```
#include "types.r"

resource 'DITL' (256) {
 { /* array DITLarray: 2 elements */
  /* [1] */
  {8, 0, 24, 112},
  CheckBox {
   enabled,
   "First Box"
  };
  /* [2] */
  {8, 175, 24, 287},
  CheckBox {
   enabled,
   "Second Box"
  }
 }
};
```

#96: SCSI Bugs

See also: The SCSI Manager
SCSI Developer's Package

Written by: Steve Flowers October 1, 1986
Modified by: Bryan Stearns November 15, 1986
Modified by: Bo3b Johnson July 1, 1987
Updated: March 1, 1988

There are a number of problems in the SCSI Manager; this note lists the ones we know about, along with an explanation of what we're doing about them. Changes made for the 2/88 release are made to more accurately reflect the state of the SCSI Manager. System 4.1 and 4.2 are very similar; one bug was fixed in System 4.2.

There are several categories of SCSI Manager problems:

1. Those in the ROM boot code
(Before the System file has been opened, and hence, before any patches could possibly fix them.)
2. Those that have been fixed in System 3.2
3. Those that have been fixed in System 4.1/4.2
4. Those that are new in System 4.1/4.2
5. Those that have not yet been fixed.

The problems in the ROM boot code can only be fixed by changing the ROMs. Most of the bugs in the SCSI Manager itself have been fixed by the patch code in the System 3.2 file. There are a few problems, though, that are not fixed with System 3.2—most of these bugs have been corrected in System 4.1/4.2. Any that are not fixed will be detailed here. ROM code for future machines will, of course, include the corrections.

ROM boot code problems

- In the process of looking for a bootable SCSI device, the boot code issues a SCSI bus reset before each attempt to read block 0 from a device. If the read fails for any reason, the boot code goes on to the next device. SCSI devices which implement the `Unit Attention` condition as defined by the Revision 17B SCSI standard will fail to boot in this case. The read will fail because the drive is attempting to report the `Unit Attention` condition for the first command it receives after the SCSI bus reset. The boot code does not read the sense bytes and does not retry the failed command; it simply resets the SCSI bus and goes on to the next device.

If no other device is bootable, the boot code will eventually cycle back to the same SCSI device ID, reset the bus (causing `Unit Attention` in the drive again), and try to read block 0 (which fails for the same reason).

The 'new' Macintosh Plus ROMs that are included in the platinum Macintosh Plus have only one change. The change was to simply do a single SCSI Bus Reset after power up instead of a Reset each time through the SCSI boot loop. This was done to allow `Unit Attention` drives to be bootable. It was an object code patch (affecting approximately 30 bytes) and no other bugs were fixed. For details on the three versions of Macintosh Plus ROMs, see Technical Note #154.

We recommend that you choose an SCSI controller which does not require the `Unit Attention` feature—either an older controller (most of the SCSI controllers currently available were designed before Revision 17B), or one of the newer Revision-17B-compatible controllers which can enable/disable `Unit Attention` as a formatting option (such as those from Seagate, Rodime, et al). Since the vast majority of Macintosh Plus computers have the ROMs which cannot use `Unit Attention` drives, we still recommend that you choose an SCSI controller that does not require the `Unit Attention` feature.

- If an SCSI device goes into the `Status` phase after being selected by the boot code, this leads to the SCSI bus being left in the `Status` phase indefinitely, and no SCSI devices can be accessed. The current Macintosh Plus boot code does not handle this change to `Status` phase, which means that the presence of an SCSI device with this behavior (as in some tape controllers we've seen) will prevent any SCSI devices from being accessed by the SCSI Manager, even if they already had drivers loaded from them. The result is that any SCSI peripheral that is turned on at boot time must not go into `Status` phase immediately after selection; otherwise, the Macintosh Plus SCSI bus will be left hanging. Unless substantially revised ROMs are released for the Macintosh Plus (highly unlikely within the next year or so), this problem will never be fixed on the Macintosh Plus, so you should design for old ROMs.

- The Macintosh Plus would try to read 256 bytes of blocks 0 and 1, ignoring the extra data. The Macintosh SE and Macintosh II try to read 512 bytes from blocks 0 and 1, ignoring errors if the sector size is larger (but not smaller) than 512 bytes. Random access devices (disks, tapes, CD ROMS, etc.) can be booted as long as the blocks are at least 512 bytes, blocks 0, 1 and other partition blocks are correctly set up, and there is a driver on it. With the new partition layout (documented in *Inside Macintosh* volume V), more than 256 bytes per sector may be required in some partition map entries. This is why we dropped support for 256-byte sectors. Disks with tag bytes (532-byte sectors) or larger block sizes (1K, 2K, etc.) can be booted on any Macintosh with an SCSI port. Of course, the driver has to take care of data blocking and de-blocking, since HFS likes to work with 512-byte sectors.

Problems with ROM SCSI Manager routines

Note that the following problems are fixed after the System file has been opened; for a device to boot properly, it must not depend on these fixes. The sample SCSI driver, available from APDA, contains an example of how to find out if the fixes are in place.

- **Prior to System file 3.2,** blind transfers (both reads and writes) would not work properly with many SCSI controllers. Since blind operation depends on the drive's ability to transfer data fast enough, it is the responsibility of the driver writer to make sure blind operation is safe for a particular device.

- **Prior to System file 3.2,** the SCSI Manager dropped a byte when the driver did two or more `SCSIReads` or `SCSIRBlinds` in a row. (Each `Read` or `RBlind` has to have a Transfer Information Block (TIB) pointer passed in.) The TIB itself can be as big and complex as you want—it is the process of returning from one `SCSIRead` or `SCSIRBlind` and entering another one (while still on the same SCSI command) that causes the first byte for the other `SCSIReads` to be lost.

  Note that this precludes use of file-system tags. Apple no longer recommends that you support tags; see Technical Note #94 for more information.

- **Prior to System file 3.2,** `SCSIStat` didn't work; the new version works correctly.

- **Running under System file 3.2,** the SCSI Manager does not check to make sure that the last byte of a write operation (to the peripheral) was handshaked while operating in pseudo-DMA mode. The SCSI Manager writes the final byte to the NCR 5380's one-byte buffer and then turns pseudo-DMA mode off shortly thereafter (reported to be 10-15 microseconds). If the peripheral is somewhat slow in actually reading the last byte of data, it asserts `REQ` after the Macintosh has already turned off pseudo-DMA mode and never gets an `ACK`. The CPU then expects to go into the `Status` phase since it thinks everything went OK, but the peripheral is still waiting for `ACK`. Unless the driver can recover from this somehow, the SCSI bus is 'hung' in the `Data Out` phase. In this case, all successive SCSI Manager calls will fail until the bus is reset.

- **Running under System file 4.1/4.2,** the SCSI Manager waits for the last byte of a write operation to be handshaked while operating in pseudo-DMA mode; it checks for a final `DRQ` (or a phase change) at the end of a `SCSIWrite` or `SCSIWBlind` before turning off the pseudo-DMA mode. Drivers that could recover from this problem by writing the last byte again if the bus was still in a `Data Out` phase will still work correctly, as long as they were checking the bus state.

- **Running under System file 3.2,** the SCSI Manager does not time out if the peripheral fails to finish transferring the expected number of bytes for polled reads and writes. (Blind operation does poll for the first byte of each requested data transfer in the Transfer Information Block.)

- **Running under System file 4.1/4.2,** `SCSIRead` and `SCSIWrite` return an error to the caller if the peripheral changes the bus phase in the middle of a transfer, as might happen if the peripheral fails to transfer the expected number of bytes. The computer is no longer left in a hung state.

- **Running under System file 3.2,** the Selection timeout value is very short (900 microseconds). Patches to the SCSI Manager **in System 4.1/4.2** ensure that this value is the recommended 250 milliseconds.

- **Running under System file 3.2,** the SCSI Manager routine `SCSIGet` (which arbitrates for the bus) will fail if the `BSY` line is still asserted. Some devices are a bit slow in releasing `BSY` after the completion of an SCSI operation, meaning that `BSY` may not have been released before the driver issues a `SCSIGet` call to start the next SCSI operation. A work-around for this is to call `SCSIGet` again if it failed the first time. (Rarely has it been necessary to try it a third time.) This assumes, of course, that the bus has not been left 'hanging' by an improperly terminated SCSI operation before calling `SCSIGet`.

- **Running under System file 4.1/4.2,** the `SCSIGet` function has been made more tolerant of devices that are slow to release the `BSY` line after a SCSI operation. The SCSI Manager now waits up to 200 milliseconds before returning an error.

Problems with the SCSI Manager that haven't been fixed yet

These problems currently exist in the Macintosh Plus, SE, and II SCSI Manager. We plan to fix these problems in a future release of the System Tools disk, but in the mean time, you should try to work around the problems (but don't "require" the problems!).

- Multiple calls to `SCSIRead` or `SCSIRBlind` after issuing a command and before calling `SCSIComplete` may not work. Suppose you want to read some mode sense data from the drive. After sending the command with `SCSICmd`, you might want to call `SCSIRead` with a TIB that reads four bytes (typically a header). After reading the field (in the four-byte header) that tells how many remaining bytes are available, you might call `SCSIRead` again with a TIB to read the remaining bytes. The problem is that the first byte of the second `SCSIRead` data will be lost because of the way the SCSI Manager handles reads in pseudo-DMA mode. The work-around is to issue two separate SCSI commands: the first to read only the four-byte header, the second to read the four-byte header plus the remaining bytes. We recommend that you **not** use a clever TIB that contains two data transfers, the second of which gets the transfer length from the first transfer's received data (the header). These two step TIBs will not work in the future. This bug will probably not be fixed.

- On read operations, some devices may be slow in deasserting `REQ` after sending the last byte to the CPU. The current SCSI Manager (all machines) will return to the caller without waiting for `REQ` to be deasserted. Usually the next call that the driver would make is `SCSIComplete`. On the Macintosh SE and II, the `SCSIComplete` call will check the bus to be sure that it is in `Status` phase. If not, the SCSI Manager will return a new error code that indicates the bus was in Data In/Data Out phase when `SCSIComplete` was called. The combination of the speed of the Macintosh II and a

slow peripheral can cause `SCSIComplete` to detect that the bus is still in Data In phase before the peripheral has finally changed the bus to `Status` phase. This results in a false error being passed back by `SCSIComplete`.

- The `scComp` (compare) TIB opcode does not work in System 4.1 on the Macintosh Plus only. It returns an error code of 4 (bad parameters). This has been fixed in System 4.2.

Other SCSI Manager Issues

- At least one third-party SCSI peripheral driver used to issue SCSI commands from a VBL task. It didn't check to see if the bus was in the free state before sending the command! This is guaranteed to wipe out any other SCSI command that may have been in progress, since the SCSI Manager on the Macintosh Plus does not mask out (or use) interrupts.

  We strongly recommend that you avoid calling the SCSI Manager from interrupt handlers (such as VBL tasks). If you must send SCSI commands from a VBL task (like for a removable media system), do a `SCSIStat` call first to see if the bus is currently busy. If it's free (`BSY` is not asserted), then it's probably safe; otherwise the VBL task should not send the command. Note that you can't call `SCSIStat` before the System file fixes are in place. Since SCSI operations during VBL are not guaranteed, you should check all errors from SCSI Manager calls.

- A new SCSI Manager call will be added in the future. This will be a high-level call; it will have some kind of parameter block in which you give a pointer to a command buffer, a pointer to your TIB, a pointer to a sense data buffer (in case something goes wrong, the SCSI Manager will automatically read the sense bytes into the buffer for you), and a few other fields. The SCSI Manager will take care of arbitration, selection, sending the command, interpreting the TIB for the data transfer, and getting the status and message bytes (and the sense bytes, if there was an error). It should make SCSI device drivers much easier to write, since the driver will no longer have to worry about unexpected phase changes, getting the sense bytes, and so on. In the future, this will be the recommended way to use the SCSI Manager.

- The SCSI Manager (all machines) does not currently support interrupt-driven (asynchronous) operations. The Macintosh Plus can never support it since there is no interrupt capability, although a polled scheme may be implemented by the SCSI Manager. The Macintosh SE has a maskable interrupt for `IRQ`, and the Macintosh II has maskable interrupts for both `IRQ` and `DRQ`. Apple is working on an implementation of the SCSI Manager that will support asynchronous operations on the Macintosh II and probably on the SE as well. Because the interrupt hardware will interact adversely with any asynchronous schemes that are polled, it is strongly recommended that third parties do not attempt asynchronous operations until the new SCSI Manager is released. Apple will not attempt to be compatible with any products that bypass some or all of the SCSI Manager. In order to implement software-based (polled) asynchronous operations it is necessary to bypass the SCSI Manager.

The SCSI Manager section of the alpha draft of *Inside Macintosh* volume V documented the Disconnect and Reselect routines which were intended to be used for asynchronous I/O. Those routines cannot be used. Those routines have been removed from the manual. Any software that uses those routines will have to be revised when the SCSI Manager becomes interrupt-driven. Drivers which send SCSI commands from VBL tasks may also have to be modified.

Hardware in the SCSI

There is some confusion on how many terminators can be used on the bus, and the best way to use them. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of the wire that goes through all of the devices. One terminator should be on the end of the system cable that comes out of the Macintosh. The other terminator would be on the very end of the last device on the chain. If you have an SE or II with an internal hard disk, there is already one terminator on the front of the chain, inside the computer.

On the Macintosh SE and II, there is additional hardware support for the SCSI bus transfers in pseudo-DMA mode. The hardware makes it possible to handshake the data in Blind mode so that the Blind mode is safe for all transfers. On the Macintosh Plus, the Blind transfers are heavily timing dependent and can overrun or underrun during the transfer with no error generated. Assuring that Blind mode is safe on the Macintosh Plus depends upon the peripheral being used. On the SE and II, the transfer is hardware assisted to prevent overruns or underruns.

Changes in SCSI for SE and II

The changes made to the SCSI Manager found in the Macintosh SE and Macintosh II are primarily bug fixes. No new functionality was added. The newer SCSI Manager is more robust and has more error checking. Since the Macintosh Plus SCSI Manager only did limited error checking, it is possible to have code that would function (with bugs) on the Macintosh Plus, but will not work correctly on the SE or II. The Macintosh Plus could mask some bugs in the caller by not checking errors. An example of this is sending or receiving the wrong number of bytes in a blind transfer. On the Macintosh Plus, no error would be generated since there was no way to be sure how many bytes were sent or received. On the SE and II, if the wrong number of bytes are transferred an error will be returned to the caller. The exact timing of transfers has changed on the SE and II as well, since the computers run at different speeds. Devices that are unwittingly dependent upon specific timing in transfers may have problems on the newer computers. To find problems of this sort it is usually only necessary to examine the error codes that are passed back by the SCSI Manager routines. The error codes will generally point out where the updated SCSI Manager found errors.

To report other bugs or make suggestions

Please send additional bug reports and suggestions to us at the address in Technical Note #0. Let us know what SCSI controller you're using in your peripheral, and whether you've had any particularly good or bad experiences with it. We'll add to this note as more information becomes available.

#97: PrSetError Problem

Written by: Mark Baumwell November 15, 1986
Updated: March 1, 1988

This note formerly described a problem in Lisa Pascal glue for the `PrSetError` routine. The glue in MPW (and most, if not all, third party compilers) does not have this problem.

Macintosh Technical Notes

#98: Short-Circuit Booleans in Lisa Pascal

Written by: Mark Baumwell
Updated:

November 15, 1986
March 1, 1988

This note formerly described problems with the Lisa Pascal compiler. These problems have been fixed in the MPW Pascal compiler.

#99: Standard File Bug in System 3.2

See also: The Standard File Package

Written by: Jim Friedlander November 15, 1986
Updated: March 1, 1988

This note formerly described a bug in Standard File in System 3.2. This bug has been fixed in more recent Systems.

#100: Compatibility with Large-Screen Displays

See also: Technical Note #2—Macintosh Compatibility Guidelines

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

A number of third-party developers have announced large-screen display peripherals for Macintosh. One of them, Radius Inc., has issued a set of guidelines for developers who wish to remain compatible with their Radius FPD; unfortunately, one of their recommendations can cause system crashes. This note suggests a more correct approach.

On the first page of the appendix to their guidelines, "How to be FPD Aware," Radius recommends the following:

> "First, to detect the presence of a Radius FPD, you should check address `$C00008...`"

Unfortunately, this assumes that you're running on a Macintosh or Macintosh Plus; this test will not work on Macintosh XL, nor on a Macintosh II. Since these displays weren't designed to work with systems other than Macintosh and Macintosh Plus, you should make sure you're running on one of these systems before addressing I/O locations (such as those for an add-on display).

Before testing for the presence of any large-screen display, you should first check the machine ID; it's the byte located at `(ROMBASE)+8` (that is, take the long integer at the low-memory location `ROMBASE [$2AE]`, and add 8 to get the address of the machine ID byte. On a Macintosh or Macintosh Plus, this address will work out to be `$400008`; however, use the low-memory location, to be compatible with future systems that may have the ROM at a different address!).

The machine ID byte will be `$00` for all current Macintosh systems. If the value isn't `$00`, you can assume that no large-screen display is present, but don't forget to follow Technical Note #2's guidelines for screen size independence!

> **Note:** If you are a developer of an add-on large-screen display, we'd be happy to review your guidelines for developers in advance of distribution; please send them to us at the address for comments in Technical Note #0. Future versions of this note may recommend general guidelines for dealing with add-on large-screen displays.

#101: CreateResFile and the Poor Man's Search Path

See also: The File Manager
The Resource Manager
Technical Note #77—HFS Ruminations

Written by: Jim Friedlander

January 12, 1987
Updated: March 1, 1988

CreateResFile checks to see if a resource file with a given name exists, and if it does, returns a dupFNErr (–48) error. Unfortunately, to do this check, CreateResFile uses a call that follows the Poor Man's Search Path (PMSP).

CreateResFile checks to see if a resource file with a given name exists, and if it does, returns a dupFNErr (–48) error. Unfortunately, to do the check, CreateResFile calls PBOpenRF, which uses the Poor Man's Search Path (PMSP). For example, if we have a resource file in the System folder named 'MyFile' (and no file with that name in the current directory) and we call CreateResFile('MyFile'), ResError will return a dupFNErr, since PBOpenRF will search the current directory first, then search the blessed folder on the same volume. This makes it impossible to use CreateResFile to create the resource file 'MyFile' in the current directory if a file with the same name already exists in a directory that's in the PMSP.

To make sure that CreateResFile will create a resource file in the current directory whether or not a resource file with the same name already exists further down the PMSP, call _Create (PBCreate or Create) before calling CreateResFile:

```
err := Create('MyFile',0,myCreator,myType);
             {0 for VRefNum means current volume/directory}
CreateResFile('MyFile');
err := ResError; {check for error}
```

In MPW C:

```
err = Create("\pMyFile",0,myCreator,myType);
CreateResFile("\pMyFile");
err = ResError();
```

This works because _Create does **not** use the PMSP. If we already have 'MyFile' in the current directory, _Create will fail with a dupFNErr, then, if 'MyFile' has an empty resource fork, CreateResFile will write a resource map, otherwise, CreateResFile will return dupFNErr. If there is no file named 'MyFile' in the current directory, _Create will create one and then CreateResFile will write the resource map.
Notice that we are intentionally ignoring the error from _Create, since we are calling it

only to assure that a file named 'MyFile' does exist in the current directory.

Please note that SFPutFile does **not** use the PMSP, but that FSDelete does. SFPutFile returns the vRefNum/WDRefNum of the volume/folder that the user selected. If your program deletes a resource file before creating one with the same name based on information returned from SFPutFile, you can use the following strategy to avoid deleting the wrong file, that is, a file that is not in the directory specified by the vRefNum/WDRefNum returned by SFPutFile, but in some other directory in the PMSP:

```
VAR
   wher   : Point;
   reply  : SFReply;
   err    : OSErr;
   oldVol : Integer;

...

   wher.h := 80; wher.v := 90;
   SFPutFile(wher,'','',NIL,reply);
   IF reply.good THEN BEGIN
       err := GetVol(NIL,oldVol); {So we can restore it later}
       err := SetVol(NIL,reply.vRefNum);{for the CreateResFile call}

   {Now for the Create/CreateResFile calls to create a resource file that
    we know is in the current directory}

       err := Create(reply.fName,reply.vRefNum,myCreator,myType);
       CreateResFile(reply.fName); {we'll use the ResError from this ...}

       CASE ResError OF
          noErr:{the create succeeded, go ahead and work with the new
                 resource file -- NOTE: at this point, we don't know
                 what's in the data fork of the file!!} ;
          dupFNErr: BEGIN {duplicate file name error}
              {the file already existed, so, let's delete it. We're now
               sure that we're deleting the file in the current directory}

                err:= FSDelete(reply.fName,reply.vRefNum);

              {now that we've deleted the file, let's create the new one,
               again, we know this will be in the current directory}

                err:= Create(reply.fName,reply.vRefNum,myCreator,myType);
                CreateResFile(reply.fName);
          END; {CASE dupFNErr}
          OTHERWISE     {handle other errors} ;
       END;             {Case ResError}
       err := SetVol(NIL,oldVol);{restore the default directory}
   END;                 {If reply.good}
...
```

In MPW C:

```
  Point         wher;
  SFReply       reply;
  OSErr         err;
  short         oldVol;


  wher.h = 80; wher.v = 90;
  SFPutFile(wher,"","",nil,&reply);
  if (reply.good )
  {
      err = GetVol(nil,&oldVol);
      /*So we can restore it later*/
      err = SetVol(nil,reply.vRefNum);/*for the CreateResFile call*/

      /*Now for the Create/CreateResFile calls to create a resource file
      that we know is in the current directory*/

      err = Create(&reply.fName,reply.vRefNum,myCreator,myType);
      CreateResFile(&reply.fName);
      /*we'll use the ResError from this ...*/

      switch (ResError())
      {
         case noErr:;/*the create succeeded, go ahead and work with the
                       new resource file -- NOTE: at this point, we don't
                       know what's in the data fork of the file!!*/
              break; /* case noErr*/
         case dupFNErr: /*duplicate file name error*/
                   /*the file already existed, so, let's delete it.
                   We're now sure that we're deleting the file in the
                   current directory*/

                   err= FSDelete(&reply.fName,reply.vRefNum);

                   /*now that we've deleted the file, let's create the
                   new one, again, we know this will be in the current
                   directory*/

                   err= Create(&reply.fName,reply.vRefNum,
                                              myCreator,myType);
                   CreateResFile(&reply.fName);
             break; /*case dupFNErr*/
        default:;     /*handle other errors*/
    }  /* switch */
    err = SetVol(nil,oldVol);/*restore the default directory*/
}                  /*if reply.good*/
```

Note: `OpenResFile` uses the PMSP too, so you may have to adopt similar strategies to make sure that you are opening the desired resource file and not some other file further down the PMSP. This is normally not a problem if you use `SFGetFile`, since `SFGetFile` does not use the PMSP, in fact, `SFGetFile` does not open or close files, so it doesn't run into this problem.

Macintosh Technical Notes

#102: HFS Elucidations

See also: The File Manager
Technical Note #77—HFS Ruminations

Written by: Bryan "Bo3b" Johnson January 12, 1987
Updated: March 1, 1988

This technical note will describe a few problems that can occur while using HFS. It will also describe ways to avoid these problems.

This technical note will discuss the following problems:

1) It is very important to be careful about how files are opened and closed. There must be no more than one close for every open.

2) Don't use Driver names, like `.Bout`, `.Print` or `.Sony`, in place of file names or the file system will become confused.

3) Be aware of the `ioFlVersNum` byte in all file calls. A number of pieces of the Macintosh system do not use, and may in fact ignore, files created with non-zero `ioFlVersNums`.

Each of these can lead to strange occurrences, as well as problems for the users. Doing any or all of these marginally illegal operations will not necessarily lead to a System Error. In some cases the confusion generated may be worse than a System Error.

One Close is always enough

If a file is closed twice, it is possible to corrupt the file system on a disk. If a program has been creating unreadable disks, this may be the cause.

One aspect of the file system that is not well documented is how it allocates access paths to files that are currently open. As a result of this, it is possible to get a rather cavalier attitude about opening and closing files. This discussion will explain why it is necessary to be very careful about opening and closing files.

When the File Manager receives an `Open` call, it will look at the parameters passed in the parameter block and create a new access path for the file that is being opened. The access path is how the File Manager keeps track of where to send data that is written, and where to get data that is read from that file. An access path is nothing more than: 1) a buffer that the file system uses to read and write data, and 2) a File Control Block that describes how the file is stored on a disk.

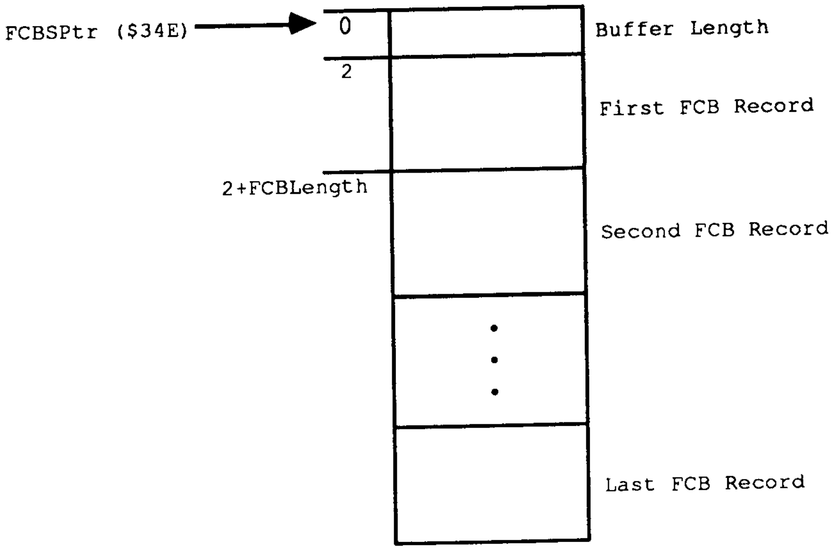
A call like:

```
    ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
```

will create the access path as a buffer and a File Control Block (FCB) in the FCB queue.

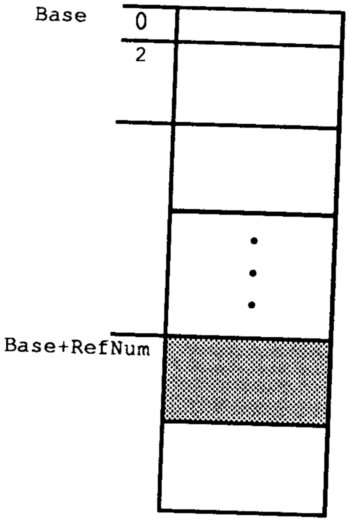
Note: The following information is here for illustrative purposes only; dependence on it may cause compatibility problems with future system software.

The structure of the queue can be visualized as:



where FCBSPtr is a low-memory global (at $34E) that holds the address of a nonrelocatable block. That block is the File Control Block buffer, and is composed of the two byte header which gives the length of the block, followed by the FCB records themselves. The records are of fixed length, and give detailed information about an open file. As depicted, any given record can be found by adding the length of the previous FCB records to the start of the block, adding 2 for the two byte header; giving an offset to the record itself. The size of the block, and hence the number of files that can be open at any given time, is determined at startup time. The call to open 'FirstFile' above will pass back the File Reference Number to that file in FirstRefNum. This is the number that will be used to access that file from that point on. The File Manager passes back an offset into the FCB queue as the RefNum. This offset is the number of bytes past the beginning of the queue to that FCB record in the queue. That FCB record will describe the file that was opened. An example of a number that might get passed back as a RefNum is $1D8. That also means that the FCB record is $1D8 bytes into the FCB block.

A visual example of a record being in use, and how the RefNum is related is:

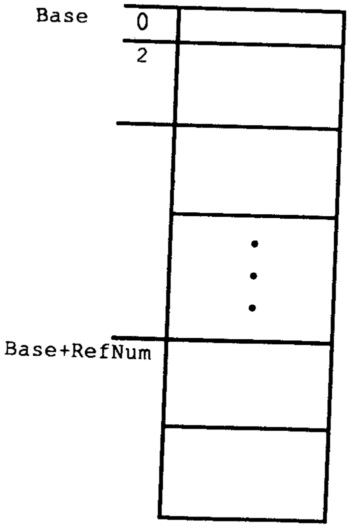


Base is merely the address of the nonrelocatable block that is the FCB buffer. FCBSPtr points to it. The RefNum (a number like $1D8) is added to Base, to give an address in the block. That address is what the file system will use to read and write to an open file, which is why you are required to pass the RefNum to the PBRead and PBWrite calls.

Since that RefNum is merely an offset into the queue, let's step through a dangerous imaginary sequence and see what happens to a given record in the FCB Buffer. Here's the sequence we will step through:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose ( FirstRefNum );
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose ( FirstRefNum ); {the wrong file gets closed!!!}
{the above line will close 'SecondFile', not 'FirstFile', which is already
 closed}
```

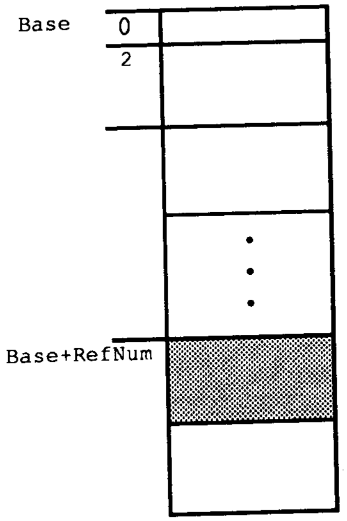
Before any operations:
the record at $1D8 is not used.



After the call:

`ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);`

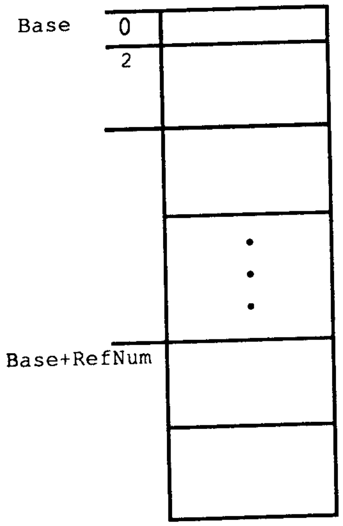
`FirstRefNum = $1D8` and the record is in use.



After the call:

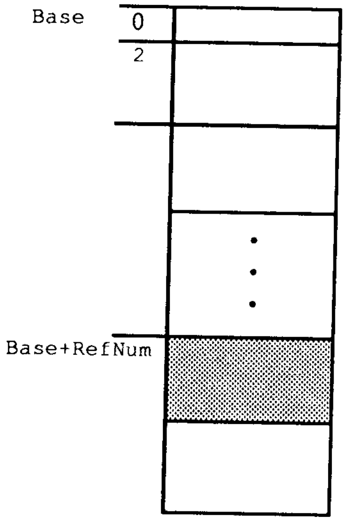
`ErrStuff := FSClose (FirstRefNum);`

`FirstRefNum is still equal to $1D8, but the FCB record is unused.`



After the call:

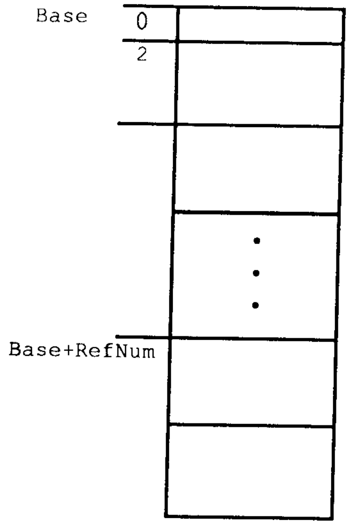
```
        ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
SecondRefNum = $1D8, FirstRefNum = $1D8, and the record is reused.
```



After the call:

```
        ErrStuff := FSClose (FirstRefNum);
The FirstRefNum = $1D8, SecondRefNum = $1D8,
```

the queue element is cleared. This happens, even though `FirstFile` was already closed. Actually, `SecondFile` was closed:



Note that the second close is using the old `RefNum`. The second close will still close a file, and in fact will return `noErr` as its result. Any subsequent accesses to the `SecondRefNum` will return an error, since the file '`SecondFile`' was closed. The File Control Blocks are reused, and since they are just offsets, it is possible to get the same file `RefNum` back for two different files. In this case, `FirstRefNum = SecondRefNum` since '`FirstFile`' was closed before opening '`SecondFile`' and the same FCB record was reused for '`SecondFile`'.

There are worse cases than this, however. As an example, think of what can happen if a program were to close a file, then the user inserted an HFS disk. The FCB could be reused for the Catalog File on that HFS disk. If the program had a generic error handler that closed all of its files, it could inadvertently close "its" file again. If it thought "its" file was still open it would do the close, which could close the Catalog file on the HFS disk. This is catastrophic for the disk since the file could easily be closed in an inconsistent state. The result is a bad disk that needs to be reformatted.

There are any number of nasty cases that can arise if a file is closed twice, reusing an old `RefNum`. A common programming practice is to have an error handler or cleanup routine that goes through the files that a program creates and closes them all, even if some may already be closed. If an FCB element was not reused, the `Close` will return the expected `fnOpnErr`. If the FCB had been reused, then the `Close` could be closing the wrong file. This can be very dangerous, particularly for all those paranoid hard disk users.

How to avoid the problem:

A very simple technique is to merely clear the `RefNum` after each close. If the variable that the program uses is cleared after each close, then there is no way of reusing a `RefNum` in the program. An example of this technique would be:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose (FirstRefNum);
FirstRefNum := 0; { We just closed it, so clear our refnum }
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose (FirstRefNum); { returns an error }
```

This makes the second `Close` pass back an error. In this case, the second close will try to close `RefNum` = 0, which will pass back a `fnOpnErr` and do no damage. **Note:** Be sure to use 0, which will never be a valid `RefNum`, since the first FCB entry is beyond the FCB queue length word. Don't confuse this with the 0 that the Resource Manager uses to represent the System file.

Thus, if an error handler were cleaning up possibly open files, it could blithely close all the files it knew about, since it would legitimately get an error back on files that are already closed. This is not done automatically, however. The programmer must be careful about the opening and closing of files. The problem can get quite complex if an error is received halfway through opening a sequence of ten files, for example. By merely clearing the `RefNum` that is stored after each close, it is possible to avoid the complexities of trying to track which files are open and which are closed.

This .file name looks outrageous.

There is a potential conflict between file names and driver names. If a file name is named something like `.Bout`, `.Print` or `.Sony`, then the file system will open the driver instead of the file. Drivers have priority on the 128K ROMs, and will always be opened before a file of the same name. This may mean that an application will get an error back

when opening these types of files, or worse, it will get back a driver RefNum from the call. What the application thought was a file open call was actually a driver open call. If the program uses that access path as a file RefNum, it is possible to get all kinds of strange things to happen. For example, if .Sony is opened, the Sony driver's RefNum would be passed back, instead of a file RefNum. If the application does a Write call using that RefNum, it will actually be a driver call, using whatever parameters happen to be in the parameter block. Disks may be searching for new life after this type of operation. If a program creates files, it should not allow a file to be created whose name begins with '.'.

This file's not my type.

This has been discussed in other places, but another aspect of the File Manager that can cause confusion is the ioFlVersNum byte that is passed to the low-level File Manager calls. This is called ioFileType from Assembly, and should not be confused with ioFVersNum. This byte must be set to zero for normal Macintosh files. There are a number of parts of the system that will not deal correctly with files that have the wrong versions: the Standard File package will not display any file with a non-zero ioFlVersNum; the Segment Loader and Resource Manager cannot open files that have non-zero ioFlVersNums. It is not sufficient to ignore this byte when a file is created. The byte must be cleared in order to avoid this type of problem. Strictly speaking, it is not a problem unless a file is being created on an MFS disk. The current system will easily allow the user to access 400K disks however, so it is better to be safe than confused.

Macintosh Technical Notes

#103: Using MaxApplZone and MoveHHi from Assembly Language

See also: Using Assembly Language
The Memory Manager
Technical Note #129—SysEnvirons

Written by: Bryan “Bo3b” Johnson January 12, 1987
Updated: March 1, 1988

When calling `MaxApplZone` and `MoveHHi` from assembly language, be sure to get the correct code.

`MaxApplZone` and `MoveHHi` were marked [Not in ROM] in *Inside Macintosh, Volumes I-III* . They are ROM calls in the 128K ROM. Since they are not in the 64K ROM, if you want your program to work on 64K ROM routines it is necessary to call the routines by a `JSR` to a glue (library) routine instead of using the actual trap macro. The glue calls the ROM routines if they are available, or executes its copy of them (linked into your program) if not.

How to do it:

Whenever you need to use these calls, just call the library routine. It will check `ROM85` to determine which ROMs are running, and do the appropriate thing.

For MDS, include the `Memory.Rel` library in your link file and use:

```
    XREF  MoveHHi      ; we need to use this 'ROM' routine
    ...
    JSR  MoveHHi       ; jump to the glue routine that will check ROM85 for us
```

For MPW link with `Interface.o` and use:

```
    IMPORT  MoveHHi    ; we need to use this
    ...
    JSR  MoveHHi       ; jump to the glue routine that will check ROM85 for us
```

Avoid calling `_MaxApplZone` or `_MoveHHi` directly if you want your software to work on the 64K ROMs, since that will assemble to an actual trap, not to a `JSR` to the library.

If your program is going to be run **only** on machines with the 128K ROM or newer, you can call the traps directly. Be sure to check for the 64K ROMs, and report an error to the user. You can check for old ROMs using the `SysEnvirons` trap as described in Technical Note #129.

Macintosh Technical Notes

#104: MPW: Accessing Globals From Assembly Language

See also: MPW Reference Manual

Written by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note demonstrates how to access MPW Pascal and MPW C globals from the MPW Assembler.

To allow access of MPW Pascal globals from the MPW Assembler, you need to identify the variables that you wish to access as external. To do this, use the {$Z+} compiler option. Using the {$Z+} option can substantially increase the size of the object file due to the additional symbol information (no additional code is generated and the symbol information is stripped by the linker). If you are concerned about object file size, you can "bracket" the variables you wish to access as external variables with {$Z+} and {$Z-}. Here's a trivial example:

Pascal Source

```
PROGRAM MyPascal;
USES
   MemTypes,QuickDraw,OSIntf,ToolIntf;

VAR
   myWRect: Rect;
{$Z+} {make the following external}
   myInt: Integer;
{$Z-} {make the following local to this file (not lexically local)}
   err: Integer;

PROCEDURE MyAsm; EXTERNAL; {routine doubles the value of myInt}

BEGIN {PROGRAM}
   myInt:= 5;
   MyAsm; {call the routine, myInt will be 10 now}
   writeln('The value of myInt after calling myAsm is ', myInt:1);
END. {PROGRAM}
```

Assembly Source for Pascal

```
          CASE    OFF         ;treat upper and lower case identically
MyAsm     PROC    EXPORT      ;CASE OFF is the assembler's default
          IMPORT  myInt:DATA  ;we need :DATA, the assembler assumes CODE
          ASL.W   #1,myInt    ;multiply by two
          RTS                 ;all done with this extensive routine, whew!
```

```
              END
```

The variable `myInt` is accessible from assembler. Neither `myWRect` nor `err` are accessible. If you try to access `myWRect`, for example, from assembler, you will get the following linker error:

```
    ### Link: Error   Undefined entry name:   MYWRECT.
```

C Source

In an MPW C program, one need only make sure that `MyAsm` is declared as an external function, that `myInt` is a global variable (capitalizations must match) and that the `CASE ON` directive is used in the Assembler:

```
#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <windows.h>
#include <events.h>
#include <textedit.h>
#include <dialogs.h>
#include <stdio.h>

extern MyAsm();     /* assembly routine that doubles the value of myInt */
short myInt;       /* we'll change the value of this variable from MyAsm */

main()
{
WindowPtr MyWindow;
Rect myWRect;

myInt = 5;
MyAsm();
printf(" The value of myInt after calling myAsm is %d\n",myInt);
} /*main*/
```

Assembly source for C

```
        CASE    ON            ;treat upper and lower case distinct
MyAsm   PROC    EXPORT        ;this is how C treats upper and lower case
        IMPORT  myInt:DATA    ;we need :DATA, the assembler assumes CODE
        ASL.W   #1,myInt      ;multiply by two
        RTS                   ;all done with this extensive routine, whew!
        END
```

#105: MPW Object Pascal Without MacApp

See also: Technical Note #93—{$LOAD};_DataInit;%_MethTables

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

Object Pascal must have a CODE segment named `%_MethTables` in order to access object methods. In MacApp this is taken care of “behind the scenes” so you don't have to worry about it . However, if you are doing a straight Object Pascal program, you must make sure that `%_MethTables` is around when you need it. If it's unloaded when you call a method, your Macintosh will begin executing wild noncode and die a gruesome and horrible death.

The MPW Pascal compiler must see some declaration of an object in order to produce a reference to the magic segment. You can achieve this cheaply by simply including `ObjIntf.p` in your `Uses` declaration. This must be in the main program, by the way. The compiler will produce a call to `%_InitObj` which is in `%_MethTables`.

If you're a more adventurous soul, you can call `%_InitObj` explicitly from the initialization section of your main program (you must use the `{$%+}` compiler directive to allow the use of “`%`” in identifiers). This will load the `%_MethTables` segment. See Technical Note #93 for ideas about locking down segments that are needed forever without fragmenting the heap.

Macintosh Technical Notes

#106: The Real Story: VCBs and Drive Numbers

See also: The File Manager
Technical Note #36—Drive Queue Element Format

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

The top of page IV-178 in The File Manager chapter of *Inside Macintosh* in attempts to explain the behavior of two fields in a volume control block when the corresponding disk is offline or ejected. Due to the fact that a little bit is left unsaid, this paragraph is rather misleading. The two fields in question are `vcbDrvNum` and `vcbDRefNum` (referred to as `ioVDrvInfo` and `ioVDRefNum` in C and Pascal). `PBHGetVInfo` can be used to access these fields.

Offline

When a mounted volume is placed offline, `vcbDrvNum` is cleared **and** `vcbDRefNum` is set to the two's complement of the drive number. Since drive numbers are assigned positive values (starting with one), this will be a negative number. If `vcbDrvNum` is zero **and** `vcbDRefNum` is negative, you know that the volume is offline.

Ejected

When a volume is ejected, `vcbDrvNum` is cleared and `vcbDRefNum` is set to the positive drive number. If `vcbDrvNum` is zero and `vcbDRefNum` is positive, you know that the volume is ejected. Ejection implies being offline. There is no such thing as "premature ejection".

Summary

	online	offline	ejected
`vcbDrvNum`	`>0 (DrvNum)`	`0`	`0`
`vcbDRefNum`	`<0 (DRefNum)`	`<0 (-DrvNum)`	`>0 (DrvNum)`

Please refrain from assuming anything about a VCB queue element beyond what is documented in *Inside Macintosh*, and don't expect it to always be 178 bytes in size. It grew when we went from MFS to HFS, and it may grow again. It's safest to use calls like `PBHGetVInfo` to get the information that you need.

Macintosh Technical Notes

#107: Nulls in Filenames

See also: The File Manager

Written by: Rick Blair March 2, 1987
Updated: March 1, 1988

Some applications (loosely speaking so as to include Desk Accessories, INITs, and what-have-you) generate or rename special files on the fly so that they are not explicitly named by the user via `SFPutFile`. Since the Macintosh file system is very liberal about filenames and only excludes colons from the list of acceptable characters, this can lead to some difficulties, both for the end user and for writers of other programs which may see these files.

Other programs which might be backing up your disk or something similar may get confused. A program written in C will think it has found the end of a string when it hits a null (ASCII code 0) character, so nulls in filenames are especially risky.

As a rule, filenames should only include characters which the user can see and edit. The only reasonable exception might be invisible files, but it can be argued that they are of dubious value anyway. You can argue "but what about my help file, I don't want it renamed" but we already have what we think is the best approach for that situation. If you can't find a configuration or other file because the user has renamed or moved it, then call `SFGetFile` and let the user find it. If the user cancels, and you can't run without the file, then quit with an appropriate message.

Please consider carefully before you put non-displaying characters in filenames!

Macintosh Technical Notes

#108: _AddDrive, _DrvrInstall, and _DrvrRemove

See also: Technical Note #36, Drive Queue Elements
SCSI Development Package (APDA)

Written by:	Jim Friedlander	March 2, 1987
Revised by:	Pete Helme	December 1988

_AddDrive, _DrvrInstall, and _DrvrRemove are used in the sample SCSI driver in the SCSI Development Package, which is available from APDA. This Technical Note documents the parameters for these calls.

Changes since March 1, 1988: Updated the _DrvrInstall text to reflect the use of register A0, which should contain a pointer to the driver when called. Also added simple glue code for _DrvrInstall and _DrvrRemove since none is available in the MPW interfaces.

_AddDrive

_AddDrive adds a drive to the drive queue, and is discussed in more detail in Technical Note #36, Drive Queue Elements:

```
FUNCTION AddDrive(DQE:DrvQEl;driveNum,refNum:INTEGER):OSErr;
```

A0 (input)	→	pointer to DQE
D0 high word(input)	→	drive number
D0 low word(input)	→	driver RefNum
D0 (output)	←	error code
		noErr (always returned)

_DrvrInstall

_DrvrInstall is used to install a driver. A DCE for the driver is created and its handle entered into the specified Unit Table position (–1 through –64). If the unit number is –4 through –9, the corresponding ROM-based driver will be replaced:

```
FUNCTION DrvrInstall(drvrHandle:Handle; refNum: INTEGER): OSErr;
```

A0 (input)	→	pointer to driver
D0 (input)	→	driver RefNum (–1 through –64)
D0 (output)	←	error code
		noErr
		badUnitErr

_DrvrRemove

_DrvrRemove is used to remove a driver. A RAM-based driver is purged from the system heap (using _ReleaseResource). Memory for the DCE is disposed:

```
FUNCTION DrvrRemove(refNum: INTEGER):OSErr;

    D0 (input)        →     Driver RefNum
    D0 (output)       ←     error code
                                noErr
                                qErr
```

Interfaces

Through a sequence of cataclysmic events, the glue code for _DrvrInstall and _DrvrRemove was never actually added to the MPW interfaces (i.e., "We forgot."), so we will include simple glue here at no extra expense to you.

It would be advisable to first lock the handle to your driver with _HLock before making either of these calls since memory may be moved.

```
;-------------------------------------------------------------
; FUNCTION DRVRInstall(drvrHandle:Handle; refNum:INTEGER):OSErr;
;-------------------------------------------------------------

DRVRInstall     PROC     EXPORT
        MOVEA.L          (SP)+, A1       ; pop return address
        MOVE.W           (SP)+, D0       ; driver reference number
        MOVEA.L          (SP)+, A0       ; handle to driver
        MOVEA.L          (A0), A0        ; pointer to driver
        _DrvrInstall                     ; $A03D
        MOVE.W           D0, (SP)        ; get error
        JMP              (A1)            ; & split
        ENDPPROC

;-------------------------------------------------------------
; FUNCTION DRVRRemove(refNum:INTEGER):OSErr;
;-------------------------------------------------------------

DRVRRemove      PROC     EXPORT
        MOVEA.L          (SP)+, A1       ; pop return address
        MOVE.W           (SP)+, D0       ; driver reference number
        _DrvrRemove                      ; $A03E
        MOVE.W           D0, (SP)        ; get error
        JMP              (A1)            ; & split
        ENDPPROC
```

#109: Bug in MPW 1.0 Language Libraries

See also: MPW Reference Manual

Written by: Scott Knaster March 2, 1987
Updated: March 1, 1988

This note formerly described a problem in the language libraries for MPW 1.0. This bug is fixed in MPW 1.0.2, available from APDA.

#110: MPW: Writing Stand-Alone Code

Revised by: Keith Rollin August 1990
Written by: Jim Friedlander March 1987

This Technical Note formerly discussed using MPW Pascal and C to write stand-alone code, such as `'WDEF'`, `'LDEF'`, `'INIT'`, and `'FKEY'` resources.
Changes since February 1990: Merged the contents of this Note into Technical Note #256, Stand-Alone Code, *ad nauseam.*

This Note formerly discussed using MPW Pascal and C to write stand-alone code. This information has been expanded and is now contained in Technical Note #256, Stand-Alone Code, *ad nauseam.*

#111: MoveHHi and SetResPurge

See also: The Memory Manager
The Resource Manager

Written by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

`SetResPurge(TRUE)` is called to make the Memory Manager call the Resource Manager before purging a block specified by a handle. If the handle is a handle to a resource, and its `resChanged` bit is set, the resource data will be written out (using `WriteResource`).

When `MoveHHi` is called, even though the handle's block is not actually being purged, the resource data specified by the handle will be written out. An application can prevent this by calling `SetResPurge(FALSE)` before calling `MoveHHi` (and then calling `SetResPurge(TRUE)` after the `MoveHHi` call).

Macintosh Technical Notes

#112: FindDItem

See also: The Dialog Manager

Written by: Rick Blair March 2, 1987
Updated: March 1, 1988

FindDItem is a potentially useful call which returns the number of a dialog item given a point in local coordinates and a dialog handle. It returns an item number of –1 if no item's rectangle overlaps the point. This is all well and good, except you don't get back quite what you would expect.

The item number returned is zero-based, so you have to add one to the result:

```
theitem := FindDItem(theDialog, thePoint) + 1;
```

Macintosh Technical Notes

#113: Boot Blocks

See also: The Segment Loader

Written by: Bo3b Johnson March 2, 1987
Updated: March 1, 1988

There are two undocumented features of the Boot Blocks. This note will describe how they currently work.

Warning: The format and functionality of the Boot Blocks will change in the future; dependence on this information may cause your program to fail on future hardware or with future System software.

The first two sectors of a bootable Macintosh disk are used to store information on how to start up the computer. The blocks contain various parameters that the system uses to startup such as the name of the system file, the name of the Finder, the first application to run at boot time, the number of events to allow, etc.

Changing System Heap Size

The boot blocks dictate what size the system heap will be after booting. Any common sector editing program will allow you to change the data in the boot blocks. Changing the system heap size is accomplished by changing two parameters in the boot blocks: the long word value at location `$86` in Block 0 indicates the size of the system heap; the word value at location `$6` is the version number of the boot blocks. Changing the version number to be greater than `$14` (`$15` is recommended) tells the ROM to use the value at `$86` for the system heap size, otherwise the value at `$86` is ignored. The `$86` location only applies to computers with more than 128K of RAM.

Secondary Sound and Video Pages

Another occasionally useful feature of the boot blocks is the ability to specify that the secondary sound and video pages be allocated at boot time. This is done before a debugger is loaded, so the debugger will load below the alternate screen. This is useful for debugging software that uses the alternate video page, like page-flipping demos or games. To allocate the second video and sound buffers, change the two bytes starting at location `$8` in the boot blocks. Change the value (normally 0) to a negative number (`$FFFF`) to allocate both video and sound buffers. Change the value to a positive number (`$0001`) to allocate only the secondary sound buffer.

Warning: MacsBug may not work properly if you allocate additional pages for sound and video.

#114: AppleShare and Old Finders

See also: *AppleShare User's Guide*

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

A rumor has been spread that if you use a pre-AppleShare Finder on a workstation to access AppleShare volumes, you can bypass AppleShare's "access privilege" mechanisms.

This is not true. Access controls are enforced by the server, **not** by the Finder. If you use an older Finder, you are still prevented (by the server) from gaining access to protected files and folders; however, you will not get the proper user-interface feedback that you would if you were using the correct Finder: for instance, folders on the server will always appear plain white (that is, without the permission feedback you'd normally get), and error messages would not be as explanatory as those from Finders that "know" about AppleShare servers.

#115: Application Configuration with Stationery Pads

See also: The File Manager
Technical Note #116—AppleShare-able Applications
Technical Note #47—Customizing SFGetFile
Technical Note #48—Bundles
"Application Development in a Shared Environment"

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

With the introduction of AppleShare (Apple's file server) there are restrictions on self-modification of application resource files and the placement of configuration files. This note describes one way to get around the necessity for configuration files.

Configuration Files

Some applications need to store information about configuration; others could benefit simply from allowing users to customize default ruler settings, window placement, fonts, etc.

There are applications which store this information as additional resources in the application's resource file; when the user changes the configuration, the application writes to itself to change the saved information.

AppleShare, however, requires that if an application is to be used by more than one user at a time, it must not need write access to itself. This means that the above method of storing configuration information cannot be used. (For more information about making your application sharable, see Technical Note #116.)

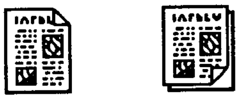
Storing configuration in a special configuration file can be a problem; the user must keep the file in the system folder or the application must search for it. This process has design issues of its own.

An alternative to configuration files: Stationery Pads

A basis for one solution to this problem was a user-interface feature of the Lisa Office System architecture. Lisa introduced the concept of "stationery pads", special documents that created copies of themselves to allow users to save a pre-set-up document for future use. On Lisa, this was the way Untitled documents were created.

Your Macintosh application can provide the option of saving a document as a stationery pad, to provide similar functionality. Here's how:

- You'll need to add a checkbox to your `SFPutFile` dialog box (if you don't know how to do this, check out Technical Note #47); if the user checks this box, save the document as you normally would, but use a different file type (the file type of a document is usually set when the document is created, using the File Manager `Create` procedure, or later using `SetFileInfo`).



A Document and its Stationery pad

- Be sure to use a different but similar icon for the stationery pad file. This is easy if you differentiate between stationery and normal files solely by file type—the Finder uses the type to determine which icon to display, see Technical Note #48 for help with the "bundle" mechanism used to associate a file type with an icon.

- When opening a stationery pad file, the window should come up named "Untitled", with the contents of the stationery pad file.

- "Revert" should re-read the stationery pad file.

- Don't forget to add the stationery pad's file type to the file-types list that you pass to Standard File, so that the new files will appear in the list when the user chooses Open. This file type should be registered with Macintosh Developer Technical Support.

Macintosh Technical Notes

#116: AppleShare-able Applications and the Resource Manager

See also: The Resource Manager
"Application Development in a Shared Environment"
Technical Note #40—Finder Flags

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

Normally, applications on an AppleShare server volume cannot be executed by more than one user at a time. This technical note explains why, and tells how you can enable your application to be shared.

The Resource Manager versus Shared Files

Part of the explanation of why applications are not automatically sharable is based on the design of the Resource Manager. The Resource Manager is a great little database. It was originally conceived as a way to keep applications localizable (a task it has performed admirably), and was found to be an excellent foundation for the Segment Loader, Font Manager, and a large part of the rest of the Macintosh operating system.

However, it was never designed to be a multi-user database. When the Resource Manager opens a resource file (such as an application), it reads the file's resource map into memory. This map remains in memory until the resource file is closed by the Segment Loader, which regains control when the application exits. Sometimes it is necessary to write the map out to disk; normally, this is only done by `UpdateResFile` and `CloseResFile`.

If two users opened the same resource file at the same time, and one of them had write access to the file and added a resource to it, the other user's Resource Manager wouldn't know about it; this would make the other user's copy of the file's original resource map invalid. This could cause (at least) a crash; if both users had write access, it's not unlikely that the resource file involved would become corrupted. Also, although you can tell the Resource Manager to write out an updated resource map, there's no way for another user to tell it to refresh the copy of the map in memory if the file changes.

What does all this have to do with running my application twice?

Your application is stored as a resource file; code segments, alert and dialog templates, etc., are resources. If you write to your application's resource file (for instance, to add configuration information, like print records), your application can't be shared.

In Apple's compatibility testing of existing applications (during development of AppleShare), we found quite a few applications, some of them quite popular, that wrote to their own resource files. So we decided, to improve the safety of using AppleShare, to always launch applications using a combination of access privileges such that only one user at a time could use a given application (these privileges will be discussed in a future Technical Note). In fact, AppleShare opens all resource files this way, unless the resource file is opened with `OpenRFPerm` and read-only permission is specified.

But my application doesn't write to itself!

We realize that many applications do not. However, there are other considerations (covered in detail, with suggestions for fixes, in "Application Development in a Shared Environment", available from APDA). In brief, here are the big ones we know about:

- Does your application create temporary files with fixed names in a fixed place (such as the directory containing the application)? Without AppleShare's protection, two applications trying to use the same temporary file could be disastrous.

- Is your application at least "conscious" of the fact that it may be in a multi-user environment? For instance, does it work correctly if a volume containing an existing document is on a locked volume? Does it check all result codes returned from File Manager calls, and `ResError` after relevant Resource Manager calls?

OK, I follow the rules. What do I do to make my application sharable?

There is a flag in each file's Finder information (stored in the file's directory entry) known as the "shared" bit. If you set this bit on your application's resource file, the Finder will launch your application using read-only permissions; if anyone else launches your application, they'll also get it read-only (their Finder will see the same "shared" bit set.).

Three important warnings accompany this information:

- The definition of the "shared" bit was incorrect in previous releases of information and software from Apple. This includes the June 16, 1986 version of Technical Note #40 (fixed in the March 2, 1987 version), as well as all versions of ResEdit before and including 1.1b3 (included with MPW 2.0). For now, the most reliable way to set this bit is to get the 1.1b3 version of ResEdit, use it to Get Info on your application, and check the box labeled "cached" (the incorrect documentation upon which ResEdit [et al.] was based called the real shared bit "cached"; the bit labeled as "shared" is the real cached bit [a currently unused but reserved bit which should be left clear]).

- By checking this bit, you're promising (to your users) that your application will work entirely correctly if launched by more than one user. This means that you follow the other rules, in addition to simply not writing to your application's own resource file. See "Application Development for a Shared Environment," and test carefully!
- Setting this bit has nothing to do with allowing your application's documents to be shared; you must design this feature into your application (it's not something that Apple system software can take care of behind your application's back.). You should realize from reading this note, however, that if you store your document's data in resource files, you won't be able to allow multiple users to access them simultaneously.

#117: Compatibility: Why & How

See Also: Technical Note #2—Compatibility Guidelines
Technical Note #7—A Few Quick Debugging Tips

Written by: Bo3b Johnson February 9, 1987
Updated: March 1, 1988

While creating or revising any program for the Macintosh, you should be aware of the most common reasons why programs fail on various versions of the Macintosh. This note will detail some common failure modes, why they occur, and how to avoid them.

We've tried to explain the issues in depth, but recognize that not everyone is interested in every issue. For example, if your application is not copy protected, you're probably not very interested in the section on copy protection. That's why we've included the outline form of the technical note. The first two pages outline the problems and the solutions that are detailed later. Feel free to skip around at will, but remember that we're sending this enormous technical note because the suggestions it provides may save you hasty compatibility revisions when we announce a new machine.

We know it's a lot, and we're here to help you if you need it. Our address (electronic and physical) is on page three—contact us with **any** questions—that's what we're here for!

Compatibility: the outline

Don't assume the screen is a fixed size

To get the screen size:

- check the QuickDraw global `screenBits.bounds`

Don't assume the screen is in a fixed location

To get the screen location:

- check the QuickDraw global `screenBits.baseAddr`

Don't assume that `rowBytes` is equal to the width of the screen

To get the number of bytes on a line:

- check the QuickDraw global `screenBits.rowBytes`

To get the screen width:

- check the QuickDraw global `screenBits.bounds.right`

To do screen-size calculations:

- Use `LongInts`

Don't write to or read from `nil` Handles or `nil` Pointers

Don't create or Use Fake Handles

To avoid creating or using fake handles:

- Always let the Memory Manager perform operations with handles
- Never write code that assigns something to a master pointer

Don't write code that modifies itself

Self modifying code will not live across incarnations of the 68000

Think carefully about code designed strictly as copy protection

To avoid copy protection-related incompatibilities:

- Avoid copy protection altogether
- Rely on schemes that don't require specific hardware
- Make sure your scheme doesn't perform illegal operations

Don't ignore errors

To get valuable information:

- Check all pertinent calls for errors
- Always write defensive code

Don't access hardware directly

To avoid hardware-related incompatibilities:

- Don't read or write the hardware
- If you can't get the support from the ROM, ask the system where the hardware is
- Use low-memory globals

Don't use bits that are reserved

To avoid compatibility problems when bit status changes:

- Don't use undocumented stuff
- When using low-memory globals, check only what you want to know

Summary

Minor bugs are getting harder and harder to get away with:

- Good luck
- We'll help
- AppleLink: MacDTS, MCI: MacDTS
- U.S. Mail: 20525 Mariani Ave.; M/S 27-T; Cupertino, CA 95014

What it Is

The basic idea is to make sure that your programs will run, regardless of which Macintosh they are being run on. The current systems to be concerned with include:

- Macintosh 128K
- Macintosh 512K
- Macintosh XL
- Macintosh 512Ke
- Macintosh Plus
- Macintosh SE
- Macintosh II

If you perform operations in a generic fashion, there is rarely any reason to know what machine is running. This means that you should avoid writing code to determine which version of the machine you are running on, unless it is absolutely necessary.

For the purposes of this discussion, the term "programs" will be used to describe any code that runs on a Macintosh. This includes applications, INITs, FKEYs, Desk Accessories and Drivers.

What the "Rules" mean

Compatibility across all Macintosh computers (which may sound like it involves more work for you) may actually mean that you have less work to do, since it may not be necessary to revise your program each time Apple brings out a new computer or System file. Users, as a group, do not understand compatibility problems; all they see is that the program does not run on their system.

The benefits of being compatible are many-fold: your customers/users stay happy, you have less programming to do, you can devote your time to more valuable goals, there are fewer versions to deal with, your code will probably be more efficient, your users will not curse you under their breath, and your outlook on life will be much merrier.

Now that we know what being compatible is all about, recognize that nobody is requiring you to be compatible with anything. Apple does not employ roving gangs of thought police to be sure that developers are following the recommended guidelines. Furthermore, when the guidelines comprise 1200 pages of turgid prose (*Inside Macintosh*), you can be expected to miss one or two of the "rules." It is no sin to be incompatible, nor is it a punishable offense. If it were, there would be no Macintosh programs, since virtually all developers would be incarcerated. What it does mean, however, is that your program will be unfavorably viewed until it steps in line with the current system (which is a moving target). If a program becomes incompatible with a new Macintosh, it usually requires rethinking the offending code, and releasing a new version. You may read something like "If the developers followed Apple guidelines, they would be compatible with the transverse-hinged diatomic quark realignment system." This means that if you made any mistakes (you read all 1200 pages carefully, right?), you will not be compatible. It is extremely difficult to remain completely compatible, particularly in a system as complex as the Macintosh. The rules haven't changed, but what you can get away with has. There are, however, a number of things that you can do to improve your odds—some of which will be explained here.

It's your choice

It is still your choice whether you will be concerned with compatibility or not. Apple will not put out a warrant for your arrest. However, if you are doing things that are specifically illegal, Apple will also not worry about "breaking" your program.

Bad Things

The following list is not intended to be comprehensive, but these are the primary reasons why programs break from one version of the system to the next. These are the current top ten commandments:

I Thou shalt not assume the screen is a fixed size.
II Thou shalt not assume the screen is at a fixed location.
III Thou shalt not assume that `rowBytes` is equal to the width of the screen.
IV Thou shalt not use `nil` handles or `nil` pointers.
V Thou shalt not create or use fake handles.
VI Thou shalt not write code that modifies itself.
VII Thou shalt think twice about code designed strictly as copy protection.
VIII Thou shalt check errors returned as function results.
IX Thou shalt not access hardware directly.
X Thou shalt not use any of the bits that are reserved (unused means reserved).

This has been determined from extensive testing of our diverse software base.

Assuming the screen is a fixed size

Do not assume that the Macintosh screen is 512 x 342 pixels. Programs that do generally have problems on (or special case for) the Macintosh XL, which has a wider screen. Most applications have to create the bounding rectangle where a window can be dragged. This is the `boundsRect` that is passed to the call:

```
    DragWindow (myWindowPtr, theEvent.where, boundsRect);
```

Some ill-advised programs create the `boundsRect` by something like:

```
    SetRect (boundsRect, 0,0,342,512);  { oops, this is hard-coded…}
```

Why it's Bad

This is bad because it is **never** necessary to specifically put in the bounding rectangle for the screen. On a Macintosh XL for example, the screen size is 760x364 (and sometimes 608x431 with alternate hardware). If a program uses the hard-coded 0,0,342,512 as a bounding rectangle, end users will not be able to move their windows past the fictitious boundary of 512. If something similar were done to the `GrowWindow` call, it would make it impossible for users to grow their window to fill the entire screen. (Always a saddening waste of valuable screen real-estate.)

Assuming screen size makes it more difficult to use the program on Macintoshes with big screens, by making it difficult to grow or move windows, or by drawing in strange places where they should not be drawing (outside of windows). Consider the case of running on a Macintosh equipped with one of the full page displays, or Ultra-Large screens. No one who paid for a big screen wants to be restricted to using only the upper-left corner of it.

How to avoid becoming a screening fascist

Never hard code the numbers 512 and 342 for screen dimensions. You should avoid using constants for system values that can change. Parameters like these are nearly always available in a dynamic fashion. Programs should read the appropriate variables while the program is running (at run-time, not at compile time).

Here's how smart programs get the screen dimensions:

```
    InitGraf(@thePort); { QuickDraw global variables have to be initialized.}
    ...
    boundsRect := screenBits.bounds;    { The Real way to get screen size }
                                          { Use QuickDraw global variable. }
```

This is smart, because the program never has to know specifically what the numbers are. All references to rectangles that need to be related to the screen (like the drag and grow areas of windows) should use `screenBits.bounds` to avoid worrying about the screen size.

Note that this does not do anything remotely like assume that "if the computer is not a standard Macintosh, then it must be an XL." Special casing for the various versions of the Macintosh has always been suspicious at best; it is now grounds for breaking. (At least with respect to screen dimensions.)

By the way, remember to take into account the menu bar height when using this rectangle. On 128K ROMs (and later) you can use the low-memory global `mBarHeight` (a word at `$BAA`). But since we didn't provide a low-memory global for the menu bar height in the 64K ROMs, you'll have to hard code it to 20 (`$14`). (You're not the only ones to forget the future holds changes.)

How to find fascist screenism in current programs

The easiest way is to exercise your program on one of the Ultra-Large screen Macintoshes. There should be no restrictions on sizing or moving the windows, and all drawing should have no problems. If there are any anomalies in the program's usage, there is probably a lurking problem. Also, do a global find in the source code to see if the numbers 512 or 342 occur in the program. If so, and if they are in reference to the screen, excise them.

Assuming the screen is at a fixed location

Some programs use a fixed screen address, assuming that the screen location will be the same on various incarnations of the Macintosh. This is not the case. For example, the screen is located at memory location `$1A700` on a 128K Macintosh, at `$7A700` on a 512K Macintosh, at `$F8000` on the Macintosh XL, and at `$FA700` on the Macintosh Plus.

Why it's Bad

When a program relies upon the screen being in a fixed location, Murphy's Law dictates that an unknowing user will run it upon a computer with the screen in a different location. This usually causes the system to crash, since the offending program will write to memory that was used for something important. Programs that crash have been proven to be less useful than those that don't.

How to avoid being a base screener

Suffice it to say that there is no way that the address of the screen will remain static, but there are rare occasions where it is necessary to go directly to the screen memory. On these occasions, there are bad ways and not-as-bad ways to do it. A bad way:

```
    myScreenBase := Pointer ($7A700);  { not good.  Hard-coded number. }
```

A not-as-bad way:

```
    InitGraf(@thePort);    { do this only once in a program. }
    ...
    myScreenBase := screenBits.baseAddr;  { Good.  Always works. }
                                        {Yet another QuickDraw global variable}
```

Using the latter approach is guaranteed to work, since QuickDraw has to know where to draw, and the operating system tells QuickDraw where the screen can be found. When in doubt, ask QuickDraw. This will work on Macintosh computers from now until forever, so if you use this approach you won't have to revise your program just because the screen moved in memory.

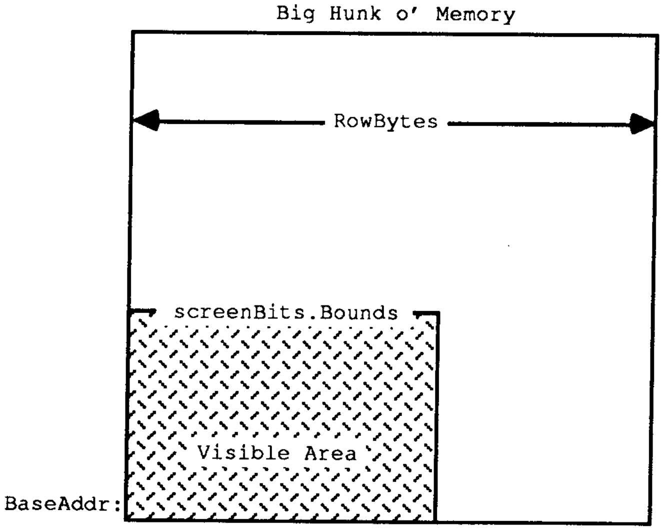
If you have a program (such as an INIT) that cannot rely upon QuickDraw being initialized (via `InitGraf`), then it is possible to use the `ScrnBase` low-memory global variable (a long word at `$824`). This method runs a distant second to asking QuickDraw, but is sometimes necessary.

How to find base screeners

The easiest way to find base screeners is to run the offending program on machines that have different screen addresses. If any addresses are being used in a base manner, the system will usually crash. The offending program may also occasionally refuse to draw. Some programs afflicted with this problem may also hang the computer (sometimes known as accessing funny space). Also, do a global find on the source code to look for numbers like `$7A700` or `$1A700`. When found, exercise caution while altering the offending lines.

Assuming that rowbytes is equal to the width of the screen

According to the definition of a bitMap found in *Inside Macintosh* (p I-144), you can see that rowBytes is the number of actual bytes in memory that are used to determine the bitMap. We know the screen is just a big hunk of memory, and we know that QuickDraw uses that memory as a bitMap. rowBytes accomplishes the translation of a big hunk of memory into a bitMap. To do this, rowBytes tells the system how long a given row is in memory and, more importantly, where in memory the next row starts. For conventional Macintoshes, rowBytes (bytes per Row) * 8 (Pixels per Byte) gives the final horizontal width of the screen as Pixels per Row. This does not have to be the case. It is possible to have a Macintosh screen where the rowBytes extends beyond what is actually visible on the screen. You can think of it as having the screen looking in on a larger bitMap. Diagrammatically, it might look like:



With an Ultra-Large screen, the number of bytes used for screen memory may be in the 500,000 byte range. Whenever calculations are being made to find various locations in the screen, the variables used should be able to handle larger screen sizes. For example, a 16 bit Integer will not be able to hold the 500,000 number, so a LongInt would be required. Do **not** assume that the screen size is 21,888 bytes long. bitMaps **can** be larger than 32K or 64K.

Why it's Bad

Programs that assume that all of the bytes in a row are visible may make bad calculations, causing drawing routines to produce unusual, and unreadable, results. Also, programs that use the rowBytes to figure out the width of the screen rectangle will find that their calculated rectangle is not the real screenBits.Bounds. Drawing into areas that are not visible will not necessarily crash the computer, but it will probably give erroneous results, and displays that don't match the normal output of the program.

Programs that assume that the number of bytes in the screen memory will be less than 32768 may have problems drawing into Ultra-Large screens, since those screens will often have more memory than a normal Macintosh screen. These particular problems do not evidence themselves by crashing the system. They generally appear as loss of

functionality (not being able to move a window to the bottom of the screen), or as drawing routines that no longer look correct. These problems can prevent an otherwise wonderful program from being used.

How to avoid being a row byter

In any calculations, the `rowBytes` variable should be thought of as the way to get to the next row on the screen. This is distinct from thinking of it as the width of the screen. The width should always be found from `screenBits.bounds.right-screenBits.bounds.left`.

It is also inappropriate to use the rectangle to decide how many bytes there are on a row. Programs that do something like:

```
bytesLine := screenBits.bounds.right DIV 8;  { bad use of bounds }
rightSide := screenBits.rowBytes * 8;      { bad use of rowBytes }
```

will find that the screen may have more `rowBytes` than previously thought. The best way to avoid being a row byter is to use the proper variables for the proper things. Without the proper mathematical basis to the screen, life becomes much more difficult. Always do things like:

```
bytesLine := screenBits.rowBytes;  { always the correct number }
rightSide := screenBits.bounds.right;  { always the correct screen size }
```

It is sometimes necessary to do calculations involving the screen. If so, be sure to use `LongInts` for all the math, and be sure to use the right variables (i.e. use `LongInts`). For example, if we need to find the address of the 500th row in the screen (500 lines from the top):

```
VAR   myAddress:   LongInt;
      myRow:       LongInt;       { so the calculations don't round off. }
      myOffset:    LongInt;       { could easily be over 32768 ... }
      bytesLine:   LongInt;

      ...
      myAddress := ord4(screenBits.baseAddr); {start w/the real base address }
      myRow := 500;                          {the row we want to address }
      bytesLine := screenBits.rowBytes;     {the real bytes per line }
      myOffset := myRow * bytesLine;        {lines * bytes per lines gives bytes }
      myAddress := myAddress + myOffset;    {final address of the 500th line }
```

This is not something you want to do if you can possibly avoid it, but if you simply must go directly to the screen, be careful. The big-screen machines (Ultra-Large screens) will thank you for it. If QuickDraw cannot be initialized, there is also the low-memory global `screenRow` (a word at `$106`) that will give you the current `rowBytes`.

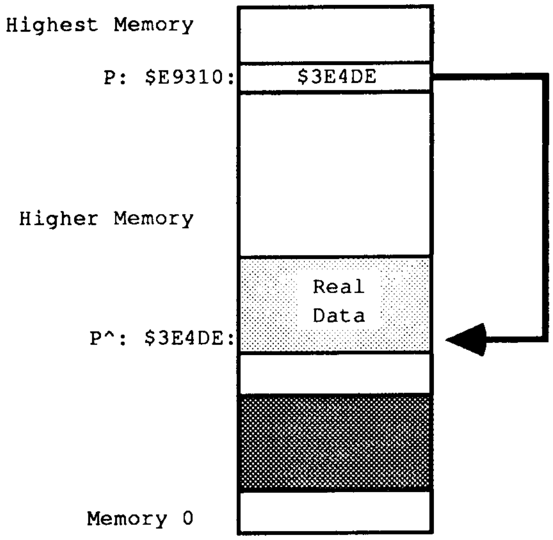
How to find row byters

To find current problems with row byter programs, run them on a machine equipped with Ultra-Large screens and see if any anomalies crop up. Look for drawing sequences that don't work right, and for drawing that clips to an imaginary edge. For source-level

inspection, look for uses of the `rowBytes` variables and be sure that they are being used in a mathematically sound fashion. Be highly suspicious of any code that uses `rowBytes` for the screen width. Any calculations involving those system variables should be closely inspected for round-off errors and improper use. Search for the number 8. If it is being used in a calculation where it is the number of bits per byte, then watch that code closely for improper conceptualization. This is code that could leap out and grab you by the throat at anytime. Be careful!

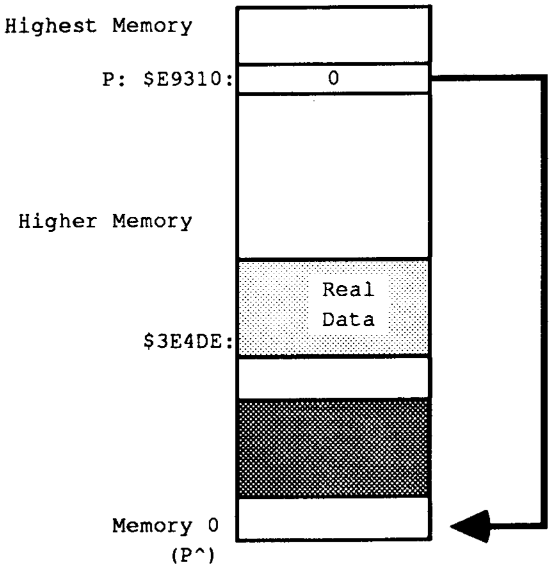
Using nil Handles or nil Pointers

A nil pointer is a pointer that has a value of 0. Recognize that pointers are merely addresses in memory. This means that a nil pointer is pointing to memory location 0. Any use of memory location 0 is strictly forbidden, since it is owned by Motorola. Trespassers may be shot on sight, but they may not die until much later. Sometimes trespassers are only wounded and act strangely. Any use of memory location 0 can be considered a bug, since there are **no** valid reasons for Macintosh programs to read or write to that memory. However, nil pointers themselves are not necessarily bad. It is occasionally necessary to pass nil pointers to ROM routines. This should not be confused with reading or writing to memory location 0. A pointer normally points to (contains the address of) a location in memory. It could look like this:



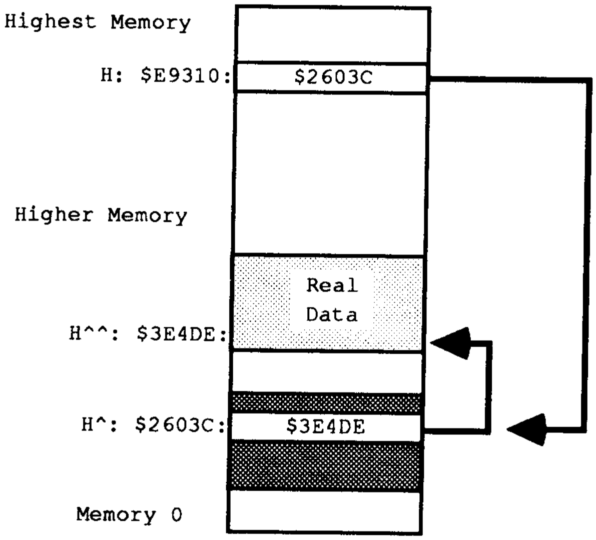
This is how a Pointer works. The address of the pointer variable itself is $E9310 (@P) and is four bytes long. The pointer points to (contains the address of) the block at $3E4DE (P). That memory location is where the actual data resides (P^).

If a pointer has been cleared to nil, it will point to memory location 0. This is OK as long as the program does not try to read from or write to that pointer. An example of a nil pointer could look like:



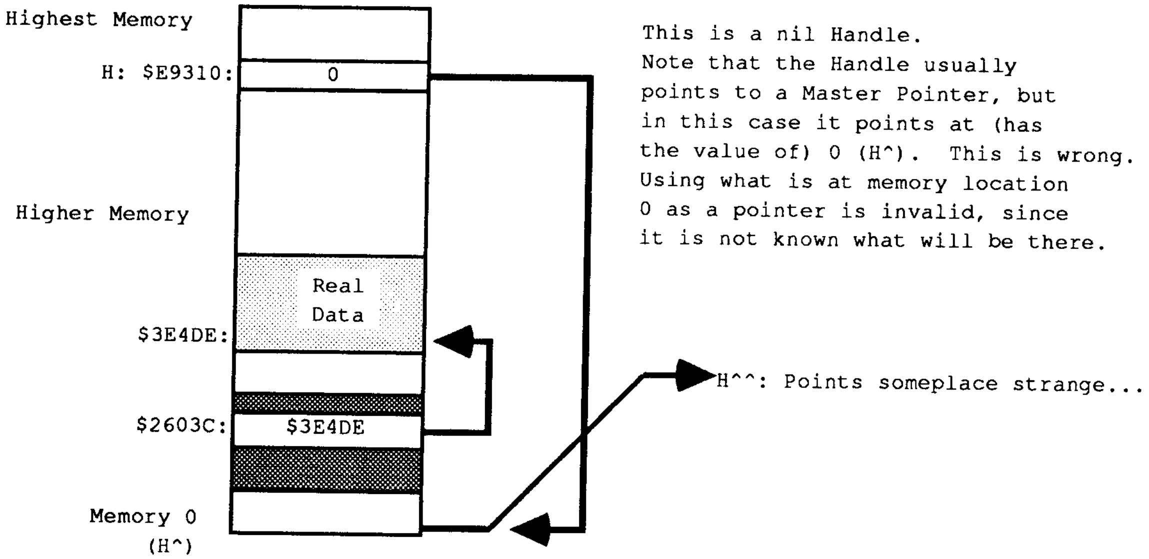
This is a nil Pointer. Note that the memory that it points to (the address) is 0 (P^). This is wrong. There is no valid data at memory location 0. Any writing to or reading from this pointer is a bug.

`nil` handles are related to the problem, since a handle is merely the address of a pointer (or a pointer to a pointer). An example of what a normal handle might look like is:



```
This is how a Handle works.
The address of the handle
variable itself (H) is $E9310.
That variable points (has the
address) to the master pointer
at location $2603C (H). That
variable is a pointer also, and
points to the real data found
at $3E4DE (H^^). The dark grey
block is a Master pointer block. It
is a group (usually 64) of Master
Pointers. One of them is the Master
Pointer at address $2603C (H^).
```

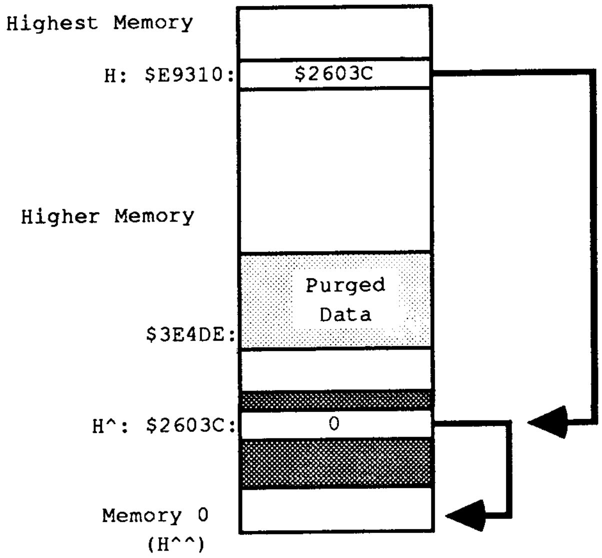
When the first pointer (`h`) becomes `nil`, that implies that memory location 0 can be used as a pointer. This is strictly illegal. There are **no** cases where it is valid to read from or write to a `nil` handle. A pictorial representation of what a `nil` handle could look like:



```
This is a nil Handle.
Note that the Handle usually
points to a Master Pointer, but
in this case it points at (has
the value of) 0 (H^). This is wrong.
Using what is at memory location
0 as a pointer is invalid, since
it is not known what will be there.
```

If the memory at 0 contains an odd number (numerically odd), then using it as a pointer will cause a system error with ID=2. This can be very useful, since that tells you exactly where the program is using this illegal handle, making it easy to fix. Unfortunately, there are cases where it is appropriate to pass a `nil` handle to ROM routines (such as `GetScrap`). These cases are rare, and it is **never** legal to read from or write to a `nil` handle.

There is also the case of an empty handle. An empty handle is one where the handle itself (the first pointer) points to a valid place in memory; that place in memory is also a pointer, and if it is nil the entire handle is termed empty. There are occasions where it is necessary to use the handle itself, but using the nil pointer that it contains is not valid. An example of an empty handle could be:



This is an Empty Handle. Note that the handle itself has a valid Master Pointer address in it $2603C (H^). The Master Pointer is nil however, which is the address of location 0 in memory. It is wrong to use the Master Pointer in this case, although there are cases where using the Handle itself is valid.

Fundamentally, any reading or writing to memory using a pointer or handle that is nil is punishable by death (of your program).

Why it's Bad

The use of nil pointers can lead to the use of make-believe data. This make-believe data often changes for different versions of the computer. This changing data makes it difficult to predict what will happen when a program uses nil pointers. Programs may not crash as a result of using a nil pointer, and they may behave in a consistent fashion. This does not mean that there isn't a bug. This merely means that the program is lucky, and that it should be playing the lottery, not running on a Macintosh. If a program acts differently on different versions of the Macintosh, you should think "could there be a nasty nil pointer problem here?" Use of a nil handle usually culminates in reading or writing to obscure places in memory. As an example:

```
VAR   myHandle:  TEHandle;

myHandle := nil;
```

That's pretty straightforward, so what's the problem? If you do something like:

```
myHandle^^.viewRect := myRect;  { very bad idea with myHandle = nil }
```

memory location zero will be used as a pointer to give the address of a TextEdit record. What if that memory location points to something in the system heap? What if it points to the sound buffer? In cases like these, eight bytes of rectangle data will be written to wherever memory location 0 points.

Use of a `nil` handle will never be useful. This memory is reserved and used by the 68000 for various interrupt vectors and Valuable Stuff. This Valuable Stuff is composed of things that you definitely do not want to change. When changed, the 68000 finds out, and decides to get back at your program in the most strange and wonderful ways. These strange results can range from a System Error all the way to erasing hard disks and destroying files. There really is no limit to the havoc that can be wreaked. This tends to keep the users on the edge of their seat, but this is not really the desired effect. As noted above, it won't necessarily cause traumatic results. A program can be doing naughty things and not get caught. This is still a bug that needs to be fixed, since it is nearly guaranteed to give different results on different versions of the Macintosh. Programs exhibiting schizophrenia have been proven to be less enjoyable to use.

How to avoid being a Niller

Whenever a program uses pointers and handles, it should ensure that the pointer or handle will not be `nil`. This could be termed defensive programming, since it assumes that everyone is out to get the program (which is not far from the truth on the Macintosh). You should always check the result of routines that claim to pass back a handle. If they pass you back a `nil` handle, you could get in trouble if you use them. Don't trust the ROM. The following example of a defensive use of a handle involves the Resource Manager. The Resource Manager passes back a handle to the resource data. There are any number of places where it may be forced to pass back a `nil` handle. For example:

```
VAR   myRezzie:  MyHandle;

myRezzie := MyHandle(GetResource(myResType, myResNumber)); { could be missing…}
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;               { We know it is OK }
```

As another example, think of how handles can be purged from memory in tight memory conditions. If a block is marked purgeable, the Memory Manager may throw it away at any time. This creates an empty handle. The defensive programmer will always make sure that the handles being used are not empty.

```
VAR   myRezzie:  myHandle;

myRezzie := myHandle(GetResource(myResType, myResNumber));  { could be
                                                               missing… }
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;     { We know it is OK }
tempHandle := NewHandle (largeBlock);  {might dispose a purgeable myRezzie}
IF myRezzie^ = nil  THEN LoadResource(Handle(myRezzie)); {Re-load empty
                                                            handle}
IF ResError = noErr  THEN
    myRezzie^^.StatusField := OK;       { guaranteed not empty, and actually
                                        gets read back in, if necessary }
```

Be especially careful of places where memory is being allocated. The `NewHandle` and `NewPtr` calls will return a `nil` handle or pointer if there is not enough memory. If you use that handle or pointer without checking, you will be guilty of being a Niller.

How to find Nillers

The best way to find these nasty `nil` pointer problems is to set memory location zero to be an **odd** number (a good choice is 'NIL!' = `$4E494C21`, which is numerically odd, as well as personality-wise). Please see Technical Note #7 for details on how to do this.

If you use TMON, you can use the extended user area with Discipline. Discipline will set memory location 0 to 'NIL!' to help catch those nasty pointer problems. If you use Macsbug, just type `SM 0 'NIL!` and go. Realize of course, that if a program has made a transgression and is actually using `nil` pointers, this may make the program crash with an ID=2 system error. This is good! This means that you have found a bug that may have been causing you untold grief. Once you know where a program crashes, it is usually very easy to use a debugger to find where the error is in the source code. When the program is compiled, turn on the debugging labels (usually a $D+ option). Set memory location 0 to be 'NIL!'. When the program crashes, look at where the program is executing and see what routine it was in (from a disassembly). Go back to that routine in the source code and remove the offending code with a grim smile on your face. Another scurvy bug has been vanquished. The intoxicating smell of victory wafts around your head.

Another way to find problems is to use a debugger to do a checksum on the first four bytes in memory (from 0 to 3 inclusive). If the program ever traps into the debugger claiming that the memory changed, see which part of the program altered memory location 0. Any code that writes to memory location zero is guilty of high treason against the state and must be removed. Remember to say, “bugs are not my friends.”

Creating or Using Fake Handles

A fake handle is one that was not manufactured by the system, but was created by the program itself. An example of a fake handle is:

```
CONST aMem = $100;
VAR   myHandle: Handle;
      myPointer: Ptr;

myPointer := Ptr (aMem);     { the address of some memory }
myHandle := @myPointer;      {the address of the pointer variable. Very bad.}
```

The normal way to create and use handles is to call the Memory Manager `NewHandle` function.

Why it's Bad

A handle that is manufactured by the program is not a legitimate handle as far as the operating system is concerned. Passing a fake handle to routines that use handles is a good way to discover the meaning of "Death by ROM." For example, think how confused the operating system would get if the fake handle were passed to `DisposHandle`. What would it dispose? It never allocated the memory, so how can it release it? Programs that manufacture handles may find that the operating system is no longer their friend.

When handles are passed to various ROM routines, there is no telling what sorts of things will be done to the handle. There are any number of normal handle manipulation calls that the ROM may use, such as `SetHandleSize`, `HLock`, `HNoPurge`, `MoveHHi` and so on. Since a program cannot guarantee that the ROM will not be doing things like this to handles that the program passes in, it is wise to make sure that a real handle is being used, so that all these type of operations will work as the ROM expects. For fake handles, the calls like `HLock` and `SetHandleSize` have no bearing. Fake handles are very easy to create, and they are very bad for the health of otherwise upstanding programs. Whenever you need a handle, get one from the Memory Manager.

As a particularly bad use of a fake handle:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

myHandle := NewHandle (SIZEOF(myStuff));    { create a new normal handle }
myHandle^ := @myStuff;   {YOW!  Intended to make myHandle a handle to
                            the myStuff record.  What it really does is
                            blow up a Master Pointer block, Heap corruption,
                            and death by Bad Heap.  Never do this. }
```

This can be a little confusing, since it is fine to use your own pointers, but very bad to use your own handles. The difference is that handles can move in memory, and pointers cannot, hence the pointers are not dangerous. This does not mean you should use pointers for everything since that causes other problems. It merely means that you have to be careful how you use the handles.

The use of fake handles usually causes system errors, but can be somewhat mysterious in its effects. Fake handles can be particularly hard to track down since they often cause damage that is not uncovered for many minutes of use. Any use of fake handles that causes the heap to be altered will usually crash the system. Heap corruption is a common failure mode. In clinical studies, 9 out of 10 programmers recommend uncorrupted heaps to their users who use heaps.

How to avoid being a fakir

The correct way to make a handle to some data is to make a copy of the data:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

errCode := PtrToHand (@myStuff, myHandle, SIZEOF(myStuff));
IF  errCode <> noErr  THEN ErrorHandler ('Out of memory');
```

Always, always, let the Memory Manager perform operations with handles. Never write code that assigns something to a master pointer, like:

```
VAR   myDeath:  Handle;
myDeath^ := stuff;   { Don't change the Master pointer. }
```

If there is code like this, it usually means the heap is being corrupted, or a fake handle is being used. It is, however, OK to pass around the handle itself, like:

```
myCopyHandle := myHandle;    { perfectly OK, nobody will yell about this. }
```

This is far different than using the ^ operator to accidentally modify things in the system. Whenever it is necessary to write code to use handles, be careful. Watch things carefully as they are being written. It is much easier to be careful on the way in than it is to try to find out why something is crashing. Be very careful of the @ operator. This operator can unleash untold problems upon unsuspecting programs. If at all possible, try to avoid using it, but if it is necessary, be absolutely sure you know what it is doing. It is particularly dangerous since it turns off the normal type checking that can help you find errors (in Pascal). In short, don't get crazy with pointer and handle manipulations, and they won't get crazy with you.

How to find fakirs

Problems of this form are particularly insidious because it can be very difficult to find them after they have been created. They tend to not crash immediately, but rather to crash sometime long after the real damage has been done. The best way to find these problems is to run the program with Discipline. (Discipline is a programmer's tool that will check all parameters passed to the ROM to see if they are legitimate. Discipline can be found as a stand-alone tool, but the most up-to-date version will be found in the Extended User Area for the TMON debugger. The User Area is public domain, but TMON itself is not. TMON has a number of other useful features, and is well worth the price.) Discipline will check handles that are passed to the ROM to see if they are real handles or not, and if not, will stop the program at the offending call. This can lead you back to the source at a point that may be close to where the bad handle was created. If a program passes the Discipline test, it will be a healthy, robust program with drastically improved odds for compatibility. Programs that do not pass Discipline can sleep poorly at night, knowing that they have broken at least one or two of the "rules."

A way to find programs that are damaging the heap is to use a debugger (TMON or Macsbug) and turn on the Heap Check operation. This will check the heap for errors at each trap call, and if the heap is corrupted will break into the debugger. Hopefully this will be close to where the code is that caused the damage. Unfortunately, it may not be close enough; this will force you to look further back.

Looking in the source code, look for all uses of the `@` operator, and examine the code carefully to see if it is breaking the rules. If it is, change it to step in line with the rest of the happy programs here in happy valley. Also, look for any code that changes a master pointer like the `myHandle^ := stuff`. Any code of this form is highly suspect, and probably a member of the Anti-Productivity League. The APL has been accused of preventing software sales and the rise of the Yen. These problems can be quite difficult to find at times, but don't give up. These fake handles are high on the list of guilty parties, and should never be trusted.

Writing code that modifies itself

Self-modifying code is software that changes itself. Code that alters itself runs into two main groupings: code that modifies the code itself and code that changes the block the code is stored in. Copy protection code often modifies the code itself, to change the way it operates (concealing the meaning of what the code does). Changing the code itself is very tricky, and also prone to having problems, particularly when the microprocessor itself changes. There are third-party upgrades available that add a 68020 to a Macintosh. Because of the 68020's cache, programs that modify themselves stand a good chance of having problems when run on a 68020. This is a compatibility point that should not be missed (nudge, nudge, wink, wink). Code that changes other code (or itself) is prone to be incompatible when the microprocessor changes.

The second group is code that changes the block that the code is stored in. Keeping variables in the CODE segment itself is an example of this. This is uncommon with high-level languages, but it is easy to do in assembly language (using the DC directive). Variables defined in the code itself should be read-only (constants). Code that modifies itself has signed a tacit agreement that says "I'm being tricky, if I die, I'll revise it."

Why it's Bad

There are now three different versions of the microprocessor, the 68000, 68010, and the 68020. They are intended to be compatible with each other, but may not be compatible with code that modifies itself. As the Macintosh evolves, the system may have compatibility problems with programs that try to "push the envelope."

How to avoid being an abuser

Well, the obvious answer is to avoid writing self-modifying code. If you feel obliged to write self-modifying code, then you are taking an oath to not complain when you break in the future. But don't worry about accidentally taking the oath: you won't do it without knowing it. If you choose to abuse, you also agree to personal visits from the Apple thought police, who will be hired as soon as we find out.

How to find abusers

Run the program on a 68020 system. If it fails, it could be related to this problem, but since there are other bugs that might cause failures, it is not guaranteed to be a self-modifying code problem. Self-modifying code is often used in copy protection, which brings us to the next big topic.

Code designed strictly as copy protection

Copy protection is used to make it difficult to make copies of a program. The basic premise is to make it impossible to copy a program with the Finder. This will not be a discussion as to the pros and cons of copy protection. Everyone has an opinion. This will be a description of reality, as it relates to compatibility.

Why it's Bad

System changes will never be made merely to cause copy protection schemes to fail, but given the choice between improving the system and making a copy protection scheme remain compatible, the system improvement will always be chosen.

- Copy protection is number one on the list of why programs fail the compatibility test.
- Copy protection by its very nature tends to do the most "illegal" things.
- Programs that are copy protected are assumed to have signed a tacit agreement to revise the program when the system changes.

Copy protection itself is not necessarily bad. What is bad is when programs that would otherwise be fully compatible do not work due only to the copy protection. This is very sad, since it requires extra work, revisions to the software, and time lost while the revision is being produced. The users are not generally humored when they can no longer use their programs. Copy protection schemes that fail generally cause system errors when they are run. They also can refuse to run when they should.

How to avoid being a protectionist

The simple answer is to do without copy protection altogether. If you think of compatibility as a probability game, if you leave out the copy protection, your odds of winning skyrocket. As noted above, copy protection is the single biggest reason why programs fail on the various versions of the Macintosh. For those who are required to use copy protection, try to rely on schemes that do not require specific hardware and make sure that the scheme used is not performing illegal operations. If a program runs, an experienced Macintosh programmer armed with a debugger can probably make a copy of it, (no matter how sophisticated the copy protection scheme) so a moderate scheme that does not break the rules is probably a better compatibility bet. The trickier and more devious the scheme, the higher the chance of breaking a rule. Tread lightly.

How to find protectionists

The easiest way to see if a scheme is being overly tricky is to run it on a Macintosh XL. Since the floppy disk hardware is different this will usually demonstrate an unwanted hardware dependency. Be wary of schemes that don't allow installation on a hard disk. If the program cannot be installed on a hard disk, it may be relying upon things that are prone to change. Don't use schemes that access the hardware directly. All Macintosh software should go through the various managers in the ROM to maintain compatibility. Any code that sidesteps the ROM will be viewed as having said "It's OK to make me revise myself."

Check errors returned as function results

All of the Operating System functions, as well as some of the Toolbox functions, will return result codes as the value of the function. Don't ignore these result codes. If a program ignores the result codes, it is possible to have any number of bad things happen to the program. The result code is there to tell the program that something went wrong; if the program ignores the fact that something is wrong, that program will probably be killed by whatever went wrong. (Bugs do not like to be ignored.) If a program checks errors, an anomaly can be nipped in the bud, before something really bizarre happens.

Why it's Bad

A program that ignores result codes is skipping valuable information. This information can often prevent a program from crashing and keep it from losing data.

How to avoid becoming a skipper

Always write code that is defensive. Assume that everyone and everything is out to kill you. Trust no one. An example of error checking is:

```
myRezzie := GetResource (myResType, myResId);
IF  myRezzie = nil  THEN  ErrorHandler ('Who stole my resource...');
```

Another example:

```
fsErrCode := FSOpen ('MyFile', myVRefNum, myFileRefNum);
IF fsErrCode <> noErr  THEN ErrorHandler (fsErrCode, 'File error');
```

And another:

```
myTPPrPort := PrOpenDoc (myTHPrint, nil, nil);
IF  PRError <> noErr  THEN  ErrorHandler (PRError, 'Printing error');
```

Any use of Operating System functions should presume that something nasty can happen, and have code to handle the nasty situations. Printing calls, File Manager calls, Resource Manager calls, and Memory Manager calls are all examples of Operating System functions that should be watched for returning errors. Always, always check the result codes from Memory Manager calls. Big memory machines are pretty common now, and it is easy to get cavalier about memory, but realize that someone will always want to run the program under Switcher, or on smaller Macintoshes. It never hurts to check, and always hurts to ignore it.

How to find skippers

This is easy: just do weird things while the program is running. Put in locked or unformatted disks while the program is running. Use unconventional command sequences. Run out of disk space. Run on 128K Macintoshes to see how the program deals with running out of memory. Run under Switcher for the same reason. (Programs that die while running under Switcher are often not Switcher's fault, and are in fact due

to faulty memory management.) Print with no printer connected to the Macintosh. Pop disks out of the drives with the Command-Shift sequence, and see if the program can deal with no disk. When a disk-switch dialog comes up, press Command-period to pass back an error to the requesting program (128K ROMs only). Torturing otherwise well-behaved programs can be quite enjoyable, and a number of users enjoy torturing the program as much as the program enjoys torturing them. For the truly malicious, run the debugger and alter error codes as they come back from various routines. Sure it's a dirty low-down rotten thing to do to a program, but we want to see how far we can push the program. (This is also a good way to check your error handling.) It's one thing to be an optimist, but it's quite another to assume that nothing will go wrong while a program is running.

Accessing hardware directly

Sometimes it is necessary to go directly to the Macintosh hardware to accomplish a specific task for which there is no ROM support. Early hard disks that used the serial ports had no ROM support. Those disks needed to use the SCC chip (the 8530 communication chip) in a high-speed clocked fashion. Although it is a valid function, it is not something that is supported in the ROM. It was therefore necessary to go play with the SCC chip directly, setting and testing various hardware registers in the chip itself. Another example of a valid function that has no ROM support is the use of the alternate video page for page-flipping animation. Since there is no ROM call to flip pages, it is necessary to go play with the right bit in the VIA chip (6522 Versatile Interface Adapter). Going directly to the hardware does not automatically throw a program into the incompatible group, but it certainly lowers its odds.

Why it's bad

Going directly to the hardware poses any number of problems for enlightened programs that are trying to maintain compatibility across the various versions of the Macintosh. On the Macintosh XL for example, a lot of the hardware is found in different locations, and in some cases the hardware doesn't exist. On the XL there is no sound chip. Programs that go directly to the sound hardware will find they don't work correctly on an XL. If the same program were to go through the Sound Manager, it would work fine, although the sound would not be the same as expected. Since the Macintosh is heavily oriented to the software side of things, expecting various hardware to always be available is not a safe bet. Choosy programmers choose to leave the hardware to the ROM.

How to avoid having a hard attack

Don't read or write the hardware. Exhaust every possible conventional approach before deciding to really get down and dirty. If there is a Manager in the ROM for the operation you wish to perform, it is far better to use the Manager than to go directly to the hardware. Compatibility at the hardware level can very rarely be maintained, but compatibility at the Manager level is a prime consideration. If a program is down to the last ditch effort, and cannot get the support from the ROM that is desired, then access the hardware in an enlightened approach. The really bad way to do it:

```
VIA := Pointer ($EFE1FE);  { sure it's the base address today...}
                            { This is bad.  Hard-coded number. }
```

The with-it, inspired programmer of the eighties does something like:

```
TYPE LongPointer = ^LongInt;

VAR  VIA: LongPointer;
     VIABase: LongInt;

VIA := Pointer ($1D4);  { the address of the low-memory global. }
VIABase := VIA^;        { get the low-memory variable's value }
                       { Now VIABase has the address of the chip }
```

The point here is that the best way to get the address of a hardware chip is to ask the system where it currently is to be found. The system always knows where the pieces of the system are, and will always know for every incarnation of the Macintosh. There are low-memory global variables for all of the pieces of hardware currently found in the Macintosh. This includes the VIA, the SCC, the Sound Chip, the IWM, and the video display. Whenever you are stuck with going to the hardware, use the low-memory globals. The fact that a program goes directly to the hardware means that it is risking imminent incompatibility, but using the low-memory global will ensure that the program has the best odds. It's like going to Las Vegas: if you don't gamble at all, you don't lose any money; if you have to gamble, play the game that you lose the least on.

How to find hard attacks

Run the suspicious program on the Macintosh XL. Nearly all of the hardware is in a different memory location on the XL. If a program has a hard-coded hardware address in it, it will fail. It may crash, or it might not perform the desired task, but it won't work as advertised. This unfortunately, is not a completely legitimate test, since the XL does not have some of the hardware of other Macintoshes, and some of the hardware that is there has the register mapping different. This means that it is possible to play by the rule of using the low-memory global and still be incompatible.

Don't use bits that are reserved

Occasionally during the life of a Macintosh programmer, there comes a time when it is necessary to bite the bullet and use a low-memory global. These are very sad days, since it has been demonstrated (by history) that low-memory global variables are a mysterious lot, and not altogether friendly. One fellow in particular is known as ROM85, a word located at $28E. This particular variable has been documented as the way to determine if a program is running on the 128K ROMs or not. Notably, the top most bit of that word is the determining bit. This means that the rest of the bits in that word are reserved, since nothing is described about any further bits. Remember, if it doesn't say, assume it's reserved. If it's reserved, don't depend upon it. Take the cautious way out and assume that the other bits that aren't documented are used for Switcher local variables, or something equally wild. An example of a bad way to do the comparison is:

```
VAR  Rom85Ptr: WordPtr;
     RomsAre64: Boolean;

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF  Rom85Ptr^ = $7FFF  THEN  RomsAre64 := False  { Bad test. }
ELSE  RomsAre64 := True;
```

This is a bad test since the comparison is testing the value of **all** of the bits, not only the one that is valid. Since the other bits are undocumented, it is impossible to know what they are used for. Assume they are used for something that is arbitrarily random, and take the safe way out.

How to avoid being bitten

```
VAR     ROM85Ptr: Ptr

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF BitTst(ROM85Ptr,0) THEN RomsAre64 := True {Good--tests only hi-bit}
ELSE  RomsAre64 := False;
```

This technique will ensure that when those bits are documented, your program won't be using them for the wrong things. Beware of trojan bits.

Don't use undocumented stuff. Be very careful when you use anything out of the ordinary stream of a high-level language. For instance, in the ROM85 case, it is very easy to make the mistake of checking for an absolute value instead of testing the actual bit that encodes the information. Whenever a program is using low-memory globals, be sure that only the information desired is being used, and not some undocumented (and hence reserved) bits. It's not always easy to determine what is reserved and what isn't, so conservative programmers always use as little as possible. Be wary of the strange bits, and accept rides from none of them. The ride you take might cause you to revise your program.

How to find those bitten

Since there are such a multitude of possible places to get killed, there is no simple way to see what programs are using illegal bits. As time goes by it will be possible to find more of these cases by running on various versions of the Macintosh, but there will probably never be a comprehensive way of finding out who is accepting strange rides, and who is not. Whenever the use of a bit changes from reserved status to active, it will be possible to find those bugs via extensive testing. From a source level, it would be advisable to look over **any** use of low-memory globals, and eye them closely for inappropriate bit usage. Do a global search for the $ (which describes those ubiquitous hexadecimal numbers), and when found see if the use of the number is appropriate. Trust no one that is not known. If they are documented, they will stay where they are, and have the same meaning. Be very careful in realms that are undocumented. Bits that suddenly jump from reserved to active status have been known to cause more than one program to have a sudden anxiety attack. It is very unnerving to watch a program go from calm and reassuring to rabid status. Users have been known to drop their keyboards in sudden shock (which is bad on the keyboards).

Summary

So what does all this mean? It means that it is getting harder and harder to get away with minor bugs in programs. The minor bugs of yesterday are the major ones of today. No one will yell at you for having bugs in your program, since all programs have bugs of one form or another. The goal should be to make the programs run as smoothly and effortlessly as possible. The end-users will never object to bug-reduced programs.

What is the best way to test a program? A reasonably comprehensive test is to exercise all of the program's functions under the following situations:

- Use Discipline to be sure the program does not pass illegal things to the ROM.
- Use heap scramble and heap purge to be sure that handles are being used correctly, and that the memory management of the program is correct.
- Run with a checksum on memory locations 0...3 to see if the program writes to these locations.
- Run on a 128K Macintosh, or under Switcher with a small partition, to see how the program deals with memory-critical situations.
- Run on a 68020 system to see if the program is 68020-compatible and to make sure that changing system speed won't confuse the program.
- Run on a Macintosh XL to be sure that the program does not assume too much about the operating system, and to test screen handling.
- Run on an Ultra-Large screen to be sure that the screen handling is correct, and that there are no hard-coded screen dimensions.
- Run on 64K ROM machines to be sure new traps are not being used when they don't exist.
- Run under both HFS and MFS to be sure that the program deals with the file system correctly. (400K floppies are usually MFS.)

If a program can live through all of this with no Discipline traps, no checksum breaks, no system errors, no anomalies, no data loss and still get useful work done, then you deserve a gold medal for programming excellence. Maybe even an extra medal for conduct above and beyond the call of duty. In any case, you will know that you have done your job about as well as it can be done, with today's version of the rules, and today's programming tools.

Sounds like a foreboding task, doesn't it? The engineers in Macintosh Technical Support are available to help you with compatibility issues (we won't always be able to talk about new products, since we love our jobs, but we can give you some hints about compatibility with what the future holds).

Good luck.

#118: How To Check and Handle Printing Errors

Revised by: Pete "Luke" Alexander October 1990
Written by: Ginger Jernigan May 1987

This Technical Note formerly described how to check and properly handle errors that occur during printing with the Printing Manager.
Changes since March 1988: Merged contents into Technical Note #161.

This Note formerly described how to check and properly handle Printing Manager errors. This information is now contained in Technical Note #161, A Printing Loop That Cares…, which also includes a table of Printing Manager error codes

#119: Determining If Color QuickDraw Exists

See: Technical Note #129—SysEnvirons

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

This note formely described a way to determine if Color QuickDraw is present on a particular machine. We now recommend that you call `SysEnvirons` to find out, as described in Technical Note #129.

#120: Principia Off-Screen Graphics Environments

Updated by: Forrest Tanaka — March 1992
Written by: Forrest Tanaka — October 1991
Inspired by: Jim Friedlander, Rick Blair, and Rich Collyer

Using Color QuickDraw to draw off screen is a common requirement of applications and other kinds of programs that run on the Macintosh. This Note discusses what Color QuickDraw needs in a graphics environment and how to create one for off-screen drawing. A brief discussion of GWorlds, which are off-screen graphics environments that are set up by the system, is given to help you decide whether to use them or the do-it-yourself techniques described in this Note for setting up an off-screen graphics environment. The author's intent is to provide concepts and routines for creating an off-screen graphics environment, and also to explain why existing routines for off-screen drawing act as they do.

Many, many thanks go to Guillermo Ortiz, Konstantin Othmer, Bruce Leak, and Jon Zap for all their expertise on this subject, Rich Collyer, Rick Blair, and Jim Friedlander for paving the way, and especially to all people who inspired this update by asking great off-screen drawing questions.

Changes since October 1991: A very embarrassing bug was found in CreateOffScreen and UpdateOffScreen. If you try to create a 16- or 32-bit off-screen graphics environment, you'll just get a paramErr. It won't do that now.

Off-Screening

The Macintosh, as with every other CPU ever made by Apple, has memory-mapped video. That is, what you see on the screen is just the visual representation of a part of memory that's reserved for the video hardware (that's stretching the truth just a bit in the case of the text screens of the original Apple computer, the Apple II line, and the Apple III because there's also a character generator in those, but the overall process still looks roughly the same). If you change the contents of a memory location in this part of memory, then you'll see the corresponding location on the screen change when the video hardware draws the next frame or field of video. The resident raster graphics package, QuickDraw in the case of the Macintosh, draws images by stuffing the right values into the right places in the part of memory reserved for the video display. The resulting image on the screen looks like a line or perhaps an oval if you asked QuickDraw to draw a line or an oval, or it could be an entire complex image if you asked QuickDraw to draw one. This is normal, on-screen drawing.

Because video memory is a part of RAM just like any other part of RAM in the memory map of the Macintosh (or almost like; video memory might exist on a NuBus™ video card, but it's still RAM), QuickDraw can be told to draw into a part of memory that isn't reserved for the video hardware, maybe into a part of your own application's heap. When you tell QuickDraw to draw into a part of memory that's not reserved for the video hardware, you can't see any of the results. This is off-screen drawing. There are plenty of perfectly good reasons to do this, such as providing storage for a paint-style document or to smoothly animate an image, but the assumption here is that you have a perfectly good reason to do this so you're more interested in the "how" of it instead of the "why" of it. If you need to know why, there are several books that cover off-screen drawing and the perfectly good reasons to do such a thing. A good place to start is Scott Knaster's book, *Macintosh Programming Secrets*, referenced at the end of this Note.

This Note is divided into these major sections:

- The introduction is the part that you're reading now.
- "The Building Blocks" provides an overview of the data structures that you need to tell Color QuickDraw to draw off screen.
- "Building the Blocks" discusses the construction and initialization of these data structures.
- "Playing With Blocks" shows an example of the use of these structures to draw off screen.
- "Put That Checkbook Away!" discusses some variations of these techniques to handle off-screen drawing for special cases.
- "The GWorld Factor" provides a brief overview of GWorlds, how to use them, and how they compare and contrast to the manual techniques that are described in most of this Note.

Those of you who aren't quite sure whether to use GWorlds or the do-it-yourself techniques might want to skip ahead for a moment to "The GWorld Factor" just in case doing it yourself is a waste of time. In any case, it's a good idea to read this whole Note because the concepts are mostly the same whether you're using GWorlds or not. GWorlds just make the process a lot easier, and they let you take advantage of the 8•24 GC video card. But, we're not in that section of the Note yet.

The Building Blocks

Before you can tell QuickDraw to draw off of the screen, you'll need to build three major data structures: a `CGrafPort`, a `PixMap`, and a `GDevice`. You'll also need a couple of tables that define the colors involved with drawing to and copying from the off-screen image: the color table and the inverse table. Of course, you'll need the pixel image itself, which is often called the "pixel buffer" or the "image buffer" or the "off-screen buffer" or just "the buffer." It's always called the "pixel image" in this Note. It doesn't necessarily buffer anything anyway.

The CGrafPort

A `CGrafPort` describes a drawing environment, and it's the color version of the `GrafPort` structure that's described on pages 147 through 155 in the QuickDraw chapter of *Inside Macintosh Volume* I. The drawing environment consists of, among other things, the size and location of the graphics pen, the foreground and background colors to use when something is drawn, the pattern to use, the region to clip all drawing to, and the portion of a pixel image that the `CGrafPort` logically exists in. Any initialized `CGrafPort` or `GrafPort` can be set as the current port through the `_SetPort` routine. The current port is a set of parameters that are implicitly passed to most QuickDraw routines.

The most important reason to build a new CGrafPort when you draw off screen rather than using an existing CGrafPort is so that switching between drawing to an off-screen graphics environment and drawing to one or more windows (each of which is an extended GrafPort or CGrafPort structure) on the screen is very easy. Some people use just one CGrafPort to share between on-screen and off-screen graphics environments, and switch their PixMap structures to switch between drawing on screen and drawing off screen. That does work, but if the off-screen and on-screen graphics environments have a different clipRgn, visRgn, pen characteristic, portRect, or any other characteristics that are different, then those must be switched at that time too. If you instead create a CGrafPort that's dedicated to one graphics environment, then a simple call to _SetPort effectively switches all these things for you at once. That's why every window on the screen comes with its own port. A simple call to _SetPort switches between the characteristics of each window even if each window has radically different drawing characteristics.

The CGrafPort data structure is more completely described in the "Color QuickDraw" chapter of *Inside Macintosh* Volume V, pages 49 through 52, and in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-12 through 16-13.

The PixMap

A pixel image alone is just a formless blob of memory. Pixel maps, defined by the PixMap structure, describe pixel images, giving them a form and structure that's suitable for Color QuickDraw to draw into them and copy from them. The PixMap structure tells you the dimensions and location in memory of the pixel image, its coordinate system, and the depth and format of the pixels. Pixel maps that describe indexed-color pixel images additionally describe the colors that are represented by the values of the pixels in the pixel image. This is done through the color table, also known as the color look-up table or CLUT. Color tables are attached to pixel maps through their pmTable field. Direct-color pixel images have pixel values that describe their own colors, and so color tables aren't needed for those.

The PixMap structure is described in the "Color QuickDraw" chapter of *Inside Macintosh* Volume V, pages 52 through 55, and in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-11 through 16-12. The concept of direct-color and indexed-color pixels is described in this same chapter on pages 16-16 through 16-18, and also in the "Color QuickDraw" chapter of the same volume on pages 17-4 through 17-10.

The GDevice

Graphics devices, defined by the GDevice structure, describe color environments. They're the most misunderstood data structure when it comes to off-screen graphics environments for three major reasons: first, they're not originally documented as being relevant to humans; second, they look as though they're only for screens; and third, it looks as though color tables describe color environments. We can dispose of these myths here: graphics devices are documented as being useful to humanity in this Note at least; they're critically important for both on-screen and off-screen drawing; and color tables describe the colors in pixel images, *not* color environments.

What's all this about color environments? In theory, there are virtually three hundred trillion colors available with Color QuickDraw through the 48-bit RGBColor record. In reality, there are never this many colors available, and in fact there might be only two. Color QuickDraw maps the theoretical color that you specify to the pixel value of the closest available color in the current color environment. This can be done with a color table, but that's not very efficient. Finding the closest available color to an RGBColor in a color table means searching the entire color table for that one closest color. If that's done just once, then performance isn't much of an issue, but if it's done many times, the performance hit could be significant. A very bad case of this is _CopyBits, where every pixel value in the source image is converted to an RGBColor by looking it up in the color table of the source PixMap. If the color table of the destination PixMap had to be searched to find the closest available color for every pixel in the source PixMap, then the performance of even the most straightforward _CopyBits call could be a lot slower than it has to be.

To avoid this performance hit, the current GDevice provides an inverse table and a device type which are used to determine the available set of colors. Inverse tables are anticolor tables. Where color tables give you a color for a given pixel value, inverse tables give you a pixel value for a given color. Every conceivable color table has a corresponding conceivable inverse table, just as every positive real number has a corresponding negative real number, or every Mr. Spock has a corresponding Mr. Spock with a goatee. The device type specifies whether the color environment uses the indexed-color, fixed-color, or direct-color model. In the direct-color model, the inverse table is empty. Only the indexed-color and direct-color models are described in this Note.

When you specify a color in an indexed-color environment, Color QuickDraw takes the RGBColor specification and converts it into a value that can be used as an index into the inverse table of the current GDevice. To do this conversion, Color QuickDraw takes the top few significant bits of each color component and combines them into part of a 16-bit word, blue bits in the least significant bits, green bits right above it, and the red bits right above green bits. Any unused bits are in the most significant bits of the 16-bit word. The resulting 16-bit word is used as an index into the inverse table. The value in the inverse table at that index is the pixel value which best represents that color in the current color environment. The number of bits of each component that are used is determined by what's called the "resolution" of the inverse table. Almost always, the resolution of an inverse table is four bits, meaning the most significant four bits of each component are used to form the index into the inverse table. Figure 1 shows how an RGBColor record is converted to an index into an inverse table when the inverse-table resolution is four.

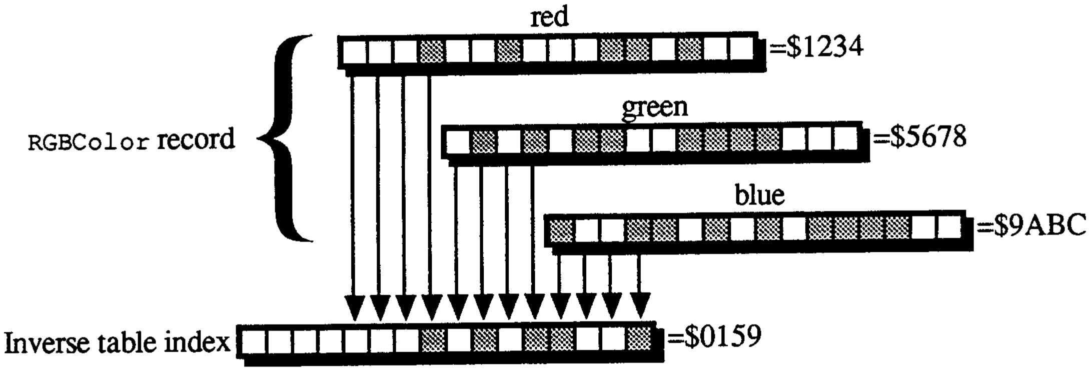


Figure 1 Conversion of RGBColor Record to Inverse-Table Index

The same process is used when _CopyBits is called with an indexed-color destination. Each pixel in the source pixel image is converted to an RGBColor either by doing a table look-up of the source pixel map's color table if the source pixel image uses indexed colors, or by expanding the pixel value to an RGBColor record if the source pixel image uses direct colors. The resulting RGBColor is then used to look up a pixel value in the inverse table of the current GDevice, and this pixel value is put into the destination pixel image.

If you specify a color in a direct-color environment, then the resulting RGBColor is converted to a direct pixel value by the processes that are shown on pages 17-6 through 17-9 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI.

Usually, inverse-table look-up involves an extra step to find what are called "hidden colors" using proprietary information that's stored at the end of the inverse table. With an inverse-table resolution of four, only 16 shades of any particular component can be distinguished, and that's often not enough. An inverse table with a resolution of five is much larger, but it still only gives you 32 shades of any component. Hidden colors are looked up after the normal inverse-table look-up to give a much more accurate representation of the specified color in the current color environment than the inverse-table look-up alone can produce. Sometimes, most notably when the arithmetic transfer modes are used or if dithering is used, the hidden colors are ignored.

When a new color table is assigned to a `PixMap` or when its existing color table is modified, then a new corresponding inverse table should be generated for the `GDevice` that'll be used when drawing into that environment. Normally, this happens automatically without you having to do any more than inform Color QuickDraw of the change. This is described in more detail in "Changing the Off-Screen Color Table" later in this Note.

Graphics devices are documented in the "Graphics Devices" chapter of *Inside Macintosh* Volume VI which supersedes the "Graphics Devices" chapter of *Inside Macintosh* Volume V. They're also discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-13 through 16-14. The inverse-table mechanism is described in the "Color Manager" chapter of *Inside Macintosh* Volume V, pages 137 through 139.

All Together Now

There are a lot of different ways to put the three structures together, and this Note discusses the architecture that's shown in Figure 2. This architecture is useful when you want a simple, atomic, off-screen graphics environment.

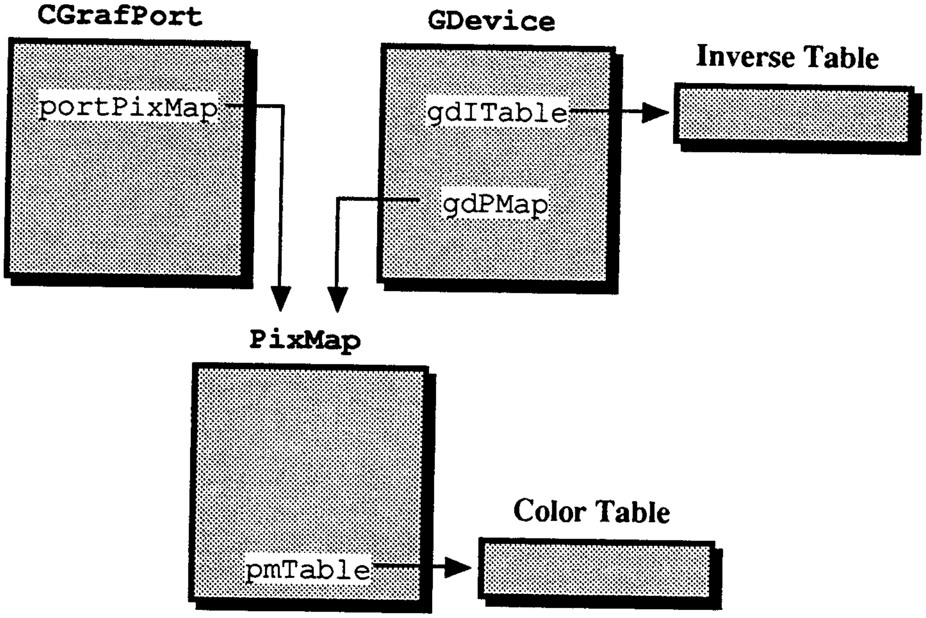


Figure 2 Relationships Between Structures for Off-Screen Drawing

Notice that there's no way to get to the `GDevice` from the `CGrafPort`, nor is there a way to get to the `CGrafPort` from the `GDevice`, though the `PixMap` can be found through either one. Your application must keep track of both the `CGrafPort` and the `GDevice`.

Building the Blocks

As with just about any algorithm, there are many ways to put the different structures together that form an off-screen graphics environment. This section covers just one way to build the architecture that's shown in Figure 2.

Building the CGrafPort

The `CGrafPort` structure is the easiest one to put together because the `_OpenCPort` routine initializes so many of the fields of the `CGrafPort` structure for you. It also allocates and initializes the structures that are attached to every `CGrafPort`, such as the `visRgn`, `clipRgn`, `grafVars` handle, and so forth. Most of these are initialized with values that are fine for general purposes, but the `visRgn`, `clipRgn`, and `portRect` fields should be set to the desired boundary rectangle of the off-screen graphics environment. What follows is an overview of each of the fields that you have to worry about when you're setting up a `CGrafPort` for drawing off screen.

`portPixMap` handle to the off-screen `PixMap`. `_OpenCPort` initializes this field to a copy of the `PixMap` that's attached to the `gdPMap` field of the current `GDevice`. An overview of setting up this `PixMap` for drawing off screen is given in "Building the PixMap" later in this Note.

`portRect` specifies the rectangular area of the associated pixel image that this `CGrafPort` controls. This field should be set to the desired rectangular area of the off-screen image because `_OpenCPort` doesn't necessarily initialize it to this size. Usually, the top-left corner of this rectangle has the coordinates (0, 0), but not necessarily so.

`visRgn` handle to the region that specifies the visible area into which you can draw. `_OpenCPort` doesn't necessarily initialize it to the size of the off-screen image, so it should be set to the same size and coordinates as the `portRect` and left at that. This field is more important for windows because parts of them can be hidden by other windows.

`clipRgn` handle to the region that specifies the logical area into which you can draw. `_OpenCPort` initializes it to cover the entire QuickDraw coordinate plane. It's usually a good idea to set it to the same size and coordinates as the `portRect` to avoid problems if the `clipRgn` is scaled or translated, which causes its signed integer coordinates to overflow and turn it into an empty region. One of the most common cases of this occurs when a picture that's created in this `CGrafPort` is drawn into a destination rectangle that's any larger than or translated from the original picture frame. Everything in the picture, including the clip region, is scaled to fit the destination rectangle. If the clip region covers the entire QuickDraw coordinate plane, then its coordinates overflow their signed integer bounds, and the clip region becomes logically empty. The result is that nothing is drawn.

The CreateOffScreen routine in Listing 1 creates an off-screen graphics environment, given a boundary rectangle, pixel depth, and color table, and it returns a new off-screen `CGrafPort` and `GDevice`, along with an error code. The desired pixel depth in bits per pixel is given in the `depth` parameter. If the pixel depth is eight or less, then an indexed-color graphics environment is created and a color table is required in the `colors` parameter. If the pixel depth is 16 or 32 bits per pixel and 32-Bit QuickDraw is available, then a direct-color graphics environment is created and the colors parameter is ignored. If 32-Bit QuickDraw isn't available, then a pixel depth of 16 or 32 bits per pixel results in CreateOffScreen doing nothing more than returning a parameter error. A description of CreateOffScreen is given following the listing.

MPW Pascal Listing 1

```
FUNCTION CreateOffScreen(
   bounds:         Rect;        {Bounding rectangle of off-screen}
   depth:          Integer;     {Desired number of bits per pixel in off-screen}
   colors:         CTabHandle;  {Color table to assign to off-screen}
   VAR retPort:    CGrafPtr;    {Returns a pointer to the new CGrafPort}
   VAR retGDevice: GDHandle     {Returns a handle to the new GDevice}
   ): OSErr;

   CONST
      kMaxRowBytes = $3FFE; {Maximum number of bytes in a row of pixels}

   VAR
      newPort:      CGrafPtr;      {Pointer to the new off-screen CGrafPort}
      newPixMap:    PixMapHandle;  {Handle to the new off-screen PixMap}
      newDevice:    GDHandle;      {Handle to the new off-screen GDevice}
      qdVersion:    LongInt;       {Version of QuickDraw currently in use}
      savedPort:    GrafPtr;       {Pointer to GrafPort used for save/restore}
      savedState:   SignedByte;    {Saved state of color table handle}
      bytesPerRow:  Integer;       {Number of bytes per row in the PixMap}
      error:        OSErr;         {Returns error code}

BEGIN
   (* Initialize a few things before we begin *)
   newPort := NIL;
   newPixMap := NIL;
   newDevice := NIL;
   error := noErr;

   (* Save the color table's current state and make sure it isn't purgeable *)
   IF colors <> NIL THEN
      BEGIN
         savedState := HGetState(Handle(colors));
         HNoPurge(Handle(colors));
      END;

   (* Calculate the number of bytes per row in the off-screen PixMap *)
   bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;

   (* Get the current QuickDraw version *)
   error := Gestalt(gestaltQuickdrawVersion, qdVersion);
   error := noErr;

   (* Make sure depth is indexed or depth is direct and 32-Bit QD installed *)
   IF (depth = 1) OR (depth = 2) OR (depth = 4) OR (depth = 8) OR
        (((depth = 16) OR (depth = 32)) AND (qdVersion >= gestalt32BitQD)) THEN
      BEGIN
         (* Maximum number of bytes per row is 16,382; make sure within range *)
         IF bytesPerRow <= kMaxRowBytes THEN
            BEGIN
               (* Make sure a color table is provided if the depth is indexed *)
               IF depth <= 8 THEN
                  IF colors = NIL THEN
                     (* Indexed depth and clut is NIL; is parameter error *)
                     error := paramErr;
            END
         ELSE
            (* # of bytes per row is more than 16,382; is parameter error *)
            error := paramErr;
      END
   ELSE
      (* Pixel depth isn't valid; is parameter error *)
      error := paramErr;

   (* If sanity checks succeed, then allocate a new CGrafPort *)
   IF error = noErr THEN
      BEGIN
         newPort := CGrafPtr(NewPtr(SizeOf (CGrafPort)));
         IF newPort <> NIL THEN
            BEGIN
               (* Save the current port *)
               GetPort(savedPort);

               (* Initialize the new CGrafPort and make it the current port *)
               OpenCPort(newPort);

               (* Set portRect, visRgn, and clipRgn to the given bounds rect *)
               newPort^.portRect := bounds;
               RectRgn(newPort^.visRgn, bounds);
               ClipRect(bounds);

               (* Initialize the new PixMap for off-screen drawing *)
               error := SetUpPixMap(depth, bounds, colors, bytesPerRow,
                     newPort^.portPixMap);
               IF error = noErr THEN
                  BEGIN
                     (* Grab the initialized PixMap handle *)
                     newPixMap := newPort^.portPixMap;

                     (* Allocate and initialize a new GDevice *)
                     error := CreateGDevice(newPixMap, newDevice);
                  END;

               (* Restore the saved port *)
               SetPort(savedPort);
            END
         ELSE
            error := MemError;
      END;

   (* Restore the given state of the color table *)
   IF colors <> NIL THEN
      HSetState(Handle(colors), savedState);

   (* One Last Look Around The House Before We Go… *)
   IF error <> noErr THEN
      BEGIN
         (* Some error occurred; dispose of everything we allocated *)
         IF newPixMap <> NIL THEN
            BEGIN
               DisposCTable(newPixMap^^.pmTable);
               DisposPtr(newPixMap^^.baseAddr);
            END;
         IF newDevice <> NIL THEN
            BEGIN
               DisposHandle(Handle(newDevice^^.gdITable));
               DisposHandle(Handle(newDevice));
            END;
         IF newPort <> NIL THEN
            BEGIN
               CloseCPort(newPort);
            DisposPtr(Ptr(newPort));
         END;
     END
   ELSE
     BEGIN
        (* Everything's OK; return refs to off-screen CGrafPort and GDevice *)
        retPort := newPort;
        retGDevice := newDevice;
     END;
   CreateOffScreen := error;
END;
```

MPW C Listing 1

```
#define kMaxRowBytes 0x3FFE /* Maximum number of bytes in a row of pixels */

OSErr CreateOffScreen(
    Rect       *bounds,     /* Bounding rectangle of off-screen */
    short      depth,       /* Desired number of bits per pixel in off-screen */
    CTabHandle colors,      /* Color table to assign to off-screen */
    CGrafPtr   *retPort,    /* Returns a pointer to the new CGrafPort */
    GDHandle   *retGDevice) /* Returns a handle to the new GDevice */
{
    CGrafPtr     newPort;     /* Pointer to the new off-screen CGrafPort */
    PixMapHandle newPixMap;   /* Handle to the new off-screen PixMap */
    GDHandle     newDevice;   /* Handle to the new off-screen GDevice */
    long         qdVersion;   /* Version of QuickDraw currently in use */
    GrafPtr      savedPort;   /* Pointer to GrafPort used for save/restore */
    SignedByte   savedState;  /* Saved state of color table handle */
    short        bytesPerRow; /* Number of bytes per row in the PixMap */
    OSErr        error;       /* Returns error code */

    /* Initialize a few things before we begin */
    newPort = nil;
    newPixMap = nil;
    newDevice = nil;
    error = noErr;

    /* Save the color table's current state and make sure it isn't purgeable */
    if (colors != nil)
    {
        savedState = HGetState( (Handle)colors );
        HNoPurge( (Handle)colors );
    }

    /* Calculate the number of bytes per row in the off-screen PixMap */
    bytesPerRow = ((depth * (bounds->right - bounds->left) + 31) >> 5) << 2;

    /* Get the current QuickDraw version */
    (void)Gestalt( gestaltQuickdrawVersion, &qdVersion );

    /* Make sure depth is indexed or depth is direct and 32-Bit QD installed */
    if (depth == 1 || depth == 2 || depth == 4 || depth == 8 ||
            ((depth == 16 || depth == 32) && qdVersion >= gestalt32BitQD))
    {
        /* Maximum number of bytes per row is 16,382; make sure within range */
        if (bytesPerRow <= kMaxRowBytes)
        {
            /* Make sure a color table is provided if the depth is indexed */
            if (depth <= 8)
              if (colors == nil)
                  /* Indexed depth and clut is NIL; is parameter error */
                  error = paramErr;
        }
        else
            /* # of bytes per row is more than 16,382; is parameter error */
            error = paramErr;
    }
    else
        /* Pixel depth isn't valid; is parameter error */
        error = paramErr;

    /* If sanity checks succeed, then allocate a new CGrafPort */
    if (error == noErr)
    {
        newPort = (CGrafPtr)NewPtr( sizeof (CGrafPort) );
        if (newPort != nil)
        {
            /* Save the current port */
            GetPort( &savedPort );

            /* Initialize the new CGrafPort and make it the current port */
            OpenCPort( newPort );

            /* Set portRect, visRgn, and clipRgn to the given bounds rect */
            newPort->portRect = *bounds;
            RectRgn( newPort->visRgn, bounds );
            ClipRect( bounds );

            /* Initialize the new PixMap for off-screen drawing */
            error = SetUpPixMap( depth, bounds, colors, bytesPerRow,
                    newPort->portPixMap );
            if (error == noErr)
            {
                /* Grab the initialized PixMap handle */
                newPixMap = newPort->portPixMap;

                /* Allocate and initialize a new GDevice */
                error = CreateGDevice( newPixMap, &newDevice );
            }

            /* Restore the saved port */
            SetPort( savedPort );
        }
        else
            error = MemError();
    }

    /* Restore the given state of the color table */
    if (colors != nil)
        HSetState( (Handle)colors, savedState );

    /* One Last Look Around The House Before We Go… */
    if (error != noErr)
    {
        /* Some error occurred; dispose of everything we allocated */
        if (newPixMap != nil)
        {
            DisposCTable( (**newPixMap).pmTable );
            DisposPtr( (**newPixMap).baseAddr );
        }
        if (newDevice != nil)
        {
            DisposHandle( (Handle)(**newDevice).gdITable );
            DisposHandle( (Handle)newDevice );
        }
        if (newPort != nil)
        {
            CloseCPort( newPort );
            DisposPtr( (Ptr)newPort );
        }
    }
    else
    {
        /* Everything's OK; return refs to off-screen CGrafPort and GDevice */
        *retPort = newPort;
        *retGDevice = newDevice;
    }
    return error;
}
```

CreateOffScreen begins by making sure that the color table, if there is one, doesn't get purged during the time that the off-screen graphics environment is created. Then, a sanity check is done for the given depth, bounds, and color table. The depth must be either 1, 2, 4, or 8 bits per pixel, or additionally 16 or 32 bits per pixel if 32-Bit QuickDraw is available. If these conditions aren't satisfied, then it's decided that there's an error in the parameter list, and CreateOffScreen does nothing more. To determine whether 32-Bit QuickDraw is available or not, the `_Gestalt` routine is used. If `_Gestalt` returns a value that's equal to or greater than the constant `gestalt32BitQD`, then 32-Bit QuickDraw is available and depths of 16 and 32 bits per pixel are supported. It's not necessary to determine whether `_Gestalt` is available or not because it's implemented as glue code in the Macintosh Programmer's Workshop.

A check is then done to determine whether the number of bytes in each row of the off-screen pixel image is too much for QuickDraw to handle. Color QuickDraw can handle up to and including 16,382 ($3FFE) bytes in each row of any pixel image. If the required number of bytes per row exceeds this amount, then CreateOffScreen decides that there's an error in the parameter list and does nothing more. The minimum number of bytes in a row that's enough to cover the given boundary rectangle at the given pixel depth is calculated with the formula:

```
bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;
```

This formula multiplies the number of pixels across the `PixMap` by the pixel depth to get the number of bits, and then this is divided by eight to get the number of bytes. This division by eight looks very strange because the number of bytes per row must be even, so this formula takes advantage of integer division and multiplication to make the result come out even. This particular formula additionally makes sure that the number of bytes per row is a multiple of four. This helps optimize the performance of Color QuickDraw operations because it allows Color QuickDraw to refer to each row beginning on a long word boundary in memory.

The last sanity check is to make sure that a color table is given as a parameter if it's needed. Indexed-color graphics environments need color tables, so if the given pixel depth is eight or less (which implies an indexed-color graphics environment) and the given color table is NIL, then CreateOffScreen decides that there's an error in the parameter list and does nothing more. If the given pixel depth is 16 or 32 (which implies a direct-color graphics environment), then CreateOffScreen ignores the given color table.

If all the sanity checks succeed, then the off-screen `CGrafPort` is allocated using a call to `_NewPtr`, and then it's initialized and opened as a `CGrafPort` by passing the resulting pointer to `_OpenCPort`. Because `_OpenCPort` makes the new `CGrafPort` the current port, the current port is first saved so that it can be restored as the current port when CreateOffScreen is done.

As mentioned above, the `_OpenCPort` doesn't necessarily initialize the `portRect`, `visRgn`, and `clipRgn` of the new `CGrafPort` to the areas that are needed for any particular off-screen graphics environment. So, the given boundary rectangle is assigned to the portRect field, _RectRgn is called to make the visRgn equal to the given boundary rectangle, and _ClipRect is called to set the clipRgn so that it's equal to the given boundary rectangle.

The PixMap in the portPixMap field needs to be initialized for off-screen drawing, and that's handled by the SetUpPixMap routine that's described and defined in "Building the PixMap" later in this Note. Similarly, the off-screen GDevice must be created and initialized. That's handled by the CreateGDevice routine that's described and defined in "Building the GDevice" later in this Note.

Once these things are done, CreateOffScreen returns a pointer to the off-screen CGrafPort in the retPort parameter and a handle to the off-screen GDevice in the retGDevice parameter. The way to use these references is described in "Playing With Blocks" later in this Note.

Building the PixMap

_OpenCPort initializes the portPixMap field of the CGrafPort it's initializing with a copy of the PixMap of the current GDevice. When the CreateOffScreen routine described earlier executes, the current GDevice is unknown. So, all the fields of the PixMap that the new CGrafPort receives must be initialized so that it can be used for drawing off screen.* What follows is an overview of each of the PixMap fields and how they should be initialized for off-screen drawing.

baseAddr — pointer to the off-screen pixel image. The off-screen pixel image is allocated as a nonrelocatable block in the heap. The size of this block of memory is calculated from the rowBytes field, described next, multiplied by the number of rows in the given boundary rectangle.

rowBytes — number of bytes in each row of the pixel image. This value is calculated from the formula that's given in the CreateOffScreen routine. The most significant bit of this field should be set so that Color QuickDraw knows that this is a PixMap rather than a BitMap. The maximum value, ignoring the most significant bit, is 16,382.

bounds — defines the coordinate system and the dimensions of the pixel image. For most off-screen drawing, this should be a rectangle that covers the entire off-screen graphics environment.

pmVersion — set of internally and externally defined flags. As of 32-Bit QuickDraw 1.2, only the baseAddr32 flag is defined externally. This flag is described in "Choosing Your Off-Screen Memory" later in this Note. For most off-screen drawing, this field is set to zero.

packType — image compression scheme for pictures. The options for this field are discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 17-22 through 17-23. In this Note, image compression isn't discussed so this field is set to zero.

* This part of these routines really bothers me because it feels impure to initialize all the PixMap fields when _OpenCPort has initialized them already, just not in a way that's any good for off-screen drawing. I tried creating the GDevice and PixMap first and then calling _OpenCPort so that it initializes its PixMap for off-screen drawing, but then you end up with two pixel maps and that makes this tougher to explain, or you have to dispose of one PixMap which seems worse than the method I'm using.

`packSize` internally used field. This field is always set to zero.

`hRes` horizontal resolution of the pixel map. By default, the QuickDraw resolution is 72 dots per inch,which is the value this Note uses. This is a fixed-point field, so the actual value in this field is $00480000.

`vRes` vertical resolution of the pixel map. See the `hRes` description.

`pixelType` format of the pixels. In indexed-color pixel maps, this field holds zero. In direct-color pixel maps, this field holds the `RGBDirect` constant, which is equal to 16.

`pixelSize` number of bits in every pixel. For indexed-color pixels, this is 1, 2, 4, or 8 bits per pixel. For direct-color pixels, this is 16 or 32 bits per pixel.

`cmpCount` number of components in every pixel. In indexed-color pixel maps, this field is set to 1. In direct-color pixel maps, this field is set to 3. Sometimes it's handy to set this field to 4 in 32-bit deep pixel maps when they're being saved in a picture. See the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI, page 17-23, for details about this.

`cmpSize` number of bits in each color component. In indexed-color pixel maps, this field is set to the same value that's in the `pixelSize` field. In 16-bit deep direct pixel maps, this field is set to 5. In 32-bit deep direct pixel maps, this field is set to 8.

`planeBytes` not currently defined. This field is set to zero.

`pmTable` handle to the color table for indexed-color pixel maps. A method to create a color table is given in "About That Creation Thing . . ." later in this Note. In direct-color pixel maps, this field contains a handle to a dummy color table, and building one of these is shown in the SetUpPixMap routine in Listing 2.

`pmReserved` not currently defined. This field is set to zero.

The SetUpPixMap routine in Listing 2 initializes the `PixMap` that's passed to it in the `aPixMap` parameter so that it can be used in an off-screen graphics environment. The `depth`, `bounds`, and `color` parameters are the same as the ones passed to the CreateOffScreen routine. The `bytesPerRow` parameter is the number of bytes in each row of the off-screen pixel image. A description of SetUpPixMap follows the listing.

MPW Pascal Listing 2

```
FUNCTION SetUpPixMap(
   depth:       Integer;      {Desired number of bits/pixel in off-screen}
   bound:       Rect;         {Bounding rectangle of off-screen}
   colors:      CTabHandle;   {Color table to assign to off-screen}
   bytesPerRow: Integer;      {Number of bytes in each row of pixels}
   aPixMap:     PixMapHandle  {Handle to the PixMap being initialized}
   ): OSErr;

   CONST
```

```
      kDefaultRes = $00480000; {Default resolution is 72 DPI; Fixed type}

   VAR
      newColors:   CTabHandle; {Color table used for the off-screen PixMap}
      offBaseAddr: Ptr;        {Pointer to the off-screen pixel image}
      error:       OSErr;      {Returns error code}

BEGIN
   error := noErr;
   newColors := NIL;
    offBaseAddr := NIL;

   (* Clone the clut if indexed color; allocate a dummy clut if direct color *)
   IF depth <= 8 THEN
      BEGIN
         newColors := colors;
         error := HandToHand(Handle(newColors));
      END
   ELSE
      BEGIN
         newColors := CTabHandle(NewHandle(SizeOf(ColorTable) -
               SizeOf(CSpecArray)));
         error := MemError;
      END;
   IF error = noErr THEN
      BEGIN
         (* Allocate pixel image; long integer multiplication avoids overflow *)
         offBaseAddr := NewPtr(LongInt(bytesPerRow) * (bound.bottom -
               bound.top));
         IF offBaseAddr <> NIL THEN
            WITH aPixMap^^ DO
               BEGIN
                  (* Initialize fields common to indexed and direct PixMaps *)
                  baseAddr := offBaseAddr;     {Point to image}
                  rowBytes := BOR(bytesPerRow, {MSB set for PixMap}
                      $8000);
                  bounds := bound;             {Use given bounds}
                  pmVersion := 0;              {No special stuff}
                  packType := 0;               {Default PICT pack}
                  packSize := 0;               {Always zero when in memory}
                  hRes := kDefaultRes;         {72 DPI default resolution}
                  vRes := kDefaultRes;         {72 DPI default resolution}
                  pixelSize := depth;          {Set number of bits/pixel}
                  planeBytes := 0;             {Not used}
                  pmReserved := 0;             {Not used}

                  (* Initialize fields specific to indexed and direct PixMaps *)
                  IF depth <= 8 THEN
                     BEGIN
                        (* PixMap is indexed *)
                        pixelType := 0;       {Indicates indexed}
                        cmpCount := 1;        {Have 1 component}
                        cmpSize := depth;     {Component size=depth}
                        pmTable := newColors; {Handle to CLUT}
                     END
                  ELSE
                     BEGIN
                        (* PixMap is direct *)
                        pixelType := RGBDirect; {Indicates direct}
                        cmpCount := 3;          {Have 3 components}
                        IF depth = 16 THEN
                           cmpSize := 5         {5 bits/component}
                        ELSE
```

```
                            cmpSize := 8;        {8 bits/component}

                         (* Initialize fields of the dummy color table *)
                         newColors^^.ctSeed := 3 * aPixMap^^.cmpSize;
                         newColors^^.ctFlags := 0;
                         newColors^^.ctSize := 0;
                         pmTable := newColors;
                     END;
               END
         ELSE
            error := MemError;
      END
   ELSE
      newColors := NIL;

   (* If no errors occurred, return a handle to the new off-screen PixMap *)
   IF error <> noErr THEN
      BEGIN
         IF newColors <> NIL THEN
            DisposCTable(newColors);
      END;

    (* Return the error code *)
    SetUpPixMap := error;
END;
```

MPW C Listing 2

```
#define kDefaultRes 0x00480000 /* Default resolution is 72 DPI; Fixed type */

OSErr SetUpPixMap(
    short        depth,        /* Desired number of bits/pixel in off-screen */
    Rect         *bounds,      /* Bounding rectangle of off-screen */
    CTabHandle   colors,       /* Color table to assign to off-screen */
    short        bytesPerRow,  /* Number of bytes per row in the PixMap */
    PixMapHandle aPixMap)      /* Handle to the PixMap being initialized */
{
    CTabHandle newColors;   /* Color table used for the off-screen PixMap */
    Ptr        offBaseAddr; /* Pointer to the off-screen pixel image */
    OSErr      error;       /* Returns error code */

    error = noErr;
    newColors = nil;
    offBaseAddr = nil;

    /* Clone the clut if indexed color; allocate a dummy clut if direct color */
    if (depth <= 8)
    {
        newColors = colors;
        error = HandToHand( (Handle *)&newColors );
    }
    else
    {
        newColors = (CTabHandle)NewHandle( sizeof (ColorTable) -
                sizeof (CSpecArray) );
        error = MemError();
    }
    if (error == noErr)
    {
        /* Allocate pixel image; long integer multiplication avoids overflow */
        offBaseAddr = NewPtr( (unsigned long)bytesPerRow * (bounds->bottom -
                bounds->top) );
        if (offBaseAddr != nil)
        {
            /* Initialize fields common to indexed and direct PixMaps */
            (**aPixMap).baseAddr = offBaseAddr;  /* Point to image */
            (**aPixMap).rowBytes = bytesPerRow | /* MSB set for PixMap */
                    0x8000;
            (**aPixMap).bounds = *bounds;        /* Use given bounds */
            (**aPixMap).pmVersion = 0;           /* No special stuff */
            (**aPixMap).packType = 0;            /* Default PICT pack */
            (**aPixMap).packSize = 0;            /* Always zero in mem */
            (**aPixMap).hRes = kDefaultRes;      /* 72 DPI default res */
            (**aPixMap).vRes = kDefaultRes;      /* 72 DPI default res */
            (**aPixMap).pixelSize = depth;       /* Set # bits/pixel */
            (**aPixMap).planeBytes = 0;          /* Not used */
            (**aPixMap).pmReserved = 0;          /* Not used */

            /* Initialize fields specific to indexed and direct PixMaps */
            if (depth <= 8)
            {
                /* PixMap is indexed */
                (**aPixMap).pixelType = 0;       /* Indicates indexed */
                (**aPixMap).cmpCount = 1;        /* Have 1 component */
                (**aPixMap).cmpSize = depth;     /* Component size=depth */
                (**aPixMap).pmTable = newColors; /* Handle to CLUT */
            }
            else
            {
                /* PixMap is direct */
                (**aPixMap).pixelType = RGBDirect; /* Indicates direct */
                (**aPixMap).cmpCount = 3;          /* Have 3 components */
                if (depth == 16)
                    (**aPixMap).cmpSize = 5;       /* 5 bits/component */
                else
                    (**aPixMap).cmpSize = 8;       /* 8 bits/component */
                (**newColors).ctSeed = 3 * (**aPixMap).cmpSize;
                (**newColors).ctFlags = 0;
                (**newColors).ctSize = 0;
                (**aPixMap).pmTable = newColors;
            }
        }
        else
            error = MemError();
    }
    else
        newColors = nil;

    /* If no errors occurred, return a handle to the new off-screen PixMap */
    if (error != noErr)
    {
        if (newColors != nil)
            DisposCTable( newColors );
    }

    /* Return the error code */
    return error;
}
```

SetUpPixMap begins by copying the given color table if an indexed-color graphics environment is being built, or allocating a dummy color table if a direct-color graphics environment is being built. A copy of the color table is made because this allows the given color table and the off-screen graphics environment's color table to be manipulated independently without interfering with each other, and this lets the off-screen graphics environment routines manipulate the color table without needing to worry about whether the color table is a 'clut' resource or not. The dummy color table is made so that routines which assume that every PixMap has a color table won't do something catastrophic if they find a NIL color table. The off-screen pixel image is then allocated as a nonrelocatable block in the application's heap.

Some of the fields of a PixMap have to be initialized differently depending upon whether the indexed-color model or the direct-color model is being used. So, the fields that are the same regardless of the color model that's being used are assigned first. Then the desired pixel depth is compared to 8. If the depth is less than or equal to 8, then the rest of the fields are initialized for the indexed-color model. Otherwise, the rest of the fields are initialized for the direct color model. In the case of the direct-color model, the dummy color table is initialized to have no cSpecArray entries and its ctSeed field is set to three times the component size. This dummy color table is then installed into the PixMap.

Once SetUpPixMap completes, the PixMap of the new CGrafPort is ready to hold an off-screen image. It's not quite ready to be drawn into with Color QuickDraw though. To do that, the off-screen GDevice is still needed; the construction and initialization of the GDevice are covered in the next section.

Building the GDevice

The _OpenCPort routine automatically allocates and initializes a PixMap, and the SetUpPixMap routine reinitializes that existing PixMap. _OpenCPort doesn't allocate nor initialize a GDevice, so one has to be created from scratch. Pages 21-20 through 21-21 of "The Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI describe the _NewGDevice routine. This routine seems as though it's the ticket to getting a GDevice for off-screen drawing, but it always allocates the new GDevice in the system heap. That's not so good because if your program unexpectedly quits or if you just forget to dispose of the GDevice before you quit for real, the GDevice gets orphaned in the system heap. To prevent this from happening, _NewGDevice should be ignored and the off-screen GDevice should instead be allocated and initialized from scratch. What follows is a description of how each field of the GDevice structure should be initialized.

gdRefNum — reference number of video driver. Off-screen graphics environments don't need to have video drivers because there's no video device associated with them, so this field is set to zero.

gdID — used to identify specific GDevice structures from color-search procedures. This isn't necessary for off-screen drawing, so this is normally set to zero.

gdType — type of GDevice. This field is set to the constant clutType (equal to zero) for an indexed-color environment and set to the constant directType (equal to 2) for a direct-color environment.

gdITable — handle to the inverse table. Initially, this field is set to an arbitrarily small handle. Later, the _MakeITable routine is used to resize and initialize this handle to a real inverse table.

gdResPref — inverse-table resolution. When _MakeITable is called by QuickDraw, the value of this field is used as the inverse-table resolution. Almost all inverse tables have a resolution of 4. There are some cases when a inverse-table resolution of 5 is useful, particularly when the arithmetic transfer modes are used with _CopyBits. See "The GDevice" earlier in this Note.

gdSearchProc — pointer to the color-search procedure. If a color-search procedure is needed, this field can be set later by calling the _AddSearch routine (see the "Color

Manager" chapter of *Inside Macintosh* Volume V, pages 145 through 147). Usually, this field is just set to NIL and left at that.

gdCompProc — pointer to the color-complement procedure. If a color-complement procedure is needed, this field can be set later by calling the _AddComp routine (see the "Color Manager" chapter of *Inside Macintosh* Volume V, pages 145 through 147). Usually, this field is set to NIL and left at that.

gdFlags — flags indicating certain states of the GDevice. This field should initially be set to zeroes. After the GDevice has been built, these flags can be set with the _SetDeviceAttrs routine (see the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI, pages 21-10 and 21-22).

gdPMap — handle to a PixMap. A handle to the PixMap of the CGrafPort that was created earlier is put into this field.

gdRefCon — miscellaneous data. _CalcCMask and _SeedCFill use this field as described on pages 71 through 72 of *Inside Macintosh* Volume V. Initially, this field is set to zero.

gdNextGD — handle to next GDevice in the GDevice list. The system maintains a linked list of GDevice records in which there's one GDevice for every screen, and the links are kept in this field. Off-screen GDevice structures should never be put into this list, so this field should be set to NIL.

gdRect — rectangle of GDevice. Strictly speaking, this field is used only for screens, but it should be the same as the bounds rectangle of the off-screen PixMap.

gdMode — current video mode. This field is used by video drivers to keep track of the current mode that the video device is in. For off-screen GDevice structures, this field should be set to -1.

gdCC... — These four fields are used only with GDevice structures for screens. For off-screen GDevice structures, these fields should be set to zero.

gdReserved — not currently defined. This field is set to zero.

The CreateGDevice routine shown below in Listing 3 allocates and initializes a GDevice structure. It takes the initialized off-screen PixMap in the basePixMap parameter and returns the initialized GDevice in the retGDevice parameter. If any error occurs, any memory that's allocated is disposed of and the result code is returned as a function result.

MPW Pascal Listing 3

```
FUNCTION CreateGDevice(
   basePixMap:     PixMapHandle; {Handle to the PixMap to base GDevice on}
   VAR retGDevice: GDHandle      {Returns a handle to the new GDevice}
   ): OSErr;

   CONST
      kITabRes = 4; {Inverse-table resolution}

   VAR
      newDevice:  GDHandle;    {Handle to the new GDevice}
      embryoITab: ITabHandle;  {Handle to the embryonic inverse table}
      error:      OSErr;       {Error code}

BEGIN
   (* Initialize a few things before we begin *)
   error := noErr;
   newDevice := NIL;
   embryoITab := NIL;

   (* Allocate memory for the new GDevice *)
   newDevice := GDHandle(NewHandle(SizeOf(GDevice)));
   IF newDevice <> NIL THEN
      BEGIN
         (* Allocate the embryonic inverse table *)
         embryoITab := ITabHandle(NewHandleClear(2));
         IF embryoITab <> NIL THEN
            BEGIN
               (* Initialize the new GDevice fields *)
               WITH newDevice^^ DO
                  BEGIN
                     gdRefNum := 0;                 {Only used for screens}
                     gdID := 0;                     {Won't normally use}
                     IF basePixMap^^.pixelSize <= 8 THEN
                        gdType := clutType          {Depth≤8; clut device}
                     ELSE
                        gdType := directType;       {Depth>8; direct device}
                     gdITable := embryoITab;        {2-byte handle for now}
                     gdResPref := kITabRes;         {Normal inv table res}
                     gdSearchProc := NIL;           {No color-search proc}
                     gdCompProc := NIL;             {No complement proc}
                     gdFlags := 0;                  {Will set these later}
                     gdPMap := basePixMap;          {Reference our PixMap}
                     gdRefCon := 0;                 {Won't normally use}
                     gdNextGD := NIL;               {Not in GDevice list}
                     gdRect := basePixMap^^.bounds; {Use PixMap dimensions}
                     gdMode := -1;                  {For nonscreens}
                     gdCCBytes := 0;                {Only used for screens}
                     gdCCDepth := 0;                {Only used for screens}
                     gdCCXData := NIL;              {Only used for screens}
                     gdCCXMask := NIL;              {Only used for screens}
                     gdReserved := 0;               {Currently unused}
                  END;

               (* Set color-device bit if PixMap isn't black & white *)
               IF basePixMap^^.pixelSize > 1 THEN
                  SetDeviceAttribute(newDevice, gdDevType, true);

               (* Set bit to indicate that the GDevice has no video driver *)
               SetDeviceAttribute(newDevice, noDriver, true);

               (* Initialize the inverse table *)
               IF basePixMap^^.pixelSize <= 8 THEN
                  BEGIN
                     MakeITable(basePixMap^^.pmTable, newDevice^^.gdITable,
                           newDevice^^.gdResPref);
                     error := QDError;
                  END;
            END
         ELSE
            error := MemError;
      END
   ELSE
      error := MemError;

   (* Handle any errors along the way *)
   IF error <> noErr THEN
```

```
      BEGIN
         IF embryoITab <> NIL THEN
            DisposHandle(Handle(embryoITab));
         IF newDevice <> NIL THEN
            DisposHandle(Handle(newDevice));
      END
   ELSE
      retGDevice := newDevice;

   (* Return a handle to the new GDevice *)
   CreateGDevice := error;
END;
```

MPW C Listing 3

```
#define kITabRes 4 /* Inverse-table resolution */

OSErr CreateGDevice(
    PixMapHandle basePixMap,  /* Handle to the PixMap to base GDevice on */
    GDHandle     *retGDevice) /* Returns a handle to the new GDevice */
{
    GDHandle   newDevice;  /* Handle to the new GDevice */
    ITabHandle embryoITab; /* Handle to the embryonic inverse table */
    Rect       deviceRect; /* Rectangle of GDevice */
    OSErr      error;      /* Error code */

    /* Initialize a few things before we begin */
    error = noErr;
    newDevice = nil;
    embryoITab = nil;

    /* Allocate memory for the new GDevice */
    newDevice = (GDHandle)NewHandle( sizeof (GDevice) );
    if (newDevice != nil)
    {
        /* Allocate the embryonic inverse table */
        embryoITab = (ITabHandle)NewHandleClear( 2 );
        if (embryoITab != nil)
        {
            /* Set rectangle of device to PixMap bounds */
            deviceRect = (**basePixMap).bounds;

            /* Initialize the new GDevice fields */
            (**newDevice).gdRefNum = 0;            /* Only used for screens */
            (**newDevice).gdID = 0;                /* Won't normally use */
            if ((**basePixMap).pixelSize <= 8)
                (**newDevice).gdType = clutType;   /* Depth≤8; clut device */
            else
                (**newDevice).gdType = directType; /* Depth>8; direct device */
            (**newDevice).gdITable = embryoITab;   /* 2-byte handle for now */
            (**newDevice).gdResPref = kITabRes;    /* Normal inv table res */
            (**newDevice).gdSearchProc = nil;      /* No color-search proc */
            (**newDevice).gdCompProc = nil;        /* No complement proc */
            (**newDevice).gdFlags = 0;             /* Will set these later */
            (**newDevice).gdPMap = basePixMap;     /* Reference our PixMap */
            (**newDevice).gdRefCon = 0;            /* Won't normally use */
            (**newDevice).gdNextGD = nil;          /* Not in GDevice list */
            (**newDevice).gdRect = deviceRect;     /* Use PixMap dimensions */
            (**newDevice).gdMode = -1;             /* For nonscreens */
            (**newDevice).gdCCBytes = 0;           /* Only used for screens */
            (**newDevice).gdCCDepth = 0;           /* Only used for screens */
            (**newDevice).gdCCXData = 0;           /* Only used for screens */
            (**newDevice).gdCCXMask = 0;           /* Only used for screens */
            (**newDevice).gdReserved = 0;           /* Currently unused */

            /* Set color-device bit if PixMap isn't black & white */
            if ((**basePixMap).pixelSize > 1)
                SetDeviceAttribute( newDevice, gdDevType, true );

            /* Set bit to indicate that the GDevice has no video driver */
            SetDeviceAttribute( newDevice, noDriver, true );

            /* Initialize the inverse table */
            if ((**basePixMap).pixelSize <= 8)
            {
                MakeITable( (**basePixMap).pmTable, (**newDevice).gdITable,
                        (**newDevice).gdResPref );
                error = QDError();
            }
        }
        else
            error = MemError();
    }
    else
        error = MemError();

    /* Handle any errors along the way */
    if (error != noErr)
    {
        if (embryoITab != nil)
            DisposHandle( (Handle)embryoITab );
        if (newDevice != nil)
            DisposHandle( (Handle)newDevice );
    }
    else
        *retGDevice = newDevice;

    /* Return a handle to the new GDevice */
    return error;
}
```

CreateGDevice begins by allocating the `GDevice` structure and an embryonic form of the inverse table in the current heap. The inverse table is allocated as two zero bytes for now; it'll be resized and initialized to be a real inverse table later in this routine. Then, each of the `GDevice` fields are initialized as described earlier.

After all the fields have been initialized, the `gdFlags` field is set through `_SetDeviceAttribute`. If the desired pixel depth is greater than 1, then the `gdDevType` bit is set. This indicates that the `GDevice` is for a color graphics environment. This bit should be set even if a gray-scale color table is used for this off-screen graphics environment. The `noDriver` bit is set because this is an off-screen `GDevice` and so there's no associated video device driver.

Finally, the inverse table is resized and initialized by calling the `_MakeITable` routine. A handle to the two-byte embryonic inverse table that was created earlier in CreateGDevice is passed as a parameter, as is a handle to the off-screen color table and the preferred inverse-table resolution.

All Fall Down

Now that we have a way to create an off-screen graphics environment, there has to be a way to get rid of it too. The DisposeOffScreen routine shown in Listing 4 does this. The CreateOffScreen routine returns an off-screen graphics environment that's represented by a `CGrafPort` and `GDevice`. The DisposeOffScreen routine takes the off-screen `CGrafPort` and `GDevice` and deallocates all the memory that's associated with them including the `CGrafPort` and its dependent structures, the `GDevice`, the `PixMap`, the color table, and the inverse table.

MPW Pascal Listing 4

```
PROCEDURE DisposeOffScreen(
   doomedPort:    CGrafPtr; {Pointer to the CGrafPort we're getting rid of}
   doomedGDevice: GDHandle  {Handle to the GDevice we're getting rid of}
   );

   VAR
      currPort:    CGrafPtr; {Pointer to the current port}
      currGDevice: GDHandle; {Handle to the current GDevice}

BEGIN
   (* Check to see whether the doomed CGrafPort is the current port *)
   GetPort(GrafPtr(currPort));
   IF currPort = doomedPort THEN
      BEGIN
         (* It is; set current port to Window Manager CGrafPort *)
         GetCWMgrPort(currPort);
         SetPort(GrafPtr(currPort));
      END;

   (* Check to see whether the doomed GDevice is the current GDevice *)
   currGDevice := GetGDevice;
   IF currGDevice = doomedGDevice THEN
      (* It is; set current GDevice to the main screen's GDevice *)
      SetGDevice(GetMainDevice);

   (* Throw everything away *)
   doomedGDevice^^.gdPMap := NIL;
   DisposGDevice(doomedGDevice);
   DisposPtr(doomedPort^.portPixMap^^.baseAddr);
   IF doomedPort^.portPixMap^^.pmTable <> NIL THEN
      DisposCTable(doomedPort^.portPixMap^^.pmTable);
   CloseCPort(doomedPort);
   DisposPtr(Ptr(doomedPort));
END;
```

MPW C Listing 4

```
void DisposeOffScreen(
    CGrafPtr doomedPort,     /* Pointer to the CGrafPort to be disposed of */
    GDHandle doomedGDevice) /* Handle to the GDevice to be disposed of */
{
    CGrafPtr currPort;     /* Pointer to the current port */
    GDHandle currGDevice; /* Handle to the current GDevice */

    /* Check to see whether the doomed CGrafPort is the current port */
    GetPort( (GrafPtr *)&currPort );
    if (currPort == doomedPort)
    {
        /* It is; set current port to Window Manager CGrafPort */
        GetCWMgrPort( &currPort );
        SetPort( (GrafPtr)currPort );
    }

    /* Check to see whether the doomed GDevice is the current GDevice */
    currGDevice = GetGDevice();
    if (currGDevice == doomedGDevice)
        /* It is; set current GDevice to the main screen's GDevice */
        SetGDevice( GetMainDevice() );

    /* Throw everything away */
    (**doomedGDevice).gdPMap = nil;
    DisposGDevice( doomedGDevice );
    DisposPtr( (**doomedPort->portPixMap).baseAddr );
    if ((**doomedPort->portPixMap).pmTable != nil)
        DisposCTable( (**doomedPort->portPixMap).pmTable );
    CloseCPort( doomedPort );
    DisposPtr( (Ptr)doomedPort );
}
```

One mildly tricky aspect of this is that we shouldn't dispose of the current graphics environment. To prevent this, the current port is retrieved by a call to _GetPort. If it returns a pointer to the same port that DisposeOffScreen is disposing, then the current port is set to the Window Manager's CGrafPort. That was an arbitrary choice, but it's the most neutral. Similarly, the current GDevice is retrieved by a call to _GetGDevice. If it returns a handle to the same GDevice that DisposeOffScreen is disposing, then the current port is set to the main screen's GDevice. Again, that's an arbitrary, neutral choice.

The inverse table, GDevice, pixel image, and color table are disposed of. Before disposing of the color table, a check is first made to see whether it's NIL. That's because it's reasonable, though not normal, for the PixMap not to have even a dummy color table if the direct-color model is being used. Then the CGrafPort is closed which deallocates all the pieces associated with the CGrafPort, including the PixMap. Once this is done, all the structures that were created by calling CreateOffScreen are deallocated.

Playing With Blocks

Now that these four routines with two entry points can create and dispose of off-screen graphics environments, how are they used? There are several phases to using an off-screen graphics environment: creating it, drawing into it, switching between it and other off-screen and on-screen graphics environments, copying images to and from it, and disposing of it. Listing 5 shows a routine called ExerciseOffScreen which is a very basic example of all of these phases.

MPW Pascal Listing 5

```
PROCEDURE ExerciseOffScreen;

  CONST
    kOffDepth   = 8;    {Number of bits per pixel in off-screen environment}
    rGrayClut   = 1600; {Resource ID of gray-scale clut}
    rColorClut  = 1601; {Resource ID of full-color clut}

  VAR
    grayPort:    CGrafPtr;   {Graphics environment for gray off screen}
    grayDevice:  GDHandle;   {Color environment for gray off screen}
    colorPort:   CGrafPtr;   {Graphics environment for color off screen}
    colorDevice: GDHandle;   {Color environment for color off screen}
    savedPort:   GrafPtr;    {Pointer to the saved graphics environment}
    savedDevice: GDHandle;   {Handle to the saved color environment}
    offColors:   CTabHandle; {Colors for off-screen environments}
    offRect:     Rect;       {Rectangle of off-screen environments}
    circleRect:  Rect;       {Rectangles for circle-drawing}
    count:       Integer;    {Generic counter}
    aColor:      RGBColor;   {Color used for drawing off screen}
    error:       OSErr;      {Error return from off-screen creation}
```

```
BEGIN
  (* Set up the rectangle for the off-screen graphics environments *)
  SetRect(offRect, 0, 0, 256, 256);

  (* Get the color table for the gray off-screen graphics environment *)
  offColors := GetCTable(rGrayClut);

  (* Create the gray off-screen graphics environment *)
  error := CreateOffScreen(offRect, kOffDepth, offColors, grayPort,
        grayDevice);

  IF error = noErr THEN
    BEGIN
      (* Get the color table for the color off-screen graphics environment *)
      offColors := GetCTable(rColorClut);

      (* Create the color off-screen graphics environment *)
      error := CreateOffScreen(offRect, kOffDepth, offColors, colorPort,
            colorDevice);

      IF error = noErr THEN
        BEGIN
          (* Save the current graphics environment *)
          GetPort(savedPort);
          savedDevice := GetGDevice;

          (* Set the current graphics environment to the gray one *)
          SetPort(GrafPtr(grayPort));
          SetGDevice(grayDevice);

          (* Draw gray-scale ramp into the gray off-screen environment *)
          FOR count := 0 TO 255 DO
            BEGIN
              aColor.red := count * 257;
              aColor.green := aColor.red;
              aColor.blue := aColor.green;
              RGBForeColor(aColor);
              MoveTo(0, count);
              LineTo(255, count);
            END;

          (* Copy gray ramp into color off-screen colorized with green *)
          SetPort(GrafPtr(colorPort));
          SetGDevice(colorDevice);
          aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
          RGBForeColor(aColor);
          CopyBits(GrafPtr(grayPort)^.portBits,
                GrafPtr(colorPort)^.portBits,
                grayPort^.portRect,
                colorPort^.portRect,
                srcCopy + ditherCopy, NIL);

          (* Draw red, green, and blue circles *)
          PenSize(8, 8);
          aColor.red := $FFFF; aColor.green := $0000; aColor.blue := $0000;
          RGBForeColor(aColor);
          circleRect := colorPort^.portRect;
          FrameOval(circleRect);
          aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
          RGBForeColor(aColor);
          InsetRect(circleRect, 20, 20);
          FrameOval(circleRect);
          aColor.red := $0000; aColor.green := $0000; aColor.blue := $FFFF;
```

```
                RGBForeColor(aColor);
                InsetRect(circleRect, 20, 20);
                FrameOval(circleRect);

                (* Copy the color off-screen environment to the current port *)
                SetPort(savedPort);
                SetGDevice(savedDevice);
                CopyBits(GrafPtr(colorPort)^.portBits, savedPort^.portBits,
                      colorPort^.portRect, savedPort^.portRect,
                      srcCopy, NIL);

                (* Dispose of the off-screen graphics environments *)
                DisposeOffScreen(grayPort, grayDevice);
                DisposeOffScreen(colorPort, colorDevice);
            END;
        END;
END;
```

MPW C Listing 5

```
#define kOffDepth  8    /* Number of bits per pixel in off-screen environment */
#define rGrayClut  1600 /* Resource ID of gray-scale clut */
#define rColorClut 1601 /* Resource ID of full-color clut */

void ExerciseOffScreen()
{
    CGrafPtr   grayPort;    /* Graphics environment for gray off screen */
    GDHandle   grayDevice;  /* Color environment for gray off screen */
    CGrafPtr   colorPort;   /* Graphics environment for color off screen */
    GDHandle   colorDevice; /* Color environment for color off screen */
    GrafPtr    savedPort;   /* Pointer to the saved graphics environment */
    GDHandle   savedDevice; /* Handle to the saved color environment */
    CTabHandle offColors;   /* Colors for off-screen environments */
    Rect       offRect;     /* Rectangle of off-screen environments */
    Rect       circleRect;  /* Rectangles for circle-drawing */
    short      count;       /* Generic counter */
    RGBColor   aColor;      /* Color used for drawing off screen */
    OSErr      error;       /* Error return from off-screen creation */

    /* Set up the rectangle for the off-screen graphics environments */
    SetRect( &offRect, 0, 0, 256, 256 );

    /* Get the color table for the gray off-screen graphics environment */
    offColors = GetCTable( rGrayClut );

    /* Create the gray off-screen graphics environment */
    error = CreateOffScreen( &offRect, kOffDepth, offColors,
            &grayPort, &grayDevice );

    if (error == noErr)
    {
        /* Get the color table for the color off-screen graphics environment */
        offColors = GetCTable( rColorClut );

        /* Create the color off-screen graphics environment */
        error = CreateOffScreen( &offRect, kOffDepth, offColors,
                &colorPort, &colorDevice );

        if (error == noErr)
        {
            /* Save the current graphics environment */
            GetPort( &savedPort );
            savedDevice = GetGDevice();
            /* Set the current graphics environment to the gray one */
            SetPort( (GrafPtr)grayPort );
            SetGDevice( grayDevice );

            /* Draw gray-scale ramp into the gray off-screen environment */
            for (count = 0; count < 256; ++count)
            {
                aColor.red = aColor.green = aColor.blue = count * 257;
                RGBForeColor( &aColor );
                MoveTo( 0, count );
                LineTo( 255, count );
            }

            /* Copy gray ramp into color off-screen colorized with green */
            SetPort( (GrafPtr)colorPort );
            SetGDevice( colorDevice );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            CopyBits( &((GrafPtr)grayPort)->portBits,
                    &((GrafPtr)colorPort)->portBits,
                    &grayPort->portRect,
                    &colorPort->portRect,
                    srcCopy | ditherCopy, nil );

            /* Draw red, green, and blue circles */
            PenSize( 8, 8 );
            aColor.red = 0xFFFF; aColor.green = 0x0000; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            circleRect = colorPort->portRect;
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0x0000; aColor.blue = 0xFFFF;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );

            /* Copy the color off-screen environment to the current port */
            SetPort( savedPort );
            SetGDevice( savedDevice );
            CopyBits( &((GrafPtr)colorPort)->portBits, &savedPort->portBits,
                    &colorPort->portRect, &savedPort->portRect,
                    srcCopy, nil );

            /* Dispose of the off-screen graphics environments */
            DisposeOffScreen( grayPort, grayDevice );
            DisposeOffScreen( colorPort, colorDevice );
        }
    }
}
```

Two off-screen graphics environments are created in the same way. A rectangle that's 256 pixels wide by 256 pixels high and with its top-left coordinate at (0, 0) is created in the `offRect` local variable. 'clut' resources are loaded from the application's resource fork to use as the color tables of the two off-screen graphics environments; a gray-scale 'clut' in the first case and a full-color 'clut' in the second case. Then, CreateOffScreen is called with the rectangle, color table, and a hard-coded pixel depth of eight bits per pixel.

If CreateOffScreen returns noErr in both cases, then the current graphics environment is saved so that it can be restored later. Graphics environments consist of the current port and the current GDevice. The current GrafPort or CGrafPort is saved with _GetPort. The current GDevice is saved with _GetGDevice.

The gray-scale off-screen graphics environment is set as the current graphics environment by calling _SetPort with its CGrafPort and calling _SetGDevice with its GDevice. A vertical gray ramp is drawn into this graphics environment with the usual set of QuickDraw calls. This graphics environment's pixel image is then copied to the full-color off-screen graphics environment with dithering and colorization with green (dithering requires 32-Bit QuickDraw and consistent colorization requires system software version 7.0; both of these features are described in Konstantin Othmer's article "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" in Issue 6 of *develop*). Before this copy happens, the full-color off-screen graphics environment must be set as the current one. Once this is done, _CopyBits can properly map colors from the gray-scale off-screen graphics environment to the full-color one which gets a green ramp image.

Red, green, and blue concentric circles are drawn into the full-color off-screen graphics environment over the green ramp. This image is then copied to the graphics environment that was the current one when ExerciseOffScreen was called. To do this, the saved graphics environment is set as the current one by what should now be the familiar calls to _SetPort and _SetGDevice. The off-screen image is then copied to the saved graphics environment with _CopyBits.

Finally, the two off-screen graphics environments are disposed of by calling the DisposeOffScreen routine that's defined in the section "All Fall Down" earlier in this Note.

Put That Checkbook Away!

The previous section covered the basics of creating and using off-screen graphics environments. This is good enough for many, if not most, needs of off-screen drawing. But there are variations to creating and maintaining an off-screen graphics environment for specific cases. This section discusses a few of the more common cases.

About That Creation Thing . . .

The CreateOffScreen routine, defined in Listing 1, takes three pieces of information: the boundary rectangle, the desired pixel depth, and the desired color table. But there's much more to these pieces than ExerciseOffScreen shows. This section describes these pieces in more detail.

The first parameter to CreateOffScreen is a rectangle which determines the size and coordinate system of the off-screen graphics environment. Usually, the top-left corner of the rectangle has the coordinate (0, 0) because it's usually easiest to draw everything using coordinates that can also be thought of as the horizontal and vertical distance in pixels from the top-left corner of the graphics environment. But in some cases, it's more convenient to have the (0, 0) coordinate somewhere else, and passing CreateOffScreen a rectangle with a nonzero coordinate in the top-left corner is an easy way to do this. The coordinate system can be translated after the off-screen graphics environment is created by using the _SetOrigin routine that's described on pages 153 through 155 of *Inside Macintosh* Volume I.

Warning: As *Inside Macintosh* Volume I, page 154, notes, the clip region of the port "sticks" to the coordinate system when you call _SetOrigin. If _SetOrigin offsets the coordinate system by a large amount, then the clip region might be moved completely outside of the port's drawing area, and nothing can be drawn into that port. After calling _SetOrigin, you should set the clip region so that you can continue drawing into the port.

The number of bits per pixel implies the maximum number of available colors in a graphics environment, at least roughly speaking. The relationship between the number of bits per pixel and the number of available colors is discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-8 through 16-9.

If an indexed-color graphics environment is being made, then a color table must be passed to CreateOffScreen. In ExerciseOffScreen, the color table is retrieved from a 'clut' resource that's in the application's resource fork with a call to _GetCTable. Because CreateOffScreen clones this color table, this 'clut' resource can be purgeable so that it can be thrown out if its memory is needed for other purposes. _GetCTable can also be passed some special constants that tell it to allocate various system color tables that can also be passed to CreateOffScreen. These special constants are described on page 17-18 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI. _GetCTable allocates memory for these system color tables, so they should be disposed of after you're done with them.

A color table could also be built from scratch by allocating it with a call to _NewHandle and then initializing it by hand. The ColorTable structure is documented on pages 48 through 49 of *Inside Macintosh* Volume V. Here's what each of the fields should be set to:

ctSeed

identification value. This is an arbitrary value that should be changed any time the contents of the color table change so that the inverse table can be kept current. When Color QuickDraw draws anything, it compares the ctSeed of the color table of the PixMap of the current GDevice against the iTabSeed field of the inverse table of the current GDevice. If they're the same, then Color QuickDraw uses colors according to that inverse table. If they're different, then Color QuickDraw first rebuilds the inverse table according to the new color table's contents and its iTabSeed is set to the value of the new color table's ctSeed; then the rebuilt inverse table is used.

When _CopyBits is called with the srcCopy transfer mode, the ctSeed fields of the source and destination pixel maps are compared. If they're the same, then _CopyBits simply transfers the source pixels to the destination with no mapping of colors. If they're different, then _CopyBits checks each entry of the color tables to determine whether they have the same colors for the same pixel values. If they do, then _CopyBits again simply transfers the source pixels to the destination with no mapping of colors. If they don't, then _CopyBits maps colors in the source PixMap to the colors in the current graphics environment according to the inverse table of the current GDevice. The ctSeed field of a color table should be changed whenever its contents are changed so that _CopyBits doesn't make the wrong assumptions about the equality of the source and destination color tables.

You can get a seed value for a new color table by assigning to it the result of the _GetCTSeed routine, documented in the "Color Manager" chapter of *Inside Macintosh* Volume V, page 143. If the contents of an existing color table are changed, then it should be passed to the _CTabChanged routine which assigns a new value to its ctSeed field. If the _CTabChanged routine isn't available (it's available with 32-Bit QuickDraw and is included with the system beginning with system software version 7.0), then the `ctSeed` field should be given a new value with another call to `_GetCTSeed`.

`ctFlags`

indicates the Boolean characteristics of a color table. If the most significant bit of `ctFlags` is clear, then the `value` field of each `ColorSpec` entry in the `ctTable` array is interpreted as the pixel value for the color that's specified in the `rgb` field in the same `ColorSpec` entry. You can build a color table with nonconsecutive pixel values this way. If this bit is set, then all the `value` fields in the color table are ignored and the index of each `ColorSpec` record in the `ctTable` array is that record's pixel value. It's your choice whether to clear this bit and set the `value` fields or set this bit and ignore the `value` fields; traditionally this bit is clear for off-screen color tables.

If the next most significant bit of `ctFlags` is set, then the `value` field of each `ColorSpec` record in the `ctTable` array is used by `_CopyBits` as an index into the color palette that's attached to the destination window, and the `rgb` field is ignored. This is documented in the "Palette Manager" chapter of *Inside Macintosh* Volume VI, page 20-17.

The other bits are reserved for future use. If you create a color table from scratch, these other bits must be set to zero. If you use a color table that's generated by the system, then these bits must be preserved.

`ctSize`

the number of color table entries minus 1. Normally, this field is set to 1, 3, 15, or 255 for 1-, 2-, 4-, and 8-bits per pixel, respectively. In special cases, it's reasonable to have less than the maximum number of entries for the pixel depth. For example, a color table for an 8-bit per pixel graphics environment could have just 150 entries, in which case the `ctSize` field should hold 149. For this case, it's still important to allocate as much space in the color table for the maximum number of entries for a pixel depth and clear the entries you're not using to zero because some parts of Color QuickDraw assume the size of a color table based on the pixel depth.

`ctTable`

array of colors and pixel values. This table defines all the available colors in the color table and their pixel values. The `value` field of each `ColorSpec` record indicates that color's pixel value if the most significant bit of `ctFlags` is clear. It's ignored if the most significant bit of `ctFlags` is set. The `value` field is used as an index into a palette if the next most significant bit of `ctFlags` is set, in which case the `rgb` field is ignored. See the discussion of the `ctFlags` field earlier in this Note for more details.

Warning: Color QuickDraw's text-drawing routines assume that the color table of the destination graphics environment has the maximum number of colors for the pixel depth of the graphics environment, and that white is the first entry in the color table and black is the last entry. If these conditions aren't satisfied, then the resulting image is unpredictable.

The code fragment in Listing 6 shows how to allocate a 256-entry color table from scratch. Color tables have a variable size, so the `_NewHandle` call has to calculate the size of the `ColorTable` record plus the maximum number of color table entries for the pixel depth multiplied by the size of a `ColorSpec` record. `kNumColors - 1` is used in the calculation because the size of the `ColorTable` record includes the size of one `ColorSpec` entry in most development environments.

MPW Pascal Listing 6

```
CONST
   kNumColors = 256; {Number of color table entries}

VAR
   newColors: CTabHandle; {Handle to the new color table}
   index:     Integer;    {Index into the table of colors}

(* Allocate memory for the color table *)
newColors := CTabHandle(NewHandleClear(SizeOf (ColorTable) +
      SizeOf(ColorSpec) * (kNumColors - 1)));
IF newColors <> NIL THEN
   BEGIN
      (* Initialize the fields *)
      newColors^^.ctSeed := GetCTSeed;
      newColors^^.ctFlags := 0;
      newColors^^.ctSize := kNumColors - 1;

      (* Initialize the table of colors *)
      FOR index := 0 TO kNumColors - 1 DO
         BEGIN
            newColors^^.ctTable[index].value := index;
            newColors^^.ctTable[index].rgb.red := someRedValue;
            newColors^^.ctTable[index].rgb.green := someGreenValue;
            newColors^^.ctTable[index].rgb.blue := someBlueValue
         END
   END
```

MPW C Listing 6

```
#define kNumColors 256 /* Number of color table entries */

CTabHandle newColors; /* Handle to the new color table */
short      index;     /* Index into the table of colors */

/* Allocate memory for the color table */
newColors = (CTabHandle)NewHandleClear( sizeof (ColorTable) +
        sizeof (ColorSpec) * (kNumColors - 1) );
if (newColors != nil)
{
    /* Initialize the fields */
    (**newColors).ctSeed = GetCTSeed();
    (**newColors).ctFlags = 0;
    (**newColors).ctSize = kNumColors - 1;

    /* Initialize the table of colors */
    for (index = 0; index < kNumColors; index++)
    {
        (**newColors).ctTable[index].value = index;
        (**newColors).ctTable[index].rgb.red = someRedValue;
        (**newColors).ctTable[index].rgb.green = someGreenValue;
        (**newColors).ctTable[index].rgb.blue = someBlueValue;
    }
}
```

Changing Your Environment

After you create an off-screen graphics environment with certain dimensions, you might later want to change its size, depth, or color table without creating a completely new graphics environment from scratch and without needing to redraw the existing image. The UpdateOffScreen routine in

Listing 7 shows just one way to do this. It takes the same parameters that CreateOffScreen (defined in Listing 1) does, but instead of creating a new `CGrafPort` and `GDevice`, it alters the ones that you pass through the `updPort` and `updGDevice` parameters. If the `newBounds` parameter specifies an empty rectangle, then the existing boundary rectangle for the off-screen graphics environment is used. Similarly, if `newDepth` is zero, then the existing depth is used; and if the `newColors` parameter is NIL, then the existing color table is used. UpdateOffScreen alters the given `CGrafPort` and `GDevice` to the new settings, but it completely replaces the `PixMap`. After all the alterations are made, the old `PixMap`'s image is copied to the new `PixMap`'s image, and then the old `PixMap` and its image are disposed.

MPW Pascal Listing 7

```
FUNCTION UpdateOffScreen(
   newBounds:  Rect;        {New bounding rectangle of off-screen}
   newDepth:   Integer;     {New number of bits per pixel in off-screen}
   newColors:  CTabHandle;  {New color table to assign to off-screen}
   updPort:    CGrafPtr;    {Returns a pointer to the updated CGrafPort}
   updGDevice: GDHandle     {Returns a handle to the updated GDevice}
   ): OSErr;

   CONST
      kMaxRowBytes = $3FFE; {Maximum number of bytes per row of pixels}

   VAR
      newPixMap:    PixMapHandle; {Handle to the new off-screen PixMap}
      oldPixMap:    PixMapHandle; {Handle to the old off-screen PixMap}
      bounds:       Rect;         {Boundary rectangle of off-screen}
      depth:        Integer;      {Depth of the off-screen PixMap}
      bytesPerRow: Integer;       {Number of bytes per row in the PixMap}
      colors:       CTabHandle;   {Colors for the off-screen PixMap}
      savedFore:    RGBColor;     {Saved foreground color}
      savedBack:    RGBColor;     {Saved background color}
      aColor:       RGBColor;     {Used to set foreground and background color}
      qdVersion:    LongInt;      {Version of QuickDraw currently in use}
      savedPort:    GrafPtr;      {Pointer to GrafPort used for save/restore}
      savedDevice: GDHandle;      {Handle to GDevice used for save/restore}
      savedState:   SignedByte;   {Saved state of color table handle}
      error:        OSErr;        {Returns error code}

BEGIN
   (* Initialize a few things before we begin *)
   newPixMap := NIL;
   error := noErr;

   (* Keep the old bounds rectangle, or get the new one *)
   IF EmptyRect(newBounds) THEN
      bounds := updPort^.portRect
   ELSE
      bounds := newBounds;

   (* Keep the old depth, or get the old one *)
   IF newDepth = 0 THEN
      depth := updPort^.portPixMap^^.pixelSize
   ELSE
      depth := newDepth;

   (* Get the old clut, or save new clut's state and make it nonpurgeable *)
   IF newColors = NIL THEN
      colors := updPort^.portPixMap^^.pmTable
   ELSE
      BEGIN
         savedState := HGetState(Handle(newColors));
      HNoPurge(Handle(newColors));
      colors := newColors;
   END;

(* Calculate the number of bytes per row in the off-screen PixMap *)
bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;

(* Get the current QuickDraw version *)
error := Gestalt (gestaltQuickdrawVersion, qdVersion);
error := noErr;

(* Make sure depth is indexed or depth is direct and 32-Bit QD installed *)
IF (depth = 1) OR (depth = 2) OR (depth = 4) OR (depth = 8) OR
      (((depth = 16) OR (depth = 32)) AND (qdVersion >= gestalt32BitQD)) THEN
   BEGIN
      (* Maximum number of bytes per row is 16,382; make sure within range *)
      IF bytesPerRow <= kMaxRowBytes THEN
         BEGIN
            (* Make sure a color table is provided if the depth is indexed *)
            IF depth <= 8 THEN
               IF colors = NIL THEN
                  (* Indexed depth and clut is NIL; is parameter error *)
                  error := paramErr;
         END
      ELSE
         (* # of bytes per row is more than 16,382; is parameter error *)
         error := paramErr;
   END
ELSE
   (* Pixel depth isn't valid; is parameter error *)
   error := paramErr;

(* If sanity checks succeed, attempt to update the graphics environment *)
IF error = noErr THEN
   BEGIN
      (* Allocate a new PixMap *)
      newPixMap := PixMapHandle(NewHandleClear(SizeOf(PixMap)));
      IF newPixMap <> NIL THEN
           BEGIN
            (* Initialize the new PixMap for off-screen drawing *)
            error := SetUpPixMap(depth, bounds, colors, bytesPerRow,
                  newPixMap);
            IF error = noErr THEN
               BEGIN
                  (* Save old PixMap and install new, initialized one *)
                  oldPixMap := updPort^.portPixMap;
                  updPort^.portPixMap := newPixMap;

                  (* Save current port & GDevice; set ones we're updating *)
                  GetPort(savedPort);
                  savedDevice := GetGDevice;
                  SetPort(GrafPtr(updPort));
                  SetGDevice(updGDevice);

                  (* Set portRect, visRgn, clipRgn to given bounds rect *)
                  updPort^.portRect := bounds;
                  RectRgn(updPort^.visRgn, bounds);
                  ClipRect(bounds);

                  (* Update the GDevice *)
                  IF newPixMap^^.pixelSize <= 8 THEN
                     updGDevice^^.gdType := clutType
                  ELSE
```

```
                        updGDevice^^.gdType := directType;
                     updGDevice^^.gdPMap := newPixMap;
                     updGDevice^^.gdRect := newPixMap^^.bounds;

                     (* Set color-device bit if PixMap isn't black & white *)
                     IF newPixMap^^.pixelSize > 1 THEN
                        SetDeviceAttribute(updGDevice, gdDevType, TRUE);
                     else
                        SetDeviceAttribute(updGDevice, gdDevType, FALSE);

                     (* Save current fore/back colors and set to B&W *)
                     GetForeColor(savedFore);
                     GetBackColor(savedBack);
                     aColor.red := 0; aColor.green := 0; aColor.blue := 0;
                     RGBForeColor(aColor);
                     aColor.red := $FFFF;
                     aColor.green := $FFFF;
                     aColor.blue := $FFFF;
                     RGBBackColor(aColor);

                     (* Copy old image to the new graphics environment *)
                     HLock(Handle(oldPixMap));
                     CopyBits(BitMapPtr(oldPixMap^)^, GrafPtr(updPort)^.portBits,
                           oldPixMap^^.bounds, updPort^.portRect,
                           srcCopy, NIL);
                     HUnlock(Handle(oldPixMap));

                     (* Restore the foreground/background color *)
                     RGBForeColor(savedFore);
                     RGBBackColor(savedBack);

                     (* Restore the saved port *)
                     SetPort(savedPort);
                     SetGDevice(savedDevice);

                     (* Get rid of the old PixMap and its dependents *)
                     DisposPtr(oldPixMap^^.baseAddr);
                     DisposeCTable(oldPixMap^^.pmTable);
                     DisposHandle(Handle(oldPixMap));
                  END;
            END
         ELSE
              error := MemError;
     END;

   (* Restore the given state of the color table *)
   IF colors <> NIL THEN
      HSetState(Handle(colors), savedState);

   (* One Last Look Around The House Before We Go... *)
   IF error <> noErr THEN
      BEGIN
         IF newPixMap <> NIL THEN
            BEGIN
               IF newPixMap^^.pmTable <> NIL THEN
                  DisposCTable(newPixMap^^.pmTable);
               IF newPixMap^^.baseAddr <> NIL THEN
                  DisposPtr(newPixMap^^.baseAddr);
               DisposHandle(Handle(newPixMap));
            END;
      END;
   UpdateOffScreen := error;
END;
```

MPW C Listing 7

```
#define kMaxRowBytes 0x3FFE /* Maximum number of bytes in a row of pixels */

OSErr UpdateOffScreen(
    Rect        *newBounds, /* New bounding rectangle of off-screen */
    short       newDepth,   /* New number of bits per pixel in off-screen */
    CTabHandle newColors,   /* New color table to assign to off-screen */
    CGrafPtr    updPort,    /* Returns a pointer to the updated CGrafPort */
    GDHandle    updGDevice) /* Returns a handle to the updated GDevice */
{
    PixMapHandle newPixMap;    /* Handle to the new off-screen PixMap */
    PixMapHandle oldPixMap;    /* Handle to the old off-screen PixMap */
    Rect         bounds;       /* Boundary rectangle of off-screen */
    short        depth;        /* Depth of the off-screen PixMap */
    short        bytesPerRow;  /* Number of bytes per row in the PixMap */
    CTabHandle   colors;       /* Colors for the off-screen PixMap */
    RGBColor     savedFore;    /* Saved foreground color */
    RGBColor     savedBack;    /* Saved background color */
    RGBColor     aColor;       /* Used to set foreground and background color */
    long         qdVersion;    /* Version of QuickDraw currently in use */
    GrafPtr      savedPort;    /* Pointer to GrafPort used for save/restore */
    GDHandle     savedDevice;  /* Handle to GDevice used for save/restore */
    SignedByte   savedState;   /* Saved state of color table handle */
    OSErr        error;        /* Returns error code */

    /* Initialize a few things before we begin */
    newPixMap = nil;
    error = noErr;

    /* Keep the old bounds rectangle, or get the new one */
    if (EmptyRect( newBounds ))
        bounds = updPort->portRect;
    else
        bounds = *newBounds;

    /* Keep the old depth, or get the old one */
    if (newDepth == 0)
        depth = (**updPort->portPixMap).pixelSize;
    else
        depth = newDepth;

    /* Get the old clut, or save new clut's state and make it nonpurgeable */
    if (newColors == nil)
        colors = (**updPort->portPixMap).pmTable;
    else
    {
        savedState = HGetState( (Handle)newColors );
        HNoPurge( (Handle)newColors );
        colors = newColors;
    }

    /* Calculate the number of bytes per row in the off-screen PixMap */
    bytesPerRow = ((depth * (bounds.right - bounds.left) + 31) >> 5) << 2;

    /* Get the current QuickDraw version */
    (void)Gestalt( gestaltQuickdrawVersion, &qdVersion );

    /* Make sure depth is indexed or depth is direct and 32-Bit QD installed */
    if (depth == 1 || depth == 2 || depth == 4 || depth == 8 ||
            ((depth == 16 || depth == 32) && qdVersion >= gestalt32BitQD))
    {
        /* Maximum number of bytes per row is 16,382; make sure within range */
        if (bytesPerRow <= kMaxRowBytes)
        {
            /* Make sure a color table is provided if the depth is indexed */
            if (depth <= 8)
                if (colors == nil)
                    /* Indexed depth and clut is NIL; is parameter error */
                    error = paramErr;
        }
        else
            /* # of bytes per row is more than 16,382; is parameter error */
            error = paramErr;
    }
    else
        /* Pixel depth isn't valid; is parameter error */
        error = paramErr;

    /* If sanity checks succeed, attempt to create a new graphics environment */
    if (error == noErr)
    {
        /* Allocate a new PixMap */
        newPixMap = (PixMapHandle)NewHandleClear( sizeof (PixMap) );
        if (newPixMap != nil)
        {
            /* Initialize the new PixMap for off-screen drawing */
            error = SetUpPixMap( depth, &bounds, colors, bytesPerRow, newPixMap );
            if (error == noErr)
            {
                /* Save the old PixMap and install the new, initialized one */
                oldPixMap = updPort->portPixMap;
                updPort->portPixMap = newPixMap;

                /* Save current port & GDevice and set ones we're updating */
                GetPort( &savedPort );
                savedDevice = GetGDevice();
                SetPort( (GrafPtr)updPort );
                SetGDevice( updGDevice );

                /* Set portRect, visRgn, and clipRgn to the given bounds rect */
                updPort->portRect = bounds;
                RectRgn( updPort->visRgn, &bounds );
                ClipRect( &bounds );

                /* Update the GDevice */
                if ((**newPixMap).pixelSize <= 8)
                    (**updGDevice).gdType = clutType;
                else
                    (**updGDevice).gdType = directType;
                (**updGDevice).gdPMap = newPixMap;
                (**updGDevice).gdRect = (**newPixMap).bounds;

                /* Set color-device bit if PixMap isn't black & white */
                if ((**newPixMap).pixelSize > 1)
                    SetDeviceAttribute( updGDevice, gdDevType, true );
                else
                    SetDeviceAttribute( updGDevice, gdDevType, false );

                /* Save current foreground/background colors and set to B&W */
                GetForeColor( &savedFore );
                GetBackColor( &savedBack );
                aColor.red = aColor.green = aColor.blue = 0;
                RGBForeColor( &aColor );
                aColor.red = aColor.green = aColor.blue = 0xFFFF;
                RGBBackColor( &aColor );

                /* Copy old image to the new graphics environment */
                HLock( (Handle)oldPixMap );
                CopyBits( (BitMapPtr)*oldPixMap, &((GrafPtr) updPort)->portBits,
                      &(**oldPixMap).bounds, &updPort->portRect,
                      srcCopy, nil );
                HUnlock( (Handle)oldPixMap );

                /* Restore the foreground/background color */
                RGBForeColor( &savedFore );
                RGBBackColor( &savedBack );

                /* Restore the saved port */
                SetPort( savedPort );
                SetGDevice( savedDevice );

                /* Get rid of the old PixMap and its dependents */
                DisposPtr( (**oldPixMap).baseAddr );
                DisposeCTable( (**oldPixMap).pmTable ) ;
                DisposHandle( (Handle)oldPixMap );
            }
        }
        else
            error = MemError();
    }

    /* Restore the given state of the color table */
    if (colors != nil)
        HSetState( (Handle)colors, savedState );

    /* One Last Look Around The House Before We Go… */
    if (error != noErr)
    {
        /* Some error occurred; dispose of everything we allocated */
        if (newPixMap != nil)
        {
            if ((**newPixMap).pmTable)
                DisposCTable( (**newPixMap).pmTable );
            if ((**newPixMap).baseAddr)
                DisposPtr ( (**newPixMap).baseAddr );
            DisposHandle( (Handle)newPixMap );
        }
    }
    return error;
}
```

UpdateOffScreen begins by checking the boundary rectangle, depth, or color table for emptiness, zero, or NIL, respectively. If any these satisfy that condition, then the existing characteristic is used. Next, the same sanity check that CreateOffScreen uses is done. If this sanity check succeeds, then a new `PixMap` is allocated, and then it's initialized by the SetUpPixMap routine that's given in Listing 2 which gives the new `PixMap` a new pixel image and its own copy of the color table. This new `PixMap` is installed into the `CGrafPort` after saving the reference to the old `PixMap`. Then, the `portRect`, `visRgn`, and `clipRgn` of the `CGrafPort` are set to the new boundary rectangle, as is the `gdRect` of the `GDevice`. The `gdType` of the `GDevice` is set either for the indexed-color or direct-color model, the `gdPMap` is set to the new `PixMap`, and the device attributes are set according to the pixel depth. Details about the settings for the `CGrafPort` and `GDevice` are in "Building the CGrafPort" and "Building the GDevice," respectively, earlier in this Note.

At this point, the off-screen graphics environment is ready with its new characteristics, but it has garbage for an image because nothing has been drawn into it yet. The old `PixMap`, pixel image, and color table are still around, so `_CopyBits` transfers the old image into the altered graphics environment. `_CopyBits` handles the mapping from the old image's characteristics to the new characteristics, so the altered graphics environment gets the best possible representation of the old image according to its new characteristics.

Changing the Off-Screen Color Table

Sometimes, it's useful to change some or all of the colors in an off-screen color table, or to replace the off-screen color table with another one, so that the existing image in an indexed-color graphics environment appears with new colors. For example, if you had an off-screen image of a blue car and wanted to see what it looked like in green, you could change all of the shades of blue in the off-screen color table to green, and then `_CopyBits` the image to the screen. Notice that this is different from calling the UpdateOffScreen routine in the previous section with a different color table. That routine tries to reproduce the colors from the original image as best it can in the new set of colors. This section discusses the case in which you want the image's colors to change.

The most obvious part of doing this is simply to get the color table from the off-screen pixel map's `pmTable` field and modify the entries, or to dispose of the off-screen graphics environment's current color table and assign the new one to it. There's one more step to complete the process though. The discussion about `GDevice` records in "The Building Blocks" in this Note discusses inverse tables and how they go hand-in-hand with color tables. If you alter or replace the color table, you have to make sure that the inverse table of the off-screen drawing environment is rebuilt according to the new colors because Color QuickDraw uses that inverse table to know what pixel values to use for the specified color. You don't have to rebuild the inverse table explicitly as long as you tell Color QuickDraw that the color table changed. To do this, all you have to do is make sure that the `ctSeed` of the changed or altered color table is set to a new value. And to do this, you can simply call `_CTabChanged`, which is documented on page 17-26 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI. `_CTabChanged` is available beginning with 32-Bit QuickDraw and it's available in system software version 7.0. If this routine isn't available, then you can still tell Color QuickDraw that the color table has been changed by calling `_GetCTSeed` and assigning its result directly to your new color table's `ctSeed` field.

The next time you draw into this off-screen drawing environment, Color QuickDraw checks the `ctSeed` of the environment's color table against the `iTabSeed` of the inverse table of the environment's `GDevice`. Because you changed the `ctSeed` of the color table either through `_CTabChanged` or `_GetCTSeed`, these two seeds are different so Color QuickDraw automatically rebuilds the inverse table of the current `GDevice` and then it copies the `ctSeed` of the color table to the `iTabSeed` of the rebuilt inverse table. Then drawing continues normally.

Follow That Screen!

One common need of off-screen graphics environments is that they have a depth and color table that matches a screen. The CreateOffScreen routine requires a color table for indexed-color environments, and a pixel depth. Because there can be more than one screen attached to a Macintosh system, you have to decide which screen's depth and color table you should use. Typically, the depth and color table of the deepest screen that contains the area that you're interested in (probably the area of a window) is used. Another option is to use the depth and color table of the screen that has the largest area of intersection with the area that you're interested in. To find the depth and color table of the screen on which you want to base an off-screen graphics environment, you must use the list of graphics devices for all screens which is maintained by the system. Every `GDevice` record for a screen has a handle to that screen's `PixMap`, and you can find the screen's depth and color table there.

Listing 8 shows a routine called CreateScreenOffScreen which creates an off-screen graphics environment that has the depth and color table of a selected screen. The first parameter, `bounds`, specifies the rectangular part of the screen area in which you're interested in global coordinates.

The screenOption parameter specifies how you want the screen to be chosen. If you pass kDeepestScreen in this parameter, CreateScreenOffScreen creates the new off-screen graphics environment with the depth and color table of the deepest screen that intersects the bounds rectangle. If you instead pass kLargestScreenArea, then the new off-screen graphics environment is created with the depth and color table of the screen with the largest area of intersection with the bounds rectangle.

MPW Pascal Listing 8

```
TYPE
   ScreenOpt = (kDeepestScreen, kLargestAreaScreen);

FUNCTION CreateScreenOffScreen(
   bounds:          Rect;       {Global rectangle of part of screen to save}
   screenOption:    ScreenOpt;  {Use deepest or largest intersection area screen?}
   VAR retPort:     CGrafPtr;   {Returns a pointer to the new CGrafPort}
   VAR retGDevice:  GDHandle    {Returns a handle to the new GDevice}
   ): OSErr;

   VAR
      baseGDevice:   GDHandle;      {GDevice to base off-screen on}
      aGDevice:      GDHandle;      {Handle to each GDevice in the GDevice list}
      basePixMap:    PixMapHandle;  {baseGDevice's PixMap}
      maxArea:       LongInt;       {Largest intersection area found}
      area:          LongInt;       {Area of rectangle of intersection}
      commonRect:    Rect;          {Rectangle of intersection}
      normalBounds:  Rect;          {bounds rectangle normalized to (0, 0)}
      error:         Integer;       {Error code}

BEGIN
   error := noErr;

   (* Different screen options require different algorithms *)
   IF screenOption = kDeepestScreen THEN
      (* Graphics Devices Manager tells us the deepest intersecting screen *)
      baseGDevice := GetMaxDevice(bounds)
   ELSE IF screenOption = kLargestAreaScreen THEN
      BEGIN
         (* Get a handle to the first GDevice in the GDevice list *)
         aGDevice := GetDeviceList;

         (* Keep looping until all GDevices have been checked *)
         maxArea := 0;
         baseGDevice := NIL;
         WHILE aGDevice <> NIL DO
            BEGIN
               (* Check to see whether screen rectangle and bounds intersect *)
               IF SectRect(aGDevice^^.gdRect, bounds, commonRect) THEN
                  BEGIN
                     (* Calculate area of intersection *)
                     area := LongInt(commonRect.bottom - commonRect.top) *
                           LongInt(commonRect.right - commonRect.left);

                     (* Keep track of largest area of intersection so far *)
                     IF area > maxArea THEN
                        BEGIN
                           maxArea := area;
                           baseGDevice := aGDevice;
                        END;
                  END;

               (* Go to the next GDevice in the GDevice list *)
```

```
                aGDevice := GetNextDevice(aGDevice);
            END;
        END
    ELSE
        error := paramErr;

    (* If no screens intersect the bounds, baseDevice is NIL *)
    IF (baseGDevice <> NIL) AND (error = noErr) THEN
        BEGIN
            (* Normalize the bounds rectangle *)
            normalBounds := bounds;
            OffsetRect(normalBounds, -normalBounds.left, -normalBounds.top);

            (* Create off-screen graphics environment w/ depth, clut of screen *)
            basePixMap := baseGDevice^^.gdPMap;
            error := CreateOffScreen(normalBounds, basePixMap^^.pixelSize,
                    basePixMap^^.pmTable, retPort, retGDevice);
        END;
    CreateScreenOffScreen := error;
END;
```

MPW C Listing 8

```
enum
{
    kDeepestScreen,
    kLargestAreaScreen,
};

OSErr CreateScreenOffScreen(
    Rect     *bounds,      /* Global rectangle of part of screen to save */
    short    screenOption, /* Use deepest or largest intersection area screen */
    CGrafPtr *retPort,     /* Returns a pointer to the new CGrafPort */
    GDHandle *retGDevice)  /* Returns a handle to the new GDevice */
{
    GDHandle     baseGDevice;  /* GDevice to base off-screen on */
    GDHandle     aGDevice;     /* Handle to each GDevice in the GDevice list */
    PixMapHandle basePixMap;   /* baseGDevice's PixMap */
    long         maxArea;      /* Largest intersection area found */
    long         area;         /* Area of rectangle of intersection */
    Rect         commonRect;   /* Rectangle of intersection */
    Rect         normalBounds; /* bounds rectangle normalized to (0, 0) */
    short        error;        /* Error code */

    error = noErr;

    /* Different screen options require different algorithms */
    if (screenOption == kDeepestScreen)
        /* Graphics Devices Manager tells us the deepest intersecting screen */
        baseGDevice = GetMaxDevice( bounds );
    else if (screenOption == kLargestAreaScreen)
    (
        /* Get a handle to the first GDevice in the GDevice list */
        aGDevice = GetDeviceList();

        /* Keep looping until all GDevices have been checked */
        maxArea = 0;
        baseGDevice = nil;
        while (aGDevice != nil)
        {
            /* Check to see whether screen rectangle and bounds intersect */
            if (SectRect( &(**aGDevice).gdRect, bounds, &commonRect ))
            {
                /* Calculate area of intersection */
                area = (long)(commonRect.bottom - commonRect.top) *
                       (long)(commonRect.right - commonRect.left);

                /* Keep track of largest area of intersection found so far */
                if (area > maxArea)
                {
                    maxArea = area;
                    baseGDevice = aGDevice;
                }
            }

            /* Go to the next GDevice in the GDevice list */
            aGDevice = GetNextDevice( aGDevice );
        }
    }
    else
        error = paramErr;

    /* If no screens intersect the bounds, baseDevice is NIL */
    if (baseGDevice != nil && error == noErr)
    {
        /* Normalize the bounds rectangle */
        normalBounds = *bounds;
        OffsetRect( &normalBounds, -normalBounds.left, -normalBounds.top );

        /* Create off-screen graphics environment w/ depth, clut of screen */
        basePixMap = (**baseGDevice).gdPMap;
        error = CreateOffScreen( &normalBounds, (**basePixMap).pixelSize,
                (**basePixMap).pmTable, retPort, retGDevice );
    }
    return error;
}
```

Finding the deepest screen that intersects an on-screen area is trivially easy because there's a Graphics Devices Manager routine that finds it called _GetMaxDevice which is documented on page 21-22 of the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI. The rectangle in global coordinates of the screen area you're interested in is passed to _GetMaxDevice, and it returns a handle to the deepest screen that intersects that area, even if the area of intersection is as small as one pixel. If no screens intersect that area, then _GetMaxDevice returns NIL.

Finding the GDevice of the screen that has the maximum area of intersection with the screen area you're interested in isn't quite so easy because there's no single Graphics Devices Manager routine to find this GDevice; you have to search the GDevice list yourself. You can get a handle to the first GDevice in the list by calling _GetDeviceList, and you can get a handle to each successive GDevice by calling _GetNextDevice. _GetDeviceList is documented on pages 21-21 through 21-22 of the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI, and _GetNextDevice is documented on page 21-22 of the same chapter. For each GDevice in the list, the area of intersection between the bounds and the gdRect of the GDevice is calculated. If the calculated area is the largest area of intersection found so far, then that area and the GDevice of that screen are remembered.

Once a winning GDevice has been chosen, either by being the deepest intersecting GDevice or the GDevice with the largest intersecting area, then CreateOffScreen routine is called with the pixel depth and color table of the PixMap of the GDevice, and the bounds rectangle normalized so that its top-left coordinate has the coordinates (0, 0). CreateOffScreen returns with the new off-screen graphics environment, and CreateScreenOffScreen returns this to the caller.

Choosing Your Off-Screen Memory

The CreateOffScreen routine in Listing 1 creates an off-screen graphics environment with its pixel image allocated as a nonrelocatable block in the application's heap. But this isn't the only way that the pixel image can be allocated. Pixel images can be big, and big blocks of nonrelocatable memory in your heap can be expensive in terms of performance, and they can cause a bad case of heap fragmentation. Why not put the pixel image in a relocatable block of memory instead? If there isn't much free memory in your heap and if MultiFinder or system software version 7.0 is running, there's memory that's not being used by any open applications, called *temporary memory* (formerly called *MultiFinder temporary memory*). Why not use this area of memory for the pixel image? Some people have NuBus cards with plenty of memory on them. Why not move the pixel image out of the heaps altogether and instead use NuBus memory for the pixel image? All of these things can be done with simple modifications to what's been discussed in this Note, and these modifications are discussed in the next few paragraphs.

How can pixel images be relocatable? After all, pixel images are referred to only by the `baseAddr` field of a `PixMap`, and the `baseAddr` is a pointer, not a handle. It's true that while QuickDraw is being used to draw into a graphics environment, the pixel image had better not move or else QuickDraw will start drawing over the area of memory that the pixel image used to be rather than where it is. But if QuickDraw isn't doing anything with the graphics environment, then it doesn't care what happens to the pixel image as long as the `baseAddr` points to it once QuickDraw starts drawing into the graphics environment. This implies a strategy: allocate the pixel image as a relocatable block and let it float in the heap; when QuickDraw is about to to draw into the graphics environment or to copy from it, lock the pixel image and copy its master pointer into the `baseAddr` field of the `PixMap`; when the drawing or copying is finished, unlock the pixel image. There are many ways to implement this, and Listing 9 shows a code fragment for one very simple method.

MPW Pascal Listing 9

```
      ...
      (* Allocate the pixel image; use long multiplication to avoid overflow *)
      offBaseAddr := NewHandle(LongInt(bytesPerRow) * (bounds^.bottom -
          bounds^.top));
      IF offBaseAddr <> NIL THEN
         BEGIN
            (* Initialize fields common to indexed and direct PixMaps *)
            aPixMap^^.baseAddr := Ptr(offBaseAddr); (* Reference the image *)
      ...


PROCEDURE LockOffScreen(
   offScreenPort: CGrafPtr {Ptr to off-screen CGrafPort}
   );

   VAR
      offImageHnd: Handle; {Handle to the off-screen pixel image}

BEGIN
   (* Get the saved handle to the off-screen pixel image *)
   offImageHnd := Handle(offScreenPort^.portPixMap^^.baseAddr);

   (* Lock the handle to the pixel image *)
   HLock(offImageHnd);

   (* Put pixel image master pointer into baseAddr so that QuickDraw can use it *)
   offScreenPort^.portPixMap^^.baseAddr := offImageHnd^;
END;
```

```
PROCEDURE UnlockOffScreen(
   offScreenPort: CGrafPtr {Ptr to off-screen port}
   );

   VAR
      offImagePtr: Ptr;     {Pointer to the off-screen pixel image}
      offImageHnd: Handle;  {Handle to the off-screen pixel image}

BEGIN
   (* Get the handle to the off-screen pixel image *)
   offImagePtr := offScreenPort^.portPixMap^^.baseAddr;
   offImageHnd := RecoverHandle(offImagePtr);

   (* Unlock the handle *)
   HUnlock(offImageHnd);

   (* Save the handle back in the baseAddr field *)
   offScreenPort^.portPixMap^^.baseAddr := Ptr(offImageHnd);
END;
```

MPW C Listing 9

```
        ...
        /* Allocate the pixel image; use long multiplication to avoid overflow */
        offBaseAddr = NewHandle( (unsigned long)bytesPerRow * (bounds->bottom -
                bounds->top) );
        if (offBaseAddr != nil)
        {
            /* Initialize fields common to indexed and direct PixMaps */
            (**aPixMap).baseAddr = (Ptr)offBaseAddr; /* Reference the image */
        ...


void LockOffScreen(
    CGrafPtr offScreenPort) /* Pointer to the off-screen CGrafPort */
{
    Handle offImageHnd; /* Handle to the off-screen pixel image */

    /* Get the saved handle to the off-screen pixel image */
    offImageHnd = (Handle)(**offScreenPort->portPixMap).baseAddr;

    /* Lock the handle to the pixel image */
    HLock( offImageHnd );

    /* Put pixel image master pointer into baseAddr so that QuickDraw can use it */
    (**offScreenPort->portPixMap).baseAddr = *offImageHnd;
}


void UnlockOffScreen(
    CGrafPtr offScreenPort) /* Pointer to the off-screen CGrafPort */
{
    Ptr    offImagePtr; /* Pointer to the off-screen pixel image */
    Handle offImageHnd; /* Handle to the off-screen pixel image */

    /* Get the handle to the off-screen pixel image */
    offImagePtr = (**offScreenPort->portPixMap).baseAddr;
    offImageHnd = RecoverHandle( offImagePtr );

    /* Unlock the handle */
    HUnlock( offImageHnd );

    /* Save the handle back in the baseAddr field */
    (**offScreenPort->portPixMap).baseAddr = (Ptr)offImageHnd;
}
```

Listing 9 starts with a code fragment from the SetUpPixMap routine that's modified so that it allocates a new handle for the off-screen pixel image instead of a new pointer. This handle is saved in the `baseAddr` field for now. When you're about to draw into the off-screen graphics environment or to copy from it, the LockOffScreen routine in Listing 9 should be called with a pointer to the off-screen graphics environment's `CGrafPort` as the parameter. It takes the handle to the pixel image from the `baseAddr` field of the off-screen graphics environment's `PixMap` and passes it to `_HLock` which makes sure the pixel image can't move in the heap. Then, the pixel image's handle is dereferenced to get the master pointer to the pixel image, and this master pointer is copied into the `baseAddr` field. Now, QuickDraw can draw into or copy from the off-screen graphics environment.

When you're finished drawing into the off-screen graphics environment, the pixel image should be unlocked, and the UnlockOffScreen routine in Listing 9 does this. The `baseAddr` field of the `PixMap` holds the pixel image's master pointer, so this is passed to `_RecoverHandle` to get the pixel image's handle. This handle is passed to `_HUnlock` to let the pixel image float in the heap again, and then this handle is saved in the `baseAddr` field.

One potentially useful addition to the LockOffScreen routine would be a call to `_MoveHHi` just before the call to `_HLock`. This helps reduce heap fragmentation while the pixel image is locked by moving it up as high in the heap as possible before locking it, allowing the other relocatable blocks to move without tripping over it. You have to be careful with `_MoveHHi` though because it not only moves the handle as high in the heap as possible, it moves other relocatable blocks out of the top of the heap to make room for the handle. This could involve moving huge amounts of memory, and it's not unusual for `_MoveHHi` to take several seconds to do this.

How do you make an off-screen graphics environment that uses temporary memory for the pixel image? Temporary memory is allocated as handles, so there's almost no difference between using temporary memory and using relocatable blocks in your own heap in the way that Listing 9 shows. All you have to do is replace the calls to `_NewHandle`, `_HLock`, and `_HUnlock` with calls to `_TempNewHandle`, `_TempHLock`, and `_TempHUnlock`. If temporary memory handles are real, then you don't even have to replace the `_HLock` and `_HUnlock` calls—they work properly with temporary memory handles that are real.You can tell whether temporary memory handles are real or not by calling `_Gestalt` with the `gestaltOSAttr` selector. If the `gestaltRealTempMemory` bit is set, then all temporary memory handles are real. See the sections "About Temporary Memory" and "Using Temporary Memory" of *Inside Macintosh* Volume VI, pages 28-33 through 28-40.

How do you make an off-screen graphics environment that stores the pixel image on a NuBus memory card? The Macintosh Memory Manager doesn't keep track of heaps on NuBus memory cards so it can't be used to allocate memory on those cards, but if applications can use that card's memory at will, then an application can set up the off-screen graphics environment with its pixel image in the NuBus card's memory simply by setting the address of the card's memory in the `baseAddr` field of the off-screen graphics environment's `PixMap` instead of allocating anything.

If your NuBus memory card doesn't require 32-bit addressing mode to access its memory, then setting the `baseAddr` to the address of the NuBus card's memory is all you have to do. Some NuBus memory cards require its memory to be accessed in 32-bit addressing mode. Without 32-Bit QuickDraw, these memory cards can't be used for storing the pixel image of an off-screen graphics environment because Color QuickDraw without 32-Bit QuickDraw always reads and writes pixel images in 24-bit addressing mode regardless of whether the pixel image is in main memory, on a NuBus video card, or on a NuBus memory card. With 32-Bit QuickDraw, Color QuickDraw automatically switches to 32-bit addressing mode before reading or writing a pixel image that's on a video card. It won't know to switch to 32-bit addressing mode if your off-screen graphics environment uses a pixel image on a NuBus memory card that's not a video card, but you can tell it to make this switch by setting bit 2 of the `pmVersion` field of the `PixMap` for the off-screen graphics environment. This is normally done by logically ORing the `pmVersion` field with the predefined constant `baseAddr32`. See "About 32-Bit Addressing" in Issue 6 of *develop*, page 36, for more details about how QuickDraw handles addressing modes.

The GWorld Factor

In May 1989, 32-Bit QuickDraw was introduced as an extension to the system. While it had a lot of new features, the GWorld mechanism was the one that made the big news. GWorlds are off-screen graphics environments that you can have the system put together in one call. There's no need for routines like CreateOffScreen, SetUpPixMap, or CreateGDevice—all of the off-screen graphics environment is set up with `_NewGWorld`. You can change most of its characteristics with `_UpdateGWorld`, set the current off-screen graphics environment with `_SetGWorld`, and get rid of the off-screen graphics environment with `_DisposeGWorld`. All the GWorld routines are described in the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI. As an example, Listing 10 shows the same routine as the ExerciseOffScreen routine that's shown in Listing 5, but Listing 10 uses GWorlds rather than the do-it-yourself routines that are defined in this Note.

MPW Pascal Listing 10

```
PROCEDURE ExerciseOffScreen;

   CONST
      kOffDepth  = 8;     {Number of bits per pixel in off-screen environment}
      rGrayClut  = 1600;  {Resource ID of gray-scale clut}
      rColorClut = 1601;  {Resource ID of full-color clut}

   VAR
      grayPort:     GWorldPtr;   {Graphics environment for gray off screen}
      colorPort:    GWorldPtr;   {Graphics environment for color off screen}
      savedPort:    GrafPtr;     {Pointer to the saved graphics environment}
      savedDevice:  GDHandle;    {Handle to the saved color environment}
      offColors:    CTabHandle;  {Colors for off-screen environments}
      offRect:      Rect;        {Rectangle of off-screen environments}
      circleRect:   Rect;        {Rectangles for circle-drawing}
      count:        Integer;     {Generic counter}
      aColor:       RGBColor;    {Color used for drawing off-screen}
      error:        OSErr;       {Error return from off-screen creation}

BEGIN
   (* Set up the rectangle for the off-screen graphics environments *)
   SetRect(offRect, 0, 0, 256, 256);

   (* Get the color table for the gray off-screen graphics environment *)
   offColors := GetCTable(rGrayClut);

   (* Create the gray off-screen graphics environment *)
   error := NewGWorld(grayPort, kOffDepth, offRect, offColors, NIL, []);

   IF error = noErr THEN
      BEGIN
         (* Get the color table for the color off-screen graphics environment *)
         offColors := GetCTable(rColorClut);

         (* Create the color off-screen graphics environment *)
         error := NewGWorld(colorPort, kOffDepth, offRect, offColors, NIL, []);

         IF error = noErr THEN
            BEGIN
               (* Save the current graphics environment *)
               GetGWorld(savedPort, savedDevice);

               (* Set the current graphics environment to the gray one *)
               SetGWorld(grayPort, NIL);

               (* Draw gray-scale ramp into the gray off-screen environment *)
               FOR count := 0 TO 255 DO
                  BEGIN
                     aColor.red := count * 257;
                     aColor.green := aColor.red;
                     aColor.blue := aColor.green;
                     RGBForeColor(aColor);
                     MoveTo(0, count);
                     LineTo(255, count);
                  END;

               (* Copy gray ramp into color off-screen colorized with green *)
               SetGWorld(colorPort, NIL);
               aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
               RGBForeColor(aColor);
               CopyBits(GrafPtr(grayPort)^.portBits,
                     GrafPtr(colorPort)^.portBits,
                     grayPort^.portRect,
                     colorPort^.portRect,
                     srcCopy + ditherCopy, NIL);

               (* Draw red, green, and blue circles *)
               PenSize(8, 8);
               aColor.red := $FFFF; aColor.green := $0000; aColor.blue := $0000;
               RGBForeColor(aColor);
               circleRect := colorPort^.portRect;
               FrameOval(circleRect);
               aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
               RGBForeColor(aColor);
               InsetRect(circleRect, 20, 20);
               FrameOval(circleRect);
               aColor.red := $0000; aColor.green := $0000; aColor.blue := $FFFF;
               RGBForeColor(aColor);
               InsetRect(circleRect, 20, 20);
               FrameOval(circleRect);

               (* Copy the color off-screen environment to the current port *)
               SetGWorld(savedPort, savedDevice);
               CopyBits(GrafPtr(colorPort)^.portBits,
                     savedPort^.portBits,
                     colorPort^.portRect,
                     savedPort^.portRect,
                     srcCopy, NIL);

               (* Dispose of the off-screen graphics environments *)
               DisposeGWorld grayPort);
               DisposeGWorld(colorPort);
            END;
      END;
END;
```

MPW C Listing 10

```
#define kOffDepth  8    /* Number of bits per pixel in off-screen environment */
```

```
#define rGrayClut  1600 /* Resource ID of gray-scale clut */
#define rColorClut 1601 /* Resource ID of full-color clut */

void ExerciseOffScreen()
{
    GWorldPtr  grayPort;    /* Graphics environment for gray off screen */
    GWorldPtr  colorPort;   /* Graphics environment for color off screen */
    CGrafPtr   savedPort;   /* Pointer to the saved graphics environment */
    GDHandle   savedDevice; /* Handle to the saved color environment */
    CTabHandle offColors;   /* Colors for off-screen environments */
    Rect       offRect;     /* Rectangle of off-screen environments */
    Rect       circleRect;  /* Rectangles for circle-drawing */
    short      count;       /* Generic counter */
    RGBColor   aColor;      /* Color used for drawing off-screen */
    OSErr      error;       /* Error return from off-screen creation */

    /* Set up the rectangle for the off-screen graphics environments */
    SetRect( &offRect, 0, 0, 256, 256 );

    /* Get the color table for the gray off-screen graphics environment */
    offColors = GetCTable( rGrayClut );

    /* Create the gray off-screen graphics environment */
    error = NewGWorld( &grayPort, kOffDepth, &offRect, offColors, nil, 0 );

    if (error == noErr)
    {
        /* Get the color table for the color off-screen graphics environment */
        offColors = GetCTable( rColorClut );

        /* Create the color off-screen graphics environment */
        error = NewGWorld( &colorPort, kOffDepth, &offRect, offColors, nil, 0 );

        if (error == noErr)
        {
            /* Save the current graphics environment */
            GetGWorld( &savedPort, &savedDevice );

            /* Set the current graphics environment to the gray one */
            SetGWorld( grayPort, nil );

            /* Draw gray-scale ramp into the gray off-screen environment */
            for (count = 0; count < 256; count++)
            {
                aColor.red = aColor.green = aColor.blue = count * 257;
                RGBForeColor( &aColor );
                MoveTo( 0, count );
                LineTo( 255, count );
            }

            /* Copy gray ramp into color off-screen colorized with green */
            SetGWorld( colorPort, nil );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            CopyBits( &((GrafPtr)grayPort)->portBits,
                    &((GrafPtr)colorPort)->portBits,
                    &grayPort->portRect,
                    &colorPort->portRect,
                    srcCopy | ditherCopy, nil );

            /* Draw red, green, and blue circles */
            PenSize( 8, 8 );
            aColor.red = 0xFFFF; aColor.green = 0x0000; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            circleRect = colorPort->portRect;
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0x0000; aColor.blue = 0xFFFF;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );

            /* Copy the color off-screen environment to the current port */
            SetGWorld( savedPort, savedDevice );
            CopyBits( &((GrafPtr)colorPort)->portBits,
                    &((GrafPtr)savedPort)->portBits,
                    &colorPort->portRect,
                    &savedPort->portRect,
                    srcCopy, nil );

            /* Dispose of the off-screen graphics environments */
            DisposeGWorld( grayPort );
            DisposeGWorld( colorPort );
        }
    }
}
```

_NewGWorld creates an off-screen graphics environment by creating a CGrafPort, PixMap, and GDevice—the same structures that you normally put together when you make an off-screen graphics environment yourself. In this aspect, and in fact in most aspects, there's nothing magical about GWorlds. Do GWorlds make the CreateOffScreen, DisposeOffScreen, and their dependents useless? That depends on what your needs are. What follows are a few issues about off-screen drawing and how that determines whether you use your own routines, such as CreateOffScreen, to create and maintain off-screen graphics environments or whether you use GWorlds for the same purpose.

I Want the Best Performance!

As mentioned in the last paragraph, there's nothing magical about GWorlds in most aspects. In one major aspect, there certainly is: the version of Color QuickDraw that runs with the 8•24 GC video card's acceleration on knows about GWorlds and can cache their CGrafPort, PixMap, GDevice, inverse table, color table, and pixel image on the 8•24 GC card if there's enough memory on it. When this is done, QuickDraw operations on the GWorld can be much faster than they'd normally be because the image data can stay in the card's memory where the fast microprocessor is, and image data doesn't have to move across NuBus in transfer operations between the GWorld and the screen. Additionally, these operations are executed asynchronously which increases the overall speed of your programs. For details about how the 8•24 GC card and GC QuickDraw work, see Guillermo Ortiz's article, "Macintosh Display Card 8•24 GC: The Naked Truth," in Issue 5 of *develop*.

8•24 GC QuickDraw doesn't know about the off-screen graphics environments that you create, so it doesn't cache its structures. All QuickDraw commands that move image data between the off-screen graphics environment and the screen have to move the data across NuBus, and that slows down the operation in comparison to keeping all the image data on the card.

If you want the highest possible drawing and copying performance with the 8•24 GC card, you must use GWorlds for your off-screen graphics environments.

I Want to Use a NuBus Memory Card for My GWorld's Off-Screen Pixel Image

One common desire is to use a NuBus memory card to hold a pixel image. Because GWorlds are so easy to set up, and because GWorlds have all the same parts that you can make for an off-screen graphics environment, it's tempting to make a GWorld and then point the `baseAddr` of the GWorld's `PixMap` at the NuBus card's memory. But GWorlds are designed to be fairly atomic structures, so they can't be changed in this way. You can change a GWorld's dimensions, depth, and color table because there's a routine (`_UpdateGWorld`) that is designed to change these things, but you can't change the pixel image without risking future compatibility.

If you want to have an off-screen graphics environment use a NuBus video card to store the pixel image, you should set up your own off-screen graphics environment rather than use GWorlds. This is covered earlier in this Note in "Choosing Your Off-Screen Memory."

I Want My Program to Work on All System Software Releases

GWorlds have been around since 32-Bit QuickDraw was released (while system software version 6.0.3 was current). Until system software version 7.0, 32-Bit QuickDraw was an optional part of the system, so you aren't guaranteed use of GWorlds even under recent system software releases. Obviously, if GWorlds aren't available and your program still has to work with off-screen graphics environments, then there's no choice but to use your own routines for creating, maintaining, and disposing of off-screen graphics environments. What's usually done in these cases is to check via `_Gestalt` whether GWorlds are available or not. If they aren't, then you create your off-screen graphics environment with your own routines. If they are, then you can use GWorlds without having to take up memory with your code for creating off-screen graphics environments yourself.

Are We There Yet?

Reliable, understandable, and maintainable off-screen drawing routines means not taking short-cuts. The most common problems that people run into with off-screen drawing routines is the appearance of strange colors and the gradual degradation of reliability as the program does more off-screen drawing. Building an off-screen graphics environment out of a CGrafPort, GDevice, and PixMap or by using GWorlds, combined with an understanding of how Color QuickDraw uses off-screen graphics environments, helps get rid of these problems. Hopefully, this Note helps you understand these things so that you can get better programs out the door faster.

Further Reference:

- Apple Computer, Inc., *Inside Macintosh* Volume I, Addison-Wesley, Reading, MA, 1985
- Apple Computer, Inc., *Inside Macintosh* Volume V, Addison-Wesley, Reading, MA, 1988.
- Apple Computer, Inc., *Inside Macintosh* Volume VI, Addison-Wesley, Reading, MA, 1991.
- Knaster, S., *Macintosh Programming Secrets*, Addison-Wesley, Reading, MA, 1988.
- Leak, B., "Realistic Color For Real-World Applications," *develop*, January 1990, 4-21.
- Ortiz, G., "Braving Offscreen GWorlds," *develop*, January 1990, 28-40.

- Ortiz, G., "Deaccelerated _CopyBits & 8•24 GC QuickDraw," *Macintosh Technical Note #289*, January 1991.
- Ortiz, G., "Macintosh Display Card 8•24 GC: The Naked Truth," *develop*, July 1990, 332-347.
- Othmer, K., "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0," *develop*, Spring 1991, 23-42.
- Tanaka, F., "Of Time and Space and _CopyBits," *Macintosh Technical Note #277*, June 1990.
- Zap, J., F. Tanaka, J. Friedlander, and G. Jernigan, "Drawing Into an Off-Screen Bitmap," *Macintosh Technical Note #41*, June 1990.

#121: Using the High-Level AppleTalk Routines

See also: The AppleTalk Manager
Inside AppleTalk
AppleTalk Manager Update

Written by: Fred A. Huxham — May 4, 1987
Updated: March 1, 1988

What you need to do in order to use high-level AppleTalk routines depends upon the interfaces you are using. Some differences are outlined below.

MPW before 2.0

When calling the old high-level AppleTalk routines, many programmers get mysterious "resource not found" errors (-192) from such seemingly harmless routines as `MPPOpen`. The resource that is not being found is 'atpl', a resource that contains all the glue code to the high-level routines. In order to use the high-level routines, your application must have this resource in its resource fork. The 'atpl' resource is included in a file called "AppleTalk" with any compilers that use this outdated version of the AppleTalk interface.

MPW 2.0 and newer

A newer version of the alternate interfaces is available in MPW 2.0; it includes bug fixes and increased Macintosh II compatibility. With this version of the interface, the 'atpl' resource is no longer used. Glue code is now linked into your application.

This will be the final release of the current-style interface. It will be supported for some time as the **alternate interface**. We have moved to a more straightforward and simple **preferred interface**, which is also implemented in MPW 2.0 and newer, and is described in the AppleTalk Manager chapter of *Inside Macintosh* vol. V. Developers are free to continue to use the alternate interface, but in the long run it will be advantageous to move to the preferred interface.

Third Party Compilers

Third party compilers use interfaces that are built from Apple's MPW interfaces. Some compilers may not have upgraded to the new interfaces yet. Contact the individual compiler manufacturers for more information.

Macintosh Technical Notes

#122: Device-Independent Printing

See also: The Printing Manager

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

The Printing Manager was designed to give Macintosh applications a device-independent method of printing, but we *have* provided device-dependent information, such as the contents of the print record. Due to the large number of printer-type drivers becoming available (even for non-printer devices) device independence is more necessary than ever. What this means to you, as a developer, is that we will no longer be providing (or supporting) information regarding the internal structure of the print record.

We realize that there are situations where the application may know the best method for printing a particular document and may want to bypass our dialogs. Unfortunately, using your own dialogs or not using the dialogs at all, requires setting the necessary fields in the print record yourself. There are a number of problems:

- Many of the fields in the print record are undocumented, and, as we change the internal architecture of the Printing Manager to accommodate new devices, those undocumented fields are likely to change.
- Each driver uses the private, and many of the public, fields in the print record differently. The implications are that you would need intimate knowledge of how each field is used by each available driver, and you would have to set the fields in the record *differently* depending on the driver chosen. As the number of available printer-type drivers increases, this can become a cumbersome task.

Summary

To be compatible with future printer-like devices, it is essential that your application print in a device-independent manner. Avoid testing undocumented fields, setting fields in the print record directly and bypassing the existing print dialogs. Use the Printing Manager dialogs, `PrintDefault` and `PrValidate` to set up the print record for you.

Macintosh Technical Notes

#123: Bugs in LaserWriter ROMs

See also:	The Printing Manager *PostScript Language Reference Manual*, Adobe Systems	
Written by:	Ginger Jernigan	May 4, 1987
Modified by:	Ginger Jernigan	July 1, 1987
Updated:		March 1, 1988

These are LaserWriter bugs that your users may encounter when printing from **any** Macintosh application. These are for your information; you cannot code around them. The bugs described here occur in the 1.0 and 2.0 LaserWriter ROMs.

To determine which ROMs their LaserWriter contains, users can look at the test page that the LaserWriter prints at start-up time. In addition to other information (detailed in the LaserWriter user's manual), the ROM version is shown at the bottom of the line graph. The original LaserWriter contained version 1.0 ROMs. The currently shipping LaserWriter and those upgraded to the LaserWriter Plus contain version 2.0 ROMs.

These are some of the problems we know of:

1. If the level of paper in the paper tray is getting low, and the user prints a document that will cause the tray to become empty, a PostScript error may occur. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

2. If a user prints more than 15 copies of a document, a timeout condition may occur causing the print job to abort. With LaserShare, this problem can occur with as few as 9 copies. This problem is a result of the LaserWriter turning AppleTalk off while it is printing. It doesn't send out any packets to tell the world it's still alive while it is printing, so the connection times out after about 2 minutes. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

3. When printing a document that contains more than 10 patterns, users may receive intermittent PostScript errors. This usually occurs when trying to print a lot of patterns, **and** a bitmap image on the same page. The code for imaging patterns allocates almost all of the available RAM for itself, so when the bitmap imaging code tries to allocate space, and there isn't enough (and it doesn't know how to reclaim memory from the previous operation), a `limitcheck` error occurs. This problem exists in 2.0 LaserWriter ROMs. It will be improved but **not** fixed in the next ROM version.

4. If a user chooses US Letter or B5 paper and has a different sized tray in the printer, and prints using manual feed, the LaserWriter will print assuming that the paper being fed manually is the same size as that in the tray. For example, if they have a US letter tray in the LaserWriter and print a document formatted for B5 letter using manual feed, the image will not be centered on the page. The printer assumes that the manually fed paper is also US letter size and prints the image positioned accordingly, despite the driver's instructions. This is a bug in the `Note` operator in PostScript, which the driver uses for specifying the US letter and B5 letter paper sizes. The workaround is to tell the user to put an B5 tray in the printer when printing B5 manually. This problem exists in the 1.0 and 2.0 ROMs and will **not** be fixed in the next ROM version.

   By the way, an interesting, but annoying, occurance of this bug happens when manually printing Legal sized documents with the 4.0 LaserWriter driver. When the Larger Print Area option in the style dialog is deselected (which is the default) the driver uses the `Note` operator to specify the page size. When the user prints the document using manual feed, and has a US letter tray in the printer, the image is shifted up on the page cutting off the top of the image. If you tell the user to turn on the Larger Print Area option in the style dialog, the driver specifies the page size using `Legal` instead of `Note` and the image is printed properly.

#124: Using Low-Level Printing Calls With AppleTalk ImageWriters

See also:	The Printing Manager	
Written by:	Ginger Jernigan	May 4, 1987
Update by:	Scott "ZZ" Zimmerman	Febuary ?, 1988
Updated:		March 1, 1988

When you use the low-level printer driver to print, you don't get the benefits of the error checking that is done when you use the high-level Printing Manager. So, if the user prints to an AppleTalk ImageWriter (including an AppleTalk ImageWriter LQ) that is busy printing another job, the driver doesn't know whether the printer is busy, offline, or disconnected. Because of this, PrError will return (and PrintErr will contain) `abortErr`.

Since there is no way to tell when you are printing to an AppleTalk ImageWriter, the only workaround for this is to use high-level Printing Manager interface.

#125: The Effect of Spool-a-page/Print-a-page on Shared Printers

See also: Printing Manager
Technical Note #72—
Optimizing for the LaserWriter—Techniques

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

This technical note discusses drawbacks of using the spool-a-page/print-a-page method of printing.

The "spool-a-page/print-a-page" method of printing prints each page of a document as a separate job instead of calling `PrPicFile` to print the entire picture file. Many applications adopted this method of printing to avoid running out of disk space while the ImageWriter driver was spooling the document to disk. As long as you are printing to a directly connected ImageWriter, you're fine, but if you are printing to remote or shared devices (like the AppleTalk ImageWriter and the LaserWriter), this method may create significant problems for the user.

When a job is initiated by the application, the driver establishes a connection with the printer via AppleTalk. When the job is completed, the driver closes the connection, allowing another job the opportunity to print. If each page is a job in itself, then the connection is closed and reopened between each page, allowing another application to print between the pages of the document, which, as you might imagine, could present a significant problem. If two people are printing to the same AppleTalk ImageWriter at the same time and their applications use the "spool-a-page/print-a-page" method of printing, the pages of each document will be interleaved at the printer.

Although there are good reasons for using this method of printing, it is only useful for a directly connected printer. From a compatibility point of view, this method of printing is built-in device dependence. Also, this method could create serious problems for other types of remote devices. Therefore, we are recommending that applications avoid using this method indiscriminately. You should check available disk space to see how much room you have before you print. If there isn't enough space for your entire document, then print as much as you can (to minimize the interleaving) before starting another job. Whenever possible, applications should use the print loop described on page II-155 in The Printing Manager chapter of *Inside Macintosh*.

Macintosh
Technical Notes

Developer Technical Support

#126: Sub(Launching) from a High-Level Language

Revised by: Rich Collyer & Mark Johnson — April 1989
Written by: Rick Blair & Jim Friedlander — May 1987

Note: Developer Technical Support takes the view that launching and sublaunching are features which are best **avoided** for compatibility (and other) reasons, but we want to make sure that when it is absolutely necessary to implement it, it is done in the safest possible way.

This Technical Note discusses the "safest" method of calling `_Launch` from a high-level language that supports inline assembly language with the option of launching or sublaunching another application.

Changes since August 1988: Incorporated Technical Note #52 on calling `_Launch` from a high-level language, changed the example to offer a choice between launching or sublaunching, added a discussion of the `_Launch` trap under MultiFinder, and updated the MPW C code to include inline assembly language.

The Segment Loader chapter of *Inside Macintosh* II-53 states the following about the `_Launch` trap:

> *"The routines below are provided for advanced programmers; they can be called only from assembly language."*

While this statement is technically true, it is easy to call `_Launch` from any high-level language which supports inline assembly code, and this Note provides examples of calling `_Launch` in MPW Pascal and C.

Before calling `_Launch`, you need to declare the inline procedure, which takes a variable of type `pLaunchStruct` as a parameter. Since the compiler pushes a pointer to this parameter on the stack, you need to include code to put this pointer into `A0`. The way to do this is with a `MOVE.L (SP)+,A0` instruction, which is `$205F` in hexadecimal, so the first word after `INLINE` is `$205F`. This instruction sets up `A0` to contain a pointer to the filename and `4(A0)` to contain the configuration parameter, so the last part of the inline is the `_Launch` trap itself, which is `$A9F2` in hexadecimal. The configuration parameter, which is normally zero, determines whether the application uses alternate screen and sound buffers. Since not all Macintosh models support these alternate buffers, you should avoid using them unless you have a specific circumstance which requires them.

The Finder does a lot of hidden cleanup and other tasks without user knowledge; therefore, it is best if you do not try to replace the Finder with a "mini" or try to launch other programs and have them return to your application. In the future, the Finder may provide better integration for applications, and you will circumvent this if you try to act in its place by sublaunching other programs.

If you have a situation where your application **must** launch another and have it return, and where you are not worried about incompatibility with future System Software versions, there is a "preferred" way of doing this which fits into the current system well. System file version 4.1 (or later) includes a mechanism for allowing a call to another application; we term this call a "sublaunch." You can perform a sublaunch by adding a set of simple extensions to the parameter block you pass to the `_Launch` trap.

_Launch and MultiFinder

Under MultiFinder, a sublaunch behaves differently than under the Finder. The application you sublaunch becomes the foreground application, and when the user quits that application, the system returns control to the next frontmost layer, which will not necessarily be your application.

If you set both high bits of `LaunchFlags`, which requests a sublaunch, your application will continue to execute after the call to `_Launch`. Under MultiFinder, the actual launch (and suspend of your application) will not happen in the `_Launch` trap, but rather after a call or more to `_WaitNextEvent`.

Under MultiFinder, `_Launch` currently returns an error if there is not enough memory to launch the desired application, if it cannot locate the desired application, or if the desired application is already open. In the latter case, that application will **not** be made active. If you attempted to launch, MultiFinder will call `_SysBeep`, your application will terminate, and control will given to the next frontmost layer. If you attempted to sublaunch, control will return to your application, and it is up to you to report the error to the user.

Currently, `_Launch` returns an error in register `D0` for a sublaunch, and you should check it for errors (`D0<0`) after any attempts at sublaunching. If `D0>=0` then your sublaunch was successful.

You should refer to the *Programmer's Guide to MultiFinder* (APDA) and Macintosh Technical Notes #180, MultiFinder Miscellanea and #205, MultiFinder Revisited: The 6.0 System Release, for further discussion of the `_Launch` trap under MultiFinder.)

Working Directories and Sublaunching With the Finder

Putting aside the compatibility issue for the moment, the only problem sublaunching creates under the **current** system is one of Working Directory Control Blocks (WDCBs). Unless the application you are launching is at the root directory or on an MFS volume, you must create a new WDCB and set it as the current directory when you launch the application.

In the example which follows, the new working directory is opened (allocated) by Standard File and its `WDRefNum` is returned in `reply.vRefNum`. If you do not use Standard File and cannot assume, for instance, that the application was in the blessed folder or root directory, then you must open a new working directory explicitly via a call to `_OpenWD`. You should give the new WDCB a `WDProcID` of `'ERIK'`, so the Finder (or another shell) would know to deallocate when it saw it was allocated by a "sublaunchee."

Although the sublaunching process is recursive (i.e., programs which are sublaunched may, in turn, sublaunch other programs), there is a limit of 40 on the number of WDCBs which can be created. With this limit, you could run out of available WDCBs very quickly if many programs were playing the shell game or neglecting to deallocate the WDCBs they had created. Make sure you check for **all** errors after calling `_PBOpenWD`. A `tMWDOErr` (–121) means that all available

WDCBs have been allocated, and if you receive this error, you should alert the user that the sublaunch failed and continue as appropriate.

> **Warning:** Although the example included in this Note covers sublaunching, Developer Technical Support strongly recommends that developers **not** use this feature of the `_Launch` trap. This trap will change in the not-too-distant future, and when it does change, applications which perform sublaunching **will** break. The only circumstance in which you could consider sublaunching is if you are implementing an integrated development system and are prepared to deal with the possibility of revising it every time Apple releases a new version of the System Software.

MPW Pascal

```
{It is assumed that the Signals are caught elsewhere; see Technical
 Note #88 for more information on the Signal mechanism}

{the extended parameter block to _Launch}
TYPE
   pLaunchStruct = ^LaunchStruct;
   LaunchStruct = RECORD
      pfName       : StringPtr;
      param        : INTEGER;
      LC           : PACKED ARRAY[0..1] OF CHAR; {extended parameters:}
      extBlockLen  : LONGINT; {number of bytes in extension = 6}
      fFlags       : INTEGER; {Finder file info flags (see below)}
      launchFlags  : LONGINT; {bit 31,30=1 for sublaunch, others reserved}
   END; {LaunchStruct}

FUNCTION LaunchIt(pLaunch: pLaunchStruct): OSErr; {< 0 means error}
   INLINE $205F, $A9F2, $3E80;
{ pops pointer into A0, calls Launch, pops D0 error code into result:
  MOVE.L  (A7)+,A0
  _Launch
  MOVE.W  D0,(A7)   ; since it MAY return }

PROCEDURE DoLaunch(subLaunch: BOOLEAN);  {Sublaunch if true and launch if false}

      VAR
         myLaunch      : LaunchStruct;    {launch structure}
         where         : Point;           {where to display dialog}
         reply         : SFReply;         {reply record}
         myFileTypes   : SFTypeList;      {we only want APPLs}
         numFileTypes  : INTEGER;
         myPB          : CInfoPBRec;
         dirNameStr    : str255;

      BEGIN
         where.h := 20;
         where.v := 20;
         numFileTypes:= 1;
         myFileTypes[0]:= 'APPL';        {applications only!}
      {Let the user choose the file to Launch}
         SFGetFile(where, '', NIL, numFileTypes, myFileTypes, NIL, reply);
```

```
    IF reply.good THEN BEGIN
       dirNameStr:= reply.fName;    {initialize to file selected}

{Get the Finder flags}
       WITH myPB DO BEGIN
           ioNamePtr:= @dirNameStr;
           ioVRefNum:= reply.vRefNum;
           ioFDirIndex:= 0;
           ioDirID:= 0;
       END; {WITH}
       Signal(PBGetCatInfo(@MyPB,FALSE));
{Set the current volume to where the target application is}
       Signal(SetVol(NIL, reply.vRefNum));

{Set up the launch parameters}
       WITH myLaunch DO BEGIN
          pfName := @reply.fName;    {pointer to our fileName}
          param := 0;                {we don't want alternate screen or sound buffers}
          LC := 'LC';                {here to tell Launch that there is non-junk next}
          extBlockLen := 6;          {length of param. block past this long word}
          {copy flags; set bit 6 of low byte to 1 for RO access:}
          fFlags := myPB.ioFlFndrInfo.fdFlags;  {from GetCatInfo}

{Test subLaunch and set LaunchFlags accordingly}
       IF subLaunch THEN
          LaunchFlags := $C0000000              {set BOTH high bits for a sublaunch}
       ELSE
          LaunchFlags := $00000000;             {Just launch then quit}
       END; {WITH}

 {launch; you might want to put up a dialog which explains that
  the selected application couldn't be launched for some reason.}
       Signal(LaunchIt(@myLaunch));
   END; {IF reply.good}

END; {DoLaunch}
```

MPW C

```
typedef struct LaunchStruct {
        char            *pfName;            /* pointer to the name of launchee */
        short int       param;
        char            LC[2];              /*extended parameters:*/
        long int        extBlockLen;        /*number of bytes in extension == 6*/
        short int       fFlags;             /*Finder file info flags (see below)*/
        long int        launchFlags;        /*bit 31,30==1 for sublaunch, others reserved*/
} *pLaunchStruct;

pascal OSErr LaunchIt( pLaunchStruct pLnch) /* < 0 means error */
        = {0x205F, 0xA9F2, 0x3E80};

/* pops pointer into A0, calls Launch, pops D0 error code into result:
         MOVE.L  (A7)+,A0
         _Launch
         MOVE.W  D0,(A7)  ; since it MAY return */

OSErr DoLaunch(subLaunch)
        Boolean                 subLaunch;      /* Sublaunch if true and launch if false    */
{  /* DoLaunch */
        struct LaunchStruct     myLaunch;
        Point                   where;          /*where to display dialog*/
        SFReply                 reply;          /*reply record*/
        SFTypeList              myFileTypes;    /* we only want APPLs */
        short int               numFileTypes=1;
        HFileInfo               myPB;
        char                    *dirNameStr;
        OSErr                   err;

        where.h = 80;
        where.v = 90;
        myFileTypes[0] = 'APPL';                /* we only want APPLs */
        /*Let the user choose the file to Launch*/
        SFGetFile(where, "", nil, numFileTypes, myFileTypes, nil, &reply);

        if (reply.good)
        {
                dirNameStr = &reply.fName;   /*initialize to file selected*/

        /*Get the Finder flags*/
                myPB.ioNamePtr= dirNameStr;
                myPB.ioVRefNum= reply.vRefNum;
                myPB.ioFDirIndex= 0;
                myPB.ioDirID = 0;
                err = PBGetCatInfo((CInfoPBPtr) &myPB,false);
                if (err != noErr)
                        return err;

        /*Set the current volume to where the target application is*/
                err = SetVol(nil, reply.vRefNum);
                if (err != noErr)
                        return err;

        /*Set up the launch parameters*/
                myLaunch.pfName = &reply.fName;     /*pointer to our fileName*/
                myLaunch.param = 0;                 /*we don't want alternate screen
                                                     or sound buffers*/
        /*set up LC so as to tell Launch that there is non-junk next*/
                myLaunch.LC[0] = 'L'; myLaunch.LC[1] = 'C';
                myLaunch.extBlockLen = 6;           /*length of param. block past
                                                     this long word*/
        /*copy flags; set bit 6 of low byte to 1 for RO access:*/
                myLaunch.fFlags = myPB.ioFlFndrInfo.fdFlags;      /*from _GetCatInfo*/
        /* Test subLaunch and set launchFlags accordingly */
        if ( subLaunch )
                myLaunch.launchFlags = 0xC0000000;   /*set BOTH hi bits for a sublaunch  */
        else
                myLaunch.launchFlags = 0x00000000;   /* Just launch then quit            */

                err = LaunchIt(&myLaunch);           /* call _Launch                     */
                if (err < 0)
                {
                /* the launch failed, so put up an alert to inform the user */
                        LaunchFailed();
                        return err;
                }
                else
                        return noErr;
        } /*if reply.good*/
} /*DoLaunch*/
```

Further Reference:

- *Inside Macintosh*, Volumes I-12, II-53, & IV-83, The Segment Loader
- *Programmer's Guide to MultiFinder* (APDA)
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #180, MultiFinder Miscellanea
- Technical Note #205, MultiFinder Revisited: The 6.0 System Release

#127: TextEdit EOL Ambiguity

See also: TextEdit

Written by: Rick Blair
May 4, 1987
Updated: March 1, 1988

`TESetSelect` may be used to position the insertion point at the end of a line. There is an ambiguity, though; should the insertion point appear at the end of the preceding line or the start of the following one? It is possible to determine what will happen, as you are about to see.

There is an internal flag used by TextEdit to determine where the insertion point at the end of a line appears. This flag is part of the `clikStuff` field in the `TERec`. It is there mainly for the use of `TEClick`, but it is also used by `TESetSelect` (although it defaults to the right side of the previous line).

The following code can be used to force the insertion point to appear at the left of the following line when it is positioned at the end of a line; in MPW Pascal:

```
TEDeactivate(tH);
tH^^.clikStuff := 255;                                  {position caret on left}
TESetSelect(eolcharpos, eolcharpos, tH);                {ambiguous point}
TEActivate(tH);
```

In MPW C:

```
TEDeactivate(tH);
(**tH).clikStuff = 255;                                 /*position caret on left*/
TESetSelect(eolcharpos, eolcharpos, tH);                /*ambiguous point*/
TEActivate(tH);
```

If you want to ensure that the caret is on the right side (to which it normally defaults) then substitute a zero for the 255.

Macintosh Technical Notes

#128: PrGeneral

See also: The Printing Manager
Technical Note #118—
How to Check and Handle Printing Errors

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

The Printing Manager architecture has been expanded to include a new procedure called `PrGeneral`. The features described here are advanced, special-purpose features, intended to solve specific problems for those applications that need them. The calls to determine printer resolution introduce a good deal of complexity into the application's code, and should be used only when necessary.

Version 2.5 (and later) of the ImageWriter driver and version 4.0 (and later) of the LaserWriter driver implement a generic Printing Manager procedure called `PrGeneral`. This procedure allows the Print Manager to expand in functionality, by allowing printer drivers to implement various new functions. The Pascal declaration of `PrGeneral` is:

```
PROCEDURE PrGeneral (pData: Ptr);
```

The `pData` parameter is a pointer to a data block. The structure of the data block is declared as follows:

```
TGnlData = RECORD {1st 8 bytes are common for all PrGeneral calls)
   iOpCode    : INTEGER;   {input}
   iError     : INTEGER;   {output}
   lReserved  : LONGINT;   {reserved for future use}
   {more fields here, depending on particular call}
END;
```

The first field is a 2-byte opcode, `iOpCode`, which acts like a routine selector. The currently available opcodes are described below.

The second field is the error result, `iError`, which is returned by the print code. This error only reflects error conditions that occur during the `PrGeneral` call. For example, if you use an opcode that isn't implemented in a particular printer driver then you will get a `OpNotImpl` error.

Here are the errors currently defined:

```
CONST
   noErr = 0;                  {everything's hunky}
   NoSuchRsl = 1;              {the resolution you chose isn't available}
   OpNotImpl = 2;              {the driver doesn't support this opcode}
```

After calling PrGeneral you should always check PrError. If noErr is returned, then you can proceed. If ResNotFound is returned, then the current printer driver doesn't support PrGeneral and you should proceed appropriately. See Technical Note #118 for details on checking errors returned by the Printing Manager.

IError is followed by a four byte reserved field (that means don't use it). The contents of the rest of the data block depends on the opcode that the application uses. There are currently five opcodes used by the ImageWriter and LaserWriter drivers.

The Opcodes

Initially, the following calls are implemented via PrGeneral:

- GetRslData (get resolution data): iOpCode = 4
- SetRsl (set resolution): iOpCode = 5
- DraftBits (bitmaps in draft mode): iOpCode = 6
- noDraftBits (no bitmaps in draft mode): iOpCode = 7
- GetRotn (get rotation): iOpCode = 8

The GetRslData and SetRsl allow the application to find out what physical resolutions the printer supports, and then specify a supported resolution. DraftBits and noDraftBits invoke a new feature of the ImageWriter, allowing bitmaps (imaged via CopyBits) to be printed in draft mode. GetRotn lets an application know whether landscape has been selected. Below is a detailed description of how each routine works.

The GetRslData Call

GetRslData (iOpCode = 4) returns a record that lets the application know what resolutions are supported by the current printer. The application can then use SetRsl (description follows) to tell the printer driver which one it will use. This is the format of the input data block for the GetRslData call:

```
TRslRg = RECORD            {used in TGetRslBlk}
   iMin, iMax: Integer;    {0 if printer only supports discrete resolutions}
END;

TRslRec = RECORD           {used in TGetRslBlk}
   iXRsl, iYRsl: Integer;  {a discrete, physical resolution}
END;
```

```
TGetRslBlk = RECORD      {data block for GetRslData call}
   iOpCode:     Integer;  {input; = getRslDataOp}
   iError:      Integer;  {output}
   lReserved:   LongInt;  {reserved for future use}
   iRgType:     Integer;  {output; version number}
   XRslRg:      TRslRg;   {output; range of X resolutions}
   YRslRg:      TRslRg;   {output; range of Y resolutions}
   iRslRecCnt:  Integer;  {output; how many RslRecs follow}
   rgRslRec:    ARRAY[1..27] OF TRslRec;  {output; number filled depends on
                                            printer type}
END;
```

The `iRgType` field is much like a version number; it determines the interpretation of the data that follows. At present, a `iRgType` value of 1 applies both to the LaserWriter and to the ImageWriter.

For variable-resolution printers like the LaserWriter, the resolution range fields `XRslRg` and `YRslRg` express the ranges of values to which the X and Y resolutions can be set. For discrete-resolution printers like the ImageWriter, the values in the resolution range fields are zero.

Note: In general, X and Y in these records are the horizontal and vertical directions of the **printer**, not the document! In landscape orientation, X is horizontal on the printer but vertical on the document.

After the resolution range information there is a word which gives the number of resolution records that contain information. These records indicate the physical resolutions at which the printer can actually print dots. Each resolution record gives an X value and a Y value.

When you call `PrGeneral` you pass in a data block that looks like this:

OpCode = 4	1 word
Error Code	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 0, max = 0	2 words
Y Resolution Range: min =0, max = 0	2 words
Resolution Record Count =0	1 word
Resolution Record #1: X = 0, Y = 0	2 words
Resolution Record #2..27	

Below is the data block returned for the LaserWriter:

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 72, max = 1500	2 words
Y Resolution Range: min = 72, max = 1500	2 words
Resolution Record Count = 1	1 word
Resolution Record #1: X = 300, Y = 300	2 words

Note that all the resolution range numbers happen to be the same for this printer. There is only one resolution record, which gives the physical X and Y resolutions of the printer (300x300).

Below is the data block returned for the ImageWriter.

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min =0, max = 0	2 words
Y Resolution Range: min = 0, max = 0	2 words
Resolution Record Count = 4	1 word
Resolution Record #1: X = 72, Y = 72	2 words
Resolution Record #2: X =144, Y = 144	2 words
Resolution Record #3: X = 80, Y = 72	2 words
Resolution Record #4: X = 160, Y = 144	2 words

All the resolution range values are zero, because only discrete resolutions can be specified for this printer. There are four resolution records giving these discrete physical resolutions.

Note that `GetRslData` always returns the same information for a particular printer type—it is **not** dependent on what the user does or on printer configuration information.

The SetRsl Call

SetRsl (iOpCode = 5) is used to specify the desired imaging resolution, after using GetRslData to determine a workable pair of values. Below is the format of the data block:

```
TSetRslBlk =    RECORD     {data block for SetRsl call}
   iOpCode:     Integer;   {input; = setRslOp}
   iError:      Integer;   {output}
   lReserved:   LongInt;   {reserved for future use}
   hPrint:      THPrint;   {input; handle to a valid print record}
   iXRsl:       Integer;   {input; desired X resolution}
   iYRsl:       Integer;   {input; desired Y resolution}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate. If the call executes successfully, the print record is updated with the new resolution; the data block comes back with 0 for the error and is otherwise unchanged.

However, if the desired resolution is not supported, the error is set to noSuchRsl and the resolution fields are set to the printer's default resolution

Note that you can undo the effect of a previous call to SetRsl by making another call that specifies an unsupported resolution (such as 0x0), forcing the default resolution.

The DraftBits Call

DraftBits (iOpCode = 6) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) Below is the format of the data block:

```
TDftBitsBlk =   RECORD     {data block for DraftBits and NoDraftBits calls}
   iOpCode:     Integer;   {input; = draftBitsOp or noDraftBitsOp}
   iError:      Integer;   {output}
   lReserved:   LongInt;   {reserved for future use}
   hPrint:      THPrint;   {input; handle to a valid print record}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate.

This call forces draft-mode (i.e., immediate) printing, and will allow bitmaps to be printed via CopyBits calls. The virtue of this is that you avoid spooling large masses of bitmap data onto the disk, and you also get better performance.

The following restrictions apply:

- This call should be made before bringing up the print dialogs because it affects their appearance. On the ImageWriter, calling DraftBits disables the landscape icon in the Style dialog, and the Best, Faster, and Draft buttons in the Job dialog.

- If the printer does not support draft mode, already prints bitmaps in draft mode, or does not print bitmaps at all, this call does nothing.

- Only text and bitmaps can be printed.

- As in the normal draft mode, landscape format is not allowed.

- Everything on the page must be strictly Y-sorted, i.e. no reverse paper motion between one string or bitmap and the next. Note that this means you can't have two or more objects (text or bitmaps) side by side; the top boundary of each object must be no higher than the bottom of the preceding object.

The last restriction is important. If you violate it, you will not like the results. But note that if you want two or more bitmaps side by side, you can combine them into one before calling `CopyBits` to print the result. Similarly, if you are just printing bitmaps you can rotate them yourself to achieve landscape printing.

The NoDraftBits Call

`NoDraftBits` (`iOpCode` = 7) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) The format of the data block is the same as that for the `DraftBits` call.

This call cancels the effect of any preceding `DraftBits` call. If there was no preceding `DraftBits` call, or the printer does not support draft-mode printing anyway, this call does nothing.

The GetRotn Call

`GetRotn` (`iOpCode` = 8) is implemented on the ImageWriter and LaserWriter. Here is the format of the data block:

```
TGetRotnBlk =  RECORD          {data block for GetRotn call}
   iOpCode:    Integer;        {input; = getRotnOp}
   iError:     Integer;        {output}
   lReserved:  LongInt;        {reserved for future use}
   hPrint:     THPrint;        {input; handle to a valid print record}
   fLandscape: Boolean;        {output; Boolean flag}
   bXtra:      SignedByte;     {reserved}
END;
```

`hPrint` should be the handle to a print record that has previously been passed to `PrValidate`.

If landscape orientation is selected in the print record, then `fLandscape` is true.

How To Use The PrGeneral Opcodes

The SetRsl and DraftBits calls may require the print code to suppress certain options in the Style and/or Job dialogs, therefore they should always be called before any call to the Style or Job dialogs. An application might use these calls as follows:

- Get a new print record by calling PrintDefault, or take an existing one from a document and call PrValidate on it.
- Call GetRslData to find out what the printer is capable of, and decide what resolution to use. Check PrError to be sure the PrGeneral call is supported on this version of the print code; if the error is ResNotFound, you have older print code and must print accordingly. But if the PrError return is 0, proceed:
- Call SetRsl with the print record and the desired resolution if you wish.
- Call DraftBits to invoke the printing of bitmaps in draft mode if you wish.

Note that if you call either SetRsl or DraftBits, you should do so before the user sees either of the printing dialogs.

#129: _Gestalt & _SysEnvirons—a Never-Ending Story

Revised by: Dave Radcliffe May 1992
Written by: Jim Friedlander May 1987

This Technical Note discusses the latest changes and enhancements in the `_Gestalt` and `_SysEnvirons` calls.

Changes since October 1991: Clarified information on Gestalt information for Macintosh PowerBook computers and added information on the Macintosh LC II and the `gestaltHardwareAttr` selector.

Introduction

Previous versions of this Note provided the latest documentation on new information the `_SysEnvirons` trap could return. DTS will continue to revise this Note to provide this information; however, as the `_Gestalt` trap is now the preferred method for determining information about a machine environment, this Note will also provide up-to-date information on `_Gestalt` selectors.

_Gestalt

This Note now documents `_Gestalt` selectors and return values added since the release of *Inside Macintosh* Volume VI. Please note that this is supplemental information; for the complete description of `_Gestalt` and its use, please refer to *Inside Macintosh* Volume VI.

The Macintosh LC II is identical to the Macintosh LC, except for the presence of an MC68030 processor, so it returns the *same* gestaltMachineType response as the Macintosh LC (i.e. 19). Developers are reminded that the gestaltMachineType selector is for informational purposes only and should not be used as a basis for programmatic decisions. As always, developers are encouraged to test for the specific features they need and not to rely on any particular machine having a particular set of features.

Note: The *Macintosh PowerBook 100 Developer Notes* and the *Macintosh PowerBook 140/170 Developer Notes*, available from APDA and on the *Developer CD Series* disc and AppleLink, incorrectly document `gestaltMachineType` response values for the Macintosh PowerBook computers. The following values are, and have always been, the correct values.

Additional Gestalt Response Values

```
{ gestaltMachineType response values }
        gestaltQuadra900              = 20;    { Macintosh Quadra 900 }
        gestaltPowerBook170           = 21;    { Macintosh PowerBook 170 }
        gestaltQuadra700              = 22;    { Macintosh Quadra 700 }
        gestaltClassicII              = 23;    { Macintosh Classic II }
        gestaltPowerBook100           = 24;    { Macintosh PowerBook 100 }
        gestaltPowerBook140           = 25;    { Macintosh PowerBook 140 }

{ gestaltKeyboardType response values }
        gestaltPwrBookADBKbd          = 12;    { PowerBook Keyboard }
        gestaltPwrBookISOADBKbd       = 13;    { PowerBook Keyboard (ISO) }
```

gestaltHardwareAttr Selector

The `gestaltHardwareAttr` selector has been a source of confusion for developers since originally documented in *Inside Macintosh* Volume VI . This section will try to reduce that confusion and also introduce additional information returned by the selector. But be warned that use of this selector for anything other than informational purposes should be deemed a compatibility risk. In other words, if you are dependent on the information returned by this selector to function on existing computers, you will almost certainly have problems on future systems.

The reason for this is that `gestaltHardwareAttr` returns very low-level hardware information. If you need to use this information, it implies you are too hardware dependent. So be very careful about using this information.

The principal source of confusion is bit 7, described as `gestaltHasSCSI`. What this bit really means is the machine is equipped with SCSI based on the 53C80 chip, which was introduced in the Macintosh Plus. This bit will be zero on the Macintosh IIfx and the Macintosh Quadra computers because they have a different low-level SCSI implementation. The Macintosh IIfx has a 53C80 compatible chip that also supports SCSI DMA. It reports this information using bit 6 of the `gestaltHardwareAttr` response. The Macintosh Quadra computers have yet another SCSI implementation based on the 53C96 chip and so report different information (see below).

Another source of confusion is bit 4 (`gestaltHasSCC`). The Macintosh IIfx and Macintosh Quadra 900 have intelligent I/O processors (IOPs) that normally isolate the hardware and make direct access to the SCC impossible. Normally, these machines will report that they do not have an SCC implying, correctly, that were you to attempt to access it directly, you would fail. However, if the user has used the Compatibility Switch control panel to enable compatibility mode, `gestaltHasSCC` will report true indicating you may access the SCC directly. But remember that doing so means you are doing direct hardware access and that there may be a day when you can't access the SCC under any circumstances.

New gestaltHardwareAttr Values for Macintosh Quadra Computers

Below are the new bits supported by the Macintosh Quadra computers. Any other bits remain undocumented and subject to change.

```
        gestaltHasSCSI961                  = 21; { 53C96 SCSI controller on internal bus
}
```

```
        gestaltHasSCSI962                    = 22; { 53C96 SCSI controller on external bus
}
```

_SysEnvirons

_SysEnvirons was the standard way to determine the features available on a given machine. The preferred method to get this information is now _Gestalt; information on _SysEnvirons is now provided only for backward compatibility.

As originally conceived, _SysEnvirons would check the versionRequested parameter to determine what level of information you were prepared to handle, but this technique means updating _SysEnvirons for every new hardware product Apple produces. With system software version 6.0, _SysEnvirons introduced version 2 of environsVersion to provide information about new hardware as we introduce it; this new version returns the same SysEnvRec as version 1.

Beginning with system software version 6.0.1, Apple releases a new version of _SysEnvirons only when engineering makes changes to its structure (that is, when they add new fields to SysEnvRec); all existing versions return accurate information about the machine environment even if part of that information was not originally defined for the version you request. For example, if you call _SysEnvirons with versionRequested = 1 on a Macintosh IIfx, it returns a machineType of envMacIIfx even though this machine type originally was not defined for version 1 of the call.

You should use version 2 of _SysEnvirons until Apple releases a newer version. MPW 3.0 defines a constant curSysEnvVers, which can be used to minimize the need for source code revisions when _SysEnvirons evolves. Regardless of the version used, however, your software should be prepared to handle unexpected values and should not make assumptions about functionality based on current expectations. For example, if your software currently requires a Macintosh II, testing for machineType >= envMacII may result in your software trying to run on a machine that does not support the features it requires, so test for specific functionality (that is, hasFPU, hasColorQD, and so on).

Warning: This test for specific functionality is particularly true of FPUs (floating-point units). Some CPUs, such as the Macintosh IIsi, may have optional, user-installed FPUs; therefore, an application should not assume that any Macintosh with a microprocessor greater than a 68000 (for example, 68020, 68030 or 68040) has an FPU (68881/68882 or built-in for the 68040). If an application makes a conditional branch to execute floating-point instructions directly, then it should first explicitly check for the presence of the FPU.

You should always check the environsVersion when returning from _SysEnvirons since the glue always returns as much information as possible, with environsVersion indicating the highest version available, even if the call returns an envSelTooBig (–5502) error.

Calling _SysEnvirons From a High-Level Language

Due to a documentation error in *Inside Macintosh* Volume V, DTS still receives questions about how to call _SysEnvirons properly from Pascal and C. *Inside Macintosh* defines the Pascal interface to _SysEnvirons as follows:

```
FUNCTION SysEnvirons (versRequested: INTEGER; VAR theWorld: SysEnvRecPtr) : OSErr;
```

Because theWorld is passed by reference (as a VAR parameter), it is not correct to pass a SysEnvRecPtr in the second argument. Pascal would then generate a pointer to this pointer and pass that to the _SysEnvirons trap in A0. (The assembly-language information is essentially correct; _SysEnvirons really does want a pointer to a SysEnvRec in A0.) The correct Pascal interface to _SysEnvirons is therefore:

```
FUNCTION SysEnvirons (versionRequested: INTEGER; VAR theWorld: SysEnvRec) : OSErr;
```

In this case, Pascal pushes a pointer to theWorld on the stack. The Pascal interface glue then pops this pointer off the stack directly into A0 and calls _SysEnvirons. Everything is copacetic.

C programmers should recognize their corresponding interface:

```
pascal OSErr SysEnvirons (short versionRequested, SysEnvRec *theWorld);
```

Inside Macintosh defines the type SysEnvPtr = ^SysEnvRec. It also sometimes refers to this type as SysEnvRecPtr. The inconsistency is insignificant because in reality MPW does not define any such type, under either name; therefore, it is never needed.

Inside Macintosh also states that "all of the Toolbox Managers must be initialized before calling SysEnvirons." This statement is not necessarily true. Startup documents (INITs), for instance, may wish to call _SysEnvirons without initializing any of the Toolbox Managers. Keep in mind that the atDrvrVersNum field returns a zero result if the AppleTalk drivers are not initialized. The system version, machine type, processor type, and other key data return normally.

Additional _SysEnvirons Constants

The following are new _SysEnvirons constants which are not documented in *Inside Macintosh*; however, you should refer to *Inside Macintosh* Volume V-1, Compatibility Guidelines, for the rest of the story.

machineType

```
envMacIIx              = 5;     { Macintosh IIx }
envMacIIcx             = 6;     { Macintosh IIcx }
envSE30                = 7;     { Macintosh SE/30 }
envPortable            = 8;     { Macintosh Portable }
envMacIIci             = 9;     { Macintosh IIci }
envMacIIfx             = 11;    { Macintosh IIfx }
envMacClassic          = 15;    { Macintosh Classic }
envMacIIsi             = 16;    { Macintosh IIsi }
envMacLC               = 17;    { Macintosh LC }
envMacQuadra900        = 18;    { Macintosh Quadra 900 }
envMacPowerBook170     = 19;    { Macintosh PowerBook 170 }
envMacQuadra700        = 20;    { Macintosh Quadra 700 }
envMacClassicII        = 21;    { Macintosh Classic II }
envMacPowerBook100     = 22;    { Macintosh PowerBook 100 }
envMacPowerBook140     = 23;    { Macintosh PowerBook 140 }
```

processor

```
env68030           = 4;    { MC68030 processor }
env68040           = 5;    { MC68040 processor }
```

keyBoardType

```
envPrtblADBKbd     = 6;    { Portable Keyboard }
envPrtblISOKbd     = 7;    { Portable Keyboard (ISO) }
envStdISOADBKbd    = 8;    { Apple Standard Keyboard (ISO) }
envExtISOADBKbd    = 9;    { Apple Extended Keyboard (ISO)

envADBKbdII        = 10;   { Apple Keyboard II }
envADBISOKbdII     = 11;   { Apple Keyboard II (ISO) }
envPwrBkADBKbd     = 12;   { PowerBook Keyboard }
envPwrBkISOKbd     = 13;   { PowerBook Keyboard (ISO) }
```

Further Reference:

- *Inside Macintosh*, Volumes V and VI, Compatibility Guidelines

#130: Clearing ioCompletion

See also: The File Manager

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

When making synchronous calls to the File Manager, it is not necessary to clear `ioCompletion` field of the parameter block, since that is done for you.

Some earlier technotes explicitly cleared `ioCompletion`, with the knowledge that this was unnecessary, to try to encourage developers to fill in all fields of parameter blocks as indicated in *Inside Macintosh.*

By the way, this is true of all parameter calls—you only have to set fields that are explicitly required.

Macintosh Technical Notes

#131: TextEdit Bugs in System 4.2

Written by: Chris Derossi June 1, 1987
Updated: March 1, 1988

This note formerly described the known bugs with the version of Styled TextEdit that was provided with System 4.1. Many of these bugs were fixed in System 4.2. This updated Technical Note describes the remaining known problems.

TEStylInsert

Calling `TEStylInsert` while the TextEdit record is deactivated causes unpredictable results, so make sure to only call `TEStylInsert` when the TextEdit record is active.

TESetStyle

When using the `doFace` mode with `TESetStyle`, the style that you pass as a parameter is `ORed` into the style of the currently selected text. If you pass the empty set (no styles) though, `TESetStyle` is supposed to remove all styles from the selected text. But `TESetStyle` checks an entire word instead of just the high-order byte of the `tsFace` field. The style information is contained completely in the high-order byte, and the low-order byte may contain garbage.

If the low-order byte isn't zero, `TESetStyle` thinks that the `tsFace` field isn't empty, so it goes ahead and ORs it with the selected text's style. Since the actual style portion of the `tsFace` field is zero, no change occurs with the text. If you want to have `TESetStyle` remove all styles from the text, you can explicitly set the `tsFace` field to zero like this:

```
VAR
   myStyle   : TextStyle;
   anIntPtr : ^Integer;

BEGIN
   ...
   anIntPtr := @myStyle.tsFace;
   anIntPtr^ := 0;
   TESetStyle(doFace, myStyle, TRUE, textH);
   ...
END;
```

TEStylNew

The line heights array does not get initialized when TEStylNew is called. Because of this, the caret is initially drawn in a random height. This is easily solved by calling TECalText immediately after calling TEStylNew. Extra calls to TECalText don't hurt anything anyway, so this will be compatible with future Systems.

An extra character run is placed at the beginning of the text which corresponds to the font, size, and style which were in the grafPort when TEStylNew was called. This can cause the line height for the first line to be too large. To avoid this, call TextSize with the desired text size before calling TEStylNew. If the text's style information cannot be determined in advance, then call TextSize with a small value (like 9) before calling TEStylNew.

TEScroll

The bug documented in Technical Note #22 remains in the new TextEdit. TEScroll called with zero for both vertical and horizontal displacements causes the insertion point to disappear. The workaround is the same as before; check to make sure that dV and dH are not both zero before calling TEScroll.

Growing TextEdit Record

TextEdit is supposed to dynamically grow and shrink the LineStarts array in the TERec so that it has one entry per line. Instead, when lines are added, TextEdit expands the array without first checking to see if it's already big enough. In addition, TextEdit never reduces the size of this array.

Because of this, the longer a particular TextEdit record is used, the larger it will get. This can be particularly nasty in programs that use a single TERec for many operations during the program's execution.

Restoring Saved TextEdit Records

Applications have used a technique for saving and restoring styled text which involves saving the contents of all of the TextEdit record handles. When restoring, TEStylNew is called and the TextEdit record's handles are disposed. The saved handles are then loaded and put into the TextEdit record. This technique should not be used for the nullStyle handle in the style record.

Instead, when TEStylNew is called, the nullStyle handle from the style record should be copied into the saved style record. This will ensure that the fields in the null-style record point to valid data.

#87: Error in FCBPBRec

See also: The File Manager

Written by: Jim Friedlander August 18, 1986
Updated: March 1, 1988

The declaration of a FCBPBRec is wrong in *Inside Macintosh Volume IV* and early versions of MPW. This has been fixed in MPW 1.0 and newer.

An error was made in the declaration of an FCBPBRec parameter block that is used in PBGetFCBInfo calls. The field ioFCBIndx was incorrectly listed as a LONGINT. The following declaration (found in *Inside Macintosh*):

```
...
ioRefNum:      INTEGER;
filler:        INTEGER;
ioFCBIndx:     LONGINT;
ioFCBFlNm:     LONGINT;
...
```

should be changed to:

```
...
ioRefNum:      INTEGER;
filler:        INTEGER;
ioFCBIndx:     INTEGER;
ioFCBFiller1:  INTEGER;
ioFCBFlNm:     LONGINT;
...
```

Macintosh Technical Notes

#88: Signals

See also: Using Assembly Language (Mixing Pascal & Assembly)

Written by: Rick Blair August 1, 1986
Updated: March 1, 1988

Signals are a form of intra-program interrupt which can greatly aid clean, inexpensive error trapping in stack frame intensive languages. A program may invoke the `Signal` procedure and immediately return to the last invocation of `CatchSignal`, including the complete stack frame state at that point.

Signals allow a program to leave off execution at one point and return control to a convenient error trap location, regardless of how many levels of procedure nesting are in between.

The example is provided with a Pascal interface, but it is easily adapted to other languages. The only qualification is that the language **must** bracket its procedures (or functions) with `LINK` and `UNLK` instructions. This will allow the signal code to clean up at procedure exit time by removing `CatchSignal` entries from its internal queue. **Note:** only procedures and/or functions that call `CatchSignal` need to be bracketed with `LINK` and `UNLK` instructions.

Important: `InitSignals` must be called from the main program so that `A6` can be set up properly.

Note that there is no limit to the number of local `CatchSignals` which may occur within a single routine. Only the last one executed will apply, of course, unless you call `FreeSignal`. `FreeSignal` will "pop" off the last `CatchSignal`. If you attempt to `Signal` with no `CatchSignals` pending, `Signal` will halt the program with a debugger trap.

`InitSignals` creates a small relocatable block in the application heap to hold the signal queue. If `CatchSignal` is unable to expand this block (which it does 5 elements at a time), then it will signal back to the last successful `CatchSignal` with `code` = 200. A `Signal(0)` acts as a `NOP`, so you may pass `OSErrs`, for instance, after making File System type calls, and, if the `OSErr` is equal to `NoErr`, nothing will happen.

`CatchSignal` may not be used in an expression if the stack is used to evaluate that expression. For example, you can't write:

```
c:= 3*CatchSignal;
```

"Gotcha" summary

1. Routines which call `CatchSignal` must have stack frames.
2. `InitSignals` must be called from the outermost (main) level.
3. Don't put the `CatchSignal` function in an expression. Assign the result to an `INTEGER variable; i.e. i:=CatchSignal.`
4. It's safest to call a procedure to do the processing after `CatchSignal` returns. See the Pascal example `TestSignals` below. This will prevent the use of a variable which may be held in a register.

Below are three separate source files. First is the Pascal interface to the signaling unit, then the assembly language which implements it in MPW Assembler format. Finally, there is an example program which demonstrates the use of the routines in the unit.

```
{File ErrSignal.p}
UNIT ErrSignal;

INTERFACE

{Call this right after your other initializations (InitGraf, etc.)--in other
words as early as you can in the application}
PROCEDURE InitSignals;

{Until the procedure which encloses this call returns, it will catch
subsequent Signal calls, returning the code passed to Signal. When
CatchSignal is encountered initially, it returns a code of zero. These calls
may "nest"; i.e. you may have multiple CatchSignals in one procedure.
Each nested CatchSignal call uses 12 bytes of heap space }
FUNCTION CatchSignal:INTEGER;

{This undoes the effect of the last CatchSignal. A Signal will then invoke
the CatchSignal prior to the last one.}
PROCEDURE FreeSignal;

{Returns control to the point of the last CatchSignal. The program will then
behave as though that CatchSignal had returned with the code parameter
supplied to Signal.}
PROCEDURE Signal(code:INTEGER);

END.
{End of ErrSignal.p}
```

Here's the assembly source for the routines themselves:

```
; ErrSignal code w. InitSignal, CatchSignal,FreeSignal, Signal
; defined
;
;                Version 1.0 by Rick Blair

            PRINT     OFF
            INCLUDE   'Traps.a'
            INCLUDE   'ToolEqu.a'
            INCLUDE   'QuickEqu.a'
            INCLUDE   'SysEqu.a'
            PRINT     ON


CatchSigErr EQU       200               ;"insufficient heap" message
SigChunks   EQU       5                 ;number of elements to expand by
FrameRet    EQU       4                 ;return addr. for frame (off A6)
SigBigA6    EQU       $FFFFFFFF         ;maximum positive A6 value


; A template in MPW Assembler describes the layout of a collection of data
; without actually allocating any memory space. A template definition starts ;
with a RECORD directive and ends with an ENDR directive.

; To illustrate how the template type feature works, the following template
; is declared and used. By using this, the assembler source appromixates very
; closely Pascal source for referencing the corresponding information.

;template for our table elements
  SigElement  RECORD    0     ;the zero is the template origin
  SigSP       DS.L      1     ;the SP at the CatchSignal-(DS.L just like EQU)
  SigRetAddr  DS.L      1     ;the address where the CatchSignal returned
  SigFRet     DS.L      1     ;return addr. for encl. procedure
  SigElSize   EQU       *     ;just like EQU 12
              ENDR


; The global data used by these routines follows. It is in the form of a
; RECORD, but, unlike above, no origin is specified, which means that memory
; space *will* be allocated.
; This data is referenced through a WITH statement at the beginning of the
; procs that need to get at this data. Since the Assembler knows when it is
; referencing data in a data module (since they must be declared before they
; are accessed), and since such data can only be accessed based on A5, there
; is no need to explicitly specify A5 in any code which references the data
; (unless indexing is used). Thus, in this program we have omitted all A5
; references when referencing the data.

  SigGlobals RECORD                     ;no origin means this is a data record
                                        ;not a template(as above)
  SigEnd      DS.L      1               ;current end of table
  SigNow      DS.L      1               ;the MRU element
  SigHandle   DC.L      0               ;handle to the table
              ENDR
```

```
InitSignals PROC     EXPORT           ;PROCEDURE InitSignals;

            IMPORT   CatchSignal
            WITH     SigElement,SigGlobals

;the above statement makes the template SigElement and the global data
;record SigGlobals available to this procedure

            MOVE.L   #SigChunks*SigElSize,D0
            _NewHandle                ;try to get a table
            BNE.S    forgetit         ;we couldn't get that!?

            MOVE.L   A0,SigHandle     ;save it
            MOVE.L   #-SigElSize,SigNow ;point "now" before start
            MOVE.L   #SigChunks*SigElSize,SigEnd ;save the end
            MOVE.L   #SigBigA6,A6     ;make A6 valid for Signal
forgetit    RTS
            ENDP

CatchSignal PROC     EXPORT           ;FUNCTION CatchSignal:INTEGER;
            IMPORT   SiggySetup,Signal,SigDeath
            WITH     SigElement,SigGlobals

            MOVE.L   (SP)+,A1         ;grab return address
            MOVE.L   SigHandle,D0     ;handle to table
            BEQ      SigDeath         ;if NIL then croak
            MOVE.L   D0,A0            ;put handle in A-register
            MOVE.L   SigNow,D0
            ADD.L    #SigElSize,D0
            MOVE.L   D0,SigNow        ;save new position
            CMP.L    SigEnd,D0        ;have we reached the end?
            BNE.S    catchit          ;no, proceed

            ADD.L    #SigChunks*SigElSize,D0 ;we'll try to expand
            MOVE.L   D0,SigEnd        ;save new (potential) end
            _SetHandleSize
            BEQ.S    @0               ;jump around if it worked!

;signals, we use 'em ourselves
            MOVE.L   SigNow,SigEnd    ;restore old ending offset
            MOVE.L   #SigElSize,D0
            SUB.L    D0,SigNow        ;ditto for current position
            MOVE.W   #catchSigErr,(SP);we'll signal a "couldn't
                                      ;                catch" error
            JSR      Signal           ;never returns of course



@0          MOVE.L   SigNow,D0

catchit     MOVE.L   (A0),A0          ;deref.
            ADD.L    D0,A0            ;point to new entry
            MOVE.L   SP,SigSP(A0)     ;save SP in entry
            MOVE.L   A1,SigRetAddr(A0) ;save return address there
            CMP.L    #SigBigA6,A6     ;are we at the outer level?
            BEQ.S    @0               ;yes, no frame or cleanup needed
            MOVE.L   FrameRet(A6),SigFRet(A0);save old frame return
                                             ;               address
            LEA      SiggyPop,A0
            MOVE.L   A0,FrameRet(A6) ;set cleanup code address
@0          CLR.W    (SP)             ;no error code (before its time)
            JMP      (A1)             ;done setting the trap

SiggyPop    JSR      SiggySetup       ;get pointer to element
            MOVE.L   SigFRet(A0),A0 ;get proc's real return address
            SUB.L    #SigElSize,D0
            MOVE.L   D0,SigNow        ;"pop" the entry
            JMP      (A0)             ;gone
            ENDP

FreeSignal  PROC     EXPORT           ;PROCEDURE FreeSignal;
            IMPORT   SiggySetup
            WITH     SigElement,SigGlobals
            JSR      SiggySetup       ;get pointer to current entry
            MOVE.L   SigFRet(A0),FrameRet(A6) ;"pop" cleanup code
            SUB.L    #SigElSize,D0
            MOVE.L   D0,SigNow        ;"pop" the entry
            RTS
            ENDP

Signal      PROC     EXPORT           ;PROCEDURE Signal(code:INTEGER);
            EXPORT   SiggySetup,SigDeath
            WITH     SigElement,SigGlobals
            MOVE.W   4(SP),D1         ;get code
            BNE.S    @0               ;process the signal if code is non-zero
            MOVE.L   (SP),A0          ;save return address
            ADDQ.L   #6,SP            ;adjust stack pointer
            JMP      (A0)             ;return to caller(code was 0)

@0          JSR      SiggySetup       ;get pointer to entry
            BRA.S    SigLoop1

SigLoop     UNLK     A6               ;unlink stack by one frame
SigLoop1    CMP.L    SigSP(A0),A6     ;is A6 beyond the saved stack?
            BLO.S    SigLoop          ;yes, keep unlinking
            MOVE.L   SigSP(A0),SP     ;bring back our SP
            MOVE.L   SigRetAddr(A0),A0 ;get return address
            MOVE.W   D1,(SP)          ;return code to CatchSignal
            JMP      (A0)             ;Houston, boost the Signal!
                     ;(or Hooston if you're from the Negative Zone)

SiggySetup  MOVE.L   SigHandle,A0
            MOVE.L   (A0),A0          ;deref.
            MOVE.L   A0,D0            ;to set CCR
            BEQ.S    SigDeath         ;nil handle means trouble
            MOVE.L   SigNow,D0        ;grab table offset to entry
            BMI.S    SigDeath         ;if no entries then give up
            ADD.L    D0,A0            ;point to current element
            RTS

SigDeath    _Debugger                 ;a signal sans catch is bad news

            ENDP
            END
```

Now for the example Pascal program:

```
PROGRAM TestSignals;
USES ErrSignal;

VAR i:INTEGER;

PROCEDURE DoCatch(s:STR255; code:INTEGER);
BEGIN
  IF code<>0 THEN BEGIN
    Writeln(s,code);
    Exit(TestSignals);
  END;
END; {DoCatch}

PROCEDURE Easy;
  PROCEDURE Never;
    PROCEDURE DoCatch(s:STR255; code:INTEGER);
    BEGIN
      IF code<>0 THEN BEGIN
        Writeln(s,code);
        Exit(Never);
      END;
    END; {DoCatch}

  BEGIN {Never}
  i:=CatchSignal;
  DoCatch('Signal caught from Never, code = ', i );

  i:=CatchSignal;
  IF i<>0 THEN DoCatch('Should never get here!',i);

  FreeSignal; {"free" the last CatchSignal}
  Signal(7); {Signal a 7 to the last CatchSignal}
  END;{Never}
BEGIN {Easy}
Never;
Signal(69);        {this won't be caught in Never}
END;{Easy}         {all local CatchSignals are freed when a procedure exits.}

BEGIN {PROGRAM}
InitSignals; {You must call this early on!}

{catch Signals not otherwise caught by the program}
i:=CatchSignal;
IF i<>0 THEN
 DoCatch('Signal caught from main, code = ',i);

Easy;
END.
```

The example program produces the following two lines of output:

```
Signal caught from Never, code = 7
Signal caught from main, code = 69
```

#89: DrawPicture Bug

Written by: Ginger Jernigan August 16, 1986
Updated: March 1, 1988

Earlier versions of this note described a bug in `DrawPicture`. This bug never occurred on 64K ROM machines, and has been fixed in System 3.2 and newer. Use of Systems older than 3.2 on non-64K ROM machines is no longer recommended.

#90: SANE Incompatibilities

Written by: Mark Baumwell August 14, 1986
Updated: March 1, 1988

Earlier versions of this note described a problem with SANE and System 2.0. Use of System 2.0 is only recommended for Macintosh 128 machines, which contain the 64K ROMs. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#91: Optimizing for the LaserWriter—Picture Comments

See also: The Print Manager
QuickDraw
Technical Note #72—
Optimizing for the LaserWriter—Techniques
Technical Note #27—MacDraw Picture Comments
PostScript Language Reference Manual, Adobe Systems
PostScript Language Tutorial and Cookbook,
Adobe Systems
LaserWriter Reference Manual

Written by:	Ginger Jernigan	November 15, 1986
Modified by:	Ginger Jernigan	March 2, 1987
Updated:		March 1, 1988

This technical note is a continuation of Technical Note #72. This technical note discusses the picture comments that the LaserWriter driver recognizes.

This technical note has been modified to include corrected descriptions of the `SetLineWidth`, `PostScriptFile` and `ResourcePS` comments and to include some additional warnings.

The implementation of QuickDraw's `picComment` facility by the LaserWriter driver allows you to take advantage of features (like rotated text) which are available in PostScript but may not be available in QuickDraw.

Warning: Using PostScript-specific comments will make your code printer-dependent and may cause compatibility problems with non-PostScript devices, so don't use them unless you absolutely have to.

Some of the picture comments below are designed to be issued along with QuickDraw commands that simulate the commented commands on the Macintosh screen. When the comments are used, the accompanying QuickDraw comments are ignored. If you are designing a picture to be printed by the LaserWriter, the structure and use of these comments must be precise, otherwise nothing will print. If another printer driver (like the ImageWriter I/II driver) has not implemented these comments, the comments are ignored and the accompanying QuickDraw commands are used.

Below are the picture comments that the LaserWriter driver recognizes:

Type	Kind	Data Size	Data	Description
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NIL	End text function
StringBegin	152	0	NIL	Begin pieces of original string
StringEnd	153	0	NIL	End pieces of original string
TextCenter	154	8	TTxtCenter	Offset to center of rotation
* LineLayoutOff	155	0	NIL	Turns LaserWriter line layout off
* LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following poly data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
picPlyClo	165	0	NIL	Close the poly
* DashedLine	180	-	TDashedLine	Draw following lines as dashed
* DashedStop	181	0	NIL	End dashed lines
* SetLineWidth	182	4	Point	Set fractional line widths
* PostScriptBegin	190	0	NIL	Set driver state to PostScript
* PostScriptEnd	191	0	NIL	Restore QuickDraw state
* PostScriptHandle	192	-	PSData	PostScript data in handle
*† PostScriptFile	193	-	FileName	FileName in data handle
* TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
*† ResourcePS	195	8	Type/ID/Index	PostScript data in a resource file
**RotateBegin	200	4	TRotation	Begin rotated port
**RotateEnd	201	0	NIL	End rotation
**RotateCenter	202	8	Center	Offset to center of rotation
**FormsPrinting	210	0	NIL	Don't clear print buffer after each page
**EndFormsPrinting	211	0	NIL	End forms printing after PrClosePage

* These comments are only implemented in LaserWriter driver 3.0 or later.
** These comments are only implemented in LaserWriter driver 3.1 or later.
† These comments are not available when background printing is enabled.

Each of these comments are discussed below in six groups: Text, Polygons, Lines, PostScript, Rotation, and Forms. Code examples are given where appropriate. For other examples of how to use picture comments for printing please see the Print example program in the Software Supplement (currently available through APDA as “Macintosh Example Applications and Sources 1.0”).

Note: The examples used in the *LaserWriter Reference Manual* are incorrect. Please use the examples presented here instead.

Text

In order to support the What-You-See-Is-What-You-Get paradigm, the LaserWriter driver uses a line layout algorithm to assure that the placement of the line on the printer closely approximates the placement of the line on the screen. This means that the printer driver gets the width of the line from QuickDraw, then tells PostScript to place the text in exactly the same place with the same width.

The `TextBegin` comment allows the application to specify the layout and the orientation of the text that follows it by specifying the following information:

```
TTxtPicRec = PACKED RECORD
  tJus:  Byte;     {0,1,2,3,4 or greater => none, left, center, right, full
                   justification }
  tFlip: Byte;     {0,1,2 => none, horizontal, vertical coordinate flip }
  tRot:  INTEGER;  {0..360 => clockwise rotation in degrees }
  tLine: Byte;     {1,2,3.. => single, 1-1/2, double.. spacing }
  tCmnt: Byte;     {Reserved }
END; { TTxtPicRec }
```

Left, right or center justification, specified by `tJust`, tells the driver to maintain only the left, right or center point, without recalculating the interword spacing. Full justification specifies that both endpoints be maintained and interword spacing be recalculated. This means that the driver makes sure that the specified points are maintained on the printer without caring whether the overall width has changed. Full justification means that the overall width of the line has been maintained. `tFlip` and `tRot` specify the orientation of the text, allowing the application to take advantage of the rotation features of PostScript. `tLine` specifies the interline spacing. When no `TextBegin` comment is used, the defaults are full justification, no rotation and single-spaced lines.

String Reconstruction

The `StringBegin` and `StringEnd` comments are used to bracket short strings of text that are actually sections of an original long string. MacDraw, for instance, breaks long strings into shorter pieces to avoid stack overflow problems with QuickDraw in the 64K ROM. When these smaller strings are bracketed by `StringBegin` and `StringEnd`, the LaserWriter driver assumes that the enclosed strings are parts of one long string and will perform its line layout accordingly. Erasing or filling of background rectangles should take place before the `StringBegin` comment to avoid confusing the process of putting the smaller strings back together.

Text Rotation

In order to rotate a text object, PostScript needs to have information concerning the center of rotation. The `TextCenter` comment provides this information when a rotation is specified in the `TextBegin` comment. This comment contains the offset from the present pen location to the center of rotation. The offset is given as the y-component, then the x-component, which are declared as fixed-point numbers. This allows the center to be in the middle of a pixel. This comment should appear after the `TextBegin` comment and before the first following `StringBegin` comment.

The associated comment data looks like this:

```
TTxtCenter = RECORD
   y,x: Fixed;       {offset from current pen location to center of rotation}
END; { TTxtCenter }
```

Right after a `TextBegin` comment, the LaserWriter driver expects to see a `TextCenter` comment specifying the center of rotation for any text enclosed within the text comment calls. It will ignore all further `CopyBits` calls, and print all standard text calls in the rotation specified by the information in `TTxtPicRec`. The center of rotation is the offset from the beginning position of the first string following the `TextCenter` comment. The printer driver also expects the string locations to be in the coordinate system of the current QuickDraw port. The printer driver rotates the entire port to draw the text so it can draw several strings with one rotation comment and one center comment. It is good practice to enclose an entire paragraph or paragraphs of text in a single rotation comment so that the driver makes the fewest number of rotations.

The printer driver can draw non-textual objects within the bounds of the text rotation comments but it must unrotate to draw the object, then re-rotate to draw the next string of text. To do this the printer driver must receive another `TextCenter` comment before each new rotation. So, rotated text and unrotated objects can be drawn inter-mixed within one `TextBegin/TextEnd` comment pair, but performance is slowed.

Note that all bit maps and all clip regions are ignored during text rotation so that clip regions can be used to clip out the strings on printers that can't take advantage of these comments. This has the unfortunate side effect of not allowing rotated text to be clipped.

Rotated text comments are not associated with landscape and portrait orientation of the printer paper as selected by the Page Setup dialog. These are rotations with reference to the current QuickDraw port only.

All of the above text comments are terminated by a `TextEnd` comment.

Turning Off Line Layout

If your application is using its own line layout algorithm (it uses its own character widths or does its own character or word placement), the printer driver doesn't need to do it too. To turn off line layout, you can use the `LineLayoutOff` comment. `LineLayoutOn` turns it on again.

Turning on `FractEnable` for the 128K ROMs has the same effect as `LineLayoutOff`. When the driver detects that `FractEnable` has been turned on, line layout is not performed. The driver assumes that all text being printed is already spaced correctly for the LaserWriter and just sends it as is.

Polygons

The polygon comments are recognized by the LaserWriter driver because they are used by MacDraw as an alternate method of defining polygons.

The PolyBegin and PolyEnd comments bracket polygon line segments, giving an alternate way to specify a polygon. All StdLine calls between these two comments are part of the polygon. The endpoints of the lines are the vertices of the polygon.

The picPlyClo comment specifies that the current polygon should be closed. This comes immediately after PolyBegin, if at all. It is not sufficient to simply check for begPt = endPt, since MacDraw allows you to create a "closed" polygon that isn't really closed. This comment is especially critical for smooth curves because it can make the difference between having a sharp corner or not in the curve.

These comments also work with the StdPoly call. If a FillRgn is encountered before the PolyEnd comment, then the polygon is filled. Unlike QuickDraw polygons, comment polygons do not require an initial MoveTo call within the scope of the polygon comment. The polygon will be drawn using the current pen location at the time the polygon comment is received. The pen must be set before the polygon comment is called.

Splines

A spline is a method used to determine the smallest number of points that define a curve. In MacDraw, splines are used as a method for smoothing polygons. The vertices of the underlying unsmoothed polygon are the control nodes for the quadratic B-spline curve which is drawn. PostScript has a direct facility for cubic B-splines and the LaserWriter translates the quadratic B-spline nodes it gets into the appropriate nodes for a cubic B-spline that will exactly emulate the original quadratic B-spline.

The PolySmooth comment specifies that the current polygon should be smoothed. This comment also contains data that provides a means of specifying which verbs to use on the smoothed polygon (bits 7 through 3 are not currently assigned):

```
TPolyVerb = PACKED RECORD
   f7, f6, f5, f4, f3, fPolyClose, fPolyFill, fPolyframe : Boolean;
END; { TPolyVerb }
```

Although the closing information is redundant with the picPlyClo comment, it is included for the convenience of the LaserWriter.

The LaserWriter uses the pen size at the time the PolyBegin comment is received to frame the smoothed polygon if framing is called for by the TPolyVerb information. When the PolyIgnore comment is received by the LaserWriter driver, all further StdLine calls are ignored until the PolyEnd comment is encountered. For polygons that are to be smoothed, set the initial pen width to zero after the PolyBegin comment so that the unsmoothed polygon will not be drawn by other printers not equipped to handle polygon comments. To fill the polygon, call StdRgn with the fill verb and the appropriate pattern set, as well as specifying fill in the PolySmooth comment.

Lines

The DashedLine and DashedLineStop comments are used to communicate PostScript information for drawing dashed lines.

The DashedLine comment contains the following additional data:

```
TDashedLine = PACKED RECORD
  offset:   SignedByte;                {Offset as specified by PostScript}
  centered: SignedByte;                {Whether dashed line should be
                                        centered to begin and end points}
  dashed:   Array[0..1] of SignedByte; {1st byte is # bytes following}
END; { TDashedLine }
```

The printer driver sets up the PostScript dashed line command, as defined on page 214 of Adobe's *PostScript Language Reference Manual*, using the parameters specified in the comment. You can specify that the dashed line be centered between the begin and end points of the lines by making the centered field nonzero.

The SetLineWidth comment allows you to set the pen width of all subsequent objects drawn. The additional data is a point. The vertical portion of the point is the numerator and the horizontal portion is the denominator of the scaling factor that the horizontal and vertical components of the pen are then multiplied by to obtain the new pen width. For example, if you have a pen size of 1,2 and in your line width comment you use 2 for the horizontal of the point and 7 for the vertical, the pen size will then be (7/2)*1 pixels wide and (7/2)*2 pixels high.

Below is an example of how to use the line comments:

```
PROCEDURE LineTest;
{This procedure shows how to do dashed lines and how to change the line width}
CONST
  DashedLine = 180;
  DashedStop = 181;
  SetLineWidth = 182;

TYPE
  DashedHdl = ^DashedPtr;
  DashedPtr = ^TDashedLine;
  TDashedLine = PACKED RECORD
    offset: SignedByte;
    Centered: SignedByte;
    dashed: Array[0..1] of SignedByte;   { the 0th element is the length }
  END; { TDashedLine }
  widhdl = ^widptr;
  widptr = ^widpt;
  widpt = Point;


VAR
  arect     : rect;
  Width     : Widhdl;
  dashedln : DashedHdl;
```

```
 BEGIN {LineTest}
   Dashedln := dashedhdl(NewHandle(sizeof(tdashedline)));
   Dashedln^^.offset := 0;         { No offset}
   Dashedln^^.centered := 0;       { don't center}
   Dashedln^^.dashed[0] := 1;      { this is the length }
   Dashedln^^.dashed[1] := 8;      { this means 8 points on, 8 points off }

   Width := widhdl(NewHandle(sizeof(widpt)));
   Width^^.h := 2;                 { denominator is 2}
   Width^^.v := 7;                 { numerator is 7}

   myPic := OpenPicture(theWorld);
     SetPen(1,2);                  { Set the pen size to 1 wide x 2 high }
     ClipRect(theWorld);
     MoveTo(20,20);
     DrawString('Do line test');
     PicComment(DashedLine,GetHandleSize(Handle(dashedln)),Handle(dashedln));
     PicComment(SetLineWidth,4,Handle(width));    {SetLineWidth}
     SetRect(arect,100,100,500,500);
     FrameRect(aRect);
     MoveTo(500,500);
     Lineto(100,100);
     PicComment(DashedStop,0,nil);              {DashedStop}
  ClosePicture;
  DisposHandle(handle(width));                  {Clean up}
  DisposHandle(handle(dashedln));
  PrintThePicture;                              {print it please}
  KillPicture(MyPic);
END; {LineTest}
```

PostScript

The PostScript comments tell the printer driver that the application is going to be communicating with the LaserWriter directly using PostScript commands instead of QuickDraw. The driver sends the accompanying PostScript to the printer with no preprocessing and no error checking. The application can specify data in the comment handle itself or point to another file which contains text to send to the printer. When the application is finished sending PostScript, the `PostScriptEnd` comment tells the printer driver to resume normal QuickDraw mode.

Any Quickdraw drawing commands made by the application between the `PostScriptBegin` and `PostScriptEnd` comments will be ignored by PostScript printers. In order to use PostScript in a device independent way, you should always include two representations of your document. The first representation should be a series of Quickdraw drawing commands. The second representation of your document should be a series of PostScript commands, sent to the Printing Manager via picture comments. This way, when you are printing to a PostScript device, the picture comments will be executed, and the Quickdraw commands ignored. When printing to a non-PostScript device, the picture comments will be ignored, and the Quickdraw commands will be executed. This method allows you to use PostScript, without having to ask the device if it supports it. This allows your application to get the best results with any printer, without being device dependent.

Here are some guidelines you need to remember:

- The graphic state set up during QuickDraw calls is maintained and is not affected by PostScript calls made with these comments.

- The header has changed a number of parameters so sometimes you won't get the results you expect. You may want to take a look at the header listed in *The LaserWriter Reference Manual* available through APDA.

- The header changes the PostScript coordinate system so that the origin is at the top-left corner of the page instead of at the bottom-left corner. This is done so that the QuickDraw coordinates that are used don't have to be remapped into the standard PostScript coordinate system. If you don't allow for this, all drawing is printed upside down. Please see the *PostScript Language Reference Manual* for details about transformation matrices.

- Don't call `showpage`. This is done for you by the driver. If you do, you won't be able to switch back to QuickDraw mode and an additional page will be printed when you call `PrClosePage`.
- Don't call `exitserver`. You may get very strange results.
- Don't call `initgraphics`. Graphics states are already set up by the header.

- Don't do anything that you expect to live across jobs.

- You won't be able to interrogate the printer to get information back through the driver.

The PostScriptBegin comment sets the driver state to prepare for the generation of PostScript by the application by calling gsave to save the current state. PostScript is then sent to the printer by using comments 192 through 195. The QuickDraw state of the driver is then restored by the PostScriptEnd comment. All QuickDraw operations that occur outside of these comments are performed; no clipping occurs as with the text rotation comments.

PostScript From a Text Handle

When the PostScriptHandle comment is used, the handle PSData points to the PostScript commands which are sent. PSData is a generic handle that points to text, without a length byte. The text is terminated by a carriage return. This comment is terminated by a PostScriptEnd comment.

Note: Due to a bug in the 3.1 LaserWriter driver, PostScriptEnd will not restore the QuickDraw state after the use of a PostScriptHandle comment. The workaround is to only use this comment at the end of your drawing, after you have made all the QuickDraw calls you need. This problem is fixed in more recent versions of the driver.

Here's an example of how to use this comment:

```
PROCEDURE PostHdl;
{this procedure shows how to use PostScript from a text Handle}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  PostScriptHandle = 192;

VAR
  MyString   : Str255;
  tempstr    : String[1];
  MyHandle   : Handle;
  err        : OSErr;

BEGIN { PostHdl }
  MyString := '/Times-Roman findfont 12 scalefont setfont 230 600 moveto
               (Hello World) show';
  tempstr:=' ';
  tempstr[1] := chr(13); {has to be terminated by a carriage return }
  MyString := Concat(MyString, tempstr); { in order for it to execute}
  err := PtrToHand (Pointer(ord(@myString)+1), MyHandle, length(MyString));
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('PostScript from a Handle');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(PostScriptHandle,length(mystring),MyHandle);
    PicComment(PostScriptEnd,0,nil);        {PostScript End}
  ClosePicture;
  DisposHandle(MyHandle);                   {Clean up}
  PrintThePicture;                          {print it please}
  KillPicture(MyPic);
END; { PostHdl }
```

Defining PostScript as QuickDraw Text

All QuickDraw text following the `TextIsPostScript` comment is sent as PostScript. No error checking is performed. This comment is terminated by a `PostScriptEnd` comment.

Here is an example:

```
PROCEDURE PostText;
{Shows how to use PostScript in strings in a QuickDraw picture}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  TextIsPostScript = 194;

BEGIN { PostTest }
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('TextIsPostScript Comment');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(TextIsPostScript,0,nil);     {following text is PostScript}
      DrawString('0 728 translate');        {move the origin and rotate the}
      DrawString('1 -1 scale');             {coordinate system}

      DrawString('newpath');
      DrawString('100 470 moveto');
      DrawString('500 470 lineto');
      DrawString('100 330 moveto');
      DrawString('500 330 lineto');
      DrawString('230 600 moveto');
      DrawString('230 200 lineto');
      DrawString('370 600 moveto');
      DrawString('370 200 lineto');
      DrawString('10 setlinewidth');
      DrawString('stroke');
      DrawString('/Times-Roman findfont 12 scalefont setfont');
      DrawString('230 600 moveto');
      DrawString('(Hello World) show');
    PicComment(PostScriptEnd,0,nil);        {PostScriptEnd}
  ClosePicture;
  PrintThePicture;                                 {print it please}
  KillPicture(MyPic);
END; { PostText }
```

PostScript From a File

The `PostScriptFile` and `ResourcePS` comments allow you to send PostScript to the printer from a resource file. Before these comments are described there are some restrictions you need to follow:

- Don't ever copy a picture containing these comments to the clipboard. If it is pasted into another application and the specified file or resource is not available, printing will be aborted and the user won't know what went wrong. This could be very confusing to a user. If you want the PostScript information to be available when printed from another application, use one of the other comments and include the information in the picture.
- Don't keep the PostScript in a separate file from the actual data file. If the data file ever gets moved without the PostScript file, when the picture is printed the data file may not be found and the print job will be aborted, again without the user knowing what went wrong. Keeping the data and PostScript in the same file will forestall many headaches for you and the user.

Now, a description of the comments:

The `PostScriptFile` comment tells the driver to use the POST type resources contained in the file `FileNameString`. `FileNameString` is declared as a `Str255`.

When this comment is encountered, the driver calls `OpenResFile` using the file name specified in `FileNameString`. It then calls `GetResource('POST',theID);` repeatedly, where `theID` begins at 501 and is incremented by one for each `GetResource` call. If the driver gets a `ResNotFound` error, it closes the specified resource file. If the first byte of the resource is a 3, 4, or 5 then the remaining data is sent and the file is closed.

The format of the POST resource is as follows: The IDs of the resources start at 501 and are incremented by one for each resource. Each resource begins with a 2 byte data field containing the data type in the first byte and a zero in the second. The possible values for the first byte are:

0	ignore the rest of this resource (a comment)
1	data is ASCII text
2	data is binary and is first converted to ASCII before being sent
3	AppleTalk end of file. The rest of the data, if there is any, is interpreted as ASCII text and will be sent after the EOF.
4	open the data fork of the current resource file and send the ASCII text there
5	end of the resource file

The second byte of the field must always be zero. Resources should be kept small, around 2K. Text and binary should not be mixed in the same resource. Make sure you include either a space or a return at the end of each PostScript string to separate it from the following command.

Here's an example:

```
PROCEDURE PostFile;
{This procedure shows how to use PostScript from a specified FILE}
CONST
  PostScriptBegin = 190;
  PostScriptFile = 193;
  PostScriptEnd = 191;

VAR
  MyString          : Str255;
  MyHandle          : Handle;
  err               : OSErr;

BEGIN   { PostFile }
  {You should never do this in a real program. This is only a test.}
  MyString := 'HardDisk:MPW:Print Examples:PSTestDoc';
  err := PtrToHand(pointer(MyString),MyHandle,length(MyString) + 1);
  MyPic := OpenPicture(theWorld);
  ClipRect(theWorld);
  MoveTo(20,20);
  DrawString('PostScriptFile Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(PostScriptFile,GetHandleSize(MyHandle),MyHandle);
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  MoveTo(50,50);
  DrawString('PostScriptEnd has terminated');
  ClosePicture;
  DisposHandle(MyHandle); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;    { PostFile }
```

Here are the resources:

```
type 'POST' {
  switch {
    case Comment:          /* this is a comment */
         key bitstring[8] = 0;
         fill byte;
         string;

    case ASCII:            /* this is just ASCII text */
         key bitstring[8] = 1;
         fill byte;
         string;

    case Bin:              /* this is binary */
         key bitstring[8] = 2;
         fill byte;
         string;

    case ATEOF:            /* this is an AppleTalk EOF */
         key bitstring[8] = 3;
         fill byte;
         string;
```

```
            case DataFork:          /* send the text in the data fork */
                key bitstring[8] = 4;
                fill byte;

            case EOF:               /* no more */
                key bitstring[8] = 5;
                fill byte;
            };
        };

        resource 'POST' (501) {
        ASCII{"0 728 translate "}};

        resource 'POST' (502) {
        ASCII{"1 -1 scale "}};

        resource 'POST' (503) {
        ASCII{"newpath "}};

        resource 'POST' (504) {
        ASCII{"100 470 moveto "}};

        resource 'POST' (505) {
        ASCII{"500 470 lineto "}};

        resource 'POST' (506) {
        ASCII{"100 330 moveto "}};

        resource 'POST' (507) {
        ASCII{"500 330 lineto "}};

        resource 'POST' (508) {
        ASCII{"230 600 moveto "}};

        resource 'POST' (509) {
        ASCII{"230 200 lineto "}};

        resource 'POST' (510) {
        ASCII{"370 600 moveto "}};

        resource 'POST' (511) {
        ASCII{"370 200 lineto "}};

        resource 'POST' (512) {
        ASCII{"10 setlinewidth "}};

        resource 'POST' (513) {
        ASCII{"stroke "}};

        resource 'POST' (514) {
        ASCII{"/Times-Roman findfont 12 scalefont setfont "}};

        resource 'POST' (515) {
        ASCII{"230 600 moveto "}};

        resource 'POST' (516) {
        ASCII{"(Hello World) show "}};
```

```
/* It will stop reading and close the file after 517 */
 resource 'POST' (517) {
 EOF
 {}};

/* it never gets here */
 resource 'POST' (518) {
 DataFork
 {}};
```

When the ResourcePS comment is encountered, the LaserWriter driver sends the text contained in the specified resource as PostScript to the printer. The additional data is defined as

```
    PSRsrc = RECORD
                PSType : ResType;
                PSID   : INTEGER;
                PSIndex: INTEGER;
             END;
```

The resource can be of type STR or STR#. If the Type is STR then the index should be 0. Otherwise an index should be given.

This comment is essentially the same as the PrintF control call to the driver. The imbedded command string it uses is '^r^n', which basically tells the driver to send the string specified by the additional data, then send a newline. For more information about printer control calls see the *LaserWriter Reference Manual.*

Here's an example:

```
PROCEDURE PostRSRC;
{This procedure shows how to get PostScript from a resource FILE}
  CONST
    PostScriptBegin = 190;
    PostScriptEnd = 191;
    ResourcePS = 195;

  TYPE
    theRSRChdl = ^theRSRCptr;
    theRSRCptr = ^theRSRC;
    theRSRC = RECORD
       theType: ResType;
       theID: INTEGER;
       Index: INTEGER;
    END;

  VAR
    temp          : Rect;
    TheResource   : theRSRChdl;
    i,j           : INTEGER;
    myport        : GrafPtr;
    err           : INTEGER;
    atemp         : Boolean;
```

```
BEGIN           { PostRSRC }
  TheResource := theRSRChdl(NewHandle(SizeOf(theRSRC)));
  TheResource^^.theID := 500;
  TheResource^^.Index := 0;
  TheResource^^.theType := 'STR ';
  HLock(Handle(TheResource));
  MyPic := OpenPicture(theWorld);
  DrawString('ResourcePS Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(ResourcePS,8,Handle(TheResource)); {Send postscript}
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  ClosePicture;
  DisposHandle(Handle(TheResource)); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;            { PostRSRC }
```

Here's the resource:

```
resource 'STR ' (500)
{"0 728 translate 1 -1 scale newpath 100 470 moveto 500 470 lineto 100 330
moveto 500 330 lineto 230 600 moveto 230 200 lineto 370 600 moveto 370 200
lineto 10 setlinewidth stroke /Times-Roman findfont 12 scalefont setfont 230
600 moveto (Hello World) show"
};
```

Rotation

The concept of rotation doesn't apply to text alone. PostScript can rotate any object. The rotation comments work exactly like text rotation except that all objects drawn between the two comments are drawn in the rotated coordinate system specified by the center of rotation comment, not just text. Also, no clipping of `CopyBits` calls occurs. These comments only work on the 3.1 and newer LaserWriter drivers.

The `RotateBegin` comment tells the driver that the following objects will be drawn in a rotated plane. This comment contains the following data structure:

```
Rotation = RECORD
  Flip:  INTEGER; {0,1,2 => none, horizontal, vertical coordinate flip }
  Angle: INTEGER; {0..360 => clockwise rotation in degrees }
END; { Rotation }
```

When you are finished, the `RotateEnd` comment returns the coordinate system to normal, terminating the rotation.

The relative center of rotation is specified by the `RotateCenter` comment in exactly the same manner as the `TextCenter` comments. The difference, however, is that this comment **must** appear **before** the `RotateBegin` comment. The data structure of the accompanying handle is exactly like that for the `TextCenter` comment.

Here's an example of how to use rotation comments:

```
PROCEDURE Test;
{This procedure shows how to do rotations}
CONST
  RotateBegin = 200;
  RotateEnd = 201;
  RotateCenter = 202;

TYPE
    rothdl = ^rotptr;
    rotptr = ^trot;
    trot = RECORD
      flip : INTEGER;
      Angle : INTEGER;
    END; { trot }
    centhdl = ^centptr;
    centptr = ^cent;
    Cent = PACKED RECORD
             yInt: INTEGER;
             yFrac: INTEGER;
             xInt: INTEGER;
             xFrac: INTEGER;
    END; { Cent }

VAR
  arect    : Rect;
  rotation : rothdl;
  center   : centhdl;
```

```
BEGIN { Test }
  rotation := rothdl(NewHandle(sizeof(trot)));
  rotation^^.flip := 0;                                 {no flip}
  rotation^^.angle := 15;        {15 degree rotation}

  center := centhdl(NewHandle(sizeof(cent)));
  center^^.xInt := 50;                                  {center at 50,50}
  center^^.yInt := 50;
  center^^.xFrac := 0;                                  {no fractional part}
  center^^.yFrac := 0;

  myPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('Begin Rotation');

    {set the center of Rotation}
    PicComment(RotateCenter,GetHandleSize(Handle(center)),Handle(center));
    {Begin Rotation}
   PicComment(RotateBegin,GetHandleSize(Handle(rotation)),Handle(rotation));
    SetRect(arect,100,100,500,500);
    FrameRect(aRect);
    MoveTo(500,500);
    Lineto(100,100);
    PicComment(RotateEnd,0,nil);                {RotateEnd}
 ClosePicture;
 DisposHandle(handle(rotation));                {Clean up}
 DisposHandle(handle(center));
 PrintThePicture;                                                    {print
it please}
 KillPicture(MyPic);
END; { Test }
```

Forms

The two form printing comments allow you to prepare a template to use for printing. When the `FormsBegin` comment is used, the LaserWriter's buffer is not cleared after `PrClosePage`. This allows you to download a form then change it for each subsequent page, inserting the information you want. `FormsEnd` allows the buffer to be cleared at the next `PrClosePage`.

#92: The Appearance of Text

See also: The Printing Manager
The Font Manager
Technical Note #91—
Optimizing for the LaserWriter—Picture Comments

Written by: Ginger Jernigan November 15, 1986
Updated: March 1, 1988

This technical note describes why text doesn't always look the way you expect depending on the environment you are in.

There are a number of Macintosh text editing applications where layout is critical. Unfortunately, text on a newer machine sometimes prints differently than text on a 64K ROM Macintosh. Let's examine some differences you should expect and why.

The differences we will consider here are only differences in the layout of text lines (line layout), not differences in the appearance of fonts or the differences between different printers. Differences in line layout may affect the position of line, paragraph and page breaks. The four variables that can affect line layout are fonts, the printer driver, the font manager mode, and ROMs.

Fonts

Every font on a Macintosh contains its own table of widths which tells QuickDraw how wide characters are on the screen. For every style point size there is a separate table which may contain widths that vary from face to face and from point size to point size. Character widths can vary between point sizes of characters even in the same face. In other words, fonts on the screen are not necessarily linearly scalable.

Non-linearity is not normally a problem since most fonts are designed to be as close to linear as possible. A font face in 6 point has very nearly the same scaled widths of the same font face in 24 point (plus or minus round-off or truncation differences). QuickDraw, however, requires only one face of any particular font to be in the System file to use it in any point size. If only a 10 point face actually exists, QuickDraw may scale that face to 9, 18, 24 (or whatever point size) by performing a linear scale of the 10 point face.

This can cause problems. Suppose a document is created on one Macintosh containing a font that only exists in that System file in one point size, say 9 point. The document is then taken to another Macintosh with a System file containing that same font but only in 24 point. The document may, in fact, appear differently on the two screens, and when it is printed, will have line breaks (and thus paragraph and page breaks) occurring in different places simply because of the differences in character widths that exist between the 9 point and 24 point faces.

The Printer Driver

Even when the printer you are using has a much higher resolution than what the screen can show, printer drivers perform line layout to match the screen layout as closely as possible.

The line layout performed by printer drivers is limited to single lines of text and does not change line break positions within multiple lines. The driver supplies metric information to the application about the page size and printable area to allow the application to determine the best place to make line and page breaks.

Printer driver line layout does affect word spacing, character spacing and even word positioning within a line. This may affect the overall appearance of text, particularly when font substitutions are made or various forms of page or text scaling are involved. But print drivers NEVER change line, paragraph or page break positions from what the application or screen specified. This means that where line breaks appear on the screen, they will always appear in the same place on the printer regardless of how the line layout may affect the appearance within the line.

Operating System and ROMs

In this context, operating system refers to the ROM trap routines which handle fonts and QuickDraw. Changes have occurred between the ROMs in the handling of fonts. Fonts in the 64K ROMs contain width tables (as described above) which are limited to integer values. Several new tables, however, have been added to fonts for the newer ROMs. The newer ROMs add an optional global width table containing fractional or fixed point decimal values. In addition, there is another optional table containing fractional values which can be scaled for the entire range of point sizes for any one face. There is also an optional table which provides for the addition (or removal) of width to a font when its style is changed to another value such as bold, outline or condensed. It is also possible, under the 128K ROMs, to add fonts to the system with inherent style properties containing their own width tables that produce different character widths from derived style widths.

One or all of the above tables may or may not be invoked depending on, first, their presence, and second, the mode of the operating system. The Font Manager in the newer ROMs allows the application to arbitrarily operate in either the fractional mode or integer mode (determined, in most cases, by the setting of `FractEnable`) as it chooses, with the default being integer. There is one case where fractional widths will be used if they exist even though fractional mode is disabled. When `FScaleDisable` is used fractional widths are always used if they exist regardless of the setting of `FractEnable`.

Differences in line layout (and thus line breaks) may be affected by any combination of the presence or absence of the optional tables, and the operating mode, either fractional or integer, of the application. Any of the combinations can produce different results from the original ROMs (and from each other).

The integer mode on the newer ROMs is very similar to, but not exactly the same as, the original 64K ROMs. When fonts with the optional tables present are used on Macintoshes with 64K ROMs, they continue to work in the old way with the integer widths. However, on newer ROMs, even in the integer mode, there may be variations in line width from what is seen on the old ROMs. In the plain text style there is very little if any difference (except if the global width table is present), but as various type styles are selected, line widths may vary more between ROMs.

Variations in the above options, by far, account for the greatest variation in the appearance of lines when a document is transported between one Macintosh and another. Line breaks may change position when documents created on one system (say a Macintosh) are moved to another system (like a Macintosh Plus). Variations are more pronounced as the number and sizes of various type styles increase within a document.

In all cases, however, a printer driver will produce exactly the same line breaks as appear on the screen with any given system combination.

Macintosh Technical Notes

#93: MPW: {$LOAD};_DataInit;%_MethTables

See also: MPW Reference Manuals

Written by: Jim Friedlander November 15, 1986
Modified by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note discusses the Pascal `{$LOAD}` directive as well as how to unload the `_DataInit` and `%_MethTables` segments.

{$LOAD}

MPW Pascal has a `{$LOAD}` directive that can dramatically speed up compiles.

```
{$LOAD HD:MPW:PLibraries:PasSymDump}
```

will combine symbol tables of all units following this directive (until another `{$LOAD}` directive is encountered), and dump them out to `HD:MPW:PLibraries:PasSymDump`. In order to avoid using fully specified pathnames, you can use `{$LOAD}` in conjunction with the `-k` option for Pascal:

```
Pascal -k "{PLibraries}" myfile
```

combined with the following lines in `myfile`:

```
USES
    {$LOAD PasSymDump}
        MemTypes,QuickDraw, OSIntf, ToolIntf, PackIntf,
    {$LOAD} {This "turns off" $LOAD for the next unit}
        NonOptimized,
    {$LOAD MyLibDump}
        MyLib;
```

will do the following: the first time a program containing these lines is compiled, two symbol table dump files (in this case `PasSymDump` and `MyLibDump`) will be created in the directory specified by the `-k` option (in this case `{PLibraries}`). No dump file will be generated for the unit `NonOptimized`. The compiler will compile `MemTypes`, `QuickDraw`, `OSIntf`, `ToolIntf`, `PackIntf` (quite time consuming) and dump those units' symbols to `PasSymDump` and it will compile the interface to `MyLib` and dump its symbols to `MyLib`. For subsequent compiles of this program (or any program that uses the same dump file(s)), the interface files won't be recompiled, the compiler will simply read in the symbol table.

Compiling a sample five line program on a Macintosh Plus/HD20SC takes 62 seconds without using the {$LOAD} directive. The same program takes 10 seconds to compile using the {$LOAD} directive (once the dump file exists). For further details about this topic, please see the *MPW Pascal Reference Manual*.

Note: If any of the units that are dumped into a dump file change, you need to make sure that the dump file is deleted, so that it can be regenerated by the Pascal compiler with the correct information. The best way to do this is to use a makefile to check the dump file against the files it depends on, and delete the dump file if it is out of date with respect to any of the units that it contains. An excellent (and well commented) example of doing this is in the *MPW Workshop Manual.*

The _DataInit Segment

The Linker will generate a segment whose resource name is %A5Init for any program compiled by the C or Pascal compilers. This segment is called by a program's main segment. This segment is loaded into the application heap and locked in place. It is up to your program to unload this segment (otherwise, it will remain locked in memory, possibly causing heap fragmentation). To do this from Pascal, use the following lines:

```
PROCEDURE _DataInit;EXTERNAL;
...

BEGIN          {main PROGRAM}
UnloadSeg(@_DataInit);
{remove data initialization code before any allocations}
...
```

From C, use the following lines:

```
extern _DataInit();
...
{ /* main */
              UnloadSeg(_DataInit);
              /*remove data initialization code before any allocations*/
...
```

For further details about Data Initialization, see the *MPW Reference Manual.*

%_MethTables and %_SelProcs

Object use in Pascal produces two segments which can cause heap problems. These are %_MethTables and %_SelProcs which are used when method calls are made. MacApp deals with them correctly, so this only applies to Object Pascal programs that don't use MacApp. You can make the segments locked and preloaded (probably the easiest route), so they will be loaded low in the heap, or you can unload them temporarily while you are doing heap initialization. In the latter case, make sure there are no method calls while they are unloaded. To reload %_MethTables and %_SelProcs, call the dummy procedure %_InitObj. %_InitObj loads %MethTables —calling any method will then load %_SelProcs.

Reminder: The linker is case sensitive when dealing with module names. Pascal converts all module names to upper-case (unless a routine is declared to be a C routine). The Assembler default is the same as the Pascal default, though it can be changed with the CASE directive. C preserves the case of module names (unless a routine is declared to be pascal, in which case the module name is converted to upper-case letters).

Make sure that any external routines that you reference are capitalized the same in both the external routine and the external declaration (especially in C). If the capitalization differs, you will get the following link error (library routine = findme, program declaration = extern FindMe();):

```
### Link: Error  Undefined entry, name: FindMe
```

#94: Tags

See also: The File Manager

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

Apple has decided to eliminate support for file-system tags on its future products; this technical note explains this decision.

Some of Apple's disk products (and some third-party products) have the ability to store 532 bytes per sector, instead of the normal 512. Twelve of the extra bytes are used to store redundant file system information, known as "tags", to be used by a scavenging utility to reconstruct damaged disks.

Apple has decided to eliminate support for these tags on its products; this was decided for several reasons:

1) Tags were implemented back when we had to deal with "Twiggy" drives on Lisa. These drives were less reliable than current drives, and it was expected that tags would be needed for data integrity.

2) We're working on a scavenging utility (Disk First Aid), and we've found that tags don't help us in reconstructing damaged disks (ie, if we can't fix it without using tags, tags wouldn't help us fix it). So, at least the first two versions of our scavenging utility will not use tags, and a third version (which we've planned for, but will probably never implement) can probably work without them.

3) 532-byte-per-sector drives and controllers tend to cost more, even at Apple's volumes. Thus, the demise of tags saves us (and our customers) money. The Apple Hard Disk 20SC currently supports tags; this may not always be the case, however; we'll probably drop the large sectors when we run out of our current stock of drives.

The Hierarchical File System (HFS) documentation didn't talk about tags because the writer had no information available about how they worked under HFS. Because of this decision, it is unlikely that we'll ever have documentation on how to correctly implement them under HFS.

Macintosh Technical Notes

#95: How To Add Items to the Print Dialogs

See also: The Printing Manager
The Dialog Manager

Written by: Ginger Jernigan
Lew Rollins
November 15, 1986

Updated: March 1, 1988

This technical note discusses how to add your own items to the Printing Manager's dialogs.

When the Printing Manager was initially designed, great care was taken to make the interface to the printer drivers as generic as possible in order to allow applications to print without being device-specific. There are times, however, when this type of non-specific interface interferes with the flexibility of an application. An application may require additional information before printing which is not part of the general Printing Manager interface. This technical note describes a method that an application can use to add its own items to the existing style and job dialogs.

Before continuing, you need to be aware of some guidelines that will increase your chances of being compatible with the printing architecture in the future:

- Only add items to the dialogs as described in this technical note. Any other methods will decrease your chances of survival in the future.

- Do not change the position of any item in the current dialogs. This means don't delete items from the existing item list or add items in the middle. Add items **only at the end** of the list.

- Don't count on an item retaining its current position in the list. If you depend on the Draft button being a particular number in the ImageWriter's style dialog item list, and we change the Draft button's item number for some reason, your program may no longer function correctly.

- Don't use more than half the screen height for your items. Apple reserves the right to expand the items in the standard print dialogs to fill the top half of the screen.

- If you are adding lots of items to the dialogs (which may confuse users), you should consider having your own separate dialog in addition to the existing Printing Manager dialogs.

The Heart

Before we talk about how the dialogs work, you need to know this: at the heart of the printer dialogs is a little-known data structure partially documented in the MacPrint interface file. It's a record called TPrDlg and it looks like this:

```
TPrDlg = RECORD   {Print Dialog: The Dialog Stream object.}
   dlg        : DialogRecord;    {dialog window}
   pFltrProc  : ProcPtr;         {filter proc.}
   pItemProc  : ProcPtr;         {item evaluating proc.}
   hPrintUsr  : THPrint;         {user's print record.}
   fDoIt      : BOOLEAN;
   fDone      : BOOLEAN;
   lUser1     : LONGINT;         {four longs reserved by Apple}
   lUser2     : LONGINT;
   lUser3     : LONGINT;
   lUser4     : LONGINT;
   iNumFst    : INTEGER;         {numeric edit items for std filter}
   iNumLst    : INTEGER;
{... plus more stuff needed by the particular printing dialog.}
END;
TPPrDlg = ^TPrDlg;               {== a dialog ptr}
```

All of the information pertaining to a print dialog is kept in the TPrDlg record. This record will be referred to frequently in the discussion below.

How the Dialogs Work

When your application calls PrStlDialog and PrJobDialog, the printer driver actually calls a routine called PrDlgMain. This function is declared as follows:

```
FUNCTION PrDlgMain (hprint: THPrint; pDlgInit: ProcPtr): BOOLEAN;
```

PrDlgMain first calls the pDlgInit routine to set up the appropriate dialog (in Dlg), dialog hook (pItemProc) and dialog event filter (pFilterProc) in the TPrDlg record (shown above). For the job dialog, the address of PrJobInit is passed to PrDlgMain. For the style dialog, the address of PrStlInit is passed. These routines are declared as follows:

```
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
```

After the initialization routine sets up the TPrDlg record, PrDlgMain calls ShowWindow (the window is initially invisible), then it calls ModalDialog, using the dialog event filter pointed to by the pFltrProc field. When an item is hit, the routine pointed to by the pItemProc field is called and the items are handled appropriately. When the OK button is hit (this includes pressing Return or Enter) the print record is validated. The print record is not validated if the Cancel button is hit.

How to Add Your Own Items

To modify the print dialogs, you need to change the TPrDlg record before the dialog is drawn on the screen. You can add your own items to the item list, replace the addresses of the standard dialog hook and event filter with the addresses of your own routines and then let the dialog code continue on its merry way.

For example, to modify the job dialog, first call PrJobInit. PrJobInit will fill in the TPrDlg record for you and return a pointer to that record. Then call PrDlgMain directly, passing in the address of your own initialization function. The example code's initialization function adds items to the dialog item list, saves the address of the standard dialog hook (in our global variable prPItemProc) and puts the address of our dialog hook into the pItemProc field of the TPrDlg record. Please note that your dialog hook **must** call the standard dialog hook to handle all of the standard dialog's items.

Note: If you wish to have an event filter, handle it the same way that you do a dialog hook.

Now, here is an example (written in MPW Pascal) that modifies the job dialog. The same code works for the style dialog if you globally replace 'Job' with 'Stl'. Also included is a function (AppendDITL) provided by Lew Rollins (originally written in C, translated for this technical note to MPW Pascal) which demonstrates a method of adding items to the item list, placing them in an appropriate place, and expanding the dialog window's rectangle.

The MPW Pascal Example Program

```
PROGRAM ModifyDialogs;

 USES
  {$LOAD PasDump.dump}
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint;

 CONST
  MyDITL    = 256;
  MyDFirstBox  = 1;     {Item number of first box in my DITL}
  MyDSecondBox = 2;

 VAR
  PrtJobDialog: TPPrDlg;  { pointer to job dialog }
  hPrintRec  : THPrint;  { Handle to print record }
  FirstBoxValue,      { value of our first additional box }
  SecondBoxValue: Integer; { value of our second addtl. box }
  prFirstItem,       { save our first item here }
  prPItemProc : LongInt;  { we need to store the old itemProc here }
  itemType   : Integer;  { needed for GetDItem/SetDItem calls }
  itemH    : Handle;
  itemBox   : Rect;
  err     : OSErr;
  {-----------------------------------------------------------------------}

PROCEDURE _DataInit;
 EXTERNAL;
```

```
  {---------------------------------------------------------------------}

  PROCEDURE CallItemHandler(theDialog: DialogPtr; theItem: Integer; theProc:
LongInt);
    INLINE $205F,$4E90;    { MOVE.L (A7)+,A0
                              JSR (A0)          }

{ this code pops off theProc and then does a JSR to it, which puts the
 real return address on the stack. }

  {---------------------------------------------------------------------}

  FUNCTION AppendDITL(theDialog: DialogPtr; theDITLID: Integer): Integer;
  { version 0.1 9/11/86 Lew Rollins of Human-Systems Interface Group}
  { this routine still needs some error checking }

{ This routine appends all of the items of a specified DITL
onto the end of a specified DLOG — We don't even need to know the format
of the DLOG }

{ this will be done in 3 steps:
 1. append the items of the specified DITL onto the existing DLOG
 2. expand the original dialog window as required
 3. return the adjusted number of the first new user item
}
    TYPE
     DITLItem    = RECORD { First, a single item }
            itmHndl: Handle; { Handle or procedure pointer for this item }
            itmRect: Rect; { Display rectangle for this item }
            itmType: SignedByte; { Item type for this item — 1 byte }
            itmData: ARRAY [0..0] OF SignedByte; { Length byte of data }
          END;  {DITLItem}

     pDITLItem   = ^DITLItem;
     hDITLItem   = ^pDITLItem;

     ItemList    = RECORD { Then, the list of items }
            dlgMaxIndex: Integer; { Number of items minus 1 }
            DITLItems: ARRAY [0..0] OF DITLItem; { Array of items }
          END;  {ItemList}

     pItemList   = ^ItemList;
     hItemList   = ^pItemList;

     IntPtr    = ^Integer;

    VAR
     offset   : Point;  { Used to offset rectangles of items being appended }
     maxRect   : Rect;  { Used to track increases in window size }
     hDITL    : hItemList; { Handle to DITL being appended }
     pItem    : pDITLItem; { Pointer to current item being appended }
     hItems   : hItemList; { Handle to DLOG's item list }
     firstItem : Integer; { Number of where first item is to be appended }
     newItems,       { Count of new items }
     dataSize,       { Size of data for current item }
     i      : Integer; { Working index }
     USB    : RECORD  {we need this because itmData[0] is unsigned}
            CASE Integer OF
```

```
                1:
                 (SBArray: ARRAY [0..1] OF SignedByte);
                2:
                 (Int: Integer);
             END;  {USB}

   BEGIN          {AppendDITL}
 {
  Using the original DLOG

  1. Remember the original window Size.
  2. Set the offset Point to be the bottom of the original window.
  3. Subtract 5 pixels from bottom and right, to be added
     back later after we have possibly expanded window.
  4. Get working Handle to original item list.
  5. Calculate our first item number to be returned to caller.
  6. Get locked Handle to DITL to be appended.
  7. Calculate count of new items.
 }
    maxRect := DialogPeek(theDialog)^.window.port.portRect;
    offset.v := maxRect.bottom;
    offset.h := 0;
    maxRect.bottom := maxRect.bottom - 5;
    maxRect.right := maxRect.right - 5;
    hItems := hItemList(DialogPeek(theDialog)^.items);
    firstItem := hItems^^.dlgMaxIndex + 2;
    hDITL := hItemList(GetResource('DITL',theDITLID));
    HLock(Handle(hDITL));
    newItems := hDITL^^.dlgMaxIndex + 1;
 {
  For each item,
   1. Offset the rectangle to follow the original window.
   2. Make the original window larger if necessary.
   3. fill in item Handle according to type.
 }

    pItem := @hDITL^^.DITLItems;
    FOR i := 1 TO newItems DO BEGIN
     OffsetRect(pItem^.itmRect,offset.h,offset.v);
     UnionRect(pItem^.itmRect,maxRect,maxRect);

     USB.Int := 0;    {zero things out}
     USB.SBArray[1] := pItem^.itmData[0];

     { Strip enable bit since it doesn't matter here. }
     WITH pItem^ DO
      CASE BAND(itmType,$7F) OF
       userItem:  { Can't do anything meaningful with user items. }
        itmHndl := NIL;
       ctrlItem + btnCtrl,ctrlItem + chkCtrl,ctrlItem + radCtrl:{build Control }
        itmHndl := Handle(NewControl(theDialog, { theWindow }
                       itmRect, { boundsRect }
                       StringPtr(@itmData[0])^, { title }
                       true, { visible }
                       0,0,1, { value, min, max }
                       BAND(itmType,$03), { procID }
                       0)); { refCon }
       ctrlItem + resCtrl: BEGIN { Get resource based Control }
      itmHndl := Handle(GetNewControl(IntPtr(@itmData[1])^, { controlID }
                      theDialog)); { theWindow }
      ControlHandle(itmHndl)^^.contrlRect := itmRect; {give it the right
                                  rectangle}
      {An actionProc for a Control should be installed here}
     END;     {Case ctrlItem + resCtrl}
     statText,editText: { Both need Handle to a copy of their text. }
      err := PtrToHand(@itmData[1], { Start of data }
                itmHndl, { Address of new Handle }
                USB.Int); { Length of text }
     iconItem:  { Icon needs resource Handle. }
      pItem^.itmHndl := GetIcon(IntPtr(@itmData[1])^); { ICON resID }
     picItem:   { Picture needs resource Handle. }
      pItem^.itmHndl := Handle(GetPicture(IntPtr(@itmData[1])^));{PICT resID}
     OTHERWISE
      itmHndl := NIL;
    END;       {Case}

   dataSize := BAND(USB.Int + 1,$FFFE);
   {now advance to next item}
   pItem := pDITLItem(Ptr(ord4(@pItem^) + dataSize + sizeof(DITLItem)));
  END;        {for}
  err := PtrAndHand
   (@hDITL^^.DITLItems,Handle(hItems),GetHandleSize(Handle(hDITL)));
  hItems^^.dlgMaxIndex := hItems^^.dlgMaxIndex + newItems;
  HUnlock(Handle(hDITL));
  ReleaseResource(Handle(hDITL));
  maxRect.bottom := maxRect.bottom + 5;
  maxRect.right := maxRect.right + 5;
  SizeWindow(theDialog,maxRect.right,maxRect.bottom,true);
  AppendDITL := firstItem;
 END;          {AppendDITL}

{-----------------------------------------------------------------------}

PROCEDURE MyJobItems(theDialog: DialogPtr; itemNo: Integer);
{
This routine replaces the routine in the pItemProc field in the
TPPrDlg record.  The steps it takes are:
1. Check to see if the item hit was one of ours. This is done by "localizing"
   the number, assuming that our items are numbered from 0..n
2. If it's one of ours  then case it and Handle appropriately
3. If it isn't one of ours then call the old item handler
}

  VAR
   MyItem,firstItem: Integer;
   thePt    : Point;
   thePart   : Integer;
   theValue   : Integer;
   debugPart  : Integer;

  BEGIN          {MyJobItems}
   firstItem := prFirstItem; { remember, we saved this in myJobDlgInit }
   MyItem := itemNo - firstItem + 1; { "localize" current item No }
   IF MyItem > 0 THEN BEGIN { if localized item > 0, it's one of ours }
    { find out which of our items was hit }
    GetDItem(theDialog,itemNo,itemType,itemH,itemBox);
```

```
      CASE MyItem OF
        MyDFirstBox: BEGIN
         { invert value of FirstBoxValue and redraw it }
         FirstBoxValue := 1 - FirstBoxValue;
         SetCtlValue(ControlHandle(itemH),FirstBoxValue);
        END;      {case MyDFirstBox}
        MyDSecondBox: BEGIN
         { invert value of SecondBoxValue and redraw it }
         SecondBoxValue := 1 - SecondBoxValue;
         SetCtlValue(ControlHandle(itemH),SecondBoxValue);
        END;      {case MyDSecondBox}
        OTHERWISE
         Debug;    { OH OH - We got an item we didn't expect }
      END;        {Case}
     END         { if MyItem > 0 }
     ELSE       { chain to standard item handler, whose address is saved
                 in prPItemProc }
      CallItemHandler(theDialog,itemNo,prPItemProc);
   END;           { MyJobItems }

  {-----------------------------------------------------------------------}

  FUNCTION MyJobDlgInit(hPrint: THPrint): TPPrDlg;
 {
 This routine appends items to the standard job dialog and sets up the
 user fields of the printing dialog record TPRDlg
 This routine will be called by PrDlgMain
 This is what it does:
 1. First call PrJobInit to fill in the TPPrDlg record.
 2. Append our items onto the old DITL. Set them up appropriately.
 3. Save the address of the old item handler and replace it with ours.
 4. Return the Fixed dialog to PrDlgMain.
 }

   VAR
    firstItem : Integer; { first new item number }

   BEGIN          {MyJobDlgInit}
    firstItem := AppendDITL(DialogPtr(PrtJobDialog),MyDITL);

    prFirstItem := firstItem; { save this so MyJobItems can find it }

    { now we'll set up our DITL items - The "First Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),FirstBoxValue);

    { now we'll set up the second of our DITL items - The "Second Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem + 1,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),SecondBoxValue);

{ Now comes the part where we patch in our item handler.  We have to save
 the old item handler address, so we can call it if one of the standard
 items is hit, and put our item handler's address
 in pItemProc field of the TPrDlg struct}

    prPItemProc := LongInt(PrtJobDialog^.pItemProc);

    { Now we'll tell the modal item handler where our routine is }
   PrtJobDialog^.pItemProc := ProcPtr(@MyJobItems);

   { PrDlgMain expects a pointer to the modified dialog to be returned.... }
   MyJobDlgInit := PrtJobDialog;

  END;          {myJobDlgInit}

 {-----------------------------------------------------------------------}

 FUNCTION Print: OSErr;

  VAR
   bool    : BOOLEAN;

  BEGIN         {Print}
   hPrintRec := THPrint(NewHandle(sizeof(TPrint)));
   PrintDefault(hPrintRec);
   bool := PrValidate(hPrintRec);
   IF (PrError <> noErr) THEN BEGIN
    Print := PrError;
    Exit(Print);
   END;         {If}

   { call PrJobInit to get pointer to the invisible job dialog }
   PrtJobDialog := PrJobInit(hPrintRec);
   IF (PrError <> noErr) THEN BEGIN
    Print := PrError;
    Exit(Print);
   END;         {If}

 {Here's the line that does it all!}
   IF NOT (PrDlgMain(hPrintRec,@MyJobDlgInit)) THEN BEGIN
    Print := cancel;
    Exit(Print);
   END;         {If}

   IF PrError <> noErr THEN Print := PrError;

   { that's all for now }

  END;          { Print }

 {-----------------------------------------------------------------------}

 BEGIN          {PROGRAM}

  UnloadSeg(@_DataInit);  {remove data initialization code before any
allocations}
  InitGraf(@thePort);
  InitFonts;
  FlushEvents(everyEvent,0);
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(NIL);
  InitCursor;

  { call the routine that does printing }
  FirstBoxValue := 0;     { value of our first additional box }
  SecondBoxValue := 0;    { value of our second addtl. box }
  PrOpen;     { Open the Print Manager }
  IF PrError = noErr THEN
   err := Print      { This actually brings up the modified Job dialog }
  ELSE BEGIN
   {tell the user that PrOpen failed}
  END;

  PrClose;     { Close the Print Manager and leave }
END.
```

The Lightspeed C Example Program

```
/* NOTE: Apple reserves the top half of the screen (where the current DITL
     items are located). Applications may use the bottom half of the
     screen to add items, but should not change any items in the top half
     of the screen.  An application should expand the print dialogs only
     as much as is absolutely necessary.
*/

/* Note: A global search and replace of 'Job' with 'Stl' will produce
     code that modifies the style dialogs */
#include <DialogMgr.h>
#include <MacTypes.h>
#include <Quickdraw.h>
#include <ResourceMgr.h>
#include <WindowMgr.h>
#include <pascal.h>
#include <printmgr.h>
#define nil 0L


static TPPrDlg PrtJobDialog;        /* pointer to job dialog */

/*    This points to the following structure:

        struct {
              DialogRecord    Dlg;          (The Dialog window)
              ProcPtr         pFltrProc;    (The Filter Proc.)
              ProcPtr         pItemProc;    (The Item evaluating proc. --
                                            we'll change this)
              THPrint         hPrintUsr;    (The user's print record.)
              Boolean         fDoIt;
              Boolean         fDone;
                  (Four longs -- reserved by Apple Computer)
              long                   lUser1;
              long                   lUser2;
              long                   lUser3;
              long                   lUser4;
    } TPrDlg; *TPPrDlg;
*/



/*    Declare 'pascal' functions and procedures */
pascal Boolean PrDlgMain();         /* Print manager's dialog handler */
pascal TPPrDlg PrJobInit();         /* Gets standard print job dialog. */
pascal TPPrDlg MyJobDlgInit();      /* Our extention to PrJobInit */
pascal void MyJobItems();           /* Our modal item handler */

#define MyDITL 256                  /* resource ID of my DITL to be spliced
                                          on to job dialog */


THPrint hPrintRec;                  /* handle to print record */
short FirstBoxValue = 0;            /* value of our first additional box */
short SecondBoxValue = 0;           /* value of our second addtl. box */
long prFirstItem;                   /* save our first item here */
long prPItemProc;             /* we need to store the old itemProc here */
```

```
/*----------------------------------------------------------------------*/
      WindowPtr   MyWindow;
      OSErr       err;
      Str255      myStr;
main()
{
      Rect        myWRect;

      InitGraf(&thePort);
      InitFonts();
      InitWindows();
      InitMenus();
      InitDialogs(nil);
      InitCursor();
      SetRect(&myWRect,50,260,350,340);

      /* call the routine that does printing */
      PrOpen();
      err = Print();

      PrClose();
} /* main */

/*----------------------------------------------------------------------*
/

OSErr Print()

{
      /* call PrJobInit to get pointer to the invisible job dialog */
      hPrintRec = (THPrint)(NewHandle(sizeof(TPrint)));
      PrintDefault(hPrintRec);
      PrValidate(hPrintRec);
      if (PrError() != noErr)
            return PrError();

      PrtJobDialog = PrJobInit(hPrintRec);
      if (PrError() != noErr)
            return PrError();


      if (!PrDlgMain(hPrintRec, &MyJobDlgInit)) /* this line does all the
                                                 stuff */
            return Cancel;

      if (PrError() != noErr)
            return PrError();

/* that's all for now */

} /* Print */

/*----------------------------------------------------------------------*
/

pascal TPPrDlg MyJobDlgInit (hPrint)
THPrint hPrint;
```

```
/* this routine appends items to the standard job dialog and sets up the
     user fields of the printing dialog record TPRDlg
     This routine will be called by PrDlgMain */
{
     short          firstItem;          /* first new item number */

     short          itemType;           /* needed for GetDItem/SetDItem call */
     Handle         itemH;
     Rect           itemBox;

     firstItem = AppendDITL (PrtJobDialog, MyDITL); /*call routine to do
                                                       this */

     prFirstItem = firstItem; /* save this so MyJobItems can find it */

/* now we'll set up our DITL items -- The "First Box" */
     GetDItem(PrtJobDialog,firstItem,&itemType,&itemH,&itemBox);
     SetCtlValue(itemH,FirstBoxValue);

/* now we'll set up the second of our DITL items  -- The "Second Box" */
     GetDItem(PrtJobDialog,firstItem+1,&itemType,&itemH,&itemBox);
     SetCtlValue(itemH,SecondBoxValue);

/* Now comes the part where we patch in our item handler.  We have to save
     the old item handler address, so we can call it if one of the
     standard items is hit, and put our item handler's address
     in pItemProc field of the TPrDlg struct
*/

     prPItemProc = (long)PrtJobDialog->pItemProc;

/* Now we'll tell the modal item handler where our routine is */
     PrtJobDialog->pItemProc = (ProcPtr)&MyJobItems;

/* PrDlgMain expects a pointer to the modified dialog to be returned.... */
     return PrtJobDialog;

} /*myJobDlgInit*/


/*----------------------------------------------------------------------------*/


/* here's the analogue to the SF dialog hook */

pascal void MyJobItems(theDialog,itemNo)
TPPrDlg     theDialog;
short       itemNo;

{ /* MyJobItems */
     short          myItem;
     short          firstItem;

     short          itemType;           /* needed for GetDItem/SetDItem call */
     Handle         itemH;
     Rect           itemBox;

     firstItem = prFirstItem; /* remember, we saved this in myJobDlgInit
*/
```

```
    myItem = itemNo-firstItem+1;  /* "localize" current item No */
    if (myItem > 0)   /* if localized item > 0, it's one of ours */
    {
        /* find out which of our items was hit */
        GetDItem(theDialog,itemNo,&itemType,&itemH,&itemBox);
        switch (myItem)
        {
            case 1:
                /* invert value of FirstBoxValue and redraw it */
                FirstBoxValue ^= 1;
                SetCtlValue(itemH,FirstBoxValue);
                break;

            case 2:
                /* invert value of SecondBoxValue and redraw it */
                SecondBoxValue ^= 1;
                SetCtlValue(itemH,SecondBoxValue);
                break;
            default: Debugger(); /* OH OH */
        } /* switch */
    } /* if (myItem > 0) */
    else /* chain to standard item handler, whose address is saved in
            prPItemProc */
    {
        CallPascal(theDialog,itemNo,prPItemProc);
    }
} /* MyJobItems */
```

The Rez Source

```
#include "types.r"

resource 'DITL' (256) {
 { /* array DITLarray: 2 elements */
   /* [1] */
   {8, 0, 24, 112},
   CheckBox {
    enabled,
    "First Box"
   };
   /* [2] */
   {8, 175, 24, 287},
   CheckBox {
    enabled,
    "Second Box"
   }
 }
};
```

Macintosh Technical Notes

#96: SCSI Bugs

See also: The SCSI Manager
SCSI Developer's Package

Written by:	Steve Flowers	October 1, 1986
Modified by:	Bryan Stearns	November 15, 1986
Modified by:	Bo3b Johnson	July 1, 1987
Updated:		March 1, 1988

There are a number of problems in the SCSI Manager; this note lists the ones we know about, along with an explanation of what we're doing about them. Changes made for the 2/88 release are made to more accurately reflect the state of the SCSI Manager. System 4.1 and 4.2 are very similar; one bug was fixed in System 4.2.

There are several categories of SCSI Manager problems:

1. Those in the ROM boot code
(Before the System file has been opened, and hence, before any patches could possibly fix them.)
2. Those that have been fixed in System 3.2
3. Those that have been fixed in System 4.1/4.2
4. Those that are new in System 4.1/4.2
5. Those that have not yet been fixed.

The problems in the ROM boot code can only be fixed by changing the ROMs. Most of the bugs in the SCSI Manager itself have been fixed by the patch code in the System 3.2 file. There are a few problems, though, that are not fixed with System 3.2—most of these bugs have been corrected in System 4.1/4.2. Any that are not fixed will be detailed here. ROM code for future machines will, of course, include the corrections.

ROM boot code problems

- In the process of looking for a bootable SCSI device, the boot code issues a SCSI bus reset before each attempt to read block 0 from a device. If the read fails for any reason, the boot code goes on to the next device. SCSI devices which implement the `Unit Attention` condition as defined by the Revision 17B SCSI standard will fail to boot in this case. The read will fail because the drive is attempting to report the `Unit Attention` condition for the first command it receives after the SCSI bus reset. The boot code does not read the sense bytes and does not retry the failed command; it simply resets the SCSI bus and goes on to the next device.

If no other device is bootable, the boot code will eventually cycle back to the same SCSI device ID, reset the bus (causing `Unit Attention` in the drive again), and try to read block 0 (which fails for the same reason).

The 'new' Macintosh Plus ROMs that are included in the platinum Macintosh Plus have only one change. The change was to simply do a single SCSI Bus Reset after power up instead of a Reset each time through the SCSI boot loop. This was done to allow `Unit Attention` drives to be bootable. It was an object code patch (affecting approximately 30 bytes) and no other bugs were fixed. For details on the three versions of Macintosh Plus ROMs, see Technical Note #154.

We recommend that you choose an SCSI controller which does not require the `Unit Attention` feature—either an older controller (most of the SCSI controllers currently available were designed before Revision 17B), or one of the newer Revision-17B-compatible controllers which can enable/disable `Unit Attention` as a formatting option (such as those from Seagate, Rodime, et al). Since the vast majority of Macintosh Plus computers have the ROMs which cannot use `Unit Attention` drives, we still recommend that you choose an SCSI controller that does not require the `Unit Attention` feature.

- If an SCSI device goes into the `Status` phase after being selected by the boot code, this leads to the SCSI bus being left in the `Status` phase indefinitely, and no SCSI devices can be accessed. The current Macintosh Plus boot code does not handle this change to `Status` phase, which means that the presence of an SCSI device with this behavior (as in some tape controllers we've seen) will prevent any SCSI devices from being accessed by the SCSI Manager, even if they already had drivers loaded from them. The result is that any SCSI peripheral that is turned on at boot time must not go into `Status` phase immediately after selection; otherwise, the Macintosh Plus SCSI bus will be left hanging. Unless substantially revised ROMs are released for the Macintosh Plus (highly unlikely within the next year or so), this problem will never be fixed on the Macintosh Plus, so you should design for old ROMs.

- The Macintosh Plus would try to read 256 bytes of blocks 0 and 1, ignoring the extra data. The Macintosh SE and Macintosh II try to read 512 bytes from blocks 0 and 1, ignoring errors if the sector size is larger (but not smaller) than 512 bytes. Random access devices (disks, tapes, CD ROMS, etc.) can be booted as long as the blocks are at least 512 bytes, blocks 0, 1 and other partition blocks are correctly set up, and there is a driver on it. With the new partition layout (documented in *Inside Macintosh* volume V), more than 256 bytes per sector may be required in some partition map entries. This is why we dropped support for 256-byte sectors. Disks with tag bytes (532-byte sectors) or larger block sizes (1K, 2K, etc.) can be booted on any Macintosh with an SCSI port. Of course, the driver has to take care of data blocking and de-blocking, since HFS likes to work with 512-byte sectors.

Problems with ROM SCSI Manager routines

Note that the following problems are fixed after the System file has been opened; for a device to boot properly, it must not depend on these fixes. The sample SCSI driver, available from APDA, contains an example of how to find out if the fixes are in place.

- **Prior to System file 3.2,** blind transfers (both reads and writes) would not work properly with many SCSI controllers. Since blind operation depends on the drive's ability to transfer data fast enough, it is the responsibility of the driver writer to make sure blind operation is safe for a particular device.

- **Prior to System file 3.2,** the SCSI Manager dropped a byte when the driver did two or more `SCSIReads` or `SCSIRBlinds` in a row. (Each `Read` or `RBlind` has to have a Transfer Information Block (TIB) pointer passed in.) The TIB itself can be as big and complex as you want—it is the process of returning from one `SCSIRead` or `SCSIRBlind` and entering another one (while still on the same SCSI command) that causes the first byte for the other `SCSIReads` to be lost.

  Note that this precludes use of file-system tags. Apple no longer recommends that you support tags; see Technical Note #94 for more information.

- **Prior to System file 3.2,** `SCSIStat` didn't work; the new version works correctly.

- **Running under System file 3.2,** the SCSI Manager does not check to make sure that the last byte of a write operation (to the peripheral) was handshaked while operating in pseudo-DMA mode. The SCSI Manager writes the final byte to the NCR 5380's one-byte buffer and then turns pseudo-DMA mode off shortly thereafter (reported to be 10-15 microseconds). If the peripheral is somewhat slow in actually reading the last byte of data, it asserts `REQ` after the Macintosh has already turned off pseudo-DMA mode and never gets an `ACK`. The CPU then expects to go into the `Status` phase since it thinks everything went OK, but the peripheral is still waiting for `ACK`. Unless the driver can recover from this somehow, the SCSI bus is 'hung' in the `Data Out` phase. In this case, all successive SCSI Manager calls will fail until the bus is reset.

- **Running under System file 4.1/4.2,** the SCSI Manager waits for the last byte of a write operation to be handshaked while operating in pseudo-DMA mode; it checks for a final `DRQ` (or a phase change) at the end of a `SCSIWrite` or `SCSIWBlind` before turning off the pseudo-DMA mode. Drivers that could recover from this problem by writing the last byte again if the bus was still in a `Data Out` phase will still work correctly, as long as they were checking the bus state.

- **Running under System file 3.2,** the SCSI Manager does not time out if the peripheral fails to finish transferring the expected number of bytes for polled reads and writes. (Blind operation does poll for the first byte of each requested data transfer in the Transfer Information Block.)

- **Running under System file 4.1/4.2,** SCSIRead and SCSIWrite return an error to the caller if the peripheral changes the bus phase in the middle of a transfer, as might happen if the peripheral fails to transfer the expected number of bytes. The computer is no longer left in a hung state.

- **Running under System file 3.2,** the Selection timeout value is very short (900 microseconds). Patches to the SCSI Manager **in System 4.1/4.2** ensure that this value is the recommended 250 milliseconds.

- **Running under System file 3.2,** the SCSI Manager routine SCSIGet (which arbitrates for the bus) will fail if the BSY line is still asserted. Some devices are a bit slow in releasing BSY after the completion of an SCSI operation, meaning that BSY may not have been released before the driver issues a SCSIGet call to start the next SCSI operation. A work-around for this is to call SCSIGet again if it failed the first time. (Rarely has it been necessary to try it a third time.) This assumes, of course, that the bus has not been left 'hanging' by an improperly terminated SCSI operation before calling SCSIGet.

- **Running under System file 4.1/4.2,** the SCSIGet function has been made more tolerant of devices that are slow to release the BSY line after a SCSI operation. The SCSI Manager now waits up to 200 milliseconds before returning an error.

Problems with the SCSI Manager that haven't been fixed yet

These problems currently exist in the Macintosh Plus, SE, and II SCSI Manager. We plan to fix these problems in a future release of the System Tools disk, but in the mean time, you should try to work around the problems (but don't "require" the problems!).

- Multiple calls to SCSIRead or SCSIRBlind after issuing a command and before calling SCSIComplete may not work. Suppose you want to read some mode sense data from the drive. After sending the command with SCSICmd, you might want to call SCSIRead with a TIB that reads four bytes (typically a header). After reading the field (in the four-byte header) that tells how many remaining bytes are available, you might call SCSIRead again with a TIB to read the remaining bytes. The problem is that the first byte of the second SCSIRead data will be lost because of the way the SCSI Manager handles reads in pseudo-DMA mode. The work-around is to issue two separate SCSI commands: the first to read only the four-byte header, the second to read the four-byte header plus the remaining bytes. We recommend that you **not** use a clever TIB that contains two data transfers, the second of which gets the transfer length from the first transfer's received data (the header). These two step TIBs will not work in the future. This bug will probably not be fixed.

- On read operations, some devices may be slow in deasserting REQ after sending the last byte to the CPU. The current SCSI Manager (all machines) will return to the caller without waiting for REQ to be deasserted. Usually the next call that the driver would make is SCSIComplete. On the Macintosh SE and II, the SCSIComplete call will check the bus to be sure that it is in Status phase. If not, the SCSI Manager will return a new error code that indicates the bus was in Data In/Data Out phase when SCSIComplete was called. The combination of the speed of the Macintosh II and a

slow peripheral can cause `SCSIComplete` to detect that the bus is still in Data In phase before the peripheral has finally changed the bus to `Status` phase. This results in a false error being passed back by `SCSIComplete`.

- The `scComp` (compare) TIB opcode does not work in System 4.1 on the Macintosh Plus only. It returns an error code of 4 (bad parameters). This has been fixed in System 4.2.

Other SCSI Manager Issues

- At least one third-party SCSI peripheral driver used to issue SCSI commands from a VBL task. It didn't check to see if the bus was in the free state before sending the command! This is guaranteed to wipe out any other SCSI command that may have been in progress, since the SCSI Manager on the Macintosh Plus does not mask out (or use) interrupts.

  We strongly recommend that you avoid calling the SCSI Manager from interrupt handlers (such as VBL tasks). If you must send SCSI commands from a VBL task (like for a removable media system), do a `SCSIStat` call first to see if the bus is currently busy. If it's free (`BSY` is not asserted), then it's probably safe; otherwise the VBL task should not send the command. Note that you can't call `SCSIStat` before the System file fixes are in place. Since SCSI operations during VBL are not guaranteed, you should check all errors from SCSI Manager calls.

- A new SCSI Manager call will be added in the future. This will be a high-level call; it will have some kind of parameter block in which you give a pointer to a command buffer, a pointer to your TIB, a pointer to a sense data buffer (in case something goes wrong, the SCSI Manager will automatically read the sense bytes into the buffer for you), and a few other fields. The SCSI Manager will take care of arbitration, selection, sending the command, interpreting the TIB for the data transfer, and getting the status and message bytes (and the sense bytes, if there was an error). It should make SCSI device drivers much easier to write, since the driver will no longer have to worry about unexpected phase changes, getting the sense bytes, and so on. In the future, this will be the recommended way to use the SCSI Manager.

- The SCSI Manager (all machines) does not currently support interrupt-driven (asynchronous) operations. The Macintosh Plus can never support it since there is no interrupt capability, although a polled scheme may be implemented by the SCSI Manager. The Macintosh SE has a maskable interrupt for `IRQ`, and the Macintosh II has maskable interrupts for both `IRQ` and `DRQ`. Apple is working on an implementation of the SCSI Manager that will support asynchronous operations on the Macintosh II and probably on the SE as well. Because the interrupt hardware will interact adversely with any asynchronous schemes that are polled, it is strongly recommended that third parties do not attempt asynchronous operations until the new SCSI Manager is released. Apple will not attempt to be compatible with any products that bypass some or all of the SCSI Manager. In order to implement software-based (polled) asynchronous operations it is necessary to bypass the SCSI Manager.

The SCSI Manager section of the alpha draft of *Inside Macintosh* volume V documented the Disconnect and Reselect routines which were intended to be used for asynchronous I/O. Those routines cannot be used. Those routines have been removed from the manual. Any software that uses those routines will have to be revised when the SCSI Manager becomes interrupt-driven. Drivers which send SCSI commands from VBL tasks may also have to be modified.

Hardware in the SCSI

There is some confusion on how many terminators can be used on the bus, and the best way to use them. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of the wire that goes through all of the devices. One terminator should be on the end of the system cable that comes out of the Macintosh. The other terminator would be on the very end of the last device on the chain. If you have an SE or II with an internal hard disk, there is already one terminator on the front of the chain, inside the computer.

On the Macintosh SE and II, there is additional hardware support for the SCSI bus transfers in pseudo-DMA mode. The hardware makes it possible to handshake the data in Blind mode so that the Blind mode is safe for all transfers. On the Macintosh Plus, the Blind transfers are heavily timing dependent and can overrun or underrun during the transfer with no error generated. Assuring that Blind mode is safe on the Macintosh Plus depends upon the peripheral being used. On the SE and II, the transfer is hardware assisted to prevent overruns or underruns.

Changes in SCSI for SE and II

The changes made to the SCSI Manager found in the Macintosh SE and Macintosh II are primarily bug fixes. No new functionality was added. The newer SCSI Manager is more robust and has more error checking. Since the Macintosh Plus SCSI Manager only did limited error checking, it is possible to have code that would function (with bugs) on the Macintosh Plus, but will not work correctly on the SE or II. The Macintosh Plus could mask some bugs in the caller by not checking errors. An example of this is sending or receiving the wrong number of bytes in a blind transfer. On the Macintosh Plus, no error would be generated since there was no way to be sure how many bytes were sent or received. On the SE and II, if the wrong number of bytes are transferred an error will be returned to the caller. The exact timing of transfers has changed on the SE and II as well, since the computers run at different speeds. Devices that are unwittingly dependent upon specific timing in transfers may have problems on the newer computers. To find problems of this sort it is usually only necessary to examine the error codes that are passed back by the SCSI Manager routines. The error codes will generally point out where the updated SCSI Manager found errors.

To report other bugs or make suggestions

Please send additional bug reports and suggestions to us at the address in Technical Note #0. Let us know what SCSI controller you're using in your peripheral, and whether you've had any particularly good or bad experiences with it. We'll add to this note as more information becomes available.

#97: PrSetError Problem

Written by: Mark Baumwell November 15, 1986
Updated: March 1, 1988

This note formerly described a problem in Lisa Pascal glue for the `PrSetError` routine. The glue in MPW (and most, if not all, third party compilers) does not have this problem.

#98: Short-Circuit Booleans in Lisa Pascal

Written by: Mark Baumwell November 15, 1986
Updated: March 1, 1988

This note formerly described problems with the Lisa Pascal compiler. These problems have been fixed in the MPW Pascal compiler.

#99: Standard File Bug in System 3.2

See also: The Standard File Package

Written by: Jim Friedlander November 15, 1986
Updated: March 1, 1988

This note formerly described a bug in Standard File in System 3.2. This bug has been fixed in more recent Systems.

#100: Compatibility with Large-Screen Displays

See also: Technical Note #2—Macintosh Compatibility Guidelines

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

A number of third-party developers have announced large-screen display peripherals for Macintosh. One of them, Radius Inc., has issued a set of guidelines for developers who wish to remain compatible with their Radius FPD; unfortunately, one of their recommendations can cause system crashes. This note suggests a more correct approach.

On the first page of the appendix to their guidelines, "How to be FPD Aware," Radius recommends the following:

> "First, to detect the presence of a Radius FPD, you should check address `$C00008`..."

Unfortunately, this assumes that you're running on a Macintosh or Macintosh Plus; this test will not work on Macintosh XL, nor on a Macintosh II. Since these displays weren't designed to work with systems other than Macintosh and Macintosh Plus, you should make sure you're running on one of these systems before addressing I/O locations (such as those for an add-on display).

Before testing for the presence of any large-screen display, you should first check the machine ID; it's the byte located at `(ROMBASE)+8` (that is, take the long integer at the low-memory location `ROMBASE` [`$2AE`], and add 8 to get the address of the machine ID byte. On a Macintosh or Macintosh Plus, this address will work out to be `$400008`; however, use the low-memory location, to be compatible with future systems that may have the ROM at a different address!).

The machine ID byte will be `$00` for all current Macintosh systems. If the value isn't `$00`, you can assume that no large-screen display is present, but don't forget to follow Technical Note #2's guidelines for screen size independence!

> **Note:** If you are a developer of an add-on large-screen display, we'd be happy to review your guidelines for developers in advance of distribution; please send them to us at the address for comments in Technical Note #0. Future versions of this note may recommend general guidelines for dealing with add-on large-screen displays.

#101: CreateResFile and the Poor Man's Search Path

See also: The File Manager
The Resource Manager
Technical Note #77—HFS Ruminations

Written by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

`CreateResFile` checks to see if a resource file with a given name exists, and if it does, returns a `dupFNErr` (–48) error. Unfortunately, to do this check, `CreateResFile` uses a call that follows the Poor Man's Search Path (PMSP).

`CreateResFile` checks to see if a resource file with a given name exists, and if it does, returns a `dupFNErr` (–48) error. Unfortunately, to do the check, `CreateResFile` calls `PBOpenRF`, which uses the Poor Man's Search Path (PMSP). For example, if we have a resource file in the System folder named '`MyFile`' (and no file with that name in the current directory) and we call `CreateResFile('MyFile')`, `ResError` will return a `dupFNErr`, since `PBOpenRF` will search the current directory first, then search the blessed folder on the same volume. This makes it impossible to use `CreateResFile` to create the resource file '`MyFile`' in the current directory if a file with the same name already exists in a directory that's in the PMSP.

To make sure that `CreateResFile` will create a resource file in the current directory whether or not a resource file with the same name already exists further down the PMSP, call `_Create` (`PBCreate` or `Create`) before calling `CreateResFile`:

```
err := Create('MyFile',0,myCreator,myType);
            {0 for VRefNum means current volume/directory}
CreateResFile('MyFile');
err := ResError; {check for error}
```

In MPW C:

```
err = Create("\pMyFile",0,myCreator,myType);
CreateResFile("\pMyFile");
err = ResError();
```

This works because `_Create` does **not** use the PMSP. If we already have '`MyFile`' in the current directory, `_Create` will fail with a `dupFNErr`, then, if '`MyFile`' has an empty resource fork, `CreateResFile` will write a resource map, otherwise, `CreateResFile` will return `dupFNErr`. If there is no file named '`MyFile`' in the current directory, `_Create` will create one and then `CreateResFile` will write the resource map.
Notice that we are intentionally ignoring the error from `_Create`, since we are calling it only to assure that a file named '`MyFile`' does exist in the current directory.

Please note that SFPutFile does **not** use the PMSP, but that FSDelete does. SFPutFile returns the vRefNum/WDRefNum of the volume/folder that the user selected. If your program deletes a resource file before creating one with the same name based on information returned from SFPutFile, you can use the following strategy to avoid deleting the wrong file, that is, a file that is not in the directory specified by the vRefNum/WDRefNum returned by SFPutFile, but in some other directory in the PMSP:

```
VAR
   wher   : Point;
   reply  : SFReply;
   err    : OSErr;
   oldVol : Integer;

...

   wher.h := 80; wher.v := 90;
   SFPutFile(wher,'','',NIL,reply);
   IF reply.good THEN BEGIN
       err := GetVol(NIL,oldVol); {So we can restore it later}
       err := SetVol(NIL,reply.vRefNum);{for the CreateResFile call}

   {Now for the Create/CreateResFile calls to create a resource file that
    we know is in the current directory}

       err := Create(reply.fName,reply.vRefNum,myCreator,myType);
       CreateResFile(reply.fName); {we'll use the ResError from this ...}

       CASE ResError OF
          noErr:{the create succeeded, go ahead and work with the new
                 resource file -- NOTE: at this point, we don't know
                 what's in the data fork of the file!!} ;
          dupFNErr: BEGIN {duplicate file name error}
              {the file already existed, so, let's delete it. We're now
               sure that we're deleting the file in the current directory}

                err:= FSDelete(reply.fName,reply.vRefNum);

              {now that we've deleted the file, let's create the new one,
               again, we know this will be in the current directory}

                err:= Create(reply.fName,reply.vRefNum,myCreator,myType);
                CreateResFile(reply.fName);
          END; {CASE dupFNErr}
          OTHERWISE      {handle other errors} ;
       END;              {Case ResError}
       err := SetVol(NIL,oldVol);{restore the default directory}
   END;                  {If reply.good}
 ...
```

In MPW C:

```
Point           wher;
SFReply         reply;
OSErr           err;
short           oldVol;


wher.h = 80; wher.v = 90;
SFPutFile(wher,"","",nil,&reply);
if (reply.good )
{
    err = GetVol(nil,&oldVol);
    /*So we can restore it later*/
    err = SetVol(nil,reply.vRefNum);/*for the CreateResFile call*/

    /*Now for the Create/CreateResFile calls to create a resource file
    that we know is in the current directory*/

    err = Create(&reply.fName,reply.vRefNum,myCreator,myType);
    CreateResFile(&reply.fName);
    /*we'll use the ResError from this ...*/

    switch (ResError())
    {
       case noErr:;/*the create succeeded, go ahead and work with the
                     new resource file -- NOTE: at this point, we don't
                     know what's in the data fork of the file!!*/
            break; /* case noErr*/
       case dupFNErr: /*duplicate file name error*/
                  /*the file already existed, so, let's delete it.
                  We're now sure that we're deleting the file in the
                  current directory*/

                  err= FSDelete(&reply.fName,reply.vRefNum);

                  /*now that we've deleted the file, let's create the
                  new one, again, we know this will be in the current
                  directory*/

                  err= Create(&reply.fName,reply.vRefNum,
                                               myCreator,myType);

                  CreateResFile(&reply.fName);
            break; /*case dupFNErr*/
        default:;     /*handle other errors*/
      }  /* switch */
      err = SetVol(nil,oldVol);/*restore the default directory*/
                  /*if reply.good*/
  }
```

Note: `OpenResFile` uses the PMSP too, so you may have to adopt similar strategies to make sure that you are opening the desired resource file and not some other file further down the PMSP. This is normally not a problem if you use `SFGetFile`, since `SFGetFile` does not use the PMSP, in fact, `SFGetFile` does not open or close files, so it doesn't run into this problem.

Macintosh Technical Notes

#102: HFS Elucidations

See also: The File Manager
Technical Note #77—HFS Ruminations

Written by: Bryan "Bo3b" Johnson January 12, 1987
Updated: March 1, 1988

This technical note will describe a few problems that can occur while using HFS. It will also describe ways to avoid these problems.

This technical note will discuss the following problems:

1) It is very important to be careful about how files are opened and closed. There must be no more than one close for every open.

2) Don't use Driver names, like `.Bout`, `.Print` or `.Sony`, in place of file names or the file system will become confused.

3) Be aware of the `ioFlVersNum` byte in all file calls. A number of pieces of the Macintosh system do not use, and may in fact ignore, files created with non-zero `ioFlVersNums`.

Each of these can lead to strange occurrences, as well as problems for the users. Doing any or all of these marginally illegal operations will not necessarily lead to a System Error. In some cases the confusion generated may be worse than a System Error.

One Close is always enough

If a file is closed twice, it is possible to corrupt the file system on a disk. If a program has been creating unreadable disks, this may be the cause.

One aspect of the file system that is not well documented is how it allocates access paths to files that are currently open. As a result of this, it is possible to get a rather cavalier attitude about opening and closing files. This discussion will explain why it is necessary to be very careful about opening and closing files.

When the File Manager receives an `Open` call, it will look at the parameters passed in the parameter block and create a new access path for the file that is being opened. The access path is how the File Manager keeps track of where to send data that is written, and where to get data that is read from that file. An access path is nothing more than: 1) a buffer that the file system uses to read and write data, and 2) a File Control Block that describes how the file is stored on a disk.

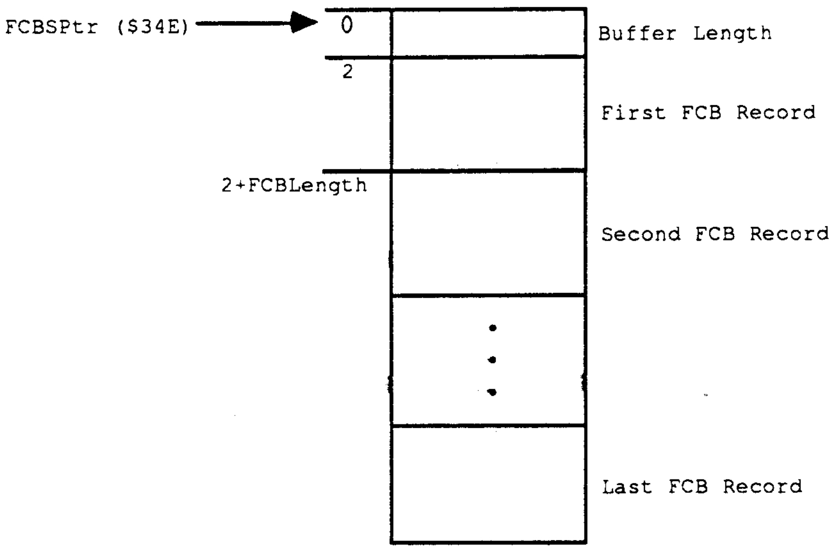
A call like:

```
     ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
```

will create the access path as a buffer and a File Control Block (FCB) in the FCB queue.

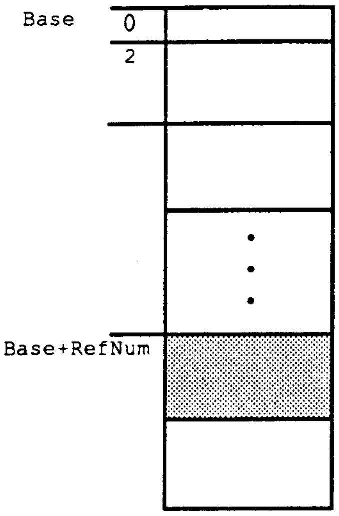
Note: The following information is here for illustrative purposes only; dependence on it may cause compatibility problems with future system software.

The structure of the queue can be visualized as:



where FCBSPtr is a low-memory global (at $34E) that holds the address of a nonrelocatable block. That block is the File Control Block buffer, and is composed of the two byte header which gives the length of the block, followed by the FCB records themselves. The records are of fixed length, and give detailed information about an open file. As depicted, any given record can be found by adding the length of the previous FCB records to the start of the block, adding 2 for the two byte header; giving an offset to the record itself. The size of the block, and hence the number of files that can be open at any given time, is determined at startup time. The call to open 'FirstFile' above will pass back the File Reference Number to that file in FirstRefNum. This is the number that will be used to access that file from that point on. The File Manager passes back an offset into the FCB queue as the RefNum. This offset is the number of bytes past the beginning of the queue to that FCB record in the queue. That FCB record will describe the file that was opened. An example of a number that might get passed back as a RefNum is $1D8. That also means that the FCB record is $1D8 bytes into the FCB block.

A visual example of a record being in use, and how the RefNum is related is:

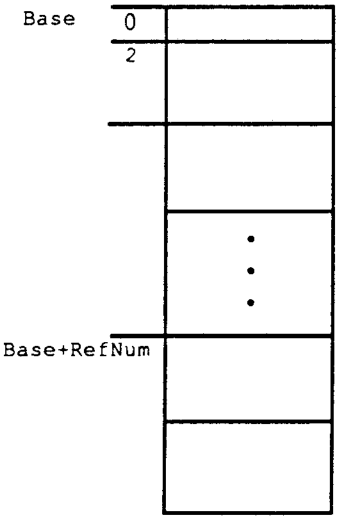


Base is merely the address of the nonrelocatable block that is the FCB buffer. FCBSPtr points to it. The RefNum (a number like $1D8) is added to Base, to give an address in the block. That address is what the file system will use to read and write to an open file, which is why you are required to pass the RefNum to the PBRead and PBWrite calls.

Since that RefNum is merely an offset into the queue, let's step through a dangerous imaginary sequence and see what happens to a given record in the FCB Buffer. Here's the sequence we will step through:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose ( FirstRefNum );
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose ( FirstRefNum ); {the wrong file gets closed!!!}
{the above line will close 'SecondFile', not 'FirstFile', which is already
 closed}
```

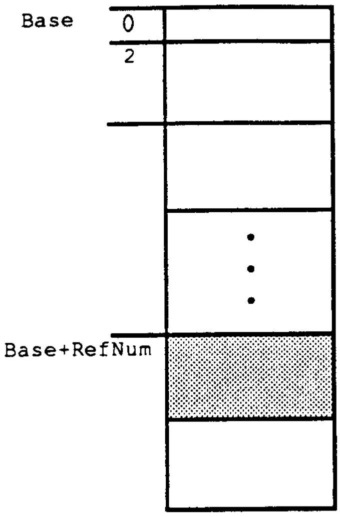
Before any operations:
the record at $1D8 is not used.



After the call:

`ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);`

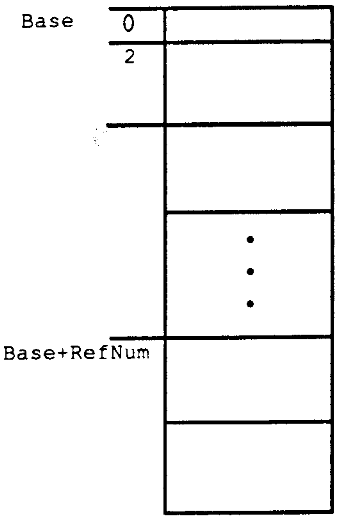
`FirstRefNum = $1D8` and the record is in use.



After the call:

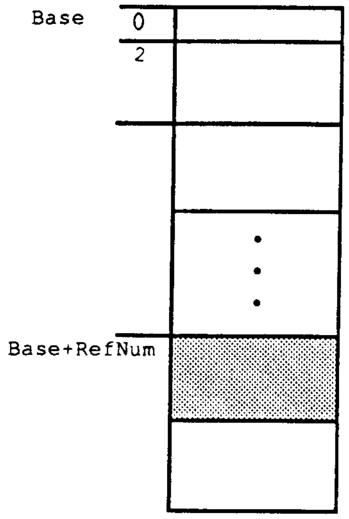
`ErrStuff := FSClose (FirstRefNum);`

`FirstRefNum is still equal to $1D8, but the FCB record is unused.`



After the call:

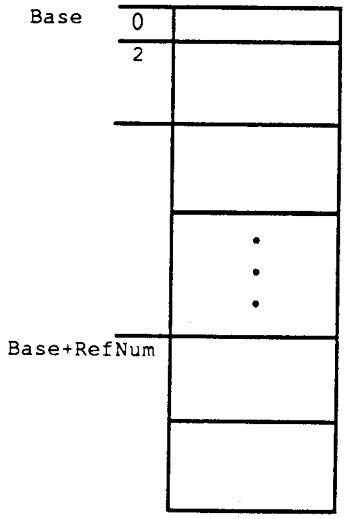
```
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
SecondRefNum = $1D8, FirstRefNum = $1D8, and the record is reused.
```



After the call:

```
ErrStuff := FSClose (FirstRefNum);
The FirstRefNum = $1D8, SecondRefNum = $1D8,
```

the queue element is cleared. This happens, even though `FirstFile` was already closed. Actually, `SecondFile` was closed:



Note that the second close is using the old `RefNum`. The second close will still close a file, and in fact will return `noErr` as its result. Any subsequent accesses to the `SecondRefNum` will return an error, since the file '`SecondFile`' was closed. The File Control Blocks are reused, and since they are just offsets, it is possible to get the same file `RefNum` back for two different files. In this case, `FirstRefNum` = `SecondRefNum` since '`FirstFile`' was closed before opening '`SecondFile`' and the same FCB record was reused for '`SecondFile`'.

There are worse cases than this, however. As an example, think of what can happen if a program were to close a file, then the user inserted an HFS disk. The FCB could be reused for the Catalog File on that HFS disk. If the program had a generic error handler that closed all of its files, it could inadvertently close "its" file again. If it thought "its" file was still open it would do the close, which could close the Catalog file on the HFS disk. This is catastrophic for the disk since the file could easily be closed in an inconsistent state. The result is a bad disk that needs to be reformatted.

There are any number of nasty cases that can arise if a file is closed twice, reusing an old `RefNum`. A common programming practice is to have an error handler or cleanup routine that goes through the files that a program creates and closes them all, even if some may already be closed. If an FCB element was not reused, the `Close` will return the expected `fnOpnErr`. If the FCB had been reused, then the `Close` could be closing the wrong file. This can be very dangerous, particularly for all those paranoid hard disk users.

How to avoid the problem:

A very simple technique is to merely clear the `RefNum` after each close. If the variable that the program uses is cleared after each close, then there is no way of reusing a `RefNum` in the program. An example of this technique would be:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose (FirstRefNum);
FirstRefNum := 0; { We just closed it, so clear our refnum }
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose (FirstRefNum); { returns an error }
```

This makes the second `Close` pass back an error. In this case, the second close will try to close `RefNum` = 0, which will pass back a `fnOpnErr` and do no damage. **Note:** Be sure to use 0, which will never be a valid `RefNum`, since the first FCB entry is beyond the FCB queue length word. Don't confuse this with the 0 that the Resource Manager uses to represent the System file.

Thus, if an error handler were cleaning up possibly open files, it could blithely close all the files it knew about, since it would legitimately get an error back on files that are already closed. This is not done automatically, however. The programmer must be careful about the opening and closing of files. The problem can get quite complex if an error is received halfway through opening a sequence of ten files, for example. By merely clearing the `RefNum` that is stored after each close, it is possible to avoid the complexities of trying to track which files are open and which are closed.

This .file name looks outrageous.

There is a potential conflict between file names and driver names. If a file name is named something like `.Bout`, `.Print` or `.Sony`, then the file system will open the driver instead of the file. Drivers have priority on the 128K ROMs, and will always be opened before a file of the same name. This may mean that an application will get an error back

when opening these types of files, or worse, it will get back a driver RefNum from the call. What the application thought was a file open call was actually a driver open call. If the program uses that access path as a file RefNum, it is possible to get all kinds of strange things to happen. For example, if .Sony is opened, the Sony driver's RefNum would be passed back, instead of a file RefNum. If the application does a Write call using that RefNum, it will actually be a driver call, using whatever parameters happen to be in the parameter block. Disks may be searching for new life after this type of operation. If a program creates files, it should not allow a file to be created whose name begins with '.'.

This file's not my type.

This has been discussed in other places, but another aspect of the File Manager that can cause confusion is the ioFlVersNum byte that is passed to the low-level File Manager calls. This is called ioFileType from Assembly, and should not be confused with ioFVersNum. This byte must be set to zero for normal Macintosh files. There are a number of parts of the system that will not deal correctly with files that have the wrong versions: the Standard File package will not display any file with a non-zero ioFlVersNum; the Segment Loader and Resource Manager cannot open files that have non-zero ioFlVersNums. It is not sufficient to ignore this byte when a file is created. The byte must be cleared in order to avoid this type of problem. Strictly speaking, it is not a problem unless a file is being created on an MFS disk. The current system will easily allow the user to access 400K disks however, so it is better to be safe than confused.

Macintosh Technical Notes

#103: Using MaxApplZone and MoveHHi from Assembly Language

See also: Using Assembly Language
The Memory Manager
Technical Note #129—SysEnvirons

Written by: Bryan "Bo3b" Johnson January 12, 1987
Updated: March 1, 1988

When calling MaxApplZone and MoveHHi from assembly language, be sure to get the correct code.

MaxApplZone and MoveHHi were marked [Not in ROM] in *Inside Macintosh, Volumes I-III* . They are ROM calls in the 128K ROM. Since they are not in the 64K ROM, if you want your program to work on 64K ROM routines it is necessary to call the routines by a JSR to a glue (library) routine instead of using the actual trap macro. The glue calls the ROM routines if they are available, or executes its copy of them (linked into your program) if not.

How to do it:

Whenever you need to use these calls, just call the library routine. It will check ROM85 to determine which ROMs are running, and do the appropriate thing.

For MDS, include the Memory.Rel library in your link file and use:

```
XREF  MoveHHi      ; we need to use this 'ROM' routine
...
JSR  MoveHHi       ; jump to the glue routine that will check ROM85 for us
```

For MPW link with Interface.o and use:

```
IMPORT  MoveHHi    ; we need to use this
...
JSR  MoveHHi       ; jump to the glue routine that will check ROM85 for us
```

Avoid calling _MaxApplZone or _MoveHHi directly if you want your software to work on the 64K ROMs, since that will assemble to an actual trap, not to a JSR to the library.

If your program is going to be run **only** on machines with the 128K ROM or newer, you can call the traps directly. Be sure to check for the 64K ROMs, and report an error to the user. You can check for old ROMs using the SysEnvirons trap as described in Technical Note #129.

Macintosh Technical Notes

#104: MPW: Accessing Globals From Assembly Language

See also: MPW Reference Manual

Written by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note demonstrates how to access MPW Pascal and MPW C globals from the MPW Assembler.

To allow access of MPW Pascal globals from the MPW Assembler, you need to identify the variables that you wish to access as external. To do this, use the {$Z+} compiler option. Using the {$Z+} option can substantially increase the size of the object file due to the additional symbol information (no additional code is generated and the symbol information is stripped by the linker). If you are concerned about object file size, you can "bracket" the variables you wish to access as external variables with {$Z+} and {$Z-}. Here's a trivial example:

Pascal Source

```
PROGRAM MyPascal;
USES
   MemTypes,QuickDraw,OSIntf,ToolIntf;

VAR
   myWRect: Rect;
{$Z+} {make the following external}
   myInt: Integer;
{$Z-} {make the following local to this file (not lexically local)}
   err: Integer;

PROCEDURE MyAsm; EXTERNAL; {routine doubles the value of myInt}

BEGIN {PROGRAM}
   myInt:= 5;
   MyAsm; {call the routine, myInt will be 10 now}
   writeln('The value of myInt after calling myAsm is ', myInt:1);
END. {PROGRAM}
```

Assembly Source for Pascal

```
        CASE    OFF            ;treat upper and lower case identically
MyAsm   PROC    EXPORT         ;CASE OFF is the assembler's default
        IMPORT  myInt:DATA     ;we need :DATA, the assembler assumes CODE
        ASL.W   #1,myInt       ;multiply by two
        RTS                    ;all done with this extensive routine, whew!
        END
```

The variable myInt is accessible from assembler. Neither myWRect nor err are accessible. If you try to access myWRect, for example, from assembler, you will get the following linker error:

```
### Link: Error   Undefined entry name:   MYWRECT.
```

C Source

In an MPW C program, one need only make sure that MyAsm is declared as an external function, that myInt is a global variable (capitalizations must match) and that the CASE ON directive is used in the Assembler:

```
#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <windows.h>
#include <events.h>
#include <textedit.h>
#include <dialogs.h>
#include <stdio.h>

extern MyAsm();     /* assembly routine that doubles the value of myInt */
short myInt;       /* we'll change the value of this variable from MyAsm */

main()
{
WindowPtr MyWindow;
Rect myWRect;

myInt = 5;
MyAsm();
printf(" The value of myInt after calling myAsm is %d\n",myInt);
} /*main*/
```

Assembly source for C

```
        CASE   ON            ;treat upper and lower case distinct
MyAsm   PROC   EXPORT        ;this is how C treats upper and lower case
        IMPORT myInt:DATA    ;we need :DATA, the assembler assumes CODE
        ASL.W  #1,myInt      ;multiply by two
        RTS                  ;all done with this extensive routine, whew!
        END
```

#105: MPW Object Pascal Without MacApp

See also: Technical Note #93—{$LOAD};_DataInit;%_MethTables

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

Object Pascal must have a CODE segment named `%_MethTables` in order to access object methods. In MacApp this is taken care of "behind the scenes" so you don't have to worry about it . However, if you are doing a straight Object Pascal program, you must make sure that `%_MethTables` is around when you need it. If it's unloaded when you call a method, your Macintosh will begin executing wild noncode and die a gruesome and horrible death.

The MPW Pascal compiler must see some declaration of an object in order to produce a reference to the magic segment. You can achieve this cheaply by simply including `ObjIntf.p` in your `Uses` declaration. This must be in the main program, by the way. The compiler will produce a call to `%_InitObj` which is in `%_MethTables`.

If you're a more adventurous soul, you can call `%_InitObj` explicitly from the initialization section of your main program (you must use the `{$%+}` compiler directive to allow the use of "`%`" in identifiers). This will load the `%_MethTables` segment. See Technical Note #93 for ideas about locking down segments that are needed forever without fragmenting the heap.

Macintosh Technical Notes

#106: The Real Story: VCBs and Drive Numbers

See also: The File Manager
Technical Note #36—Drive Queue Element Format

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

The top of page IV-178 in The File Manager chapter of *Inside Macintosh* in attempts to explain the behavior of two fields in a volume control block when the corresponding disk is offline or ejected. Due to the fact that a little bit is left unsaid, this paragraph is rather misleading. The two fields in question are `vcbDrvNum` and `vcbDRefNum` (referred to as `ioVDrvInfo` and `ioVDRefNum` in C and Pascal). `PBHGetVInfo` can be used to access these fields.

Offline

When a mounted volume is placed offline, `vcbDrvNum` is cleared **and** `vcbDRefNum` is set to the two's complement of the drive number. Since drive numbers are assigned positive values (starting with one), this will be a negative number. If `vcbDrvNum` is zero **and** `vcbDRefNum` is negative, you know that the volume is offline.

Ejected

When a volume is ejected, `vcbDrvNum` is cleared and `vcbDRefNum` is set to the positive drive number. If `vcbDrvNum` is zero and `vcbDRefNum` is positive, you know that the volume is ejected. Ejection implies being offline. There is no such thing as "premature ejection".

Summary

	online	offline	ejected
`vcbDrvNum`	`>0 (DrvNum)`	`0`	`0`
`vcbDRefNum`	`<0 (DRefNum)`	`<0 (-DrvNum)`	`>0 (DrvNum)`

Please refrain from assuming anything about a VCB queue element beyond what is documented in *Inside Macintosh*, and don't expect it to always be 178 bytes in size. It grew when we went from MFS to HFS, and it may grow again. It's safest to use calls like `PBHGetVInfo` to get the information that you need.

Macintosh Technical Notes

#107: Nulls in Filenames

See also: The File Manager

Written by: Rick Blair March 2, 1987
Updated: March 1, 1988

Some applications (loosely speaking so as to include Desk Accessories, INITs, and what-have-you) generate or rename special files on the fly so that they are not explicitly named by the user via `SFPutFile`. Since the Macintosh file system is very liberal about filenames and only excludes colons from the list of acceptable characters, this can lead to some difficulties, both for the end user and for writers of other programs which may see these files.

Other programs which might be backing up your disk or something similar may get confused. A program written in C will think it has found the end of a string when it hits a null (ASCII code 0) character, so nulls in filenames are especially risky.

As a rule, filenames should only include characters which the user can see and edit. The only reasonable exception might be invisible files, but it can be argued that they are of dubious value anyway. You can argue "but what about my help file, I don't want it renamed" but we already have what we think is the best approach for that situation. If you can't find a configuration or other file because the user has renamed or moved it, then call `SFGetFile` and let the user find it. If the user cancels, and you can't run without the file, then quit with an appropriate message.

Please consider carefully before you put non-displaying characters in filenames!

Macintosh Technical Notes

#108: _AddDrive, _DrvrInstall, and _DrvrRemove

See also: Technical Note #36, Drive Queue Elements
SCSI Development Package (APDA)

Written by: Jim Friedlander March 2, 1987
Revised by: Pete Helme December 1988

_AddDrive, _DrvrInstall, and _DrvrRemove are used in the sample SCSI driver in the SCSI Development Package, which is available from APDA. This Technical Note documents the parameters for these calls.
Changes since March 1, 1988: Updated the _DrvrInstall text to reflect the use of register A0, which should contain a pointer to the driver when called. Also added simple glue code for _DrvrInstall and _DrvrRemove since none is available in the MPW interfaces.

_AddDrive

_AddDrive adds a drive to the drive queue, and is discussed in more detail in Technical Note #36, Drive Queue Elements:

```
FUNCTION AddDrive(DQE:DrvQEl;driveNum,refNum:INTEGER):OSErr;
```

A0 (input)	→	pointer to DQE
D0 high word(input)	→	drive number
D0 low word(input)	→	driver RefNum
D0 (output)	←	error code
		noErr (always returned)

_DrvrInstall

_DrvrInstall is used to install a driver. A DCE for the driver is created and its handle entered into the specified Unit Table position (–1 through –64). If the unit number is –4 through –9, the corresponding ROM-based driver will be replaced:

```
FUNCTION DrvrInstall(drvrHandle:Handle; refNum: INTEGER): OSErr;
```

A0 (input)	→	pointer to driver
D0 (input)	→	driver RefNum (–1 through –64)
D0 (output)	←	error code
		noErr
		badUnitErr

_DrvrRemove

_DrvrRemove is used to remove a driver. A RAM-based driver is purged from the system heap (using _ReleaseResource). Memory for the DCE is disposed:

```
FUNCTION DrvrRemove(refNum: INTEGER):OSErr;
```

D0 (input)	→	Driver RefNum
D0 (output)	←	error code
		noErr
		qErr

Interfaces

Through a sequence of cataclysmic events, the glue code for _DrvrInstall and _DrvrRemove was never actually added to the MPW interfaces (i.e., "We forgot."), so we will include simple glue here at no extra expense to you.

It would be advisable to first lock the handle to your driver with _HLock before making either of these calls since memory may be moved.

```
;-----------------------------------------------------------------
; FUNCTION DRVRInstall(drvrHandle:Handle; refNum:INTEGER):OSErr;
;-----------------------------------------------------------------

DRVRInstall     PROC    EXPORT
        MOVEA.L         (SP)+, A1       ; pop return address
        MOVE.W          (SP)+, D0       ; driver reference number
        MOVEA.L         (SP)+, A0       ; handle to driver
        MOVEA.L         (A0), A0        ; pointer to driver
        _DrvrInstall                    ; $A03D
        MOVE.W          D0, (SP)        ; get error
        JMP             (A1)            ; & split
        ENDPPROC


;-----------------------------------------------------------------
; FUNCTION DRVRRemove(refNum:INTEGER):OSErr;
;-----------------------------------------------------------------

DRVRRemove      PROC    EXPORT
        MOVEA.L         (SP)+, A1       ; pop return address
        MOVE.W          (SP)+, D0       ; driver reference number
        _DrvrRemove                     ; $A03E
        MOVE.W          D0, (SP)        ; get error
        JMP             (A1)            ; & split
        ENDPPROC
```

#109: Bug in MPW 1.0 Language Libraries

See also: MPW Reference Manual

Written by: Scott Knaster March 2, 1987
Updated: March 1, 1988

This note formerly described a problem in the language libraries for MPW 1.0. This bug is fixed in MPW 1.0.2, available from APDA.

Macintosh
Technical Notes

Developer Technical Support

#110: MPW: Writing Stand-Alone Code

Revised by: Keith Rollin
Written by: Jim Friedlander

February 1990
March 1987

MPW Pascal and C can be used to write stand-alone code such as `'WDEF'`, `'LDEF'`, `'INIT'`, and `'FKEY'` resources. This Technical Note, which is not intended to be a complete discussion of the issues involved in writing stand-alone code resources, shows how to produce such stand-alone code using the MPW Pascal and C compilers and the linker, and includes an example of an `'INIT'` and a shell for making a `'WDEF'`.

Changes since March 1988: Added a note about the 32K size limit on stand-alone code resources; included an example of how to load and execute stand-alone code from an application; and added references to Technical Note #256, Globals in Stand-Alone Code, concerning the use of global variables, and Technical Note #240, Using MPW for Non-Macintosh 68000 Systems, concerning breaking the 32K limit.

Size Does Matter

There is a somewhat hard size limit of 32K bytes on code segments, including `'CODE'` resources in an application and stand-alone code such as `'XCMD'`, `'FKEY'`, `'DRVR'`, and `'WDEF'` resources. This limitation exists because Macintosh code has to be relocatable, requiring the use of PC-relative instructions. Unfortunately, the 68000 supports only 16-bit signed offsets for the purpose. These offsets limit code to a maximum jump of 32K bytes either forward or backward. For a procedure at the beginning of a code segment to branch to a procedure at the very end, that procedure cannot be more than 32K bytes away.

This limitation applies to all code segments, including those that comprise an application. All those whizzy 790K word processors and spreadsheets are actually composed of many, many `'CODE'` resources, all of which are smaller than 32K. However, special support is available for applications in the form of a jump table. This jump table keeps track of the entry points within these code segments, so that there is a way to branch from one to another. Unfortunately, you cannot do the same thing for stand-alone code resources, as the system doesn't support the use of more than one jump table. For more information on the jump table, see *Inside Macintosh*, Volume II, The Segment Loader.

The reason why this 32K limit is only a "somewhat hard" limit is because, if you are really determined, you can break this limit. If you can write your code in such a way that you don't ever need to make a jump that is longer than 32K bytes, then you should be able to get away with stretching the limit. For more of the gory details, see the section "Segmenting and the Jump Table" in Technical Note #240, Using MPW for Non-Macintosh 68000 Systems.

Calling Stand-Alone Code From An Application

Assume that you are writing an application and would like to support external routines in the form of stand-alone code. Applications like HyperCard and Apple File Exchange support such a mechanism. How do you go about putting in this functionality?

The first thing to do is establish some standard means for communicating. This is shown with HyperCard 'XCMD' resources, where a clearly defined parameter block is passed between HyperCard and the 'XCMD'.

```
XCmdBlock = RECORD
        paramCount:  INTEGER;
        params:      ARRAY [1..16] OF Handle;
        returnValue: Handle;
        passFlag:    BOOLEAN;
        entryPoint:  ProcPtr;            {to call back to HyperCard}
        request:     INTEGER;
        result:      INTEGER;
        inArgs:      ARRAY [1..8] OF LONGINT;
        outArgs:     ARRAY [1..4] OF LONGINT;
        END;
XCmdPtr = ^XCmdBlock;
```

When HyperCard calls an 'XCMD', it passes a pointer to this parameter block. The entry point to such an 'XCMD' could be declared as follows:

```
PROCEDURE XStringWidth(paramPtr: XCmdPtr);
```

To call the 'XCMD', you need to load it into memory, lock it down, fill in a parameter block, and then call the 'XCMD'. When you are done, you need to remove the 'XCMD' from memory:

```
h := Get1NamedResource('XCMD', 'XStringWidth');
HLock(h);
WITH parameterBlock DO BEGIN
    < fill it in >
END;
CallXCMD(@parameterBlock, h);
HUnlock(h);
```

CallXCMD is some in-line code that takes the Handle h and executes the necessary machine language commands to jump to it. It does this by taking the handle off of the stack, turning it into a pointer to the stand-alone code, and performing a JSR to it. In this way, the parameter block is left on the stack for the stand-alone code to access:

```
PROCEDURE CallXCMD(pb: XCMDPtr; XCMD: Handle);
       INLINE $205F,    { MOVE.L (A7)+,A0 }
              $2050,    { MOVE.L (A0),A0 }
              $4E90;    { JSR (A0) }
```

Writing an 'INIT' in Pascal

An `'INIT'` resource is stand-alone code that is executed on startup in the manner specified in the System Resource File and Startup Manager chapters of *Inside Macintosh*. `'INIT'` resources are commonly written is assembly language, but can also be written in high-level languages such as Pascal and C. Following is the source for a simple, but nonetheless highly obnoxious, `'INIT'` written in MPW Pascal:

```
UNIT MyInit; {stand-alone code is written as a UNIT}

INTERFACE

USES
   MemTypes, QuickDraw, OSIntf, ToolIntf;

PROCEDURE BeepTwice;

IMPLEMENTATION

   PROCEDURE BeepTwice;

   VAR finalTicks: LongInt;

   BEGIN {BeepTwice}
      SysBeep(1);
      Delay(120, finalTicks); {Delay two seconds, this'll annoy 'em!}
      SysBeep(1);
   END;  {BeepTwice}

END.  {UNIT}
```

That's all there is to the Pascal. Now you can compile and link the code to produce a stand-alone module. Following are the commands that you use:

```
pascal Init.p
```

Compile the unit to output file Init.p.o.

```
link ∂
        -rt INIT=0 ∂                    # resource type and ID
        -ra =16                         # INITs must be locked
        -m BEEPTWICE ∂                  # Pascal generates uppercase module names
        Init.p.o ∂                      # Link this object file first!!! Then ...
        "{Libraries}"Interface.o ∂      # need this for the glue for Delay()
        -o MyInit                       # output to this file
```

This links the INIT, puts it in the file MyInit and gives the INIT the resource type `'INIT'`, `ID = 0`. You should also set the "locked" bit in the resource attributes of the INIT (`'INIT'` resources must be marked locked because INIT 31 does not lock them). The main entry point is specified by the `-m` option. Pascal Units do not have a main entry point, and, since you are linking with "{Libraries}"Interface.o (you need the glue for `_Delay`), you need to tell the linker what to strip against. You could link this without the `-m` option, but then all the code for "{Libraries}"Interface.o would wind up in the `'INIT'`, making it much larger than it needs to be. Notice also that you need to capitalize `BEEPTWICE`, since Pascal converts module names to upper case.

Next you specify the files with which you wish to link. Since the linker links files in the order they are specified, you need to list Init.p.o first, otherwise the first instruction for your code is not

BeepTwice, but rather the glue for _Delay (which is disastrous). 'INIT' resources are entered at the beginning, regardless of where the main entry point is.

If you have any doubts about what the entry point is (and you can read assembler) you can use the command DumpCode MyInit -rt INIT to look at the code. In this case, the first code that is executed should be:

```
LINK A6, #$FFFC         ; make room for the local var 'long'
```

If you had incorrectly specified "{Libraries}"Interface.o first, the first code executed would have been the following glue for _Delay (and the code for the 'INIT' would never have been executed):

```
MOVE.L (A7)+,D0         ; execute ourselves
MOVE.L (A7)+,A1         ; address of VAR parameter
...                     ; finish getting ready for Delay
_Delay                  ; do the Delay
MOVE.L D0,(A1)          ; the VAR parameter
RTS                     ; return - but to where???
LINK A6, #$FFFC         ; the correct code, but it'll never
 ...                    ; be executed
```

```
SetFile MyInit -t INIT -c JAF1 && ∂
    duplicate -y MyInit "{SystemFolder}"
```

This command sets the file type of MyInit to "INIT" (so that the INIT 31 mechanism runs it) and the creator to "JAF1". (Yes, JAF1 is registered with Developer Technical Support. Is **your** file type?) If the SetFile succeeds, you then courageously duplicate the INIT into the system folder, so it is executed the next time the system is rebooted.

That's all there is to it.

Now for a couple of caveats. First of all, you cannot easily use globals in stand-alone code. If you put the line VAR gLong: Longint; right after the keyword INTERFACE, the code compiles and links okay, and probably executes okay. You get **no** warning that you are using someone else's global space. If you use the statement gLong := 4; the long word value four is placed at -4(A5), thus destroying whatever was there (generally, the start of the application's globals). This is not really a problem with 'INIT' resources (it definitely is a problem in the 'WDEF' example below), but, in general, you should not use globals in stand-alone code.

Another limitation of stand-alone code is that it cannot use other globals such as QuickDraw globals. For example, if you try to make a QuickDraw call such as SetPort(@thePort); (which uses the QuickDraw global variable thePort) you are informed about your transgression:

```
### link: Error  Undefined entry, name: QUICKDRAW
 Referenced from: BEEPTWICE in file: Init.p.o
```

You can access QuickDraw globals from stand-alone code by using A5 (available from high-level languages in the low-memory global CurrentA5 (a long word at $904)) which is a pointer to a pointer to thePort (@thePort = (A5)). Some of the standard Pascal library routines require the use of globals, you get similar linker errors if you use these routines.

If something isn't working correctly, you might look for inadvertent use of globals. If your use of globals is intentional, then make sure you are using them in accordance with Technical Note #256, Globals in Stand-Alone Code.

Writing an 'INIT' in C

Following is the source for the same 'INIT' in MPW C:

```
#include <OSUtils.h>

void BeepTwice()
{
        long int finalTicks;

        SysBeep(1);
        Delay(120,&finalTicks);
        SysBeep(1);
}
```

The link instruction for C is:

```
link ∂
        -rt INIT=0 ∂                         # resource type
        -ra =16 ∂                            # INITs must be locked
        -m BeepTwice ∂                       # note that C is case sensitive
        TN110Init.c.o ∂                      # link this object file first!!! then ...
        "{CLibraries}"CInterface.o ∂         # need this for the glue for Delay()
        -o tn110INIT                         # output to this file
```

Writing a 'WDEF' in Pascal

Writing a 'WDEF' is like writing an 'INIT', except that 'WDEF' resources have standard headers that are incorporated into the code. In this example, the 'WDEF' is the Pascal MyWindowDef. To create the header, you use an assembly language stub:

```
StdWDEF     MAIN EXPORT                      ; this will be the entry point
            IMPORT MyWindowDef               ; name of Pascal FUNCTION that is the WDEF
                                             ; we IMPORT externally referenced routines
                                             ; from Pascal (in this case, just this one)
            BRA.S     @0                     ; branch around the header to the actual code
            DC.W      0                      ; flags word
            DC.B      'WDEF'                 ; type
            DC.W      3                      ; ID number
            DC.W      0                      ; version
@0          JMP       MyWindowDef            ; this calls the Pascal WDEF
            END
```

Now for the Pascal source for the 'WDEF'. Only the shell of what needs to be done is listed, the actual code is left as an exercise for the reader (for further information about writing a 'WDEF', see *Inside Macintosh*, Volume I, The Window Manager (pp. 297-302).

```
UNIT WDef;

INTERFACE

  USES MemTypes, QuickDraw, OSIntf, ToolIntf;

{this is the only external routine}
  FUNCTION MyWindowDef(varCode: Integer; theWindow: WindowPtr; message: Integer;
                       param: LongInt): LongInt; {As defined in IM p. I-299}

IMPLEMENTATION
```

```
        FUNCTION MyWindowDef(varCode: Integer; theWindow: WindowPtr; message: Integer;
                             param: LongInt): LongInt;

          TYPE
            RectPtr = ^Rect;

          VAR
            aRectPtr : RectPtr;

        {here are the routines that are dispatched to by MyWindowDef}

          PROCEDURE DoDraw(theWind: WindowPtr; DrawParam: LongInt);
            BEGIN {DoDraw}
              {Fill in the code!}
            END; {DoDraw}

          FUNCTION DoHit(theWind: WindowPtr; theParam: LongInt): LongInt;
            BEGIN {DoHit}
              {Code for this FUNCTION goes here}
            END;  {DoHit}

          PROCEDURE DoCalcRgns(theWind: WindowPtr);
            BEGIN {DoCalcRgns}
              {Code for this PROCEDURE goes here}
            END;  {DoCalcRgns}

          PROCEDURE DoGrow(theWind: WindowPtr; theGrowRect: Rect);
            BEGIN {DoGrow}
              {Code for this PROCEDURE goes here}
            END;  {DoGrow}

          PROCEDURE DoDrawSize(theWind: WindowPtr);
            BEGIN {DoDrawSize}
              {Code for this PROCEDURE goes here}
            END;  {DoDrawSize}

{now for the main body to MyWindowDef}
         BEGIN  { MyWindowDef }
         {case out on the message and jump to the appropriate routine}
           MyWindowDef := 0; {initialize the function result}

           CASE message OF

             wDraw:  { draw window frame}
               DoDraw(theWindow,param);

             wHit:   { tell what region the mouse was pressed in}
               MyWindowDef := DoHit(theWindow,param);

             wCalcRgns: { calculate structRgn and contRgn}
               DoCalcRgns(theWindow);

             wNew:   { do any additional initialization}
               { we don't need to do any}
               ;

             wDispose:{ do any additional disposal actions}
               { we don't need to do any}
               ;

             wGrow:  { draw window's grow image}
               BEGIN
                 aRectPtr := RectPtr(param);
                 DoGrow(theWindow,aRectPtr^);
               END; {CASE wGrow}
            wDrawGIcon:{ draw Size box in content region}
              DoDrawSize(theWindow);

          END; {CASE}
        END; {MyWindowDef}
END. {of UNIT}
```

Following are the MPW shell commands necessary to build this 'WDEF':

```
pascal MyWDEF.p
asm MyWDEF.a
link    -rt WDEF=3 ∂
        MyWDEF.a.o ∂ # MUST link with this first
        MyWDEF.p.o ∂
        "{Libraries}"Interface.o ∂
        -o MyWDEF3
```

Notice that you do **not** need the -m option; since MyWDEF.a.o contains the main entry point, the linker knows what to strip against.

That's all there is to it.

Writing a 'WDEF' in C

Writing a 'WDEF' in MPW C is very similar to writing one in Pascal. You can use the same assembly language header, and all you need to make sure of is that the main dispatch routine (in this case: MyWindowDef) is first in your source file. Here's the same 'WDEF' shell in MPW C:

```
/* first, the mandatory includes */
#include <types.h>
#include <quickdraw.h>
#include <resources.h>
#include <fonts.h>
#include <windows.h>
#include <menus.h>
#include <textedit.h>
#include <events.h>

/* declarations */
void DoDrawSize();
void DoGrow();
void DoCalcRgns();
long int DoHit();
void DoDraw();

/*---------------------- Main Proc within WDEF -----------------------*/
pascal long int MyWindowDef(varCode,theWindow,message,param)
short int    varCode;
WindowPtr    theWindow;
short int    message;
long int     param;

{    /* MyWindowDef */

  Rect        *aRectPtr;
  long int    theResult=0;      /*this is what the function returns, init to 0 */
  switch (message)
  {
    case wDraw:                /* draw window frame*/
      DoDraw(theWindow,param);
      break;
    case wHit:                 /* tell what region the mouse was pressed in*/
      theResult = DoHit(theWindow,param);
      break;
    case wCalcRgns:            /* calculate structRgn and contRgn*/
      DoCalcRgns(theWindow);
      break;
    case wNew:                 /* do any additional initialization*/
      break;                   /* nothing here */
    case wDispose:             /* do any additional disposal actions*/
      break;                   /* we don't need to do any*/
    case wGrow:                /* draw window's grow image*/
      aRectPtr = (Rect *)param;
      DoGrow(theWindow,*aRectPtr);
      break;
    case wDrawGIcon:           /* draw Size box in content region*/
      DoDrawSize(theWindow);
      break;
  }  /* switch */
  return theResult;
}    /* MyWindowDef */

/* here are the routines that are dispatched to by MyWindowDef

/*---------------------------- DoDraw function -----------------------------*/
void DoDraw(WindToDraw,DrawParam)
WindowPtr    WindToDraw;
long int     DrawParam;

{  /* DoDraw */
     /* code for DoDraw goes here */
}  /* DoDraw */

/*---------------------------- DoHit function -----------------------------*/
long int DoHit(WindToTest,theParam)
WindowPtr    WindToTest;
long int     theParam;

{  /* DoHit */
     /* code for DoHit goes here */
}  /* DoHit */

/*------------------------- DoCalcRgns procedure --------------------------*/
void DoCalcRgns(WindToCalc)
WindowPtr    WindToCalc;

{  /* DoCalcRgns */
     /* code for DoCalcRgns goes here */
}  /* DoCalcRgns */

/*---------------------------- DoGrow procedure ----------------------------*/
void DoGrow(WindToGrow,theGrowRect)
WindowPtr    WindToGrow;
Rect         theGrowRect;

{  /* DoGrow */
     /* code for DoGrow goes here */
}  /* DoGrow */
```

```
/*------------------------ DoDrawSize procedure --------------------------*/
void DoDrawSize(WindToDraw)
WindowPtr    WindToDraw;

{  /* DoDrawSize */
       /* code for DoDrawSize goes here */
}  /* DoDrawSize */
```

To link this 'WDEF', you can use the following link command:

```
Link     -rt WDEF=3 ∂
         tn110.WDEFHeader.a.o ∂  # must link with this first
         tn110.wdef.c.o ∂
         "{CLibraries}"CInterface.o ∂
         -o tn110.wdef
```

Further Reference:

- *Inside Macintosh*, Volume I, The Window Manager
- *Inside Macintosh*, Volume II, The Segment Loader
- *Inside Macintosh*, Volume II, The System Resource File
- *Inside Macintosh*, Volume V, The Start Manager
- *MPW Reference Manual*
- Technical Note #240, Using MPW for Non-Macintosh 68000 Systems
- Technical Note #256, Globals in Stand-Alone Code?

Macintosh Technical Notes

#111: MoveHHi and SetResPurge

See also: The Memory Manager
The Resource Manager

Written by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

`SetResPurge(TRUE)` is called to make the Memory Manager call the Resource Manager before purging a block specified by a handle. If the handle is a handle to a resource, and its `resChanged` bit is set, the resource data will be written out (using `WriteResource`).

When `MoveHHi` is called, even though the handle's block is not actually being purged, the resource data specified by the handle will be written out. An application can prevent this by calling `SetResPurge(FALSE)` before calling `MoveHHi` (and then calling `SetResPurge(TRUE)` after the `MoveHHi` call).

#112: FindDItem

See also: The Dialog Manager

Written by: Rick Blair March 2, 1987
Updated: March 1, 1988

FindDItem is a potentially useful call which returns the number of a dialog item given a point in local coordinates and a dialog handle. It returns an item number of –1 if no item's rectangle overlaps the point. This is all well and good, except you don't get back quite what you would expect.

The item number returned is zero-based, so you have to add one to the result:

```
    theitem := FindDItem(theDialog, thePoint) + 1;
```

Macintosh Technical Notes

#113: Boot Blocks

See also: The Segment Loader

Written by: Bo3b Johnson March 2, 1987
Updated: March 1, 1988

There are two undocumented features of the Boot Blocks. This note will describe how they currently work.

Warning: The format and functionality of the Boot Blocks will change in the future; dependence on this information may cause your program to fail on future hardware or with future System software.

The first two sectors of a bootable Macintosh disk are used to store information on how to start up the computer. The blocks contain various parameters that the system uses to startup such as the name of the system file, the name of the Finder, the first application to run at boot time, the number of events to allow, etc.

Changing System Heap Size

The boot blocks dictate what size the system heap will be after booting. Any common sector editing program will allow you to change the data in the boot blocks. Changing the system heap size is accomplished by changing two parameters in the boot blocks: the long word value at location `$86` in Block 0 indicates the size of the system heap; the word value at location `$6` is the version number of the boot blocks. Changing the version number to be greater than `$14` (`$15` is recommended) tells the ROM to use the value at `$86` for the system heap size, otherwise the value at `$86` is ignored. The `$86` location only applies to computers with more than 128K of RAM.

Secondary Sound and Video Pages

Another occasionally useful feature of the boot blocks is the ability to specify that the secondary sound and video pages be allocated at boot time. This is done before a debugger is loaded, so the debugger will load below the alternate screen. This is useful for debugging software that uses the alternate video page, like page-flipping demos or games. To allocate the second video and sound buffers, change the two bytes starting at location `$8` in the boot blocks. Change the value (normally 0) to a negative number (`$FFFF`) to allocate both video and sound buffers. Change the value to a positive number (`$0001`) to allocate only the secondary sound buffer.

Warning: MacsBug may not work properly if you allocate additional pages for sound and video.

#114: AppleShare and Old Finders

See also: *AppleShare User's Guide*

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

A rumor has been spread that if you use a pre-AppleShare Finder on a workstation to access AppleShare volumes, you can bypass AppleShare's "access privilege" mechanisms.

This is not true. Access controls are enforced by the server, **not** by the Finder. If you use an older Finder, you are still prevented (by the server) from gaining access to protected files and folders; however, you will not get the proper user-interface feedback that you would if you were using the correct Finder: for instance, folders on the server will always appear plain white (that is, without the permission feedback you'd normally get), and error messages would not be as explanatory as those from Finders that "know" about AppleShare servers.

#115: Application Configuration with Stationery Pads

See also: The File Manager
Technical Note #116—AppleShare-able Applications
Technical Note #47—Customizing SFGetFile
Technical Note #48—Bundles
"Application Development in a Shared Environment"

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

With the introduction of AppleShare (Apple's file server) there are restrictions on self-modification of application resource files and the placement of configuration files. This note describes one way to get around the necessity for configuration files.

Configuration Files

Some applications need to store information about configuration; others could benefit simply from allowing users to customize default ruler settings, window placement, fonts, etc.

There are applications which store this information as additional resources in the application's resource file; when the user changes the configuration, the application writes to itself to change the saved information.

AppleShare, however, requires that if an application is to be used by more than one user at a time, it must not need write access to itself. This means that the above method of storing configuration information cannot be used. (For more information about making your application sharable, see Technical Note #116.)

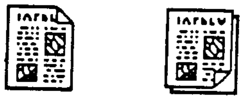
Storing configuration in a special configuration file can be a problem; the user must keep the file in the system folder or the application must search for it. This process has design issues of its own.

An alternative to configuration files: Stationery Pads

A basis for one solution to this problem was a user-interface feature of the Lisa Office System architecture. Lisa introduced the concept of "stationery pads", special documents that created copies of themselves to allow users to save a pre-set-up document for future use. On Lisa, this was the way Untitled documents were created.

Your Macintosh application can provide the option of saving a document as a stationery pad, to provide similar functionality. Here's how:

- You'll need to add a checkbox to your `SFPutFile` dialog box (if you don't know how to do this, check out Technical Note #47); if the user checks this box, save the document as you normally would, but use a different file type (the file type of a document is usually set when the document is created, using the File Manager `Create` procedure, or later using `SetFileInfo`).



A Document and its Stationery pad

- Be sure to use a different but similar icon for the stationery pad file. This is easy if you differentiate between stationery and normal files solely by file type—the Finder uses the type to determine which icon to display, see Technical Note #48 for help with the "bundle" mechanism used to associate a file type with an icon.

- When opening a stationery pad file, the window should come up named "Untitled", with the contents of the stationery pad file.

- "Revert" should re-read the stationery pad file.

- Don't forget to add the stationery pad's file type to the file-types list that you pass to Standard File, so that the new files will appear in the list when the user chooses Open. This file type should be registered with Macintosh Developer Technical Support.

#116: AppleShare-able Applications and the Resource Manager

See also: The Resource Manager
"Application Development in a Shared Environment"
Technical Note #40—Finder Flags

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

Normally, applications on an AppleShare server volume cannot be executed by more than one user at a time. This technical note explains why, and tells how you can enable your application to be shared.

The Resource Manager versus Shared Files

Part of the explanation of why applications are not automatically sharable is based on the design of the Resource Manager. The Resource Manager is a great little database. It was originally conceived as a way to keep applications localizable (a task it has performed admirably), and was found to be an excellent foundation for the Segment Loader, Font Manager, and a large part of the rest of the Macintosh operating system.

However, it was never designed to be a multi-user database. When the Resource Manager opens a resource file (such as an application), it reads the file's resource map into memory. This map remains in memory until the resource file is closed by the Segment Loader, which regains control when the application exits. Sometimes it is necessary to write the map out to disk; normally, this is only done by `UpdateResFile` and `CloseResFile`.

If two users opened the same resource file at the same time, and one of them had write access to the file and added a resource to it, the other user's Resource Manager wouldn't know about it; this would make the other user's copy of the file's original resource map invalid. This could cause (at least) a crash; if both users had write access, it's not unlikely that the resource file involved would become corrupted. Also, although you can tell the Resource Manager to write out an updated resource map, there's no way for another user to tell it to refresh the copy of the map in memory if the file changes.

What does all this have to do with running my application twice?

Your application is stored as a resource file; code segments, alert and dialog templates, etc., are resources. If you write to your application's resource file (for instance, to add configuration information, like print records), your application can't be shared.

In Apple's compatibility testing of existing applications (during development of AppleShare), we found quite a few applications, some of them quite popular, that wrote to their own resource files. So we decided, to improve the safety of using AppleShare, to always launch applications using a combination of access privileges such that only one user at a time could use a given application (these privileges will be discussed in a future Technical Note). In fact, AppleShare opens all resource files this way, unless the resource file is opened with `OpenRFPerm` and read-only permission is specified.

But my application doesn't write to itself!

We realize that many applications do not. However, there are other considerations (covered in detail, with suggestions for fixes, in "Application Development in a Shared Environment", available from APDA). In brief, here are the big ones we know about:

- Does your application create temporary files with fixed names in a fixed place (such as the directory containing the application)? Without AppleShare's protection, two applications trying to use the same temporary file could be disastrous.

- Is your application at least "conscious" of the fact that it may be in a multi-user environment? For instance, does it work correctly if a volume containing an existing document is on a locked volume? Does it check all result codes returned from File Manager calls, and `ResError` after relevant Resource Manager calls?

OK, I follow the rules. What do I do to make my application sharable?

There is a flag in each file's Finder information (stored in the file's directory entry) known as the "shared" bit. If you set this bit on your application's resource file, the Finder will launch your application using read-only permissions; if anyone else launches your application, they'll also get it read-only (their Finder will see the same "shared" bit set.).

Three important warnings accompany this information:

- The definition of the "shared" bit was incorrect in previous releases of information and software from Apple. This includes the June 16, 1986 version of Technical Note #40 (fixed in the March 2, 1987 version), as well as all versions of ResEdit before and including 1.1b3 (included with MPW 2.0). For now, the most reliable way to set this bit is to get the 1.1b3 version of ResEdit, use it to Get Info on your application, and check the box labeled "cached" (the incorrect documentation upon which ResEdit [et al.] was based called the real shared bit "cached"; the bit labeled as "shared" is the real cached bit [a currently unused but reserved bit which should be left clear]).

- By checking this bit, you're promising (to your users) that your application will work entirely correctly if launched by more than one user. This means that you follow the other rules, in addition to simply not writing to your application's own resource file. See "Application Development for a Shared Environment," and test carefully!

- Setting this bit has nothing to do with allowing your application's documents to be shared; you must design this feature into your application (it's not something that Apple system software can take care of behind your application's back.). You should realize from reading this note, however, that if you store your document's data in resource files, you won't be able to allow multiple users to access them simultaneously.

#117: Compatibility: Why & How

See Also: Technical Note #2—Compatibility Guidelines
Technical Note #7—A Few Quick Debugging Tips

Written by: Bo3b Johnson February 9, 1987
Updated: March 1, 1988

While creating or revising any program for the Macintosh, you should be aware of the most common reasons why programs fail on various versions of the Macintosh. This note will detail some common failure modes, why they occur, and how to avoid them.

We've tried to explain the issues in depth, but recognize that not everyone is interested in every issue. For example, if your application is not copy protected, you're probably not very interested in the section on copy protection. That's why we've included the outline form of the technical note. The first two pages outline the problems and the solutions that are detailed later. Feel free to skip around at will, but remember that we're sending this enormous technical note because the suggestions it provides may save you hasty compatibility revisions when we announce a new machine.

We know it's a lot, and we're here to help you if you need it. Our address (electronic and physical) is on page three—contact us with **any** questions—that's what we're here for!

Compatibility: the outline

Don't assume the screen is a fixed size

To get the screen size:

- check the QuickDraw global `screenBits.bounds`

Don't assume the screen is in a fixed location

To get the screen location:

- check the QuickDraw global `screenBits.baseAddr`

Don't assume that `rowBytes` is equal to the width of the screen

To get the number of bytes on a line:

- check the QuickDraw global `screenBits.rowBytes`

To get the screen width:

- check the QuickDraw global `screenBits.bounds.right`

To do screen-size calculations:

- Use `LongInts`

Don't write to or read from `nil` Handles or `nil` Pointers

Don't create or Use Fake Handles

To avoid creating or using fake handles:

- Always let the Memory Manager perform operations with handles
- Never write code that assigns something to a master pointer

Don't write code that modifies itself

Self modifying code will not live across incarnations of the 68000

Think carefully about code designed strictly as copy protection

To avoid copy protection-related incompatibilities:

- Avoid copy protection altogether
- Rely on schemes that don't require specific hardware
- Make sure your scheme doesn't perform illegal operations

Don't ignore errors

To get valuable information:

- Check all pertinent calls for errors
- Always write defensive code

Don't access hardware directly

To avoid hardware-related incompatibilities:

- Don't read or write the hardware
- If you can't get the support from the ROM, ask the system where the hardware is
- Use low-memory globals

Don't use bits that are reserved

To avoid compatibility problems when bit status changes:

- Don't use undocumented stuff
- When using low-memory globals, check only what you want to know

Summary

Minor bugs are getting harder and harder to get away with:

- Good luck
- We'll help
- AppleLink: MacDTS, MCI: MacDTS
- U.S. Mail: 20525 Mariani Ave.; M/S 27-T; Cupertino, CA 95014

What it Is

The basic idea is to make sure that your programs will run, regardless of which Macintosh they are being run on. The current systems to be concerned with include:

- Macintosh 128K
- Macintosh 512K
- Macintosh XL
- Macintosh 512Ke
- Macintosh Plus
- Macintosh SE
- Macintosh II

If you perform operations in a generic fashion, there is rarely any reason to know what machine is running. This means that you should avoid writing code to determine which version of the machine you are running on, unless it is absolutely necessary.

For the purposes of this discussion, the term "programs" will be used to describe any code that runs on a Macintosh. This includes applications, INITs, FKEYs, Desk Accessories and Drivers.

What the "Rules" mean

Compatibility across all Macintosh computers (which may sound like it involves more work for you) may actually mean that you have less work to do, since it may not be necessary to revise your program each time Apple brings out a new computer or System file. Users, as a group, do not understand compatibility problems; all they see is that the program does not run on their system.

The benefits of being compatible are many-fold: your customers/users stay happy, you have less programming to do, you can devote your time to more valuable goals, there are fewer versions to deal with, your code will probably be more efficient, your users will not curse you under their breath, and your outlook on life will be much merrier.

Now that we know what being compatible is all about, recognize that nobody is requiring you to be compatible with anything. Apple does not employ roving gangs of thought police to be sure that developers are following the recommended guidelines. Furthermore, when the guidelines comprise 1200 pages of turgid prose (*Inside Macintosh*), you can be expected to miss one or two of the "rules." It is no sin to be incompatible, nor is it a punishable offense. If it were, there would be no Macintosh programs, since virtually all developers would be incarcerated. What it does mean, however, is that your program will be unfavorably viewed until it steps in line with the current system (which is a moving target). If a program becomes incompatible with a new Macintosh, it usually requires rethinking the offending code, and releasing a new version. You may read something like "If the developers followed Apple guidelines, they would be compatible with the transverse-hinged diatomic quark realignment system." This means that if you made any mistakes (you read all 1200 pages carefully, right?), you will not be compatible. It is extremely difficult to remain completely compatible, particularly in a system as complex as the Macintosh. The rules haven't changed, but what you can get away with has. There are, however, a number of things that you can do to improve your odds—some of which will be explained here.

It's your choice

It is still your choice whether you will be concerned with compatibility or not. Apple will not put out a warrant for your arrest. However, if you are doing things that are specifically illegal, Apple will also not worry about "breaking" your program.

Bad Things

The following list is not intended to be comprehensive, but these are the primary reasons why programs break from one version of the system to the next. These are the current top ten commandments:

I Thou shalt not assume the screen is a fixed size.
II Thou shalt not assume the screen is at a fixed location.
III Thou shalt not assume that `rowBytes` is equal to the width of the screen.
IV Thou shalt not use `nil` handles or `nil` pointers.
V Thou shalt not create or use fake handles.
VI Thou shalt not write code that modifies itself.
VII Thou shalt think twice about code designed strictly as copy protection.
VIII Thou shalt check errors returned as function results.
IX Thou shalt not access hardware directly.
X Thou shalt not use any of the bits that are reserved (unused means reserved).

This has been determined from extensive testing of our diverse software base.

Assuming the screen is a fixed size

Do not assume that the Macintosh screen is 512 x 342 pixels. Programs that do generally have problems on (or special case for) the Macintosh XL, which has a wider screen. Most applications have to create the bounding rectangle where a window can be dragged. This is the `boundsRect` that is passed to the call:

```
DragWindow (myWindowPtr, theEvent.where, boundsRect);
```

Some ill-advised programs create the `boundsRect` by something like:

```
SetRect (boundsRect, 0,0,342,512);  { oops, this is hard-coded…}
```

Why it's Bad

This is bad because it is **never** necessary to specifically put in the bounding rectangle for the screen. On a Macintosh XL for example, the screen size is 760x364 (and sometimes 608x431 with alternate hardware). If a program uses the hard-coded 0,0,342,512 as a bounding rectangle, end users will not be able to move their windows past the fictitious boundary of 512. If something similar were done to the `GrowWindow` call, it would make it impossible for users to grow their window to fill the entire screen. (Always a saddening waste of valuable screen real-estate.)

Assuming screen size makes it more difficult to use the program on Macintoshes with big screens, by making it difficult to grow or move windows, or by drawing in strange places where they should not be drawing (outside of windows). Consider the case of running on a Macintosh equipped with one of the full page displays, or Ultra-Large screens. No one who paid for a big screen wants to be restricted to using only the upper-left corner of it.

How to avoid becoming a screening fascist

Never hard code the numbers 512 and 342 for screen dimensions. You should avoid using constants for system values that can change. Parameters like these are nearly always available in a dynamic fashion. Programs should read the appropriate variables while the program is running (at run-time, not at compile time).

Here's how smart programs get the screen dimensions:

```
InitGraf(@thePort); { QuickDraw global variables have to be initialized.}
...
boundsRect := screenBits.bounds;    { The Real way to get screen size }
                                     { Use QuickDraw global variable. }
```

This is smart, because the program never has to know specifically what the numbers are. All references to rectangles that need to be related to the screen (like the drag and grow areas of windows) should use `screenBits.bounds` to avoid worrying about the screen size.

Note that this does not do anything remotely like assume that "if the computer is not a standard Macintosh, then it must be an XL." Special casing for the various versions of the Macintosh has always been suspicious at best; it is now grounds for breaking. (At least with respect to screen dimensions.)

By the way, remember to take into account the menu bar height when using this rectangle. On 128K ROMs (and later) you can use the low-memory global `mBarHeight` (a word at `$BAA`). But since we didn't provide a low-memory global for the menu bar height in the 64K ROMs, you'll have to hard code it to 20 (`$14`). (You're not the only ones to forget the future holds changes.)

How to find fascist screenism in current programs

The easiest way is to exercise your program on one of the Ultra-Large screen Macintoshes. There should be no restrictions on sizing or moving the windows, and all drawing should have no problems. If there are any anomalies in the program's usage, there is probably a lurking problem. Also, do a global find in the source code to see if the numbers 512 or 342 occur in the program. If so, and if they are in reference to the screen, excise them.

Assuming the screen is at a fixed location

Some programs use a fixed screen address, assuming that the screen location will be the same on various incarnations of the Macintosh. This is not the case. For example, the screen is located at memory location `$1A700` on a 128K Macintosh, at `$7A700` on a 512K Macintosh, at `$F8000` on the Macintosh XL, and at `$FA700` on the Macintosh Plus.

Why it's Bad

When a program relies upon the screen being in a fixed location, Murphy's Law dictates that an unknowing user will run it upon a computer with the screen in a different location. This usually causes the system to crash, since the offending program will write to memory that was used for something important. Programs that crash have been proven to be less useful than those that don't.

How to avoid being a base screener

Suffice it to say that there is no way that the address of the screen will remain static, but there are rare occasions where it is necessary to go directly to the screen memory. On these occasions, there are bad ways and not-as-bad ways to do it. A bad way:

```
myScreenBase := Pointer ($7A700);  { not good.  Hard-coded number. }
```

A not-as-bad way:

```
InitGraf(@thePort);    { do this only once in a program. }
...
myScreenBase := screenBits.baseAddr;  { Good.  Always works. }
                                  {Yet another QuickDraw global variable}
```

Using the latter approach is guaranteed to work, since QuickDraw has to know where to draw, and the operating system tells QuickDraw where the screen can be found. When in doubt, ask QuickDraw. This will work on Macintosh computers from now until forever, so if you use this approach you won't have to revise your program just because the screen moved in memory.

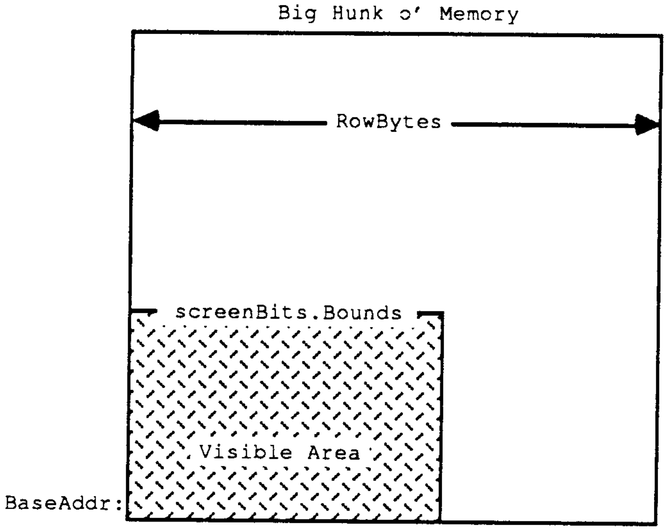
If you have a program (such as an INIT) that cannot rely upon QuickDraw being initialized (via `InitGraf`), then it is possible to use the `ScrnBase` low-memory global variable (a long word at `$824`). This method runs a distant second to asking QuickDraw, but is sometimes necessary.

How to find base screeners

The easiest way to find base screeners is to run the offending program on machines that have different screen addresses. If any addresses are being used in a base manner, the system will usually crash. The offending program may also occasionally refuse to draw. Some programs afflicted with this problem may also hang the computer (sometimes known as accessing funny space). Also, do a global find on the source code to look for numbers like `$7A700` or `$1A700`. When found, exercise caution while altering the offending lines.

Assuming that rowbytes is equal to the width of the screen

According to the definition of a bitMap found in *Inside Macintosh* (p I-144), you can see that rowBytes is the number of actual bytes in memory that are used to determine the bitMap. We know the screen is just a big hunk of memory, and we know that QuickDraw uses that memory as a bitMap. rowBytes accomplishes the translation of a big hunk of memory into a bitMap. To do this, rowBytes tells the system how long a given row is in memory and, more importantly, where in memory the next row starts. For conventional Macintoshes, rowBytes (bytes per Row) * 8 (Pixels per Byte) gives the final horizontal width of the screen as Pixels per Row. This does not have to be the case. It is possible to have a Macintosh screen where the rowBytes extends beyond what is actually visible on the screen. You can think of it as having the screen looking in on a larger bitMap. Diagrammatically, it might look like:



With an Ultra-Large screen, the number of bytes used for screen memory may be in the 500,000 byte range. Whenever calculations are being made to find various locations in the screen, the variables used should be able to handle larger screen sizes. For example, a 16 bit Integer will not be able to hold the 500,000 number, so a LongInt would be required. Do **not** assume that the screen size is 21,888 bytes long. bitMaps **can** be larger than 32K or 64K.

Why it's Bad

Programs that assume that all of the bytes in a row are visible may make bad calculations, causing drawing routines to produce unusual, and unreadable, results. Also, programs that use the rowBytes to figure out the width of the screen rectangle will find that their calculated rectangle is not the real screenBits.Bounds. Drawing into areas that are not visible will not necessarily crash the computer, but it will probably give erroneous results, and displays that don't match the normal output of the program.

Programs that assume that the number of bytes in the screen memory will be less than 32768 may have problems drawing into Ultra-Large screens, since those screens will often have more memory than a normal Macintosh screen. These particular problems do not evidence themselves by crashing the system. They generally appear as loss of

functionality (not being able to move a window to the bottom of the screen), or as drawing routines that no longer look correct. These problems can prevent an otherwise wonderful program from being used.

How to avoid being a row byter

In any calculations, the `rowBytes` variable should be thought of as the way to get to the next row on the screen. This is distinct from thinking of it as the width of the screen. The width should always be found from `screenBits.bounds.right-screenBits.bounds.left`.

It is also inappropriate to use the rectangle to decide how many bytes there are on a row. Programs that do something like:

```
bytesLine := screenBits.bounds.right DIV 8;  { bad use of bounds }
rightSide := screenBits.rowBytes * 8;       { bad use of rowBytes }
```

will find that the screen may have more `rowBytes` than previously thought. The best way to avoid being a row byter is to use the proper variables for the proper things. Without the proper mathematical basis to the screen, life becomes much more difficult. Always do things like:

```
bytesLine := screenBits.rowBytes;  { always the correct number }
rightSide := screenBits.bounds.right;  { always the correct screen size }
```

It is sometimes necessary to do calculations involving the screen. If so, be sure to use `LongInts` for all the math, and be sure to use the right variables (i.e. use `LongInts`). For example, if we need to find the address of the 500th row in the screen (500 lines from the top):

```
VAR  myAddress:  LongInt;
     myRow:      LongInt;      { so the calculations don't round off. }
     myOffset:   LongInt;      { could easily be over 32768 ... }
     bytesLine:  LongInt;

     ...
     myAddress := ord4(screenBits.baseAddr); {start w/the real base address }
     myRow := 500;                           {the row we want to address }
     bytesLine := screenBits.rowBytes;       {the real bytes per line }
     myOffset := myRow * bytesLine;          {lines * bytes per lines gives bytes }
     myAddress := myAddress + myOffset;      {final address of the 500th line }
```

This is not something you want to do if you can possibly avoid it, but if you simply must go directly to the screen, be careful. The big-screen machines (Ultra-Large screens) will thank you for it. If QuickDraw cannot be initialized, there is also the low-memory global `screenRow` (a word at `$106`) that will give you the current `rowBytes`.

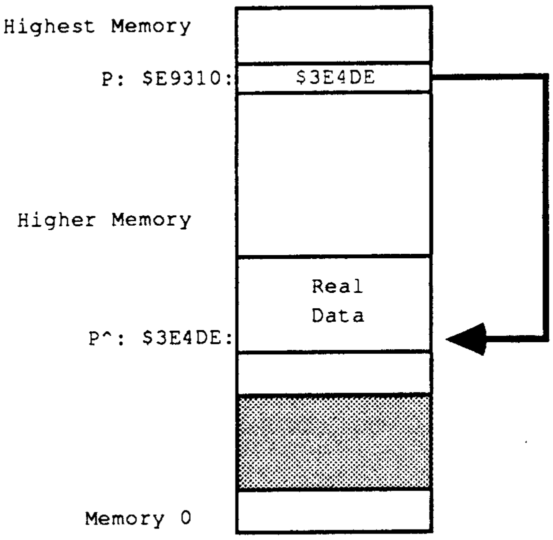
How to find row byters

To find current problems with row byter programs, run them on a machine equipped with Ultra-Large screens and see if any anomalies crop up. Look for drawing sequences that don't work right, and for drawing that clips to an imaginary edge. For source-level

inspection, look for uses of the `rowBytes` variables and be sure that they are being used in a mathematically sound fashion. Be highly suspicious of any code that uses `rowBytes` for the screen width. Any calculations involving those system variables should be closely inspected for round-off errors and improper use. Search for the number 8. If it is being used in a calculation where it is the number of bits per byte, then watch that code closely for improper conceptualization. This is code that could leap out and grab you by the throat at anytime. Be careful!

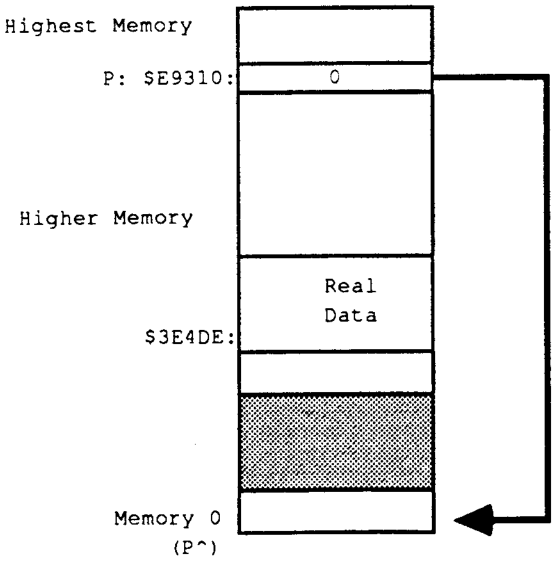
Using nil Handles or nil Pointers

A nil pointer is a pointer that has a value of 0. Recognize that pointers are merely addresses in memory. This means that a nil pointer is pointing to memory location 0. Any use of memory location 0 is strictly forbidden, since it is owned by Motorola. Trespassers may be shot on sight, but they may not die until much later. Sometimes trespassers are only wounded and act strangely. Any use of memory location 0 can be considered a bug, since there are **no** valid reasons for Macintosh programs to read or write to that memory. However, nil pointers themselves are not necessarily bad. It is occasionally necessary to pass nil pointers to ROM routines. This should not be confused with reading or writing to memory location 0. A pointer normally points to (contains the address of) a location in memory. It could look like this:



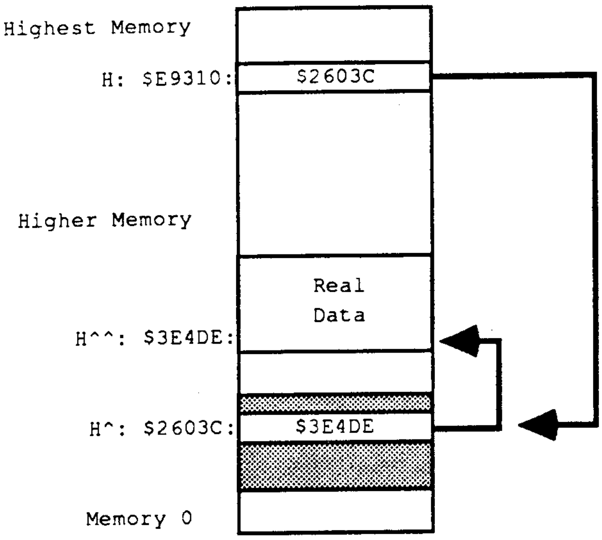
This is how a Pointer works. The address of the pointer variable itself is $E9310 (@P) and is four bytes long. The pointer points to (contains the address of) the block at $3E4DE (P). That memory location is where the actual data resides (P^).

If a pointer has been cleared to nil, it will point to memory location 0. This is OK as long as the program does not try to read from or write to that pointer. An example of a nil pointer could look like:



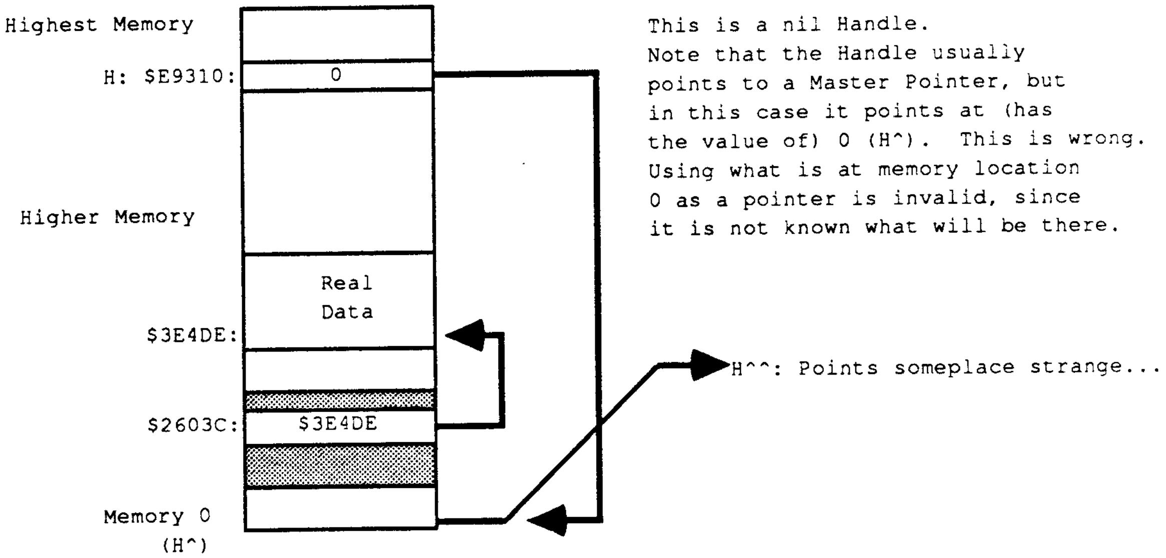
This is a nil Pointer. Note that the memory that it points to (the address) is 0 (P^). This is wrong. There is no valid data at memory location 0. Any writing to or reading from this pointer is a bug.

nil handles are related to the problem, since a handle is merely the address of a pointer (or a pointer to a pointer). An example of what a normal handle might look like is:



This is how a Handle works. The address of the handle variable itself (H) is $E9310. That variable points (has the address) to the master pointer at location $2603C (H). That variable is a pointer also, and points to the real data found at $3E4DE (H^^). The dark grey block is a Master pointer block. It is a group (usually 64) of Master Pointers. One of them is the Master Pointer at address $2603C (H^).

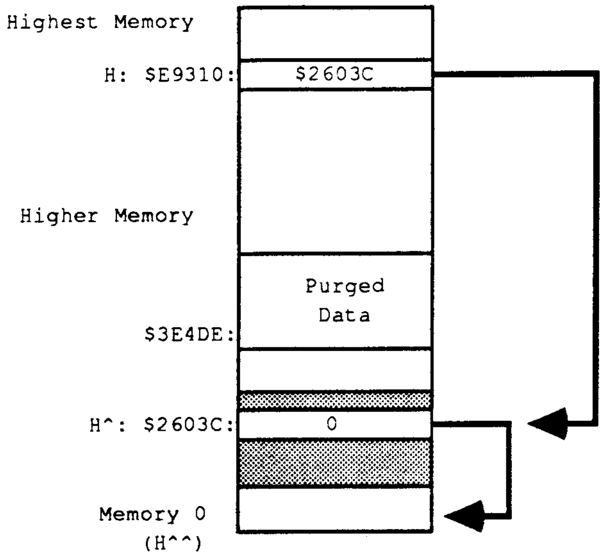
When the first pointer (h) becomes nil, that implies that memory location 0 can be used as a pointer. This is strictly illegal. There are **no** cases where it is valid to read from or write to a nil handle. A pictorial representation of what a nil handle could look like:



This is a nil Handle. Note that the Handle usually points to a Master Pointer, but in this case it points at (has the value of) 0 (H^). This is wrong. Using what is at memory location 0 as a pointer is invalid, since it is not known what will be there.

If the memory at 0 contains an odd number (numerically odd), then using it as a pointer will cause a system error with ID=2. This can be very useful, since that tells you exactly where the program is using this illegal handle, making it easy to fix. Unfortunately, there are cases where it is appropriate to pass a nil handle to ROM routines (such as GetScrap). These cases are rare, and it is **never** legal to read from or write to a nil handle.

There is also the case of an empty handle. An empty handle is one where the handle itself (the first pointer) points to a valid place in memory; that place in memory is also a pointer, and if it is `nil` the entire handle is termed empty. There are occasions where it is necessary to use the handle itself, but using the `nil` pointer that it contains is not valid. An example of an empty handle could be:



This is an Empty Handle. Note that the handle itself has a valid Master Pointer address in it $2603C (H^). The Master Pointer is nil however, which is the address of location 0 in memory. It is wrong to use the Master Pointer in this case, although there are cases where using the Handle itself is valid.

Fundamentally, any reading or writing to memory using a pointer or handle that is `nil` is punishable by death (of your program).

Why it's Bad

The use of `nil` pointers can lead to the use of make-believe data. This make-believe data often changes for different versions of the computer. This changing data makes it difficult to predict what will happen when a program uses `nil` pointers. Programs may not crash as a result of using a `nil` pointer, and they may behave in a consistent fashion. This does not mean that there isn't a bug. This merely means that the program is lucky, and that it should be playing the lottery, not running on a Macintosh. If a program acts differently on different versions of the Macintosh, you should think "could there be a nasty `nil` pointer problem here?" Use of a `nil` handle usually culminates in reading or writing to obscure places in memory. As an example:

```
VAR   myHandle:  TEHandle;

myHandle := nil;
```

That's pretty straightforward, so what's the problem? If you do something like:

```
myHandle^^.viewRect := myRect;  { very bad idea with myHandle = nil }
```

memory location zero will be used as a pointer to give the address of a TextEdit record. What if that memory location points to something in the system heap? What if it points to the sound buffer? In cases like these, eight bytes of rectangle data will be written to wherever memory location 0 points.

Use of a nil handle will never be useful. This memory is reserved and used by the 68000 for various interrupt vectors and Valuable Stuff. This Valuable Stuff is composed of things that you definitely do not want to change. When changed, the 68000 finds out, and decides to get back at your program in the most strange and wonderful ways. These strange results can range from a System Error all the way to erasing hard disks and destroying files. There really is no limit to the havoc that can be wreaked. This tends to keep the users on the edge of their seat, but this is not really the desired effect. As noted above, it won't necessarily cause traumatic results. A program can be doing naughty things and not get caught. This is still a bug that needs to be fixed, since it is nearly guaranteed to give different results on different versions of the Macintosh. Programs exhibiting schizophrenia have been proven to be less enjoyable to use.

How to avoid being a Niller

Whenever a program uses pointers and handles, it should ensure that the pointer or handle will not be nil. This could be termed defensive programming, since it assumes that everyone is out to get the program (which is not far from the truth on the Macintosh). You should always check the result of routines that claim to pass back a handle. If they pass you back a nil handle, you could get in trouble if you use them. Don't trust the ROM. The following example of a defensive use of a handle involves the Resource Manager. The Resource Manager passes back a handle to the resource data. There are any number of places where it may be forced to pass back a nil handle. For example:

```
VAR   myRezzie:  MyHandle;

myRezzie := MyHandle(GetResource(myResType, myResNumber)); { could be missing…}
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;                { We know it is OK }
```

As another example, think of how handles can be purged from memory in tight memory conditions. If a block is marked purgeable, the Memory Manager may throw it away at any time. This creates an empty handle. The defensive programmer will always make sure that the handles being used are not empty.

```
    VAR   myRezzie:  myHandle;

    myRezzie := myHandle(GetResource(myResType, myResNumber));  { could be
                                                                    missing… }
    IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
    ELSE  myRezzie^^.myRect := newRect;     { We know it is OK }
    tempHandle := NewHandle (largeBlock);  {might dispose a purgeable myRezzie}
    IF myRezzie^ = nil  THEN LoadResource(Handle(myRezzie)); {Re-load empty
                                                                  handle}
    IF ResError = noErr  THEN
        myRezzie^^.StatusField := OK;      { guaranteed not empty, and actually
                                            gets read back in, if necessary }
```

Be especially careful of places where memory is being allocated. The NewHandle and NewPtr calls will return a nil handle or pointer if there is not enough memory. If you use that handle or pointer without checking, you will be guilty of being a Niller.

How to find Nillers

The best way to find these nasty `nil` pointer problems is to set memory location zero to be an **odd** number (a good choice is 'NIL!' = `$4E494C21`, which is numerically odd, as well as personality-wise). Please see Technical Note #7 for details on how to do this.

If you use TMON, you can use the extended user area with Discipline. Discipline will set memory location 0 to 'NIL!' to help catch those nasty pointer problems. If you use Macsbug, just type `SM 0 'NIL!` and go. Realize of course, that if a program has made a transgression and is actually using `nil` pointers, this may make the program crash with an ID=2 system error. This is good! This means that you have found a bug that may have been causing you untold grief. Once you know where a program crashes, it is usually very easy to use a debugger to find where the error is in the source code. When the program is compiled, turn on the debugging labels (usually a $D+ option). Set memory location 0 to be 'NIL!'. When the program crashes, look at where the program is executing and see what routine it was in (from a disassembly). Go back to that routine in the source code and remove the offending code with a grim smile on your face. Another scurvy bug has been vanquished. The intoxicating smell of victory wafts around your head.

Another way to find problems is to use a debugger to do a checksum on the first four bytes in memory (from 0 to 3 inclusive). If the program ever traps into the debugger claiming that the memory changed, see which part of the program altered memory location 0. Any code that writes to memory location zero is guilty of high treason against the state and must be removed. Remember to say, “bugs are not my friends.”

Creating or Using Fake Handles

A fake handle is one that was not manufactured by the system, but was created by the program itself. An example of a fake handle is:

```
CONST aMem = $100;
VAR   myHandle: Handle;
      myPointer: Ptr;

myPointer := Ptr (aMem);      { the address of some memory }
myHandle  := @myPointer;      {the address of the pointer variable. Very bad.}
```

The normal way to create and use handles is to call the Memory Manager NewHandle function.

Why it's Bad

A handle that is manufactured by the program is not a legitimate handle as far as the operating system is concerned. Passing a fake handle to routines that use handles is a good way to discover the meaning of "Death by ROM." For example, think how confused the operating system would get if the fake handle were passed to DisposHandle. What would it dispose? It never allocated the memory, so how can it release it? Programs that manufacture handles may find that the operating system is no longer their friend.

When handles are passed to various ROM routines, there is no telling what sorts of things will be done to the handle. There are any number of normal handle manipulation calls that the ROM may use, such as SetHandleSize, HLock, HNoPurge, MoveHHi and so on. Since a program cannot guarantee that the ROM will not be doing things like this to handles that the program passes in, it is wise to make sure that a real handle is being used, so that all these type of operations will work as the ROM expects. For fake handles, the calls like HLock and SetHandleSize have no bearing. Fake handles are very easy to create, and they are very bad for the health of otherwise upstanding programs. Whenever you need a handle, get one from the Memory Manager.

As a particularly bad use of a fake handle:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

myHandle := NewHandle (SIZEOF(myStuff));    { create a new normal handle }
myHandle^ := @myStuff;  {YOW!  Intended to make myHandle a handle to
                           the myStuff record.  What it really does is
                           blow up a Master Pointer block, Heap corruption,
                           and death by Bad Heap.  Never do this. }
```

This can be a little confusing, since it is fine to use your own pointers, but very bad to use your own handles. The difference is that handles can move in memory, and pointers cannot, hence the pointers are not dangerous. This does not mean you should use pointers for everything since that causes other problems. It merely means that you have to be careful how you use the handles.

The use of fake handles usually causes system errors, but can be somewhat mysterious in its effects. Fake handles can be particularly hard to track down since they often cause damage that is not uncovered for many minutes of use. Any use of fake handles that causes the heap to be altered will usually crash the system. Heap corruption is a common failure mode. In clinical studies, 9 out of 10 programmers recommend uncorrupted heaps to their users who use heaps.

How to avoid being a fakir

The correct way to make a handle to some data is to make a copy of the data:

```
VAR    myHandle:  Handle;
       myStuff:  myRecord;

errCode := PtrToHand (@myStuff, myHandle, SIZEOF(myStuff));
IF  errCode <> noErr  THEN ErrorHandler ('Out of memory');
```

Always, always, let the Memory Manager perform operations with handles. Never write code that assigns something to a master pointer, like:

```
VAR   myDeath:  Handle;
myDeath^ := stuff;  { Don't change the Master pointer. }
```

If there is code like this, it usually means the heap is being corrupted, or a fake handle is being used. It is, however, OK to pass around the handle itself, like:

```
myCopyHandle := myHandle;    { perfectly OK, nobody will yell about this. }
```

This is far different than using the ^ operator to accidentally modify things in the system. Whenever it is necessary to write code to use handles, be careful. Watch things carefully as they are being written. It is much easier to be careful on the way in than it is to try to find out why something is crashing. Be very careful of the @ operator. This operator can unleash untold problems upon unsuspecting programs. If at all possible, try to avoid using it, but if it is necessary, be absolutely sure you know what it is doing. It is particularly dangerous since it turns off the normal type checking that can help you find errors (in Pascal). In short, don't get crazy with pointer and handle manipulations, and they won't get crazy with you.

How to find fakirs

Problems of this form are particularly insidious because it can be very difficult to find them after they have been created. They tend to not crash immediately, but rather to crash sometime long after the real damage has been done. The best way to find these problems is to run the program with Discipline. (Discipline is a programmer's tool that will check all parameters passed to the ROM to see if they are legitimate. Discipline can be found as a stand-alone tool, but the most up-to-date version will be found in the Extended User Area for the TMON debugger. The User Area is public domain, but TMON itself is not. TMON has a number of other useful features, and is well worth the price.) Discipline will check handles that are passed to the ROM to see if they are real handles or not, and if not, will stop the program at the offending call. This can lead you back to the source at a point that may be close to where the bad handle was created. If a program passes the Discipline test, it will be a healthy, robust program with drastically improved odds for compatibility. Programs that do not pass Discipline can sleep poorly at night, knowing that they have broken at least one or two of the "rules."

A way to find programs that are damaging the heap is to use a debugger (TMON or Macsbug) and turn on the Heap Check operation. This will check the heap for errors at each trap call, and if the heap is corrupted will break into the debugger. Hopefully this will be close to where the code is that caused the damage. Unfortunately, it may not be close enough; this will force you to look further back.

Looking in the source code, look for all uses of the `@` operator, and examine the code carefully to see if it is breaking the rules. If it is, change it to step in line with the rest of the happy programs here in happy valley. Also, look for any code that changes a master pointer like the `myHandle^ := stuff`. Any code of this form is highly suspect, and probably a member of the Anti-Productivity League. The APL has been accused of preventing software sales and the rise of the Yen. These problems can be quite difficult to find at times, but don't give up. These fake handles are high on the list of guilty parties, and should never be trusted.

Writing code that modifies itself

Self-modifying code is software that changes itself. Code that alters itself runs into two main groupings: code that modifies the code itself and code that changes the block the code is stored in. Copy protection code often modifies the code itself, to change the way it operates (concealing the meaning of what the code does). Changing the code itself is very tricky, and also prone to having problems, particularly when the microprocessor itself changes. There are third-party upgrades available that add a 68020 to a Macintosh. Because of the 68020's cache, programs that modify themselves stand a good chance of having problems when run on a 68020. This is a compatibility point that should not be missed (nudge, nudge, wink, wink). Code that changes other code (or itself) is prone to be incompatible when the microprocessor changes.

The second group is code that changes the block that the code is stored in. Keeping variables in the CODE segment itself is an example of this. This is uncommon with high-level languages, but it is easy to do in assembly language (using the DC directive). Variables defined in the code itself should be read-only (constants). Code that modifies itself has signed a tacit agreement that says "I'm being tricky, if I die, I'll revise it."

Why it's Bad

There are now three different versions of the microprocessor, the 68000, 68010, and the 68020. They are intended to be compatible with each other, but may not be compatible with code that modifies itself. As the Macintosh evolves, the system may have compatibility problems with programs that try to "push the envelope."

How to avoid being an abuser

Well, the obvious answer is to avoid writing self-modifying code. If you feel obliged to write self-modifying code, then you are taking an oath to not complain when you break in the future. But don't worry about accidentally taking the oath: you won't do it without knowing it. If you choose to abuse, you also agree to personal visits from the Apple thought police, who will be hired as soon as we find out.

How to find abusers

Run the program on a 68020 system. If it fails, it could be related to this problem, but since there are other bugs that might cause failures, it is not guaranteed to be a self-modifying code problem. Self-modifying code is often used in copy protection, which brings us to the next big topic.

Code designed strictly as copy protection

Copy protection is used to make it difficult to make copies of a program. The basic premise is to make it impossible to copy a program with the Finder. This will not be a discussion as to the pros and cons of copy protection. Everyone has an opinion. This will be a description of reality, as it relates to compatibility.

Why it's Bad

System changes will never be made merely to cause copy protection schemes to fail, but given the choice between improving the system and making a copy protection scheme remain compatible, the system improvement will always be chosen.

- Copy protection is number one on the list of why programs fail the compatibility test.
- Copy protection by its very nature tends to do the most "illegal" things.
- Programs that are copy protected are assumed to have signed a tacit agreement to revise the program when the system changes.

Copy protection itself is not necessarily bad. What is bad is when programs that would otherwise be fully compatible do not work due only to the copy protection. This is very sad, since it requires extra work, revisions to the software, and time lost while the revision is being produced. The users are not generally humored when they can no longer use their programs. Copy protection schemes that fail generally cause system errors when they are run. They also can refuse to run when they should.

How to avoid being a protectionist

The simple answer is to do without copy protection altogether. If you think of compatibility as a probability game, if you leave out the copy protection, your odds of winning skyrocket. As noted above, copy protection is the single biggest reason why programs fail on the various versions of the Macintosh. For those who are required to use copy protection, try to rely on schemes that do not require specific hardware and make sure that the scheme used is not performing illegal operations. If a program runs, an experienced Macintosh programmer armed with a debugger can probably make a copy of it, (no matter how sophisticated the copy protection scheme) so a moderate scheme that does not break the rules is probably a better compatibility bet. The trickier and more devious the scheme, the higher the chance of breaking a rule. Tread lightly.

How to find protectionists

The easiest way to see if a scheme is being overly tricky is to run it on a Macintosh XL. Since the floppy disk hardware is different this will usually demonstrate an unwanted hardware dependency. Be wary of schemes that don't allow installation on a hard disk. If the program cannot be installed on a hard disk, it may be relying upon things that are prone to change. Don't use schemes that access the hardware directly. All Macintosh software should go through the various managers in the ROM to maintain compatibility. Any code that sidesteps the ROM will be viewed as having said "It's OK to make me revise myself."

Check errors returned as function results

All of the Operating System functions, as well as some of the Toolbox functions, will return result codes as the value of the function. Don't ignore these result codes. If a program ignores the result codes, it is possible to have any number of bad things happen to the program. The result code is there to tell the program that something went wrong; if the program ignores the fact that something is wrong, that program will probably be killed by whatever went wrong. (Bugs do not like to be ignored.) If a program checks errors, an anomaly can be nipped in the bud, before something really bizarre happens.

Why it's Bad

A program that ignores result codes is skipping valuable information. This information can often prevent a program from crashing and keep it from losing data.

How to avoid becoming a skipper

Always write code that is defensive. Assume that everyone and everything is out to kill you. Trust no one. An example of error checking is:

```
    myRezzie := GetResource (myResType, myResId);
    IF  myRezzie = nil  THEN  ErrorHandler ('Who stole my resource...');
```

Another example:

```
    fsErrCode := FSOpen ('MyFile', myVRefNum, myFileRefNum);
    IF fsErrCode <> noErr  THEN ErrorHandler (fsErrCode, 'File error');
```

And another:

```
    myTPPrPort := PrOpenDoc (myTHPrint, nil, nil);
    IF  PRError <> noErr  THEN  ErrorHandler (PRError, 'Printing error');
```

Any use of Operating System functions should presume that something nasty can happen, and have code to handle the nasty situations. Printing calls, File Manager calls, Resource Manager calls, and Memory Manager calls are all examples of Operating System functions that should be watched for returning errors. Always, always check the result codes from Memory Manager calls. Big memory machines are pretty common now, and it is easy to get cavalier about memory, but realize that someone will always want to run the program under Switcher, or on smaller Macintoshes. It never hurts to check, and always hurts to ignore it.

How to find skippers

This is easy: just do weird things while the program is running. Put in locked or unformatted disks while the program is running. Use unconventional command sequences. Run out of disk space. Run on 128K Macintoshes to see how the program deals with running out of memory. Run under Switcher for the same reason. (Programs that die while running under Switcher are often not Switcher's fault, and are in fact due

to faulty memory management.) Print with no printer connected to the Macintosh. Pop disks out of the drives with the Command-Shift sequence, and see if the program can deal with no disk. When a disk-switch dialog comes up, press Command-period to pass back an error to the requesting program (128K ROMs only). Torturing otherwise well-behaved programs can be quite enjoyable, and a number of users enjoy torturing the program as much as the program enjoys torturing them. For the truly malicious, run the debugger and alter error codes as they come back from various routines. Sure it's a dirty low-down rotten thing to do to a program, but we want to see how far we can push the program. (This is also a good way to check your error handling.) It's one thing to be an optimist, but it's quite another to assume that nothing will go wrong while a program is running.

Accessing hardware directly

Sometimes it is necessary to go directly to the Macintosh hardware to accomplish a specific task for which there is no ROM support. Early hard disks that used the serial ports had no ROM support. Those disks needed to use the SCC chip (the 8530 communication chip) in a high-speed clocked fashion. Although it is a valid function, it is not something that is supported in the ROM. It was therefore necessary to go play with the SCC chip directly, setting and testing various hardware registers in the chip itself. Another example of a valid function that has no ROM support is the use of the alternate video page for page-flipping animation. Since there is no ROM call to flip pages, it is necessary to go play with the right bit in the VIA chip (6522 Versatile Interface Adapter). Going directly to the hardware does not automatically throw a program into the incompatible group, but it certainly lowers its odds.

Why it's bad

Going directly to the hardware poses any number of problems for enlightened programs that are trying to maintain compatibility across the various versions of the Macintosh. On the Macintosh XL for example, a lot of the hardware is found in different locations, and in some cases the hardware doesn't exist. On the XL there is no sound chip. Programs that go directly to the sound hardware will find they don't work correctly on an XL. If the same program were to go through the Sound Manager, it would work fine, although the sound would not be the same as expected. Since the Macintosh is heavily oriented to the software side of things, expecting various hardware to always be available is not a safe bet. Choosy programmers choose to leave the hardware to the ROM.

How to avoid having a hard attack

Don't read or write the hardware. Exhaust every possible conventional approach before deciding to really get down and dirty. If there is a Manager in the ROM for the operation you wish to perform, it is far better to use the Manager than to go directly to the hardware. Compatibility at the hardware level can very rarely be maintained, but compatibility at the Manager level is a prime consideration. If a program is down to the last ditch effort, and cannot get the support from the ROM that is desired, then access the hardware in an enlightened approach. The really bad way to do it:

```
VIA := Pointer ($EFE1FE);  { sure it's the base address today...}
                            { This is bad.  Hard-coded number. }
```

The with-it, inspired programmer of the eighties does something like:

```
TYPE LongPointer = ^LongInt;

VAR  VIA: LongPointer;
     VIABase: LongInt;

VIA := Pointer ($1D4);  { the address of the low-memory global. }
VIABase := VIA^;        { get the low-memory variable's value }
                       { Now VIABase has the address of the chip }
```

The point here is that the best way to get the address of a hardware chip is to ask the system where it currently is to be found. The system always knows where the pieces of the system are, and will always know for every incarnation of the Macintosh. There are low-memory global variables for all of the pieces of hardware currently found in the Macintosh. This includes the VIA, the SCC, the Sound Chip, the IWM, and the video display. Whenever you are stuck with going to the hardware, use the low-memory globals. The fact that a program goes directly to the hardware means that it is risking imminent incompatibility, but using the low-memory global will ensure that the program has the best odds. It's like going to Las Vegas: if you don't gamble at all, you don't lose any money; if you have to gamble, play the game that you lose the least on.

How to find hard attacks

Run the suspicious program on the Macintosh XL. Nearly all of the hardware is in a different memory location on the XL. If a program has a hard-coded hardware address in it, it will fail. It may crash, or it might not perform the desired task, but it won't work as advertised. This unfortunately, is not a completely legitimate test, since the XL does not have some of the hardware of other Macintoshes, and some of the hardware that is there has the register mapping different. This means that it is possible to play by the rule of using the low-memory global and still be incompatible.

Don't use bits that are reserved

Occasionally during the life of a Macintosh programmer, there comes a time when it is necessary to bite the bullet and use a low-memory global. These are very sad days, since it has been demonstrated (by history) that low-memory global variables are a mysterious lot, and not altogether friendly. One fellow in particular is known as ROM85, a word located at $28E. This particular variable has been documented as the way to determine if a program is running on the 128K ROMs or not. Notably, the top most bit of that word is the determining bit. This means that the rest of the bits in that word are reserved, since nothing is described about any further bits. Remember, if it doesn't say, assume it's reserved. If it's reserved, don't depend upon it. Take the cautious way out and assume that the other bits that aren't documented are used for Switcher local variables, or something equally wild. An example of a bad way to do the comparison is:

```
VAR  Rom85Ptr: WordPtr;
     RomsAre64: Boolean;

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF  Rom85Ptr^ = $7FFF  THEN  RomsAre64 := False  { Bad test. }
ELSE  RomsAre64 := True;
```

This is a bad test since the comparison is testing the value of **all** of the bits, not only the one that is valid. Since the other bits are undocumented, it is impossible to know what they are used for. Assume they are used for something that is arbitrarily random, and take the safe way out.

How to avoid being bitten

```
VAR     ROM85Ptr: Ptr

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF BitTst(ROM85Ptr,0) THEN RomsAre64 := True {Good--tests only hi-bit}
ELSE  RomsAre64 := False;
```

This technique will ensure that when those bits are documented, your program won't be using them for the wrong things. Beware of trojan bits.

Don't use undocumented stuff. Be very careful when you use anything out of the ordinary stream of a high-level language. For instance, in the ROM85 case, it is very easy to make the mistake of checking for an absolute value instead of testing the actual bit that encodes the information. Whenever a program is using low-memory globals, be sure that only the information desired is being used, and not some undocumented (and hence reserved) bits. It's not always easy to determine what is reserved and what isn't, so conservative programmers always use as little as possible. Be wary of the strange bits, and accept rides from none of them. The ride you take might cause you to revise your program.

How to find those bitten

Since there are such a multitude of possible places to get killed, there is no simple way to see what programs are using illegal bits. As time goes by it will be possible to find more of these cases by running on various versions of the Macintosh, but there will probably never be a comprehensive way of finding out who is accepting strange rides, and who is not. Whenever the use of a bit changes from reserved status to active, it will be possible to find those bugs via extensive testing. From a source level, it would be advisable to look over **any** use of low-memory globals, and eye them closely for inappropriate bit usage. Do a global search for the $ (which describes those ubiquitous hexadecimal numbers), and when found see if the use of the number is appropriate. Trust no one that is not known. If they are documented, they will stay where they are, and have the same meaning. Be very careful in realms that are undocumented. Bits that suddenly jump from reserved to active status have been known to cause more than one program to have a sudden anxiety attack. It is very unnerving to watch a program go from calm and reassuring to rabid status. Users have been known to drop their keyboards in sudden shock (which is bad on the keyboards).

Summary

So what does all this mean? It means that it is getting harder and harder to get away with minor bugs in programs. The minor bugs of yesterday are the major ones of today. No one will yell at you for having bugs in your program, since all programs have bugs of one form or another. The goal should be to make the programs run as smoothly and effortlessly as possible. The end-users will never object to bug-reduced programs.

What is the best way to test a program? A reasonably comprehensive test is to exercise all of the program's functions under the following situations:

- Use Discipline to be sure the program does not pass illegal things to the ROM.
- Use heap scramble and heap purge to be sure that handles are being used correctly, and that the memory management of the program is correct.
- Run with a checksum on memory locations 0...3 to see if the program writes to these locations.
- Run on a 128K Macintosh, or under Switcher with a small partition, to see how the program deals with memory-critical situations.
- Run on a 68020 system to see if the program is 68020-compatible and to make sure that changing system speed won't confuse the program.
- Run on a Macintosh XL to be sure that the program does not assume too much about the operating system, and to test screen handling.
- Run on an Ultra-Large screen to be sure that the screen handling is correct, and that there are no hard-coded screen dimensions.
- Run on 64K ROM machines to be sure new traps are not being used when they don't exist.
- Run under both HFS and MFS to be sure that the program deals with the file system correctly. (400K floppies are usually MFS.)

If a program can live through all of this with no Discipline traps, no checksum breaks, no system errors, no anomalies, no data loss and still get useful work done, then you deserve a gold medal for programming excellence. Maybe even an extra medal for conduct above and beyond the call of duty. In any case, you will know that you have done your job about as well as it can be done, with today's version of the rules, and today's programming tools.

Sounds like a foreboding task, doesn't it? The engineers in Macintosh Technical Support are available to help you with compatibility issues (we won't always be able to talk about new products, since we love our jobs, but we can give you some hints about compatibility with what the future holds).

Good luck.

Macintosh Technical Notes

#118: How to Check and Handle Printing Errors

See also: The Printing Manager

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

This technical note describes how to check and properly handle errors that occur during printing with the high-level printing calls.

Most people are aware of the need for checking File Manager errors, Resource Manager errors, and the like, but sometimes Printing Manager errors get neglected; you should always check for error conditions while printing. This can be done by calling `PrError`. Errors returned by `PrError` will include any Printing Manager errors (and some AppleTalk and OS errors) that occur during printing.

The best place to start is with the code fragment on page 155 of*Inside Macintosh*, vol. II:

```
myPrPort := PrOpenDoc (prRecHdl, NIL, NIL);  {open printing grafPort}
FOR pg := 1 TO myPgCount DO                  {page loop: ALL pages of document}
  IF PrError = noErr THEN
     BEGIN
     PrOpenPage(myPrPort,NIL);               {start new page}
     IF PrError = noErr THEN
        MyDrawingProc(pg);                   {draw page with QuickDraw}
     PrClosePage(myPrPort);                  {end current page}
     END;
PrCloseDoc(myPrPort);
IF prRecHdl^^.prJob.bJDocLoop = bSpoolLoop AND PrError = noErr THEN
     BEGIN
     MySwapOutProc;                          {swap out code and data}
     PrPicFile(prRecHdl,NIL,NIL,NIL,myStRec);  {print spooled document}
     END;
IF PrError <> noErr THEN MyPrErrAlertProc;   {report any errors}
```

Here are some error-handling guidelines:

- You should avoid calling `PrError` within your `PrIdle` procedure; errors that occur while it is executing are usually temporary and serve only as internal flags for communication within the printer driver—they are not intended for the application. If you absolutely must call `PrError` within your idle procedure, and an error occurs, never abort printing within the idle procedure itself. Wait until the last called printing procedure returns and then check to see if the error still remains. Attempting to abort printing within an idle procedure is a guarantee of certain death.

- If you detect that an error has occurred after the completion of a printing routine, just stop where you are, i. e. stop drawing. Proceed to the next print procedure to close any open calls you have made. For example, if you called `PrOpenDoc` and received an error, skip to the next `PrCloseDoc`. Or if you called `PrOpenPage` and got an error, skip to the next `PrClosePage` and `PrCloseDoc`. Remember that if some `PrOpen…` procedure has been called, then you must call the corresponding `PrClose…` procedure to ensure that printing closes properly and that all temporary memory allocations are released and returned to the heap.

- Do not raise any alerts or dialogs to report an error until the end of the print loop. At the end of the print loop, check for the error again; if there is no error assume that printing completed normally. If it's still there, you can raise an alert.

  This is important for two reasons. First, if an alert is raised in the middle of the print loop, it can cause errors that will terminate an otherwise normal job. For example, if the printer is an AppleTalk printer, the connection can be terminated abnormally. While your alert is sitting there waiting for a response from the user, the driver is unable to respond to AppleTalk requests coming in from the printer. If the printer doesn't hear from the Macintosh within a short time period (30 seconds) then it will timeout, assuming that the Macintosh is no longer there. This results in the connection being broken prematurely causing another error that the application has to respond to.

  The driver may also have already put up its own alert in response to the error. In this instance, the driver will post an error to let your application know that something went wrong and that it's time to abort printing. For example, when the driver detects that the version of Laser Prep that has been downloaded to the LaserWriter is different from the version that the user is trying to print with, the LaserWriter driver raises the appropriate alert telling the user that the printer was initialized with an incompatible version of the driver and gives the option of reinitializing. If the user chooses to cancel, the driver posts an error to let the application know that it needs to abort, but since the driver has already taken care of the error by putting up an alert, the error is reset to zero before the printing loop is complete. The application should check for the error again at the end of the printing loop and if it still indicates an error, it should raise an alert.

#119: Determining If Color QuickDraw Exists

See: Technical Note #129—SysEnvirons

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

This note formely described a way to determine if Color QuickDraw is present on a particular machine. We now recommend that you call `SysEnvirons` to find out, as described in Technical Note #129.

Macintosh
Technical Notes

Developer Technical Support

#120: Drawing Into an Off-Screen Pixel Map

Revised by: Rich Collyer — April 1989
Written by: Jim Friedlander & Rick Blair — May 1987

This Technical Note provides a simple example of drawing to, then copying from, an off-screen pixel map.

Changes since October 1988: Made changes to the code which convert `GDevice` color look-up tables (clut) to pixel map color look-up tables so `_CopyBits` will copy the color information correctly. This information is especially important for color printing.

The following example demonstrates how to draw something in an off-screen pixel map, and then use `_CopyBits` to copy it back to the screen. It handles the case of multiple screens with different pixel depths. Before making any calls to Color QuickDraw, you must make sure it is present (refer to Technical Note #129, _SysEnvirons: System 6.0 and Beyond).

MPW Pascal

```
CONST
       OffLeft        = 10;
       OffTop         = 30;
       OffBottom      = 250;
       OffRight       = 400;

       {These constants for the bounds of the off-screen PixMap are chosen because we
       know what the extent of the drawing will be and we want to restrict the size of
       the map as much as possible.}

TYPE
       BitMapPtr      = ^BitMap;      {for type coercion in the _CopyBits call}

VAR
       offRowBytes    : LONGINT;
       sizeOfOff      : LONGINT;
       myBits         : Ptr;
       destRect       : Rect;
       globRect       : Rect;
       bRect          : Rect;
       theDepth       : INTEGER;
       i              : INTEGER;
       err            : INTEGER;
       myCGrafPort    : CGrafPort;
       myCGrafPtr     : CGrafPtr;
       ourCMHandle    : CTabHandle;
       theMaxDevice   : GDHandle;
       oldDevice      : GDHandle;
```

First you create a color window, then you need to determine the device with the maximum depth to which you will copy the off-screen image with `_CopyBits`.

```
myCWindow := GetNewCWindow(SomeID,NIL,WindowPtr(-1));

SetPort(myCWindow); {set to this port for the localToGlobals that follow}

SetRect(bRect,OffLeft,OffTop,OffRight,OffBottom);
IF NOT SectRect(myCWindow^.portRect,bRect,globRect) THEN
       NothingToCopy;  {nothing to do, clean up and EXIT}

{still here, so let's convert to globals}
LocalToGlobal(globRect.topLeft);
LocalToGlobal(globRect.botRight);

{figure out how much space we need for our pixel image.
we will call GetMaxDevice and get the pixel map from that --
we do this to cover the case where the pixel image that we wish
to CopyBits to spans multiple devices (of possibly different depths)}

theMaxDevice:= GetMaxDevice(globRect);{get the maxDevice}
```

You need to set `theGDevice` to the device with the maximum pixel depth (the one you found in the last step), so the pixel map of the new `CGrafPort` will be copied from one of the proper depth. Now you should open a new `CGrafPort` to use for your off-screen drawing.

```
oldDevice := GetGDevice;       {save theGDevice so we can restore it later}
SetGDevice(theMaxDevice);      {Set to the maxdevice}

myCGrafPtr := @myCGrafPort;    {initialize this guy}
OpenCPort(myCGrafPtr);         {open a new color port - this calls InitCPort}
theDepth:= myCGrafPtr^.portPixMap^^.pixelSize;
```

You are now ready to calculate the size of the pixel image you will need, then you can set the location-specific and size-specific information of the pixel map. Since Color QuickDraw distinguishes between a bitmap and a pixel map by checking the high bit of `rowBytes`, you need to add $8000 to `OffRowBytes` as shown.

```
{similar formula to Technical Note #41, except we must include pixel depth}
offRowBytes := ((((theDepth * (OffRight - OffLeft)) + 15)) DIV 16) * 2;
{make sure LONGINT math is done on the next line!}
sizeOfOff := LONGINT(OffBottom - OffTop) * offRowBytes;
OffSetRect(bRect, - OffLeft, - OffTop);     {adjust for local coordinates}

{Set up baseAddr, rowBytes,bounds and pixelSize of the PixMap in our fresh, new CPort}
myBits := NewPtr(sizeOfOff);                {allocate space for the pixel image}
{real programs do error checking here}

WITH myCGrafPtr^.portPixMap^^ DO BEGIN
       baseAddr := myBits;
       rowbytes := offRowBytes + $8000;     {remember to be a PixMap}
       bounds := bRect;
END;                                        {with}
```

Next you can clone the color table of the `maxDevice` and put it into your off-screen pixel map.

```
ourCMHandle := theMaxDevice^^.gdPMap^^.pmTable;
err := HandToHand(Handle(ourCMHandle));     {clone it}
{real programs do error checking here}
FOR i := 0 TO ourCMHandle^^.ctSize DO
       ourCMHandle^^.ctTable[i].value := i;
ourCMHandle^^.ctFlags := BAnd (ourCMHandle^^.ctFlags ,$7fff);
ourCMHandle^^.ctSeed := GetCTSeed();
{ This code is necessary for converting GDevice cluts to Pixmap cluts }

{put the cloned, correctly set-up Color Table into the off-screen map}
myCGrafPtr^.portPixMap^^.pmTable := ourCMHandle;
{Set the port to the off-screen port}
SetPort(GrafPtr(myCGrafPtr));
```

Now you can call `DrawIt` (which in turn calls `FillInColor`) to draw an image in the off-screen port.

```
FUNCTION FillInColor(r,g,b: Integer): RGBColor;
{small utility routine to return an RGBColor}
       VAR
              theColor       : RGBColor;

       BEGIN  {FillInColor}
              WITH theColor DO BEGIN
                     red := r;
                     green := g;
                     blue := b;
              END;
              FillInColor := theColor;
       END;   {FillInColor}

PROCEDURE DrawIt;

       VAR
              OvalRect              : Rect;
              myRed,myBlue,myWhite,
              myGreen, myBlack      : RGBColor;

       BEGIN  { DrawIt }
              {get our colors set up}
              myRed := FillInColor(-1,0,0);
              myBlue := FillInColor(0,0,-1);
              myGreen := FillInColor(0,-1,0);
              myWhite := FillInColor(-1,-1,-1);
              myBlack := FillInColor(0,0,0);
              PenMode(PatCopy);
              RGBBackColor(myBlue);                {set the backcolor of the current port}
              EraseRect(thePort^.portRect);        {blue it out}
              RGBBackColor(myWhite);               {set back to white}

              RGBForeColor(myRed);                 {set the forecolor of the current port}
              SetRect(OvalRect,30,30,190,150);
              PaintOval(OvalRect);

              InsetRect(OvalRect,1,20);
              EraseOval(OvalRect);                 {erase oval to white}

              RGBForeColor(myGreen);               {draw the final oval in green}
              InsetRect(OvalRect,40,1);
              PaintOval(OvalRect);
              RGBForeColor(myBlack);
       END;   { DrawIt }
```

Since you are done drawing, you need to set thePort and theGDevice back to their former values, and then you can draw the image on the screen by calling _CopyBits to copy the bits from the portPix of the off-screen pixel map to the portPix of MyCWindow.

```
        SetPort(MyCWindow);
        SetGDevice(oldDevice);

        destRect := bRect;
        OffSetRect(destRect,OffLeft,OffTop);        {adjust for coordinates}
        CopyBits(BitMapPtr(MyCGrafPtr^.portPixMap^)^, MyCWindow^.portBits,
               bRect, destRect, 0, NIL);
```

Finally, you clean up after yourself by closing the CGrafPort you created, freeing the space you reserved for the pixel image of the off-screen pixel map, and disposing of the color table you allocated.

```
        CloseCPort(myCGrafPtr);                     {Close our port}
        DisposPtr(MyBits);                          {clean up}
        DisposHandle(Handle(ourCMHandle));          {get rid of color table we cloned}
```

MPW C

You should note that most of the Pascal comments also apply to this C code, so if you are not sure what the C code is doing, try referring to the equivalent Pascal code and comments to gain a better understanding.

```
/* Define constants for the Off-Screen Rect */
#define        OffLeft        30
#define        OffTop         30
#define        OffBottom      250
#define        OffRight       400

/* typedef BitMapPtr for use during CopyBits operation */
typedef        BitMap *BitMapPtr;

long           offRowBytes, sizeOfOff;
Ptr            myBits;
Rect           destRect, globRect, bRect;
int            theDepth, i, err;
CGrafPort      myCGrafPort;
CGrafPtr       myCGrafPtr;
CTabHandle     ourCMHandle;
GDHandle       theMaxDevice, oldDevice;
Point          tempP;
```

Create a color window on screen. In MPW C, myWindow is declared as a WindowPtr, not a CWindowPtr, which is contrary to the way *Inside Macintosh*, Volume V documents it.

```
        myWindow = GetNewCWindow(SomeID,nil,(WindowPtr) -1);

        /* set to this port for the localToGlobals that follow */
        SetPort((WindowPtr) myWindow);

        SetRect(&bRect,OffLeft,OffTop,OffRight,OffBottom);
        if (!SectRect(&(*myWindow).portRect,&bRect,&globRect))
               ExitToShell();                       /*nothing to do, clean up and EXIT*/
```

Since MPW does not have `topLeft` or `botRight` elements for `Rect` structures, you need to set the `tempPoint`, call `_LocalToGlobal`, then reset `globRect`.

```
tempP.v = globRect.top;
tempP.h = globRect.left;
LocalToGlobal(&tempP);
globRect.top = tempP.v;
globRect.left = tempP.h;

tempP.v = globRect.bottom;
tempP.h = globRect.right;
LocalToGlobal(&tempP);
globRect.bottom = tempP.v;
globRect.right = tempP.h;

theMaxDevice = GetMaxDevice(&globRect);    /*get the maxDevice*/

oldDevice = GetGDevice();                  /*save theGDevice so we can
                                             restore it later*/
SetGDevice(theMaxDevice);                  /*Set to the maxdevice*/
```

Now you can set up the off-screen pixel map.

```
myCGrafPtr = &myCGrafPort;                 /*initialize this guy*/
OpenCPort(myCGrafPtr);                     /*open a new color port,
                                             this calls InitCPort*/
theDepth = (**(*myCGrafPtr).portPixMap).pixelSize;

/* Bitshift and adjust for local coordinates */
offRowBytes = (((theDepth * (OffRight - OffLeft)) + 15) >> 4) << 1;
sizeOfOff = (long) (OffBottom - OffTop) * offRowBytes;
OffsetRect(&bRect, - OffLeft, - OffTop);

myBits = NewPtr(sizeOfOff);

/* Remember to be a PixMap */
(**(*myCGrafPtr).portPixMap).baseAddr = myBits;
(**(*myCGrafPtr).portPixMap).rowBytes = offRowBytes + 0x8000;
(**(*myCGrafPtr).portPixMap).bounds = bRect;

ourCMHandle = (**(**theMaxDevice).gdPMap).pmTable;
err = HandToHand(&((Handle) ourCMHandle));
/* Real programs do error checking here */
for (i = 0; i <= (**ourCMHandle ).ctSize; ++i)
        (**ourCMHandle ).ctTable[i].value = i;
(**ourCMHandle ).ctFlags &= 0x7fff;
(**ourCMHandle ).ctSeed = GetCTSeed();
/* This code is necessary for converting GDevice cluts to Pixmap cluts */

(**(*myCGrafPtr).portPixMap).pmTable = ourCMHandle;
SetPort((GrafPtr) myCGrafPtr);
```

```
/**********************************************/
/*                                            */
/*     function for setting the wanted color  */
/*                                            */
/**********************************************/
RGBColor FillInColor(r,g,b)
int     r,g,b;

{       /*FillInColor*/

        RGBColor        theColor;

        theColor.red = r;
        theColor.green = g;
        theColor.blue = b;
        return (theColor);
}

/******************************************************/
/*                                                    */
/*  Drawing routine which makes the background blue   */
/* then draws a red oval, white oval, and green oval  */
/* After drawing to the off-screen it CopyBits to the */
/*                      screen                        */
/*                                                    */
/******************************************************/
void DrawIt()

{
        Rect            OvalRect;
        RGBColor        myRed,myBlue,myWhite,myGreen,myBlack;

        myRed = FillInColor(-1,0,0);
        myBlue = FillInColor(0,0,-1);
        myGreen = FillInColor(0,-1,0);
        myWhite = FillInColor(-1,-1,-1);
        myBlack = FillInColor(0,0,0);
        PenMode(patCopy);
        RGBBackColor(&myBlue);
        EraseRect(&(*qd.thePort).portRect);
        RGBBackColor(&myWhite);
        RGBForeColor(&myRed);
        SetRect(&OvalRect,30,30,190,150);
        PaintOval(&OvalRect);

        InsetRect(&OvalRect,1,20);
        EraseOval(&OvalRect);

        RGBForeColor(&myGreen);
        InsetRect(&OvalRect,40,1);
        PaintOval(&OvalRect);
        RGBForeColor(&myBlack);

        SetPort((WindowPtr) myWindow);
        SetGDevice(oldDevice);

        destRect = bRect;
        OffsetRect(&destRect,OffLeft,OffTop);
        CopyBits((BitMapPtr) *(*myCGrafPtr).portPixMap,
                &(*myWindow).portBits,&bRect, &destRect, 0, nil);

        return;
}
```

Once again, you clean up after yourself.

```
        CloseCPort(myCGrafPtr);
        DisposPtr(myBits);
        DisposHandle((Handle) ourCMHandle);
```

Note: For optimal performance, you want to make sure that the source and destination pixel maps are aligned.

Further Reference:

- *Inside Macintosh*, Volumes I-11 & IV-23, QuickDraw
- *Inside Macintosh*, Volume V-39, Color QuickDraw
- Technical Note #41, Drawing Into an Off-Screen Bitmap
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond

Macintosh Technical Notes

#121: Using the High-Level AppleTalk Routines

See also: The AppleTalk Manager
Inside AppleTalk
AppleTalk Manager Update

Written by: Fred A. Huxham May 4, 1987
Updated: March 1, 1988

What you need to do in order to use high-level AppleTalk routines depends upon the interfaces you are using. Some differences are outlined below.

MPW before 2.0

When calling the old high-level AppleTalk routines, many programmers get mysterious "resource not found" errors (-192) from such seemingly harmless routines as `MPPOpen`. The resource that is not being found is 'atpl', a resource that contains all the glue code to the high-level routines. In order to use the high-level routines, your application must have this resource in its resource fork. The 'atpl' resource is included in a file called "AppleTalk" with any compilers that use this outdated version of the AppleTalk interface.

MPW 2.0 and newer

A newer version of the alternate interfaces is available in MPW 2.0; it includes bug fixes and increased Macintosh II compatibility. With this version of the interface, the 'atpl' resource is no longer used. Glue code is now linked into your application.

This will be the final release of the current-style interface. It will be supported for some time as the **alternate interface**. We have moved to a more straightforward and simple **preferred interface**, which is also implemented in MPW 2.0 and newer, and is described in the AppleTalk Manager chapter of *Inside Macintosh* vol. V. Developers are free to continue to use the alternate interface, but in the long run it will be advantageous to move to the preferred interface.

Third Party Compilers

Third party compilers use interfaces that are built from Apple's MPW interfaces. Some compilers may not have upgraded to the new interfaces yet. Contact the individual compiler manufacturers for more information.

#122: Device-Independent Printing

See also: The Printing Manager

Written by: Ginger Jernigan

May 4, 1987
Updated: March 1, 1988

The Printing Manager was designed to give Macintosh applications a device-independent method of printing, but we *have* provided device-dependent information, such as the contents of the print record. Due to the large number of printer-type drivers becoming available (even for non-printer devices) device independence is more necessary than ever. What this means to you, as a developer, is that we will no longer be providing (or supporting) information regarding the internal structure of the print record.

We realize that there are situations where the application may know the best method for printing a particular document and may want to bypass our dialogs. Unfortunately, using your own dialogs or not using the dialogs at all, requires setting the necessary fields in the print record yourself. There are a number of problems:

- Many of the fields in the print record are undocumented, and, as we change the internal architecture of the Printing Manager to accommodate new devices, those undocumented fields are likely to change.

- Each driver uses the private, and many of the public, fields in the print record differently. The implications are that you would need intimate knowledge of how each field is used by each available driver, and you would have to set the fields in the record *differently* depending on the driver chosen. As the number of available printer-type drivers increases, this can become a cumbersome task.

Summary

To be compatible with future printer-like devices, it is essential that your application print in a device-independent manner. Avoid testing undocumented fields, setting fields in the print record directly and bypassing the existing print dialogs. Use the Printing Manager dialogs, `PrintDefault` and `PrValidate` to set up the print record for you.

#123: Bugs in LaserWriter ROMs

See also: The Printing Manager
PostScript Language Reference Manual, Adobe Systems

Written by:	Ginger Jernigan	May 4, 1987
Modified by:	Ginger Jernigan	July 1, 1987
Updated:		March 1, 1988

These are LaserWriter bugs that your users may encounter when printing from **any** Macintosh application. These are for your information; you cannot code around them. The bugs described here occur in the 1.0 and 2.0 LaserWriter ROMs.

To determine which ROMs their LaserWriter contains, users can look at the test page that the LaserWriter prints at start-up time. In addition to other information (detailed in the LaserWriter user's manual), the ROM version is shown at the bottom of the line graph. The original LaserWriter contained version 1.0 ROMs. The currently shipping LaserWriter and those upgraded to the LaserWriter Plus contain version 2.0 ROMs.

These are some of the problems we know of:

1. If the level of paper in the paper tray is getting low, and the user prints a document that will cause the tray to become empty, a PostScript error may occur. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

2. If a user prints more than 15 copies of a document, a timeout condition may occur causing the print job to abort. With LaserShare, this problem can occur with as few as 9 copies. This problem is a result of the LaserWriter turning AppleTalk off while it is printing. It doesn't send out any packets to tell the world it's still alive while it is printing, so the connection times out after about 2 minutes. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

3. When printing a document that contains more than 10 patterns, users may receive intermittent PostScript errors. This usually occurs when trying to print a lot of patterns, **and** a bitmap image on the same page. The code for imaging patterns allocates almost all of the available RAM for itself, so when the bitmap imaging code tries to allocate space, and there isn't enough (and it doesn't know how to reclaim memory from the previous operation), a `limitcheck` error occurs. This problem exists in 2.0 LaserWriter ROMs. It will be improved but **not** fixed in the next ROM version.

4. If a user chooses US Letter or B5 paper and has a different sized tray in the printer, and prints using manual feed, the LaserWriter will print assuming that the paper being fed manually is the same size as that in the tray. For example, if they have a US letter tray in the LaserWriter and print a document formatted for B5 letter using manual feed, the image will not be centered on the page. The printer assumes that the manually fed paper is also US letter size and prints the image positioned accordingly, despite the driver's instructions. This is a bug in the `Note` operator in PostScript, which the driver uses for specifying the US letter and B5 letter paper sizes. The workaround is to tell the user to put an B5 tray in the printer when printing B5 manually. This problem exists in the 1.0 and 2.0 ROMs and will **not** be fixed in the next ROM version.

   By the way, an interesting, but annoying, occurance of this bug happens when manually printing Legal sized documents with the 4.0 LaserWriter driver. When the Larger Print Area option in the style dialog is deselected (which is the default) the driver uses the `Note` operator to specify the page size. When the user prints the document using manual feed, and has a US letter tray in the printer, the image is shifted up on the page cutting off the top of the image. If you tell the user to turn on the Larger Print Area option in the style dialog, the driver specifies the page size using `Legal` instead of `Note` and the image is printed properly.

#124: Using Low-Level Printing Calls With AppleTalk ImageWriters

See also:	The Printing Manager	
Written by:	Ginger Jernigan	May 4, 1987
Update by:	Scott "ZZ" Zimmerman	Febuary ?, 1988
Updated:		March 1, 1988

When you use the low-level printer driver to print, you don't get the benefits of the error checking that is done when you use the high-level Printing Manager. So, if the user prints to an AppleTalk ImageWriter (including an AppleTalk ImageWriter LQ) that is busy printing another job, the driver doesn't know whether the printer is busy, offline, or disconnected. Because of this, PrError will return (and PrintErr will contain) `abortErr`.

Since there is no way to tell when you are printing to an AppleTalk ImageWriter, the only workaround for this is to use high-level Printing Manager interface.

Macintosh Technical Notes

#125: The Effect of Spool-a-page/Print-a-page on Shared Printers

See also: Printing Manager
Technical Note #72—
Optimizing for the LaserWriter—Techniques

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

This technical note discusses drawbacks of using the spool-a-page/print-a-page method of printing.

The "spool-a-page/print-a-page" method of printing prints each page of a document as a separate job instead of calling `PrPicFile` to print the entire picture file. Many applications adopted this method of printing to avoid running out of disk space while the ImageWriter driver was spooling the document to disk. As long as you are printing to a directly connected ImageWriter, you're fine, but if you are printing to remote or shared devices (like the AppleTalk ImageWriter and the LaserWriter), this method may create significant problems for the user.

When a job is initiated by the application, the driver establishes a connection with the printer via AppleTalk. When the job is completed, the driver closes the connection, allowing another job the opportunity to print. If each page is a job in itself, then the connection is closed and reopened between each page, allowing another application to print between the pages of the document, which, as you might imagine, could present a significant problem. If two people are printing to the same AppleTalk ImageWriter at the same time and their applications use the "spool-a-page/print-a-page" method of printing, the pages of each document will be interleaved at the printer.

Although there are good reasons for using this method of printing, it is only useful for a directly connected printer. From a compatibility point of view, this method of printing is built-in device dependence. Also, this method could create serious problems for other types of remote devices. Therefore, we are recommending that applications avoid using this method indiscriminately. You should check available disk space to see how much room you have before you print. If there isn't enough space for your entire document, then print as much as you can (to minimize the interleaving) before starting another job. Whenever possible, applications should use the print loop described on page II-155 in The Printing Manager chapter of *Inside Macintosh*.

Macintosh
Technical Notes

Developer Technical Support

#126: Sub(Launching) from a High-Level Language

Revised by: Rich Collyer & Mark Johnson — April 1989
Written by: Rick Blair & Jim Friedlander — May 1987

Note: Developer Technical Support takes the view that launching and sublaunching are features which are best **avoided** for compatibility (and other) reasons, but we want to make sure that when it is absolutely necessary to implement it, it is done in the safest possible way.

This Technical Note discusses the "safest" method of calling `_Launch` from a high-level language that supports inline assembly language with the option of launching or sublaunching another application.

Changes since August 1988: Incorporated Technical Note #52 on calling `_Launch` from a high-level language, changed the example to offer a choice between launching or sublaunching, added a discussion of the `_Launch` trap under MultiFinder, and updated the MPW C code to include inline assembly language.

The Segment Loader chapter of *Inside Macintosh* II-53 states the following about the `_Launch` trap:

> *"The routines below are provided for advanced programmers; they can be called only from assembly language."*

While this statement is technically true, it is easy to call `_Launch` from any high-level language which supports inline assembly code, and this Note provides examples of calling `_Launch` in MPW Pascal and C.

Before calling `_Launch`, you need to declare the inline procedure, which takes a variable of type `pLaunchStruct` as a parameter. Since the compiler pushes a pointer to this parameter on the stack, you need to include code to put this pointer into A0. The way to do this is with a `MOVE.L (SP)+,A0` instruction, which is `$205F` in hexadecimal, so the first word after `INLINE` is `$205F`. This instruction sets up `A0` to contain a pointer to the filename and `4(A0)` to contain the configuration parameter, so the last part of the inline is the `_Launch` trap itself, which is `$A9F2` in hexadecimal. The configuration parameter, which is normally zero, determines whether the application uses alternate screen and sound buffers. Since not all Macintosh models support these alternate buffers, you should avoid using them unless you have a specific circumstance which requires them.

The Finder does a lot of hidden cleanup and other tasks without user knowledge; therefore, it is best if you do not try to replace the Finder with a "mini" or try to launch other programs and have them return to your application. In the future, the Finder may provide better integration for applications, and you will circumvent this if you try to act in its place by sublaunching other programs.

If you have a situation where your application **must** launch another and have it return, and where you are not worried about incompatibility with future System Software versions, there is a "preferred" way of doing this which fits into the current system well. System file version 4.1 (or later) includes a mechanism for allowing a call to another application; we term this call a "sublaunch." You can perform a sublaunch by adding a set of simple extensions to the parameter block you pass to the `_Launch` trap.

_Launch and MultiFinder

Under MultiFinder, a sublaunch behaves differently than under the Finder. The application you sublaunch becomes the foreground application, and when the user quits that application, the system returns control to the next frontmost layer, which will not necessarily be your application.

If you set both high bits of `LaunchFlags`, which requests a sublaunch, your application will continue to execute after the call to `_Launch`. Under MultiFinder, the actual launch (and suspend of your application) will not happen in the `_Launch` trap, but rather after a call or more to `_WaitNextEvent`.

Under MultiFinder, `_Launch` currently returns an error if there is not enough memory to launch the desired application, if it cannot locate the desired application, or if the desired application is already open. In the latter case, that application will **not** be made active. If you attempted to launch, MultiFinder will call `_SysBeep`, your application will terminate, and control will given to the next frontmost layer. If you attempted to sublaunch, control will return to your application, and it is up to you to report the error to the user.

Currently, `_Launch` returns an error in register `D0` for a sublaunch, and you should check it for errors (`D0<0`) after any attempts at sublaunching. If `D0>=0` then your sublaunch was successful.

You should refer to the *Programmer's Guide to MultiFinder* (APDA) and Macintosh Technical Notes #180, MultiFinder Miscellanea and #205, MultiFinder Revisited: The 6.0 System Release, for further discussion of the `_Launch` trap under MultiFinder.)

Working Directories and Sublaunching With the Finder

Putting aside the compatibility issue for the moment, the only problem sublaunching creates under the **current** system is one of Working Directory Control Blocks (WDCBs). Unless the application you are launching is at the root directory or on an MFS volume, you must create a new WDCB and set it as the current directory when you launch the application.

In the example which follows, the new working directory is opened (allocated) by Standard File and its `WDRefNum` is returned in `reply.vRefNum`. If you do not use Standard File and cannot assume, for instance, that the application was in the blessed folder or root directory, then you must open a new working directory explicitly via a call to `_OpenWD`. You should give the new WDCB a `WDProcID` of `'ERIK'`, so the Finder (or another shell) would know to deallocate when it saw it was allocated by a "sublaunchee."

Although the sublaunching process is recursive (i.e., programs which are sublaunched may, in turn, sublaunch other programs), there is a limit of 40 on the number of WDCBs which can be created. With this limit, you could run out of available WDCBs very quickly if many programs were playing the shell game or neglecting to deallocate the WDCBs they had created. Make sure you check for **all** errors after calling `_PBOpenWD`. A `tMWDOErr` (–121) means that all available

WDCBs have been allocated, and if you receive this error, you should alert the user that the sublaunch failed and continue as appropriate.

Warning: Although the example included in this Note covers sublaunching, Developer Technical Support strongly recommends that developers **not** use this feature of the _Launch trap. This trap will change in the not-too-distant future, and when it does change, applications which perform sublaunching **will** break. The only circumstance in which you could consider sublaunching is if you are implementing an integrated development system and are prepared to deal with the possibility of revising it every time Apple releases a new version of the System Software.

MPW Pascal

```
{It is assumed that the Signals are caught elsewhere; see Technical
 Note #88 for more information on the Signal mechanism}

{the extended parameter block to _Launch}
TYPE
   pLaunchStruct = ^LaunchStruct;
   LaunchStruct = RECORD
      pfName       : StringPtr;
      param        : INTEGER;
      LC           : PACKED ARRAY[0..1] OF CHAR; {extended parameters:}
      extBlockLen  : LONGINT; {number of bytes in extension = 6}
      fFlags       : INTEGER; {Finder file info flags (see below)}
      launchFlags  : LONGINT; {bit 31,30=1 for sublaunch, others reserved}
   END; {LaunchStruct}

FUNCTION LaunchIt(pLaunch: pLaunchStruct): OSErr; {< 0 means error}
   INLINE $205F, $A9F2, $3E80;
{ pops pointer into A0, calls Launch, pops D0 error code into result:
  MOVE.L  (A7)+,A0
  _Launch
  MOVE.W  D0,(A7)   ; since it MAY return }

PROCEDURE DoLaunch(subLaunch: BOOLEAN);  {Sublaunch if true and launch if false}

        VAR
           myLaunch      : LaunchStruct;    {launch structure}
           where         : Point;           {where to display dialog}
           reply         : SFReply;         {reply record}
           myFileTypes   : SFTypeList;      {we only want APPLs}
           numFileTypes  : INTEGER;
           myPB          : CInfoPBRec;
           dirNameStr    : str255;

        BEGIN
           where.h := 20;
           where.v := 20;
           numFileTypes:= 1;
           myFileTypes[0]:= 'APPL';        {applications only!}
        {Let the user choose the file to Launch}
           SFGetFile(where, '', NIL, numFileTypes, myFileTypes, NIL, reply);
      IF reply.good THEN BEGIN
         dirNameStr:= reply.fName;    {initialize to file selected}

   {Get the Finder flags}
         WITH myPB DO BEGIN
             ioNamePtr:= @dirNameStr;
             ioVRefNum:= reply.vRefNum;
             ioFDirIndex:= 0;
             ioDirID:= 0;
         END; {WITH}
         Signal(PBGetCatInfo(@MyPB,FALSE));
   {Set the current volume to where the target application is}
         Signal(SetVol(NIL, reply.vRefNum));

   {Set up the launch parameters}
         WITH myLaunch DO BEGIN
            pfName := @reply.fName;    {pointer to our fileName}
            param := 0;                {we don't want alternate screen or sound buffers}
            LC := 'LC';                {here to tell Launch that there is non-junk next}
            extBlockLen := 6;          {length of param. block past this long word}
            {copy flags; set bit 6 of low byte to 1 for RO access:}
            fFlags := myPB.ioFlFndrInfo.fdFlags;  {from GetCatInfo}

   {Test subLaunch and set LaunchFlags accordingly}
         IF subLaunch THEN
            LaunchFlags := $C0000000              {set BOTH high bits for a sublaunch}
         ELSE
            LaunchFlags := $00000000;             {Just launch then quit}
         END; {WITH}

   {launch; you might want to put up a dialog which explains that
    the selected application couldn't be launched for some reason.}
         Signal(LaunchIt(@myLaunch));
      END; {IF reply.good}

   END; {DoLaunch}
```

MPW C

```
typedef struct LaunchStruct {
        char            *pfName;                /* pointer to the name of launchee */
        short int       param;
        char            LC[2];                  /*extended parameters:*/
        long int        extBlockLen;            /*number of bytes in extension == 6*/
        short int       fFlags;                 /*Finder file info flags (see below)*/
        long int        launchFlags;            /*bit 31,30==1 for sublaunch, others reserved*/
} *pLaunchStruct;

pascal OSErr LaunchIt( pLaunchStruct pLnch) /* < 0 means error */
        = {0x205F, 0xA9F2, 0x3E80};

/* pops pointer into A0, calls Launch, pops D0 error code into result:
          MOVE.L  (A7)+,A0
          _Launch
          MOVE.W  D0,(A7)  ; since it MAY return */

OSErr DoLaunch(subLaunch)
        Boolean                 subLaunch;      /* Sublaunch if true and launch if false    */
{  /* DoLaunch */
        struct LaunchStruct     myLaunch;
        Point                   where;          /*where to display dialog*/
        SFReply                 reply;          /*reply record*/
        SFTypeList              myFileTypes;    /* we only want APPLs */
        short int               numFileTypes=1;
        HFileInfo               myPB;
        char                    *dirNameStr;
        OSErr                   err;

        where.h = 80;
        where.v = 90;
        myFileTypes[0] = 'APPL';                /* we only want APPLs */
        /*Let the user choose the file to Launch*/
        SFGetFile(where, "", nil, numFileTypes, myFileTypes, nil, &reply);

        if (reply.good)
        {
                dirNameStr = &reply.fName;      /*initialize to file selected*/

        /*Get the Finder flags*/
                myPB.ioNamePtr= dirNameStr;
                myPB.ioVRefNum= reply.vRefNum;
                myPB.ioFDirIndex= 0;
                myPB.ioDirID = 0;
                err = PBGetCatInfo((CInfoPBPtr) &myPB,false);
                if (err != noErr)
                        return err;

        /*Set the current volume to where the target application is*/
                err = SetVol(nil, reply.vRefNum);
                if (err != noErr)
                        return err;

        /*Set up the launch parameters*/
                myLaunch.pfName = &reply.fName;         /*pointer to our fileName*/
                myLaunch.param = 0;                     /*we don't want alternate screen
                                                          or sound buffers*/
        /*set up LC so as to tell Launch that there is non-junk next*/
                myLaunch.LC[0] = 'L'; myLaunch.LC[1] = 'C';
                myLaunch.extBlockLen = 6;               /*length of param. block past
                                                          this long word*/
        /*copy flags; set bit 6 of low byte to 1 for RO access:*/
                myLaunch.fFlags = myPB.ioFlFndrInfo.fdFlags;        /*from _GetCatInfo*/
        /* Test subLaunch and set launchFlags accordingly */
        if ( subLaunch )
               myLaunch.launchFlags = 0xC0000000;  /*set BOTH hi bits for a sublaunch  */
        else
               myLaunch.launchFlags = 0x00000000;  /* Just launch then quit            */

               err = LaunchIt(&myLaunch);          /* call _Launch                     */
               if (err < 0)
               {
               /* the launch failed, so put up an alert to inform the user */
                      LaunchFailed();
                      return err;
               }
               else
                      return noErr;
        } /*if reply.good*/
} /*DoLaunch*/
```

Further Reference:

- *Inside Macintosh*, Volumes I-12, II-53, & IV-83, The Segment Loader
- *Programmer's Guide to MultiFinder* (APDA)
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #180, MultiFinder Miscellanea
- Technical Note #205, MultiFinder Revisited: The 6.0 System Release

Macintosh Technical Notes

#127: TextEdit EOL Ambiguity

See also: TextEdit

Written by: Rick Blair May 4, 1987
Updated: March 1, 1988

TESetSelect may be used to position the insertion point at the end of a line. There is an ambiguity, though; should the insertion point appear at the end of the preceding line or the start of the following one? It is possible to determine what will happen, as you are about to see.

There is an internal flag used by TextEdit to determine where the insertion point at the end of a line appears. This flag is part of the clikStuff field in the TERec. It is there mainly for the use of TEClick, but it is also used by TESetSelect (although it defaults to the right side of the previous line).

The following code can be used to force the insertion point to appear at the left of the following line when it is positioned at the end of a line; in MPW Pascal:

```
TEDeactivate(tH);                                {position caret on left}
tH^^.clikStuff := 255;                           {ambiguous point}
TESetSelect(eolcharpos, eolcharpos, tH);
TEActivate(tH);
```

In MPW C:

```
TEDeactivate(tH);                                /*position caret on left*/
(**tH).clikStuff = 255;                          /*ambiguous point*/
TESetSelect(eolcharpos, eolcharpos, tH);
TEActivate(tH);
```

If you want to ensure that the caret is on the right side (to which it normally defaults) then substitute a zero for the 255.

Macintosh Technical Notes

#128: PrGeneral

See also: The Printing Manager
Technical Note #118—
How to Check and Handle Printing Errors

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

The Printing Manager architecture has been expanded to include a new procedure called `PrGeneral`. The features described here are advanced, special-purpose features, intended to solve specific problems for those applications that need them. The calls to determine printer resolution introduce a good deal of complexity into the application's code, and should be used only when necessary.

Version 2.5 (and later) of the ImageWriter driver and version 4.0 (and later) of the LaserWriter driver implement a generic Printing Manager procedure called `PrGeneral`. This procedure allows the Print Manager to expand in functionality, by allowing printer drivers to implement various new functions. The Pascal declaration of `PrGeneral` is:

```
PROCEDURE PrGeneral (pData: Ptr);
```

The `pData` parameter is a pointer to a data block. The structure of the data block is declared as follows:

```
TGnlData = RECORD {1st 8 bytes are common for all PrGeneral calls)
   iOpCode    : INTEGER;  {input}
   iError     : INTEGER;  {output}
   lReserved : LONGINT;  {reserved for future use}
   {more fields here, depending on particular call}
END;
```

The first field is a 2-byte opcode, `iOpCode`, which acts like a routine selector. The currently available opcodes are described below.

The second field is the error result, `iError`, which is returned by the print code. This error only reflects error conditions that occur during the `PrGeneral` call. For example, if you use an opcode that isn't implemented in a particular printer driver then you will get a `OpNotImpl` error.

Here are the errors currently defined:

```
CONST
   noErr = 0;                {everything's hunky}
   NoSuchRsl = 1;            {the resolution you chose isn't available}
   OpNotImpl = 2;            {the driver doesn't support this opcode}
```

After calling PrGeneral you should always check PrError. If noErr is returned, then you can proceed. If ResNotFound is returned, then the current printer driver doesn't support PrGeneral and you should proceed appropriately. See Technical Note #118 for details on checking errors returned by the Printing Manager.

IError is followed by a four byte reserved field (that means don't use it). The contents of the rest of the data block depends on the opcode that the application uses. There are currently five opcodes used by the ImageWriter and LaserWriter drivers.

The Opcodes

Initially, the following calls are implemented via PrGeneral:

- GetRslData (get resolution data): iOpCode = 4
- SetRsl (set resolution): iOpCode = 5
- DraftBits (bitmaps in draft mode): iOpCode = 6
- noDraftBits (no bitmaps in draft mode): iOpCode = 7
- GetRotn (get rotation): iOpCode = 8

The GetRslData and SetRsl allow the application to find out what physical resolutions the printer supports, and then specify a supported resolution. DraftBits and noDraftBits invoke a new feature of the ImageWriter, allowing bitmaps (imaged via CopyBits) to be printed in draft mode. GetRotn lets an application know whether landscape has been selected. Below is a detailed description of how each routine works.

The GetRslData Call

GetRslData (iOpCode = 4) returns a record that lets the application know what resolutions are supported by the current printer. The application can then use SetRsl (description follows) to tell the printer driver which one it will use. This is the format of the input data block for the GetRslData call:

```
TRslRg = RECORD             {used in TGetRslBlk}
   iMin, iMax: Integer;     {0 if printer only supports discrete resolutions}
END;

TRslRec = RECORD            {used in TGetRslBlk}
   iXRsl, iYRsl: Integer;  {a discrete, physical resolution}
END;
```

```
TGetRslBlk = RECORD        {data block for GetRslData call}
   iOpCode:     Integer;   {input; = getRslDataOp}
   iError:      Integer;   {output}
   lReserved:   LongInt;   {reserved for future use}
   iRgType:     Integer;   {output; version number}
   XRslRg:      TRslRg;    {output; range of X resolutions}
   YRslRg:      TRslRg;    {output; range of Y resolutions}
   iRslRecCnt:  Integer;   {output; how many RslRecs follow}
   rgRslRec:    ARRAY[1..27] OF TRslRec;  {output; number filled depends on
                                            printer type}
END;
```

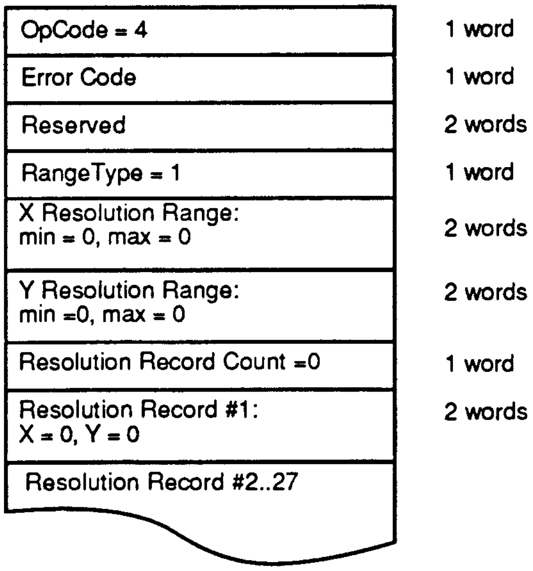
The `iRgType` field is much like a version number; it determines the interpretation of the data that follows. At present, a `iRgType` value of 1 applies both to the LaserWriter and to the ImageWriter.

For variable-resolution printers like the LaserWriter, the resolution range fields `XRslRg` and `YRslRg` express the ranges of values to which the X and Y resolutions can be set. For discrete-resolution printers like the ImageWriter, the values in the resolution range fields are zero.

Note: In general, X and Y in these records are the horizontal and vertical directions of the **printer**, not the document! In landscape orientation, X is horizontal on the printer but vertical on the document.

After the resolution range information there is a word which gives the number of resolution records that contain information. These records indicate the physical resolutions at which the printer can actually print dots. Each resolution record gives an X value and a Y value.

When you call `PrGeneral` you pass in a data block that looks like this:



OpCode = 4	1 word
Error Code	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 0, max = 0	2 words
Y Resolution Range: min =0, max = 0	2 words
Resolution Record Count =0	1 word
Resolution Record #1: X = 0, Y = 0	2 words
Resolution Record #2..27	

Below is the data block returned for the LaserWriter:

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 72, max = 1500	2 words
Y Resolution Range: min = 72, max = 1500	2 words
Resolution Record Count = 1	1 word
Resolution Record #1: X = 300, Y = 300	2 words

Note that all the resolution range numbers happen to be the same for this printer. There is only one resolution record, which gives the physical X and Y resolutions of the printer (300x300).

Below is the data block returned for the ImageWriter.

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min =0, max = 0	2 words
Y Resolution Range: min = 0, max = 0	2 words
Resolution Record Count = 4	1 word
Resolution Record #1: X = 72, Y = 72	2 words
Resolution Record #2: X =144, Y = 144	2 words
Resolution Record #3: X = 80, Y = 72	2 words
Resolution Record #4: X = 160, Y = 144	2 words

All the resolution range values are zero, because only discrete resolutions can be specified for this printer. There are four resolution records giving these discrete physical resolutions.

Note that `GetRslData` always returns the same information for a particular printer type—it is **not** dependent on what the user does or on printer configuration information.

The SetRsl Call

SetRsl (iOpCode = 5) is used to specify the desired imaging resolution, after using GetRslData to determine a workable pair of values. Below is the format of the data block:

```
TSetRslBlk =    RECORD      {data block for SetRsl call}
   iOpCode:     Integer;    {input; = setRslOp}
   iError:      Integer;    {output}
   lReserved:   LongInt;    {reserved for future use}
   hPrint:      THPrint;    {input; handle to a valid print record}
   iXRsl:       Integer;    {input; desired X resolution}
   iYRsl:       Integer;    {input; desired Y resolution}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate. If the call executes successfully, the print record is updated with the new resolution; the data block comes back with 0 for the error and is otherwise unchanged.

However, if the desired resolution is not supported, the error is set to noSuchRsl and the resolution fields are set to the printer's default resolution

Note that you can undo the effect of a previous call to SetRsl by making another call that specifies an unsupported resolution (such as 0x0), forcing the default resolution.

The DraftBits Call

DraftBits (iOpCode = 6) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) Below is the format of the data block:

```
TDftBitsBlk =   RECORD      {data block for DraftBits and NoDraftBits calls}
   iOpCode:     Integer;    {input; = draftBitsOp or noDraftBitsOp}
   iError:      Integer;    {output}
   lReserved:   LongInt;    {reserved for future use}
   hPrint:      THPrint;    {input; handle to a valid print record}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate.

This call forces draft-mode (i.e., immediate) printing, and will allow bitmaps to be printed via CopyBits calls. The virtue of this is that you avoid spooling large masses of bitmap data onto the disk, and you also get better performance.

The following restrictions apply:

- This call should be made before bringing up the print dialogs because it affects their appearance. On the ImageWriter, calling DraftBits disables the landscape icon in the Style dialog, and the Best, Faster, and Draft buttons in the Job dialog.

- If the printer does not support draft mode, already prints bitmaps in draft mode, or does not print bitmaps at all, this call does nothing.

- Only text and bitmaps can be printed.

- As in the normal draft mode, landscape format is not allowed.

- Everything on the page must be strictly Y-sorted, i.e. no reverse paper motion between one string or bitmap and the next. Note that this means you can't have two or more objects (text or bitmaps) side by side; the top boundary of each object must be no higher than the bottom of the preceding object.

The last restriction is important. If you violate it, you will not like the results. But note that if you want two or more bitmaps side by side, you can combine them into one before calling `CopyBits` to print the result. Similarly, if you are just printing bitmaps you can rotate them yourself to achieve landscape printing.

The NoDraftBits Call

`NoDraftBits` (`iOpCode = 7`) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) The format of the data block is the same as that for the `DraftBits` call.

This call cancels the effect of any preceding `DraftBits` call. If there was no preceding `DraftBits` call, or the printer does not support draft-mode printing anyway, this call does nothing.

The GetRotn Call

`GetRotn` (`iOpCode` = 8) is implemented on the ImageWriter and LaserWriter. Here is the format of the data block:

```
TGetRotnBlk =  RECORD          {data block for GetRotn call}
   iOpCode:    Integer;        {input; = getRotnOp}
   iError:     Integer;        {output}
   lReserved:  LongInt;        {reserved for future use}
   hPrint:     THPrint;        {input; handle to a valid print record}
   fLandscape: Boolean;        {output; Boolean flag}
   bXtra:      SignedByte;     {reserved}
END;
```

`hPrint` should be the handle to a print record that has previously been passed to `PrValidate`.

If landscape orientation is selected in the print record, then `fLandscape` is true.

How To Use The PrGeneral Opcodes

The SetRsl and DraftBits calls may require the print code to suppress certain options in the Style and/or Job dialogs, therefore they should always be called before any call to the Style or Job dialogs. An application might use these calls as follows:

- Get a new print record by calling PrintDefault, or take an existing one from a document and call PrValidate on it.

- Call GetRslData to find out what the printer is capable of, and decide what resolution to use. Check PrError to be sure the PrGeneral call is supported on this version of the print code; if the error is ResNotFound, you have older print code and must print accordingly. But if the PrError return is 0, proceed:

- Call SetRsl with the print record and the desired resolution if you wish.

- Call DraftBits to invoke the printing of bitmaps in draft mode if you wish.

Note that if you call either SetRsl or DraftBits, you should do so before the user sees either of the printing dialogs.

Macintosh
Technical Notes

Developer Technical Support

#129: _SysEnvirons: System 6.0 and Beyond

Revised by: Guillermo Ortiz & Dave Radcliffe — October 1989
Written by: Jim Friedlander — May 1987

This Technical Note discusses changes and enhancements in the `_SysEnvirons` call in System Software 6.0 and later.

Changes since April 1989: Added `machineType` constants for the Macintosh Portable and IIci. Also added `keyBoardType` constants for the Portable and ISO keyboards.

_SysEnvirons and New Machines

`_SysEnvirons` is the standard way to determine the features available on a given machine, and its main characteristic is that it continually evolves to provide the necessary information as new machines and System Software appear. As originally conceived, `_SysEnvirons` would check the `versionRequested` parameter to determine what level of information you were prepared to handle, but this technique means updating `_SysEnvirons` for every new hardware product Apple produces. With System Software 6.0, `_SysEnvirons` introduced version 2 of `environsVersion` to provide information about new hardware as we introduce it; this new version returns the same `SysEnvRec` as version 1.

Beginning with System Software 6.0.1, Apple only releases a new version of `_SysEnvirons` when engineering make changes to its structure (i.e., when they add new fields to `SysEnvRec`); all existing versions will return accurate information about the machine environment even if part of that information was not originally defined for the version you request. For example, if you call `_SysEnvirons` with `versionRequested = 1` on a Macintosh IIx, it will return a `machineType` of `envMacIIx` even though this machine type originally was not defined for version 1 of the call.

You should use version 2 of `_SysEnvirons` until Apple releases a newer version. Regardless of the version used, however, your software should be prepared to handle unexpected values and should not make assumptions about functionality based on current expectations. For example, if your software currently requires a Macintosh II, testing for `machineType >= envMacII` may result in your software trying to run on a machine which will not support the features it requires, so test for specific functionality (i.e., `hasFPU`, `hasColorQD`, etc.).

You should always check the `environsVersion` when returning from `_SysEnvirons` since the glue always returns as much information as possible, with `environsVersion` indicating the highest version available, even if the call returns an `envSelTooBig` (–5502) error.

New Constants

The following are new `_SysEnvirons` constants which are not documented in *Inside Macintosh*; however, you should refer to *Inside Macintosh*, Volume V-1, Compatibility Guidelines, for the rest of the story.

```
machineType
envMacIIx = 5                {Macintosh IIx}
envMacIIcx = 6               {Macintosh IIcx}
envSE30 = 7                  {Macintosh SE/30}
envPortable = 8              {Macintosh Portable}
envMacIIci = 9               {Macintosh IIci}

processor
env68030 = 4                 {MC68030 processor}

keyBoardType
envPortADBKbd = 6            {Portable Keyboard}
envPortISOADBKbd = 7         {Portable Keyboard (ISO)}
envStdISOADBKbd = 8          {Apple Standard Keyboard (ISO)}
envExtISOADBKbd = 9          {Apple Extended Keyboard (ISO)}
```

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines

#130: Clearing ioCompletion

See also: The File Manager

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

When making synchronous calls to the File Manager, it is not necessary to clear `ioCompletion` field of the parameter block, since that is done for you.

Some earlier technotes explicitly cleared `ioCompletion`, with the knowledge that this was unnecessary, to try to encourage developers to fill in all fields of parameter blocks as indicated in *Inside Macintosh.*

By the way, this is true of all parameter calls—you only have to set fields that are explicitly required.

Macintosh Technical Notes

#131: TextEdit Bugs in System 4.2

Written by: Chris Derossi June 1, 1987
Updated: March 1, 1988

This note formerly described the known bugs with the version of Styled TextEdit that was provided with System 4.1. Many of these bugs were fixed in System 4.2. This updated Technical Note describes the remaining known problems.

TEStylInsert

Calling `TEStylInsert` while the TextEdit record is deactivated causes unpredictable results, so make sure to only call `TEStylInsert` when the TextEdit record is active.

TESetStyle

When using the `doFace` mode with `TESetStyle`, the style that you pass as a parameter is `ORed` into the style of the currently selected text. If you pass the empty set (no styles) though, `TESetStyle` is supposed to remove all styles from the selected text. But `TESetStyle` checks an entire word instead of just the high-order byte of the `tsFace` field. The style information is contained completely in the high-order byte, and the low-order byte may contain garbage.

If the low-order byte isn't zero, `TESetStyle` thinks that the `tsFace` field isn't empty, so it goes ahead and ORs it with the selected text's style. Since the actual style portion of the `tsFace` field is zero, no change occurs with the text. If you want to have `TESetStyle` remove all styles from the text, you can explicitly set the `tsFace` field to zero like this:

```
VAR
   myStyle   : TextStyle;
   anIntPtr : ^Integer;

BEGIN
   ...
   anIntPtr := @myStyle.tsFace;
   anIntPtr^ := 0;
   TESetStyle(doFace, myStyle, TRUE, textH);
   ...
END;
```

TEStylNew

The line heights array does not get initialized when TEStylNew is called. Because of this, the caret is initially drawn in a random height. This is easily solved by calling TECalText immediately after calling TEStylNew. Extra calls to TECalText don't hurt anything anyway, so this will be compatible with future Systems.

An extra character run is placed at the beginning of the text which corresponds to the font, size, and style which were in the grafPort when TEStylNew was called. This can cause the line height for the first line to be too large. To avoid this, call TextSize with the desired text size before calling TEStylNew. If the text's style information cannot be determined in advance, then call TextSize with a small value (like 9) before calling TEStylNew.

TEScroll

The bug documented in Technical Note #22 remains in the new TextEdit. TEScroll called with zero for both vertical and horizontal displacements causes the insertion point to disappear. The workaround is the same as before; check to make sure that dV and dH are not both zero before calling TEScroll.

Growing TextEdit Record

TextEdit is supposed to dynamically grow and shrink the LineStarts array in the TERec so that it has one entry per line. Instead, when lines are added, TextEdit expands the array without first checking to see if it's already big enough. In addition, TextEdit never reduces the size of this array.

Because of this, the longer a particular TextEdit record is used, the larger it will get. This can be particularly nasty in programs that use a single TERec for many operations during the program's execution.

Restoring Saved TextEdit Records

Applications have used a technique for saving and restoring styled text which involves saving the contents of all of the TextEdit record handles. When restoring, TEStylNew is called and the TextEdit record's handles are disposed. The saved handles are then loaded and put into the TextEdit record. This technique should not be used for the nullStyle handle in the style record.

Instead, when TEStylNew is called, the nullStyle handle from the style record should be copied into the saved style record. This will ensure that the fields in the null-style record point to valid data.

Macintosh Technical Notes

#132: AppleTalk Interface Update

See also: The AppleTalk Manager
Inside AppleTalk (for ZIP information)
Technical Note #121—
Using the High-Level AppleTalk Routines

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

Technical Note #121 announced that we would be moving to a simplified AppleTalk Manager interface. That interface is available now, as part of MPW 2.0 and newer.

Documentation for this new interface is contained in the AppleTalk Manager chapter of *Inside Macintosh Volume V*. This technical note contains some of the preliminary documentation for this interface and some useful points about information about it, and AppleTalk in general.

The original AppleTalk Pascal Interfaces, known as `ABPasIntf`, were designed to simplify use of AppleTalk from high-level languages. Instead, they've caused us a few compatibility problems. We've decided to encourage use of the same interface that assembly-language AppleTalk uses, a parameter-block interface in the same style as the low-level interfaces to the File and Device Managers.

The original calls are still supported (and will be for a while) as an "alternate" interface, but we suggest that you consider moving to the new "preferred" calls. Be warned that use of the original calls may cause compatibility problems with future system software. Also, new protocols (like ASP, the AppleTalk Session Protocol) are only provided with the new interfaces.

The new interface uses parameter blocks like those used by the File and Device Managers; you fill out the call-specific fields of the block, and a small amount of glue code (provided with development environments like MPW) turns the parameter block into a `Control` call to the appropriate AppleTalk driver.

Most calls have an interface like:

```
    FUNCTION PSomeCall(thePBPtr: ATPPBptr; asyncFlag: BOOLEAN): OSErr;
```

The glue fills in the fields `csCode` and `ioRefNum` with the appropriate value for the call you're making.

Synchronous and Asynchronous calls

You can still make calls synchronously ("do it now") or asynchronously ("start it now, finish it soon"). If you choose to make a call asynchronously, be sure to provide a completion routine in the `ioCompletion` field (to be called when the call finally finishes), or poll the `ioResult` field of the parameter block (the call is done if `ioResult` is less than or equal to 0).

You must not move or dispose of a parameter block before the call finishes; when the call does complete, you are responsible for throwing the parameter block away (if you allocated it using Memory Manager routines).

Note that the alternate interfaces generated a network event on completion of an asynchronous call; this service is not provided by the preferred interfaces, partly because of future compatibility problems. See Technical Note #142 for background information.

Packed data structures

Several of the data structures used by the new interfaces are packed; Pascal doesn't deal well with these structures. Special calls are provided for building LAP and DPP write-data structures, NBP names-table elements, and ATP buffer data structures.

For example, when registering a name (using `PRegisterName`), you'll use a `NamesTableEntry` structure. This structure consists of a few unpacked fields, followed by an entity-name: three strings (representing the object, type, and zone fields of the name) packed together. You can call `NBPSetNTE` to pack the strings into the `NamesTableEntry` structure. When you remove the name (`PRemoveName`), you'll use the entity-name by itself; you can use `NBPSetEntity` to pack it in.

Zone Interface Protocol

A function, `GetBridgeAddress`, is provided to obtain the node ID of a bridge, for use in ZIP transactions (zero is returned if no bridge is present on your network). You make ZIP calls using ATP requests, as described in the *Inside AppleTalk* chapter on ZIP.

Macintosh Technical Notes

#133: Am I Talking To A Spooler?

See also: *PostScript Language Reference Manual*
Adobe Systems Document Structuring Conventions

Written by: Ginger Jernigan July 1, 1987
Updated: March 1, 1988

When the LaserShare spooler is on an AppleTalk network, it acts like a LaserWriter-type device, which can be chosen and communicated with much like a real LaserWriter. Some applications, however, must communicate with a LaserWriter directly, not a spooler. If this is true for your application, you can check whether you are actually talking to a real LaserWriter by sending to the LaserWriter the following query:

```
%!PS-Adobe-1.2 Query
%%Title: Query to Spooler/Non-Spooler status
%%?BeginSpoolerQuery
(0) = flush
%%?EndSpoolerQuery 1
%%EOF
```

(The query has to be sent using the Printer Access Protocol (PAP). The object code for PAP is available from Licensing.) If the string returned begins with a '%%' then it is a status string and you can ignore it and wait for another string. If the LaserWriter is actually a LaserShare spooler, then the string that is returned will be '1'. If the LaserWriter is a real LaserWriter then the string returned will be '0'.

#134: Hard Disk Medic & Booting Camp

See also: Hard Disk Users Manual
Technical Note 154—Macintosh Plus ROMs
Technical Note 113—Boot Blocks
Technical Note 67—Finding the 'Blessed Folder'

Written by: Bo3b Johnson July 1, 1987
Updated: March 1, 1988

The death of a hard disk with megabytes worth of data can be exceedingly traumatic. This technical note will describe techniques for recovering a hard disk and the data that is on it. The discussion will also include some tips on how to avoid problems.

You should never need this information. However, software problems can wreak havoc upon otherwise functional disks. When they have the equivalent of a heart attack, there are a number of steps that can be taken to try to recover the disk. There are occasions when the disk itself is not bad, and it may be possible to correct the disk without having to reformat the disk and restore the data from a backup. This note will describe some of the steps that can be used with Apple Hard Disks, but most of the information pertains to all hard disks. For example, the HD SC Setup program is specific to the Apple drives, but there is probably a similar utility for every hard disk. This is primarily a discussion of what to do from the user standpoint, but there are a few suggestions on ways of retrieving data via programmatic means.

This discussion will focus on the SCSI disks since they are more complex in terms of the booting sequence. For other hard disks, like the standard HD-20, most of the information still applies, but SCSI-specific sequences can be ignored. For example, the standard HD-20 also has an installer program, although it is different than HD SC Setup.

Attack of the Nasties

There are a number of unusual conditions that a hard disk may get itself in:

1) The data is intact, but the hard disk won't boot.
2) The SCSI disk won't boot and only shows up after running HD SC Setup.
3) The disk will boot but hangs part way through the boot process.
4) There are data errors while the disk is running.
5) The disk is very slow returning to the Finder.
6) The computer crashes or hangs when returning to the Finder.
7) The disk appears in a "This disk is bad" dialog.

8) The disk never shows up at all.

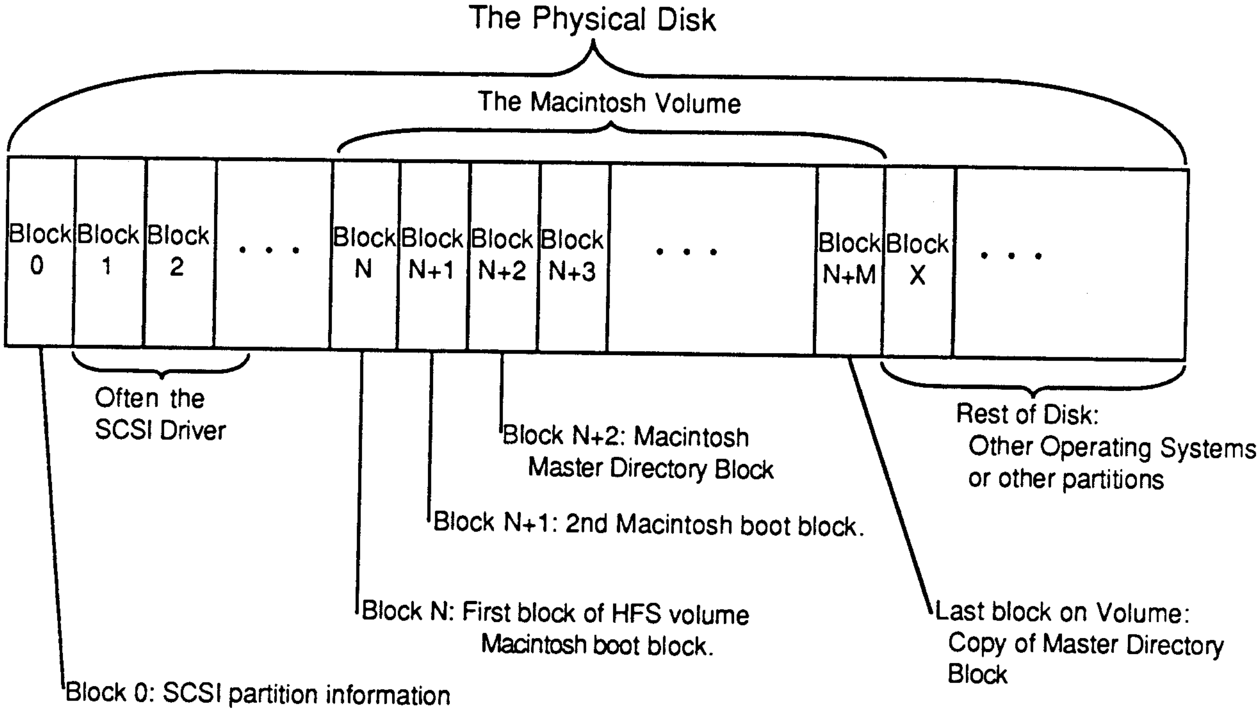
These problems can develop from a number of sources, including system crashes, rebooting at bad times, power fluctuations, malicious software, old software, buggy software, etc. In general, these problems will be software-related, since the hardware itself is very rarely defective.

This technical note will discuss:

1) The normal stages in the booting process.
2) Results of errors during the various stages in the booting process.
3) A step-by-step procedure to follow in order to maximize your chances of recovering the disk and the data.

A Boot to the Head

This discussion will detail a normal boot process of a Macintosh with a single hard disk attached. For clarity, this section will deliberately ignore potential problems and the complexities involved in different configurations. The following sections will detail some errors that may occur, and give more information in terms of what the ROM will do to boot the system. A SCSI disk can be thought of in the following fashion:


The Physical Disk
The Macintosh Volume
Block 0
Block 1
Block 2
. . .
Block N
Block N+1
Block N+2
Block N+3
. . .
Block N+M
Block X
. . .
Often the SCSI Driver
Block N+2: Macintosh Master Directory Block
Rest of Disk: Other Operating Systems or other partitions
Block N+1: 2nd Macintosh boot block.
Block N: First block of HFS volume Macintosh boot block.
Last block on Volume: Copy of Master Directory Block
Block 0: SCSI partition information

The important thing to note from this diagram is that the Macintosh volume is a subset of the entire SCSI Disk. There can be more than one Macintosh volume on a given disk, or even other volumes that are not Macintosh volumes.

1) Check the SCSI port:

Immediately after the RAM check, the system looks at the SCSI port to see if there are any drives connected. If a SCSI drive is found the system reads the SCSI partition information in block 0. This block is specific to SCSI drives and is always found at block 0 of the disk. The SCSI Manager then reads in the SCSI driver from the disk. Once the driver is loaded into memory, the system will use the driver to read and write blocks from the disk, instead of the ROM boot code. The driver reads and writes blocks relative to the beginning of the Macintosh volume on the SCSI drive, which can start anywhere on the physical disk.

2) Decide which disk is to be the startup disk:

The Macintosh then looks at the floppy disks to see if there is a disk that it should try to use. If so, it will always boot from the floppy. If there are no floppy disks, the startup hard disk is chosen. The Macintosh boot blocks are read off of the chosen disk to determine if the volume is bootable. The two Macintosh boot blocks (same boot blocks as those found on floppies) are read using the SCSI Driver. The Macintosh boot blocks are found as the first two blocks on the Macintosh **volume**, but are much higher in terms of where they are found on the **disk** itself. See the figure for the difference between the Macintosh volume and the SCSI disk. The driver cannot normally read the SCSI partition information, or any blocks outside of the Macintosh volume.

3) Execute the Macintosh boot blocks:

The boot blocks are composed of strings and parameters which determine various system functions, and code that finishes the job of booting the system.

The hard disk is mounted as a volume, using the `PBMountVol` call. The volume has the two Macintosh boot blocks, as well as the volume header. The `PBMountVol` will use the driver to read the volume header and other information from the disk. Once the volume is mounted, there are only volume reads and writes, and the driver is responsible for the actual SCSI disk reads.

The System file is opened on the volume. The patch code for the current ROM is read into the system, including the patches to the SCSI Manager.

The Finder is launched.

4) The Finder uses the Desktop file on the volume to draw the desktop.

The Icons that make up the desktop representation of the Macintosh volume are stored in the Desktop file. The Desktop file is invisible and used only by the Finder.

That is a rather simplistic view of the boot process. There are a number of complications that arise due to the wild variety of devices that can be attached to a Macintosh. The full boot process is essentially a series of special cases, leading to the final booted System at the Finder's desktop (or in the startup application). The following section will go into painstaking detail in order to give you enough information to determine what step in the boot process failed.

Tough Boots

To further explain the boot process:

1) Check the SCSI port:

a) Before starting the boot process, the screen will be filled with a grey pattern.

b) Before the Macintosh will check for any SCSI devices, it will first reset the SCSI bus using a `SCSIReset`. This is to make sure the bus was not left in a bad state.

c) The Macintosh will then start a cycle through all 7 SCSI IDs (from 6..0) to see which disks are connected, and keeps a table of all disks that are connected.

d) For each disk that is connected to the Macintosh, the ROM boot code will use the SCSI Manager to read in the SCSI partition information to find where the driver is located on the disk. The signature of the SCSI partition information is also checked to be sure that the device is valid.

e) The SCSI Manager will then be used to read the driver into memory. Once the driver is loaded for a given disk, the driver is called to install itself. The driver will usually post a Disk Inserted event to have its volume mounted by the Finder.

f) Steps d and e are repeated for each disk connected. At this point, there may be a number of drivers in memory, but there are no volumes, since none have been mounted yet. Generally there is one driver per disk, but some drivers can handle more than one disk at a time.

2) Decide which disk is to be the startup disk:

a) The next stage is to determine which volume will become the startup disk. If there is a floppy available it will always be the startup disk. During this process the disk chosen as the startup disk is not known to be valid. The System file and boot blocks are checked later.

b) The standard HD-20 is connected to the system in a fashion that is very similar to a floppy, so if a bootable HD-20 is connected it will be the startup disk.

c) There is no search for floppy devices like there is for SCSI disks since the driver for the floppies will post a Disk Inserted event when it detects a floppy in the drive. The first floppy device that is found will be used as the startup disk. If there are multiple floppy devices, the others will be mounted by the Finder, not at boot time. The SCSI devices that are online are not mounted at this time, either. There is a pending Disk Inserted event for each disk that will be handled by the Finder.

d) At boot time, there is only one volume that is mounted (during execution of the Macintosh boot blocks). The others will be mounted when their Disk Inserted event is processed at a `GetNextEvent` call.

e) On the new Control Panel there is a Control Device (cdev) called the Startup Device. This Startup Device cdev allows the user to choose which device the system should try to boot from first. This can only be used on the Macintosh II and SE. The drive number, driver reference number, and driver OS type are stored in parameter RAM to allow a chosen device to be the boot disk. The floppy drives will still have precedence over the SCSI devices. The standard HD-20 can be chosen as the Startup Device as well, since it uses a different driver reference number. If the drive number that is stored as the Startup Device is invalid, or had a read/write error, then another disk in the chain will be chosen as the next bootable candidate. Remember that there is only one boot/startup/system disk, and it is the only one that is explicitly mounted at boot time. All other devices in the system will be handled once the system is booted.

3) **Execute the Macintosh boot blocks:**

a) Once the Startup Disk has been chosen (whether floppy, SCSI or other disk) then it is time to read the Macintosh boot blocks off of blocks 0 and 1 of the volume. Those boot blocks determine various parameters in the system, such as whether a Macsbug-like debugger will be loaded, the name of the startup program (not always the Finder), how big to make the event queue, how big to make the system heap, and so on. They also contain a signature identifying them as Macintosh boot blocks, and a version number to differentiate between different boot blocks.

b) After the boot blocks are read and the signature verified, the smiling Macintosh is displayed on the screen. The smiling Macintosh basically means that valid Macintosh boot blocks were found.

c) On 64K ROMs the boot blocks are executed by jumping to the code that follows the header information in boot block 0. On the newer Macintoshes the boot block version number is checked, and if it is 'old' the boot blocks will be skipped. The same code that would have been found in the boot blocks is found in the ROM itself. Regardless of which kind of Macintosh it is, the following steps apply. For the newer Macintoshes the boot blocks are usually used only for the parameters stored in the header.

d) Do the `PBMountVol` on the chosen startup volume. If `PBMountVol` fails, the process starts over at the point where a startup disk is being chosen (step 2 above). The failing volume is marked out of the list of candidates so that it won't be used again.

e) Find the System file and create a Working Directory, if needed, for the System folder. This is only done for HFS volumes of course, and the directory ID is set to the blessed folder. The blessed folder is saved in the volume header as part of the `FinderInfo` field. See Technical Note #67 for more information on the blessed folder. If the directory ID is wrong, the System file won't be found, causing it to start over again (at step 2 above). If the Working Directory was created successfully, that `WDRefNum` is set as the default volume with `SetVol`.

f) The System file is opened with `OpenResFile`. If the file could not be opened, the process starts over again at the point where a suitable boot device is being chosen (step 2 again).

g) The Startup Screen is loaded and displayed. If there was no Startup Screen, the normal "Welcome to Macintosh" message will be displayed. The Startup Screen or "Welcome..." means that the System file was found and opened successfully. On the Macintosh Plus and 64K ROM machines, the Startup Screen is displayed before the System file is opened. (reverse steps f and g)

h) The debugger and disassembler are installed if found. The names of the debugger and disassembler are found in the header of the boot blocks and are usually Macsbug and Disassembler respectively.

i) The **data** fork of the System file is opened and executed. The data fork contains code to read in the PTCH resources which patch the ROM.

j) The INITs that are in the System file are executed. The last INIT is INIT 31 which then looks in the System Folder for other INITs to be executed.

k) The file specified by the boot blocks as the startup application (Set Startup at the Finder) is found on the volume, using another field in the `FinderInfo` field of the volume header in order to get the Directory ID. If the file exists, it is launched. If not, the Finder is launched. If the Finder is not found, `SysError` is called with error code of 41 which is the "Can't launch Finder" alert.

4) The Finder uses the Desktop file on the volume to draw the desktop.

If the startup application was the Finder, it opens the Desktop file on the startup volume in order to draw the desktop. When it finishes with the startup volume, it calls `GetNextEvent`. If there are any pending Disk Inserted events, the volume specified is mounted (by the ROM) and the result passed to the Finder. If `PBMountVol` failed for any reason, the bad result will be passed to the Finder. At that point the Finder would put up the "This disk is damaged" alert and ask if the volume should be initialized or ejected. If ejected, the driver for that volume still exists, but the volume is unmounted. For each volume that the Finder sees, it opens the Desktop file on the volume to get the information that it needs to build the desktop. If the Desktop file was not found on a volume, it is created. If there are any errors while creating or using the Desktop file, the Finder will display the "This disk needs minor repairs" message. If the OK button is clicked, the Finder will delete the old file and create a new one. If that fails, the volume is unmounted and deemed unusable by the Finder. This happens if the disk is locked, or too full to add a Desktop file. If that was the startup volume, the computer is rebooted since it was forced to unmount the startup volume, and cannot run if there is no startup volume.

If you follow the previous sequence closely, you can predict what errors are causing a given end result. For example, if you have the effect where the smiley Macintosh appears, but immediately goes away and the disk does not boot, you can look through the sequence to see what might be going wrong. In this case, we know that the boot blocks were found on our startup volume, since the smiley Macintosh was displayed. We know that the System file was not found, or failed to open, since we never got the Welcome message. This usually calls for throwing away all of the System Folders on the volume, and starting again with a new System Folder to fix the problem. If there is more than one System Folder on a volume it is possible to confuse the system.

Other tidbits of information that may be useful (in no particular order) some which will be mentioned in the step-by-step operation below:

1) The SCSI cables have a lot of wires in them, and are rather bulky because of it. It is best to avoid bending the cables too much or too often, since the wires inside will break if overstressed. Don't put wild kinks in the cable in order to make it fit behind the Macintosh.
2) If there is no default volume stored in the parameter RAM with the Startup Device cdev, then the first drive that is in the drive queue will be the Startup Device. Since SCSI drives are added in highest ID order, that means the larger SCSI IDs will have a higher 'priority'. Macintosh IIs will default to the internal hard disk.
3) If the parameter RAM is trashed for some reason, the boot process can fail since a driver OS type is stored as well. If the OS type is wrong, the ROM will skip that driver, making the disk unbootable. On the Macintosh II/SE, the battery is no longer removable to fix parameter RAM problems. To correct this problem the Control Panel now has a feature that will allow you to clear parameter RAM. Holding down the Option-Command-Shift keys while opening the Control Panel will reset parameter RAM, forcing it to be rebuilt and therefore losing all of your settings, but possibly fixing some booting problems.

4) The Macintosh II and SE both have a new feature that will allow you to skip having the any hard disk mounted. Holding down the Option-Command-Shift-Delete combination will have the startup code skip the SCSI hard disks on the system. This can be useful if you are booting an old System file that does not understand HFS disks (like System 2.0/Finder 4.1), and want to avoid having your hard disks on line while you do something shaky. With external hard disks it is easier to just turn them off, but with internal disks it is not so easy.
5) Since the parameter RAM can be trashed in a manner that makes it impossible to boot a volume (looking for the wrong OS type), a new feature was added to the HD SC Setup program to have it fix this problem as well. If you have version 1.3 or greater, the parameter RAM bytes that determine booting will be reset to fix some boot problems that occur. The parameter RAM is fixed when the Update button is clicked. This does not invalidate the rest of parameter RAM, it merely fixes the bytes used for the Startup Device.
6) When the Finder copies a new System Folder onto a disk that does not already have a System Folder, that new folder will become the blessed folder. Its Directory ID will be saved in the volume header. In addition, the Macintosh boot blocks will be copied from the current startup device to the destination device. This is the best way to fix System Folder or Macintosh boot block problems. In order for the blessed folder to be set correctly, all System Folders on the volume should be deleted before copying the new folder there.
7) If the Desktop file is damaged for whatever reason, it can be deleted with a number of programs. This will force the Finder to rebuild it from scratch. You can also have the Finder rebuild the Desktop file by holding down the Option-Command keys when the Finder is launched. When the Desktop file is rebuilt you lose the Finder Comments in the Get Info boxes.
8) On the 64K ROMs, whenever something goes wrong during booting (like System file not found, bad boot blocks, and so on) the Sad Mac Icon is displayed. Starting with the 128K ROMs, whenever something goes wrong the ROM jumps back to the start to try to find another disk to use.

Bo3b's Boot Repair

This section will detail step-by-step processes that can be used to fix some common booting and volume problems. It is not intended to cover every possible case. The purpose of the preceding sections was to give you the information that will allow you to figure out what might be going wrong.

For most hard disk users, it is not sufficient to merely have the device running. It is generally a good idea to make the system as robust as possible in order to avoid some of the problems that might cause a volume to become wholly unreadable. The ultimate fix is to reinitialize the volume from scratch and rebuild the volume with the Finder or a restore operation that uses the File Manager. This is guaranteed to fix anything except hardware problems, and will give you the most solid system. If your system is acting funny, you can try the following sequence that is the next best thing to initializing the disk. This sequence will not make you rebuild the disk, but can be fooled by some disk problems. If everything passes, then the disk is in good shape; maybe not perfect, but good.

1) Power down the entire system, including the hard disk that is suspect.
2) Run the HD SC Setup program (or equivalent) and Update the drivers on the disk. For HD SC, this also fixes the parameter RAM. For non-Apple drives, the parameter RAM can be reset with the Control Panel.
3) Run the Test Disk option in HD SC Setup (or equivalent). If the test fails, reinitialize the volume, since it is not worth risking future problems.
4) Run the Disk First Aid utility. This utility will work on all HFS volumes. Have it check the volume for consistency. If it reports any errors, you can have it fix the problem, but the safest tack is to reinitialize. There are some problems that Disk First Aid won't catch. If Disk First Aid says the volume cannot be verified, it is time to reinitialize.
5) Rebuild the Desktop file by holding down Option-Command when returning to the Finder.

If you can successfully perform all of these steps, the volume will be as solid as it can get without reinitializing the disk. If things are still funny, it is time to take the last recourse, reinitialize.

Based on the previous sections, it is now time to go through all of the Nasties to give a step-by-step sequence for fixing these problems.

1) The data is intact, but the hard disk won't boot.

This is for the case where the volume won't boot, but if the computer is booted with a floppy disk the volume shows up at the desktop and can run normally. For this case, we know that the driver is being loaded and working, since the volume shows up at the desktop. The volume is also mountable, since it shows up with no problem. This implies that the Macintosh boot blocks are wrong, or the blessed folder is wrong. Clues such as the smiling Macintosh can tell you how far the process got before it failed. For example, if the smiling Macintosh never appeared, we know that Macintosh boot blocks were not read successfully. When the volume is fixed and bootable, it would be a good idea to go through the steps above to make the volume as solid as possible.

The sequence to follow:

a) Power down the entire computer, including the hard disk. Try to boot again. If it works, you are done.
b) Use the Control Panel's Startup Device to set the hard disk as the Startup Device. This will also reset some of the bytes in parameter RAM. Try rebooting to see if it has fixed the problem.
c) Run HD SC Setup (or equivalent) and perform the Update Drivers procedure. In the HD SC Setup case this will also rewrite the parameter RAM. If you are not using HD SC Setup, blast the parameter RAM with the Control Panel. Try rebooting.
d) Delete **all** System Folders from the hard disk. Using Find File or something similar, be sure that there are no stray copies of the System or Finder buried in some long lost folder. Copy a new System Folder to the volume, using the Finder. This process will fix bad boot blocks, as well as a bad blessed folder. Try rebooting.
e) If it still won't boot, there is something very strange happening. Whenever things get too weird it is usually time to start over: reinitialize.

2) The disk won't boot and only shows up after running HD SC Setup.

The disk does not even show up at the Finder when the system is booted with a floppy. After running the HD SC Setup (or equivalent) the volume will appear on the desktop and be usable. The HD SC Setup and most similar utilities will do an explicit `PBMountVol` of the volume in order to make the volume usable. Since the volume does not show up at the Finder at first, this implies that the driver itself is not getting loaded or is working improperly, since there was no Disk Inserted Event for the Finder to use.

The sequence:

a) Power down completely, including the hard disk.

b) Run HD SC Setup (or equivalent) and Update the Drivers. For non-Apple drives, update the drivers on the volume (this rewrites the SCSI partition information as well) using the utility that came with the disk. Reset the parameter RAM using the Control Panel.

c) If it still cannot be booted or does not show up at the Finder after booting with a floppy, the volume is too weird and should be reinitialized.

3) The disk will boot but hangs part way through the boot process.

This is when you can see the volume is being accessed by the run light (LED) on the front panel, and the booting seems to work but never makes it to the Finder. This implies that all is well until the System tries to actually launch the Finder or Startup Application. It could also be that the System file is causing something to hang.

The sequence:

a) Power down completely.

b) Boot with a floppy so that the floppy is the startup disk and the volume in question can be seen at the Finder.

c) Delete all System Folders on the hard disk. Put a new System Folder on the disk. This will presumably fix a corrupted System file.

d) If still funky, show the disk who's boss.

4) There are data errors while the disk is running.

This case usually evidences itself by messages at the Finder when trying to copy files. Messages like "The file ^0 could not be read and was skipped" usually mean that the drive is passing back I/O errors. This usually means that there is a hardware failure, but it can occasionally be caused by bad sectors on the disk itself. If the sectors are actually bad, it is generally necessary to reinitialize the volume.

The sequence:

a) Power down completely. Reboot and see if the same file gives the same error.

b) Run the HD SC Setup (or utility that came with your drive) and perform the Test operation. This will fail if there are bad blocks on the device. If there are bad blocks, it is necessary to reinitialize the volume.

c) Check the SCSI terminators to be sure they are plugged in correctly. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of this wire that goes through all of the devices. If you have a Macintosh II or SE with an internal drive, that drive will already have a terminator inside the Macintosh at the front of the cable.

d) Make sure the SCSI cables you are using are OK, by swapping them with known good ones. If the problem disappears, the cable is suspect.
e) Swap the terminators in use with known good ones to be sure they are OK.
f) Try the drive and cable on a different Macintosh to be sure the Macintosh is OK.

5) The disk is very slow returning to the Finder.

If the computer has gotten slower with age, it is probably due to a problem with the Desktop file. If a volume has been used for a long time, the Desktop file can grow to be very large (Hundreds of K). Reading and using a file that big can slow down the Finder when it is drawing the desktop. If you have a large number of files in the root directory, this will also slow the computer down. A large number (500-1000) of files in a given folder can cause performance problems as well. If a volume has been used for a long time, it can also have become fragmented.

The sequence:

a) Rebuild the Desktop file and see if it gets faster.
b) Look for large numbers of files in a given directory and break them up into other folders if needed.
c) Run Disk First Aid to be sure the volume is not damaged.
d) Reinitialize the volume and restore the data using File Manager calls to fix a fragmentation problem. Using the Finder, or a backup program that reads and writes files is a way to use only File Manager calls. You cannot fix a fragmentation problem by doing an image backup and restore.

6) The computer crashes or hangs when returning to the Finder.

This can happen if the Desktop file becomes corrupted. There are occasions when this can happen if the HFS structures on the volume are damaged.

The sequence:

a) Rebuild the Desktop file.
b) Run Disk First Aid to be sure the volume is not damaged; a boot floppy with the Set Startup set to Disk First Aid can allow you to test a volume that cannot be displayed at the Finder.
c) The path of ultimate recourse if nothing else seems wrong with the volume.

7) The disk appears in a “This disk is bad” dialog.

This is the worst of the possible errors that generally happen to hard disks. If the message is “This disk is bad” or “This is not a Macintosh disk”, the HFS structures on the volume have been damaged. In particular, the Master Directory block on the volume has been damaged. The driver and SCSI partition information are probably OK, since this dialog shows up when the Finder tries to mount a damaged volume. This means that the `PBMountVol` call failed. Don’t click the Initialize button unless you are sure you want the volume to be erased. In these cases, it is nearly always better to just reinitialize the volume after you have saved whatever information you can.

The sequence:

a) Power down completely. Occasionally the controller in the hard disk itself can crash.
b) Run Disk First Aid. For these cases, it is usually necessary to create a boot floppy with Set Startup set to Disk First Aid. When the floppy is booted, Disk First Aid will be run before the Disk Inserted events are processed. When Disk First Aid sees the Disk Inserted event it will check the result from the `PBMountVol` and still allow you to test the volume, even if it can't be mounted.
c) If Disk First Aid cannot repair the disk, it might be worth writing a simple program to call the driver to read and write blocks. There is a copy of the Master Directory Block on the end of the volume, and the volume can sometimes be fixed by copying that block over a damaged block in sector 2. You can write a program that will find out how big the volume is by looking in the Drive Queue Element for the volume, reading the block that is one sector from the end (N-1), and writing that copy over sector 2. At this point, the volume is probably inconsistent, but it may allow you to use it long enough to get information off of it. It is sometimes possible to have Disk First Aid repair the volume at this point as well. Copying the sectors can also be done with sector edit utilities, if you can get them to recognize the volume at all.
d) If making a new copy of sector 2 does not work, but the driver is still being loaded at boot time, it is possible to write a program that will read sectors from the disk looking for information that you might need. You can have a reader program go through blocks looking for a specific pattern, like a known file name. This is usually done in desperation, but sometimes there is no other choice. If the data desired can be found in some form, it can sometimes be massaged back to a useful form much easier than recreating it.
e) Sometimes the volume will be so badly damaged that the SCSI partition information is also damaged and cannot be fixed with the Update in the hard disk utility. In this case, it is usually still possible to perform direct SCSI reads, without going through the driver. Using the driver is preferable, since it knows how to talk to the drive better than you would, but sometimes the driver is not available. Using direct SCSI reads should be a last ditch effort since the SCSI Manager can be very challenging to use. This should only be used if there is irreplaceable data on the volume that cannot be read by any other means.
f) Even if the volume is recovered, it still should be reinitialized (after the data is recovered) to be sure that any hidden damage is repaired.

8) The disk never shows up at all.

The disk appears to be missing. The volume does not show up at the Finder, and does not show up in HD SC Setup. At boot time the access light (LED) does not flash. This is usually a hardware problem as well. The drive is not responding to SCSI requests at all, so the system cannot tell a drive is attached.

The sequence:

a) Power down the system, including the hard disk.
b) Make sure that the SCSI ID on the drive does not conflict with any other in the system, including the Macintosh, which is ID 7. (If you have an internal hard drive, it should be ID 0.)

c) Check the SCSI terminators to be sure they are plugged in correctly. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of this wire that goes through all of the devices. If you have a Macintosh II or SE with an internal drive, that drive will already have one terminator inside the Macintosh at the front of the cable.

d) Make sure the SCSI cables you are using are OK, by swapping them with known good ones.

e) Swap the terminators in use with known good ones to be sure they are OK.

f) Try the drive and cable on a different Macintosh to be sure the Macintosh is OK.

These boots are made for wokking

Remember, the goal here is to make the system be as stable as possible. If things are acting strange, it doesn't hurt to go through the entire process of testing the drive. The test procedure takes a little time but is non-destructive for the data that is there. If something catastrophic has happened to the disk, it is better to spend some time backing up the data, initializing the volume, and restoring the data than it is to lose some work later on due to some other permutation of the same problem. Unless you are sure that the volume is in an undamaged state, you are better off using a file-by-file backup operation than an image backup, ~~since the image backup will~~ copy any damage as well as the data.

If there are situations that you run into that are not covered by this technical note, please let us know so that they can added.

If this technical note helps even one person save some data that would otherwise be lost, it will have been worthwhile. Hope it helps.

#135: Getting through CUSToms

See also: Technical Note #88—Signals
Technical Note #110—MPW: Writing Standalone Code

Written by: Rick Blair July 1, 1987
Updated: March 1, 1988

This technical note provides a way for developers to allow sophisticated users to add code to an off-the-shelf application. Using this scheme, the user can easily install the code module; the application has to know how to call it and, optionally, be able to respond to a set of predefined calls from the custom package.

Note

The following code makes heavy use of features of the Macintosh Programmer's Workshop. It also assumes a basic familiarity with the standard Sample program included with MPW. The Pascal code (which is here only as an example implementation of the mechanism) is presented as only those sections which *differ* from Sample.p. The assembly language code also includes MPW-only features, such as record templates. Some of these are explained in Technical Note #88, "Signals."

In addition, since the order in which parameters to various routines are passed is critical, special care will have to be taken in writing interfaces for use with C. It is probably best to declare them as Pascal in the C source.

Concepts

Basically, we create a code resource of type CUST with an entry point at the beginning which takes several parameters on the stack; this code is reached via a dispatching routine which is written in assembly language.

The data passed on the stack to this dispatcher includes:

- a selector (to specify the operation desired)
- the address of a section of application globals (for communication back and forth between the application and the module when the stack parameters are insufficient)
- a handle which references the custom code resource on the stack.

Other parameters may be added (as long as they are pushed on the stack before the required ones) if desired. Since these extra parameters would **always** have to be included in any calls to a given package, it might be more convenient to use the application global space area which is accessed through the `appaddr` parameter.

Template

Your application must contain the following global data and procedure declarations to support this model:

```
VAR
      custhandle: Handle;

      {the following globals constitute the data known to the custom code}
      appdispatch: ProcPtr; {address of dispatch routine custom code can call}
      {examples of further application globals for the custom package:}
      (*
      paramptr: Ptr; {general pointer used as param. to appdispatch code}
      paramword1: INTEGER;
      paramword2: INTEGER;
      CUSTerr: INTEGER;
      *)
      {any other globals the module should get at}

    {the two assembly language glue routines which are linked into the
     application}
    PROCEDURE CustomInit(resID: INTEGER; VAR custhandle: Handle);
     EXTERNAL; {the routine used to set up the custhandle resource handle}

    PROCEDURE CustomCall({application & package-specific paramters}
                selector: INTEGER; appaddr: UNIV Ptr; ourhandle: Handle);
     EXTERNAL; {this is the code dispatcher}

   {this is called by the custom package to perform a service which is more
     easily provided by the application; since we pass a pointer to it to the
     package, CustDispatch must be at the outermost nesting level in the main
     segment}
    PROCEDURE CustDispatch(selector: INTEGER);

     BEGIN
        CASE selector OF
         {.
          .
          .}
        END; {CASE}
     END; {CustDispatch}

   {your initialization code should contain the following:}

     {Custom package initialization stuff}
     appdispatch := @CustDispatch; {put pointer where the package can see it}
     CustomInit(69,custhandle);    {our CUST resource has ID = 69}

  {then whenever you want to invoke the package you use CustomCall}
```

You must also assemble CustomInit and CustomCall and link them with into your application. The custom package itself can be written in any language which can produce stand-alone code. See Technical Note #110 for how to write stand-alone code in MPW Pascal.

The example

`CustomCall` is only referenced once in this example. When a variety of unrelated functions are provided, however, it is more convenient to provide a separate interfacing procedure to invoke each one and have them make their own `CustomCall` calls.

Note that this example is somewhat contrived; you probably wouldn't "externalize" the code for finding a word or sequence of characters like this. This is an idealized situation. More realistic uses would be: to add-on special routines to a database to perform custom calculations or the like; allow for localization when code is required (and hooks aren't already provided); let documents carry around code which may vary among software versions, etc. so that older documents would be able to work alongside the new ones, etc.

What it does

We simply add a new menu to the sample program which allows **Find** by characters or word. We just pass the menu item to the package and let it do the finding; it then calls back to the application dispatch routine to highlight text or display the "not found" message.

The Pascal source for the example application appears first:

```
{$R-}
{$D+}
PROGRAM P;

USES
 {$LOAD ::PInterfaces:most.dump}
 Memtypes,Quickdraw,OSIntf,ToolIntf,PackIntf {,MacPrint}
 {$LOAD}
 , {$U ErrSignal.p} ErrSignal;

CONST
 appleID = 128; {resource IDs/menu IDs for Apple, File and Edit menus}
 fileID = 129;
 editID = 130;
 findID = 131;

 appleM = 1; {index for each menu in myMenus (array of menu handles)}
 fileM = 2;
 editM = 3;
 findM = 4;

 menuCount = 4; {total number of menus}
```

```
    windowID = 128; {resource ID for application's window}

    undoCommand = 1; {menu item numbers identifying commands in Edit menu}
    cutCommand = 3;
    copyCommand = 4;
    pasteCommand = 5;
    clearCommand = 6;

    findcharsCommand = 1; {menu items for Custom menu}
    findwordCommand = 2;

    aboutMeCommand = 1; {menu item in apple menu for About sample item}

    aboutMeDLOG = 128;
    findDLOG = 129;
    infoDLOG = 130;

    {application dispatching code selectors}
    hilightSel = 0;
    notifySel = 1;

  VAR
  •
  •
  •
   errCode: INTEGER;
   dlogString: Str255;
   custhandle: Handle;

   {here is the area known to the custom code}
   appdispatch: ProcPtr; {address of dispatch routine custom code can call}
   {examples of further application globals for the custom package}
   paramptr: Ptr; {general pointer used as param. to appdispatch code}
   paramword1: INTEGER;
   paramword2: INTEGER;
   {any other globals the module should get at}


PROCEDURE CustomInit(resID: INTEGER; VAR custhandle: Handle);
 EXTERNAL; {the routine used to set up the custhandle resource handle}

PROCEDURE  CustomCall(text: Ptr; count: INTEGER; findstr: StringPtr;
                selector: INTEGER; appaddr: UNIV Ptr; ourhandle: Handle);
 EXTERNAL; {this is the code dispatcher}


{this will do the "about" dialog and the info dialog requested by the
custom pack.}

PROCEDURE ShowADialog(meDlog: INTEGER);

 CONST
    okButton = 1;
    authorItem = 2;
    languageItem = 3;
    infoItem = 2;
```

```
VAR
   itemHit,itemType: INTEGER;
   itemHdl: Handle;
   itemRect: Rect;
   theDialog: DialogPtr;

BEGIN
   theDialog := GetNewDialog(meDlog,NIL,WindowPtr( - 1));

   CASE meDlog OF
      aboutMeDLOG: BEGIN
      GetDitem(theDialog,authorItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,'Ming The Vaseless');
         GetDitem(theDialog,languageItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,'Pascal et al');
      END;

          infoDLOG: BEGIN {display the message requested by the custom
                          package}
         GetDitem(theDialog,infoItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,StringPtr(paramptr)^);
      END;
   END; {CASE}

   REPEAT
      ModalDialog(NIL,itemHit)
   UNTIL (itemHit = okButton);

   CloseDialog(theDialog);
END; {of ShowADialog}


{this will put up the Find dialog to allow the user to type in the
characters to search for}
FUNCTION DoCustomDialog: BOOLEAN;

CONST
   okButton = 1;
   cancelButton = 2;
   fixedItem = 3;
   editItem = 4;

VAR
   itemHit,itemType: INTEGER;
   itemHdl: Handle;
   itemRect: Rect;
   theDialog: DialogPtr;

BEGIN
   theDialog := GetNewDialog(findDLOG,NIL,WindowPtr( - 1));
   GetDitem(theDialog,editItem,itemType,itemHdl,itemRect);
   SetIText(itemHdl,dlogString);
   TESetSelect(0,MAXINT,DialogPeek(theDialog)^.textH);
```

```
      REPEAT
         ModalDialog(NIL,itemHit)
      UNTIL (itemHit IN [okButton,cancelButton]);
      GetIText(itemHdl,dlogString);
      DoCustomDialog := itemHit = okButton;

      CloseDialog(theDialog);
   END; {of DoCustomDialog}


  PROCEDURE DoCommand(mResult: LONGINT);
  •
  •
  •
  (* partial procedure fragment *)

  {here is one of the case sections for the DoCommand procedure}

        findID:
           IF DoCustomDialog THEN
              BEGIN
              MoveHHi(Handle(textH)); {stop it from fragmenting the heap}
              WITH textH^^ DO BEGIN
               HLock(hText); {since we don't know what the package might
                                              be up to}
               {now call the package to find characters or words}
                CustomCall(POINTER(ORD(hText^) + selEnd),
                 teLength - selEnd, @dlogString, theItem, @appdispatch,
                 custhandle);
                HUnLock(textH^^.hText);
              END; {WITH}
              END;

      END; {OF menu CASE} {to indicate completion of command,}
      HiliteMenu(0); {call Menu Manager to unhighlight }
      {menu title (highlighted by     }
      {MenuSelect)         }
   END; {OF DoCommand}

{this is called by the custom package to set the new selection or display a
  message; it must be in CODE 1 at the outermost lexical level}
  PROCEDURE CustDispatch(selector: INTEGER);

   BEGIN
      CASE selector OF
         hilightSel: {hilight the characters selected by the custom pack.}
         {paramptr=pointer to text to select, paramword1&paramword2=start,end
                chars}
            WITH textH^^ DO
             {we'll subtract the start of text from paramptr to get the base
                offset...}
                  TESetSelect(ORD(paramptr) - StripAddress (ORD(hText^)) +
                      paramword1, ORD(paramptr) - StripAddress (ORD(hText^))
                       + paramword2,textH);
```

```
       notifySel: {put up message per request from custom pack.}
       {paramptr points to string to display}
          ShowADialog(infoDLOG);

    END; {CASE}
 END; {CustDispatch}

BEGIN {main program}
 { Initialization }
 InitGraf(@thePort); {initialize QuickDraw}
 InitFonts; {initialize Font Manager}
 FlushEvents(everyEvent - diskMask,0); {call OS Event Mgr to discard
                                      non-disk-inserted events}
 InitWindows; {initialize Window Manager}
 InitMenus; {initialize Menu Manager}
 TEInit; {initialize TextEdit}
 InitDialogs(NIL); {initialize Dialog Manager}
 InitCursor; {call QuickDraw to make cursor (pointer) an arrow}

 InitSignals;
 errCode := CatchSignal;
 IF errCode <> 0 THEN BEGIN
    Debugger;
    Exit(P);
 END;

 SetUpMenus; {set up menus and menu bar}
 UnLoadSeg(@SetUpMenus); {remove the once-only code}

 {Custom package initialization stuff}
 appdispatch := @CustDispatch;
CustomInit(69,custhandle); {should test custhandle for NIL and alert
                             the user}
 dlogString := '';
...
{etc. with the rest of initialization and the main event loop}
END.

; now for the assembly language code
; first, the dispatching and initializing code that must be linked into
; the application
; CustomCalling
; Custom packages initializing and dispatching
;
;   Rick Blair         May, 1987
;
;          PRINT   OFF
;          INCLUDE 'Traps.a'
;          INCLUDE 'ToolEqu.a'
;          INCLUDE 'QuickEqu.a'
;          INCLUDE 'SysEqu.a'
;          PRINT   ON

           LOAD    'most.dmp'  ; from a dump of the files above

appdata    EQU                 12
```

```
;Initialize a custom module
; Pascal call format:
;  CustomInit(resID:INTEGER;VAR custhandle:Handle);
;
; This will load the CUST module with the given resource ID, install a
; handle to it in custhandle, and set the module's appdata pointer to
; point to the address appaddr.
;
resID       EQU                 8
custhandle  EQU                 4

CustomInit  PROC     EXPORT
            SUBQ.L   #4,A7        ;make room for handle from GetResource
            MOVE.L   #'CUST',-(A7)
            MOVE.W   resID+8(A7),-(A7);resource ID
            _GetResource
            MOVE.L   (A7)+,A0
            MOVE.L   custhandle(A7),A1
            MOVE.L   A0,(A1)      ;store handle in app's custhandle global
;(return with nil handle if GR failed)
            MOVE.L   (A7),A0      ;get return address
            ADD.L    #10,A7       ;strip everything
            JMP      (A0)         ;adieu


;Call a custom module
;Pascal format:
; CustomCall( {parameters as desired} selector: INTEGER; appaddr: Ptr;
;              module: Handle);
;
;This will call the code whose handle is passed on the stack. If the
;application was written in assembly language you would just
;dereference the handle and call it directly (you wouldn't need this at
;           all).
;
CustomCall  PROC     EXPORT
            IMPORT   Signal
            MOVE.L   4(A7),A0     ;get handle
            MOVE.L   (A0),D0
            BNE.S    @0           ;if hasna' been purged, ga' ahead
            MOVE.L   A0,-(A7)     ;push handle
            _LoadResource
            MOVE.W   ResErr,-(A7)
            JSR      Signal       ;Signal is a NOP if a zero is passed to it
            MOVE.L   4(A7),A0     ;handle again
; we don't lock the handle here (we can't save it so we can unlock it
;  later), so it's up to the package to lock/unlock itself
@0          MOVE.L   (A0),A0      ;dereference
            JMP      (A0)         ;call CUST code

            END
```

```
; here is the module for the custom package itself

; CustomPack
; Example custom code package
;
;   Rick Blair      May, 1987
;
; This demonstrates the recommend structure of a code module which a
; sophisticated user could add to an existing application which supported
; this mechanism. Aside from allowing for multiple routines within the
; module (via a selector), provision is made for calling a routine
; dispatcher within the application itself.

;Finding text
;We support a call to find a string anywhere within a block of text
; (selector=0), and one to find the string only as a separate "word"
; with spaces around it (selector=1).
;PROCEDURE CustomCall(text:Ptr; count:INTEGER; findstr:^STRING;
;                   selector:INTEGER; appaddr: UNIV Ptr; ourhandle:Handle);
;Rather than return a result indicating whether they succeeded or not,
;these routines take whatever action is appropriate (the application
;may not even know what these routines actually do).
;Once a call succeeds or fails, it then takes action by making a call to
;one of the services provided by the application. In this case the two
;functions provided are just what we need; the ability to select text and
;the ability to put up a message saying "Text not found".

           STRING   ASIS

;          PRINT    OFF
;          INCLUDE  'Traps.a'
;          INCLUDE  'ToolEqu.a'
;          INCLUDE  'QuickEqu.a'
;          INCLUDE  'SysEqu.a'
;          PRINT    ON

           LOAD     'most.dmp'  ; from a dump of the files above

CustPack   PROC     EXPORT

           BRA.S    Entry       ;skip header

           DC.W     0           ;flags
           DC.B     'CUST'      ;custom add-on code module
           DC.W     69          ;resource ID (picked by Mr. Peabody &
                                ; Sherman)
           DC.W     $10         ;version 1.0


StackFrame RECORD   {A6Link},DECR
paramsize  EQU      *-8
;       call-specific parameters… (optional)
text       DS.L     1           ;pointer to text block
count      DS.W     1           ;word count of characters in text
findstr    DS.L     1           ;pointer to p-string to find
;       selector(word, optional - you might only have 1 call)
selector   DS.W     1
```

```
fcharsCmd   EQU     1           ; selector for "find characters"
fwordCmd    EQU     2           ; selector for "find word"
;       pointer to app. globals(long)
appaddr     DS.L    1
;       handle to this resource(long)
ourhandle   DS.L    1
;       TOS:return address (long)
return      DS.L    1
;the stack link is built off the origin of the saved old A6 on the stack
A6Link      DS.L    1
LocalSize   EQU     *
            ENDR

;offsets into our application globals area
AppGlobals  RECORD  {appdispatch},DECR
appdispatchDS.L     1
paramptr    DS.L    1
paramword1  DS.W    1
paramword2  DS.W    1
;CUSTerr    DS.W    1           ;if we had possible errors
            ENDR

Entry
            WITH    StackFrame,AppGlobals
            LINK    A6,#LocalSize
;           MOVEM.L …           ;we'd save any non-trashable regs here
;first lock us down…
            MOVE.L  ourhandle(A6),A0
            _HLock

            MOVE.W  selector(A6),D0
            CMP.W   #fcharsCmd,D0
            BEQ.S   charfind    ;go find characters
            CMP.W   #fwordCmd,D0
            BEQ.S   wordfind    ;go find a word
;well, M. App didn't call us with a selector we know, so…

;unlock ourselves, clean up, return
; (if we wanted to return an error code we could stuff it into the app.
;  global area)
duhn        MOVE.L  ourhandle(A6),A0
            _HUnLock
;           MOVEM.L …           ;restore any registers here
            UNLK    A6
            MOVE.L  (A7)+,A0    ;return address
            ADD.L   #paramsize,A7;strip parameters
            JMP     (A0)

;selector codes for calls to application
hilight     EQU     0           ;highlight characters, please
notify      EQU     1           ;beep a little

;find the string "findstr" anywhere in the block "text"
charfind
            JSR     findchars   ;see if findstr is anywhere in text
            BEQ.S   nofind      ;if not then skip
            JSR     calcsels    ;compute selstart and selend
didfind     MOVE.L  appaddr(A6),A0 ;get pointer to appl. globals area
```

```
                MOVE.L  text(A6),paramptr(A0) ;setup text pointer and…
                MOVE.W  D0,paramword1(A0) ;start character position,
                MOVE.W  D1,paramword2(A0) ;end character position
                MOVE.W  #hilight,-(A7) ;pass proper selector
goapp           MOVE.L  appdispatch(A0),A0 ;get dispatch address
                JSR     (A0)            ;call the application to select the range
                BRA.S   duhn            ;return to application (dejà vu)

nofind          MOVE.L  appaddr(A6),A0  ;get pointer to appl. globals area
                LEA     oopstring,A1    ;get pointer to "Not found" message
                MOVE.L  A1,paramptr(A0) ;put string pointer in "paramptr"
                MOVE.W  #notify,-(A7) ;tell app. to display message
                BRA.S   goapp

;figure selstart and selend
calcsels        NEG.W   D0              ;negate # characters unskipped in text
                SUBQ.W  #1,D0           ;include 1st character
                ADD.W   count(A6),D0;compute 1st character position for select
                MOVE.L  findstr(A6),A1
                MOVE.B  (A1),D1         ;get length of string
                EXT.W   D1
                ADD.W   D0,D1           ;compute last char. pos. for select
                RTS

;find the characters, but only if surrounded by space (including end or
; beg.)
;we could extend the test to check for other delimiters (";",".",etc.)
wordfind
                JSR                     findchars
wloop           BEQ.S   nofind
                MOVE.W  D0,D2           ;save count of text remaining
                JSR     calcsels        ;figure start and end offsets
                MOVE.L  text(A6),A1 ;point to text
                TST.W   D0              ;start=beginning of text?
                BEQ.S   @0              ;yep, so it passes
                CMP.B   #' ',-1(A1,D0) ;preceded by a space?
                BNE.S   @1              ;nope, keep looking
@0              CMP.W   count(A6),D1 ;D1=length of text?
                BEQ.S   didfind         ;yep, so it passes
                CMP.B   #' ',(A1,D1) ;followed by a space?
                BEQ.S   didfind         ;yes, so we've found it

;this wasn't paydirt, so keep panning
@1              MOVE.W  D2,D0           ;restore chars remaining count
                BMI.S   nofind          ;forget it if we ran out of text
                JSR     bigloop         ;keep looking
                BRA.S   wloop


;this code will find the string if it lies anywhere in the text
findchars       MOVE.L  text(A6),A0 ;point A0 to chars to search
                MOVE.W  count(A6),D0;size of text block
bigloop         MOVE.L  findstr(A6),A1;point A1 to chars to find
                MOVE.W  (A1)+,D1        ;get length byte and 1st char. (skip 'em)
                CMP.W   #255,D1
                BGT.S   @1              ;enter loop if length<>0
                ADDQ.L  #4,A7           ;strip findchar's return address
                BRA     duhn            ;return having done nothing
```

```
;search for first character
@0          CMP.B    (A0)+,D1      ;this one match 1st character?
@1          DBEQ     D0,@0         ;branch until found or done 'em all
            BNE.S    cnofind       ;skip out if no match on 1st character

            MOVE.B   -2(A1),D1     ;length of findstr
            EXT.W    D1
            SUBQ.W   #1,D1         ;length sans 1st character
            BEQ.S    cfound        ;if Length(findstr)=1, we're done
            CMP.W    D1,D0
            BLT.S    cnofind       ;fail if findstr is longer than text left
            MOVE.L   A0,D2         ;save this character position
            CMP.W    D1,D1         ;force EQuality
            BRA.S    @3            ;enter loop

@2          CMP.B    (A0)+,(A1)+ ;match so far?
@3          DBNE     D1,@2         ;check until mismatch or end of findstr

            MOVEA.L D2,A0          ;restore position (cc's unaffected)
            BNE.S    bigloop       ;if no match then keep looking

cfound      MOVEQ    #1,D1         ;return TRUE
                     RTS

cnofind     SUB.W    D1,D1         ;return FALSE
                     RTS


            STRING   PASCAL
oopstring   DC.B     'Pattern not found.'

            END


#additions to the resource file


resource 'DLOG' (129, "Find dialog") {
        {72, 64, 164, 428},
        dBoxProc,
        visible,
        noGoAway,
        0x0,
        129,
        "Find"
};

resource 'DLOG' (130, "Info") {
        {66, 102, 224, 400},
        dboxproc, visible, nogoaway, 0x0, 130, ""
};
```

```
resource 'DITL' (130) {
        {
/* 1 */ {130, 205, 150, 284},
            button {
                    enabled,
                    "OK already"
            };
/* 2 */ {8, 32, 120, 296},
            /* info */
            statictext {
                    disabled,
                    ""
            }
        }
};

resource 'DITL' (129) {
        { /* array DITLarray: 4 elements */
          /* [1] */
          {64, 48, 84, 121},
          Button {
                  enabled,
                  "OK"
          };
          /* [2] */
          {64, 231, 84, 304},
          Button {
                  enabled,
                  "Cancel"
          };
          /* [3] */
          {8, 8, 24, 352},
          StaticText {
                  disabled,
                  "Find what?"
          };
          /* [4] */
          {32, 8, 48, 352},
          EditText {
                  disabled,
                  ""
          }
        }
};

resource 'MENU' (131, "Custom", preload) {
        131, textMenuProc, 0x3, enabled, "Custom",
        {
           "Find Chars…",
                   noicon, "F", nomark, plain;
           "Find Word…",
                   noicon, "W", nomark, plain
        }
};
```

```
type 'CTST' as 'STR ';

resource 'CTST' (0) {
        "Custom Application - Version 1.0"
};

include "CustomPack.code";


#  This makefile puts the program together incl. the CUST pack.

CustomTest          ƒƒ  CustomCalling.a.o CustomTest.p.o ErrSignal.a.o
# the predefined rule for assembly will build CustomCalling.a.o,
#  CustomPack.code
        Link CustomTest.p.o CustomCalling.a.o ErrSignal.a.o ∂
           "{Libraries}"Interface.o ∂
           "{Libraries}"Runtime.o ∂
           "{PLibraries}"Paslib.o ∂
           -o CustomTest
CustomPack.code     ƒ          CustomPack.a.o
        Link CustomPack.a.o -rt CUST=69 -o CustomPack.code
# Put the resource file together (including the custom code resource)
CustomTest          ƒƒ         CustomTest.r CustomPack.code
        Rez CustomTest.r -a -o CustomTest
```

#136: Register A5 Within GrowZone Functions

See also: The Memory Manager
Technical Note #25—Register A5 Within Trap Patches

Written by: Chris Derossi July 1, 1987
Updated: March 1, 1988

If you have a grow zone function, it may get called when a system routine is trying to allocate memory. Because this can happen, you can't be guaranteed that register A5 will be correct.

If your grow zone function depends on A5, you should save register A5, load A5 from the low-memory global CurrentA5 (a long word at $904), and restore the caller's A5 before you exit.

From high-level languages, you can also use the Operating System Utility calls SetUpA5 and RestoreA5 (page 386 of *Inside Macintosh Volume II*). SetUpA5 stores the 'old' A5 on the stack and puts the value stored at CurrentA5 into A5. Make sure to call RestoreA5 when you're done so that it can pop the saved value of A5 off the stack.

Your grow zone function depends on A5 if it does any of the following:

- Accesses your application's global variables (which are stored at negative offsets from A5).
- Accesses the QuickDraw globals. (A5 contains the address of a pointer to the QuickDraw global variables.)
- Makes any ROM trap calls.
- Makes any intersegment calls to routines in your application.

To do any of these, A5 needs to contain the value from CurrentA5. Please note that this is different than the method for calling the ROM from trap patches, where A5 should retain the value it had upon entry to your patch.

#137: AppleShare 1.1 Server FPMove Bug

See also:	*AppleTalk Filing Protocol*	
Written by:	Rich Andrews	June 16, 1987
Modified by:	Bryan Stearns	July 1, 1987
Updated:		March 1, 1988

A bug has been discovered in AppleShare 1.1's implementation of the AppleTalk Filing Protocol `FPMove` call. This bug **only** affects developers implementing custom workstation access code that will access AppleShare 1.1 servers from non-Macintosh systems (such as MS-DOS systems); if the guidelines below are not followed, data loss may result.

The AppleShare file server supports an AFP call known as `FPMove`, used to move a file or directory tree from one place to another on an AppleShare volume. In addition to moving, the caller can specify a new name for the file or directory being moved; in essence, a move and a rename can be accomplished by a single call.

The AppleShare 1.1 server implements this call as follows: the file is moved from the source directory to an invisible holding directory, renamed, then moved to the destination directory. The problem occurs when a locked file is moved and renamed in this manner: the initial move succeeds, the rename fails, and the file is left in the holding directory (essentially lost, as it will be deleted when the server is shut down).

Macintosh AppleShare 1.1 workstation software never uses the move-and-rename combination, so this problem cannot occur on a Macintosh; however, if you're implementing your own workstation-access software for some other machine or operating system, and wish to use this feature, you must follow this procedure:

When a move and rename call comes from the native file system, issue an `FPGetFileDirParms` call to see if the object is a locked file. If it is, issue an `FPSetFileParms` call to unlock the file. Then issue the `FPMove` call, followed by another `FPSetFileParms` call to lock the file again.

AFP does not allow locked files to be renamed, whereas some native file systems (such as MS-DOS) do. You must therefore preflight for this condition to maintain transparency.

This problem will be corrected in a future version of the AppleShare server software.

#138: Using KanjiTalk with a non-Japanese Macintosh Plus

See also: *KanjiTalk Usage Notes*
Script Manager Developers Package

Written by: Priscilla Oppenheimer July 1, 1987
Updated: March 1, 1988

This Technical Note describes the minor differences between using KanjiTalk with the Japanese Macintosh Plus and KanjiTalk with a standard Macintosh Plus.

There are two differences between the Japanese Macintosh Plus and the standard Macintosh Plus: The Japanese Macintosh Plus has the Kanji 12 and 18 point fonts in ROM and it is shipped with the Kana keyboard. It is not necessary to have this keyboard in order to use KanjiTalk. (See the KanjiTalk Usage Notes for details on how to use it with a non-Kana keyboard.) It is, however, necessary to have 12 point Kanji in order to use KanjiTalk; the 18 point Kanji is optional.

When using KanjiTalk with a standard (non-Japanese) Macintosh, the user supplies these fonts on disk and the Macintosh loads them into RAM. At boot time, the Macintosh looks for the 12 point Kanji font file in the system folder of the boot disk. If it cannot find the font, it will look through the root directory of all mounted volumes. (The font has to be at the root level; it cannot be in a folder.) If it still doesn't find the font, it will prompt the user to insert a disk with the font file in the root directory. Once KanjiTalk finds the 12 point font, it will go through the same process looking for the 18 point font. The user can cancel this search if the optional 18 point font is not necessary.

When KanjiTalk finds the fonts, it loads them into memory. The 12 point font takes up approximately 100K of memory and the optional 18 point font takes up approximately 250K of memory. The KanjiTalk code itself takes up about 180K of memory. Because the fonts take up quite a bit of memory, many applications will not work on a Macintosh 512K with the Kanji fonts installed.

Accessing the fonts from ROM is faster, but we have not noticed any significant speed problems when the fonts are accessed from RAM. There is, however, a noticeable difference in speed when the Macintosh is booted. It takes a couple of seconds to load the 12 point font and about 6 seconds to load the 18 point font.

Note that the Japanese Macintosh is unique; Apple has not produced other foreign versions of the Macintosh for different scripts. The introduction of the Arabic Interface System, for example, did not include an Arabic ROM version.

#139: Macintosh Plus ROM Versions

Written by: Cameron Birse July 1, 1987
Updated: March 1, 1988

Readers Digest condensed version of Macintosh Plus ROM history, or the truth according to Bo3bdar the everpresent:

1st version (Lonely Hearts, checksum `4D 1E EE E1`):

Bug in the SCSI driver; won't boot if external drive is turned off. We only produced about one and a half months worth of these.

2nd version (Lonely Heifers, checksum `4D 1E EA E1`):

Fixed boot bug. This version is the vast majority of beige Macintosh Pluses.

3rd version (Loud Harmonicas, checksum `4D 1F 81 72`):

Fixed bug for drives that return Unit Attention on power up or reset. Basically took the SCSI bus Reset command out of the boot sequence loop, so it will only reset once during boot sequence. This version shipped with the platinum Macintosh Pluses.

And Bo3bdar saith: “Thou shalt not rev them damn ROMs no more!”

Later that same day...

Bo3bdar Saith Also:

> Lonely Heifer was about a 2 byte change,
> Loud Harmonica was about 30 byte change.
> No other bug fixes in SCSI or elsewhere.
> Modified object code directly.
> Not possible to get a specific ROM since they are all the same part number.
> Shouldn't rely on a specific ROM, there will be no upgrade.
> Bo3b Bo3b a boola, a wiff Ba2m Bo1om.

#140: Why PBHSetVol is Dangerous

See also: The File Manager

Written by: Chris Derossi July 1, 1987
Updated: March 1, 1988

This note explains `PBHSetVol`, and why its use is not recommended.

`PBHSetVol`, like `SetVol` and `PBSetVol`, allows you to set the current default volume and directory to be used with subsequent File Manager calls. Unlike `SetVol` and `PBSetVol`, though, `PBHSetVol` lets you specify the volume and the directory separately, using the `ioVRefNum` and `ioWDDirID` fields.

`PBHSetVol` lets you specify a `WDRefNum` for the `ioVRefNum` in addition to a partial pathname in `ioNamePtr`. `PBHSetVol` will start at the specified working directory and use the partial pathname to determine the final directory. This directory might not correspond to an already existing working directory, so the File Manager cannot refer to this directory with a `WDRefNum`. Instead it must use the actual volume `refNum` and the `dirID` number (which is assigned when the directory is created, and doesn't change).

The net effect of all of this is, if you call `PBHSetVol`, the File Manager stores the actual volume `RefNum` as the default volume, and the default `DirID` separately. This happens on all calls to `PBHSetVol`. Subsequent calls to `GetVol` or `PBGetVol` will return only the volume `RefNum` in the `ioVRefNum` field of the parameter block. If any code tries to use the `RefNum` returned by `GetVol`, it will be accessing the root of the volume, and not the current default directory as expected.

This is particularly nasty for desk accessories because they don't know that your code has called `PBHSetVol` and they don't get what they expect if they call `GetVol`.

It is therefore recommended that you avoid using `PBHSetVol` because of this side effect. None of the other 'H' calls that allow you to specify a `DirID` do this, so they're still OK.

#141: Maximum Number of Resources in a File

See also: The Resource Manager

Written by: Cameron Birse July 1, 1987
Updated: March 1, 1988

This note describes the limitation of the number of resources in a single resource file.

There is a limit to the number of the resources in a single resource file. This limitation is imposed by the resource map. There are two bytes at the end of the resource map which are the offset from the beginning of the resource map to the beginning of the resource names list. If there is only one type of resource, then the overhead, from the beginning of the resource map to the beginning of the reference list, is 38 bytes. Since the offset is a two byte value, and is a **signed** number, its highest possible value is 32767. This is the limitation. If you subtract 38 bytes for the overhead, and divide the difference by 12 (the number of bytes for each reference) you get about 2727.4—the limit to the number of resources in a single file is 2727.

The Resource Manager was not intended to manage large numbers of resources, and as a result, its performance is particularly bad with many resources. Because of these restrictions, we recommend that developers avoid using the Resource Manager as a data base tool.

Macintosh Technical Notes

#142: Avoid Use of Network Events

See also: AppleTalk Manager

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

Future System software enhancements will not support network events. This note gives hints on weaning your application from the use of network events.

What are network events?

When the Event Manager was designed, an event number was reserved for future support of "network events". Later, when the AppleTalk Pascal Interfaces were written, a completion routine was created that, when an asynchronous AppleTalk operation finished, would post an event using `networkEvt` in the `evtNum` field.

Only the AppleTalk Pascal Interfaces generate network events. Assembly-language users of the AppleTalk drivers (and those who called the AppleTalk drivers directly from high-level languages, using `PBControl` calls) either provide a completion routine of their own, or poll the `ioResult` field of the parameter block passed with the call (when `ioResult` became negative or zero, the call is complete).

Why not use network events?

In some cases, network events can be lost. If the Event Manager finds that the queue is full while posting an event, it discards the oldest event. In a situation (such as a server) where multiple asynchronous ATP requests may complete at once, there is a chance that events may be dropped off the end of the queue. This is more likely if the same machine is also handling user-interface events (like keypresses and mouse actions).

Also, in developing improvements to our operating system, it has become apparent that to continue support of network events, we would have to compromise future enhancements to our system. So, future versions of the Macintosh operating system may ignore network events instead of passing them to the application.

How can I tell that my calls have completed without using network events?

As described on page II-275 of *Inside Macintosh*, you can poll the `abResult` field of the call's `ABusRecord`; when this value becomes negative or zero, the call has completed. You can do this in your main event loop.

With this technique, you can ignore any network events returned by `GetNextEvent`, since the AppleTalk Pascal Interfaces will be posting events anyway. If your application starts enough asynchronous operations, it's possible that their network events will cause other non-network events to be lost. To prevent this, you should call `FlushEvents(networkMask,0)` frequently to purge any accumulated network events from the event queue.

You may also consider using the new preferred high-level interface calls; see Technical Note #132 for more information.

#143: Don't Call ADBReInit on the SE with System 4.1

See also: The Apple Desktop Bus

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

Because of a bug (which causes auto-repeat) in the ROM version of the Macintosh SE keyboard driver, a patch was placed in System 4.1. If `ADBReInit` is called, the ROM version of the keyboard driver will be reloaded, and the RAM version of the driver with the patches will not be used. Therefore, it is recommended that `ADBReInit` not be called on the Macintosh SE until the problem is fixed. (There is no need to call `ADBReInit`.) This problem will not occur with the Macintosh II ROM version of the keyboard driver.

#144: Macintosh II Color Monitor Connections

Revised by: Wayne Correia February 1990
Written by: Mark Baumwell July 1987

This Technical Note describes how to connect the Macintosh II Video Card to third-party monitors. **Changes since March 1988:** Updated for newer Macintosh II Video Cards, including the Macintosh IIci On-Board Video (OBV).

Following are the pinout descriptions of the Macintosh II Video Cards and the Macintosh IIci On-Board Video (OBV):

Macintosh II Video Card Pin	**Signal Name**
1,6,11,13,14	Ground
2	Red
3	C-Sync (composite sync)
4	Monitor ID, Bit 1 (ground this pin to signal that a 640 x 480 monitor is connected)
5,12	Green (with sync)
7	Monitor ID, Bit 2
9	Blue
10	Monitor ID, Bit 3
8,15	Not connected

Note: The Macintosh II High-Resolution Display Video Card is the newer replacement for the original four- and eight-bit Macintosh II Video Card (M0211 and M5640). This new card is sold in four- and eight-bit configurations (M0322 and M0324, respectively).

Note: The newer Macintosh II Video Cards and Macintosh IIci OBV require that pin 4 (Monitor ID, Bit 1) be connected to Ground to signal the connection of a 640 x 480 monitor. Do not connect pins 7 or 10 as they are unused on original Macintosh II Video Cards and there are built-in pullup resistors on the newer Macintosh II Video Card and Macintosh IIci to terminate these pins when not in use.

Sony Multiscan (CPD-1302)

To connect a Macintosh II to a Sony Multiscan monitor, you need to make an adapter cable from the video card to the monitor (which has a 9-pin D-type connector). Following is the pinout description for the adapter cable (using the automatic sync-on-green configuration):

Macintosh II Video Card Pin	Sony Pin	Signal Name
1	1	Ground
2	3	Red
4	1	Ground
5	4	Green (sync)
9	5	Blue

NEC MultiSync (JC-140IP3A)

To connect a Macintosh II to a NEC MultiSync monitor, you need to make an adapter cable from the video card to the monitor (which has a 9-pin D-type connector). Following is the pinout description for the adapter cable (using the automatic sync-on-green configuration):

Macintosh II Video Card Pin	NEC Pin	Signal Name
1	6,7,8,9	Ground
2	1	Red
4	6,7,8,9	Ground
5	2	Green (sync)
9	3	Blue

The monitor must be set to Analog mode and Manual mode. This adaptor cable also works with an equivalent monitor such as the Taxan Super Vision 770.

Macintosh Technical Notes

#145: Debugger FKEY

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

It is often more convenient to enter the debugger using the keyboard rather than having to reach around to press the interrupt switch. This technical note shows how to make a simple FKEY that will trap to the debugger.

This technical note shows how to make a simple FKEY that will trap to the debugger. It is written in MPW Assembler. The assembler source is given below.

MPW Assembler source file listing:

```
; File: DebugKey.a
;
; An FKEY to invoke the debugger via command-shift-8
;
; To build this:
;      Asm DebugKey.a
;      Link DebugKey.a.o -o "{SystemFolder}System" -rt FKEY=6

DebugKey        MAIN

                BRA.S           CallDB   ;Invoke the debugger

                ;standard header

                DC.W            $0000    ;flags
                DC.L            'FKEY'   ;'FKEY' is 464B4559 hex
                DC.W            $0008    ;FKEY Number
                DC.W            $0000    ;Version number

CallDB          DC.W            $A9FF    ;Debugger trap
                RTS

                END
```

Macintosh Technical Notes

#146: Notes on MPW Pascal's -mc68881 Option

See also: *Apple Numerics Manual*
MPW Pascal Reference

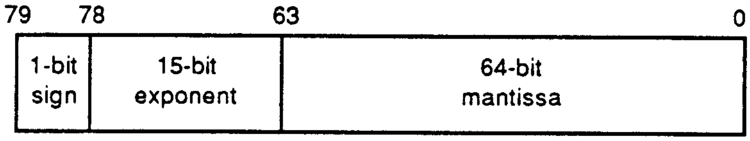
Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

For improved performance, the MPW Pascal compiler (version 2.0 and newer) represents `Extended` values in 96 bits (instead of 80, as with software SANE) when the `-mc68881` option is used. This can cause problems when using non-SANE system calls that expect 80-bit `Extended` values.

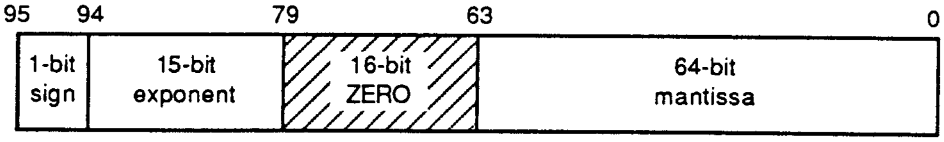
The Pascal Compiler and `Extended` values

The MPW 2.0 Pascal compiler provides a command-line option, `-mc68881`, to generate inline code to use the Motorola 68881 Floating-Point Coprocessor (included with Macintosh II). This allows you to sacrifice compatibility with other Macintosh systems (those not equipped with the 68020/68881 combination) in exchange for much-increased numeric performance.

When this option is used, the compiler stores all `Extended` values in the 96-bit format used by the 68881, instead of the 80-bit software SANE format:



80-bit Software SANE Format



96-bit 68881 Format

This affects all procedures that accept floating-point values as arguments, since all floating-point arguments are converted to `Extended` before being passed, no matter how they're declared (that is, `Real`, `Single`, `Double`, or `Comp`).
You must link with a special SANELib library file ("SANE881Lib.o") when compiling with

this option; the interface source file “SANE.p” contains conditional-compilation statements to make sure that the correct library's interface is compiled. In this situation, SANE procedures are used for certain transcendental functions only (see note below), and these functions (in “SANE881Lib.o”) expect their `Extended` parameters in 96-bit format.

However, numeric routines that are not compiled by Pascal (such as any assembly-language routines that you've written) have no way of finding out that their parameters will be in 96-bit format. If you don't wish to (or can't) rewrite these routines for 96-bit values, you can use the SANELib routines `X96ToX80` and `X80ToX96` to convert back and forth; it might be simplest to define a new interface routine to make the conversions happen automatically:

```
{An assembly-language function that accepts }
{an 80-bit Extended parameter and returns an}
{80-bit result (We've changed the types to  }
{reflect that these are not 96-bit values). }

FUNCTION FPFunc(x: Extended80): Extended80; EXTERNAL;

{Given that we're compiling in -mc68881 mode,}
{call our assembly-language function. Note   }
{that the compiler thinks that Extended      }
{values are 96 bits long, but FPFunc wants an}
{80-bit parameter and produces an 80-bit     }
{result; we convert.                         }

FUNCTION FPFunc96(x: Extended): Extended; {x is a 96-bit extended!}
BEGIN
  {convert our argument, call the function, then convert the result}
  MyFPFunc := X80ToX96(FPFunc(X96ToX80(x))); {call the real FPFunc}
END;
```

It's best to avoid compiling some parts of an application with the `-mc68881` option on, and other parts with it off; very strange bugs can occur if you try this. Note that 80-bit code and 96-bit code cannot reference the same `Extended` variables. There is no way to tell whether a given stored value is in 80-bit format or 96-bit format.

SANE on Macintosh II

The version of SANE provided in the Macintosh II ROM recognizes the presence of the 68881 and uses it for most calculations automatically. SANE still expects (and produces) 80-bit-format `Extended` values; it converts to and from 96-bit format internally when using the 68881.

A Note about 68881 Accuracy and Numeric Compatibility

SANE is more accurate than the 68881 when calculating results of certain functions (Sin, Cos, Arctan, Exp, Ln, Tan, Exp1, Exp2, Ln1, and Log2). To maintain this accuracy, SANE doesn't use 68881 instructions to directly perform these functions. Thus, the results you'll get from SANE calculations will still be identical on all Macintosh systems.

To preserve this numeric compatibility with other SANE implementations, MPW Pascal normally doesn't generate inline 68881 calls to the above functions, even when the `-mc68881` option is used; instead, it generates SANE calls to accomplish them. If you're willing to sacrifice numeric compatibility to gain extra speed, you can override this compiler feature with the compile-time variable `Elems881`; include the option "`-d Elems881=TRUE`" on the compiler command line to cause the compiler to generate direct 68881 instructions.

For certain other transcendental functions provided by the 68881 that aren't provided by SANE, MPW Pascal will generate direct 68881 calls if the `-mc68881` option is on, independent of the setting of the `Elems881` variable. These operations are Arctanh, Cosh, Sinh, Tanh, Log10, Exp10, Arccos, Arcsin, and Sincos.

#147: Finder Notes: “Get Info” Default & Icon Masks

See also: Technical Note #48—Bundles

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

The Finder has undergone a couple of changes you should keep in mind when creating the “bundle” information for your application.

Creator String will be the default “Get Info” comment text

The “creator” (or “signature”) string (contained in a resource whose type is your application’s four-character creator type, and whose ID is 0) will be used as the default for the comment text displayed by the Finder’s “Get Info” command. Thus, you should set up this string (when you build your application) to contain the name of your program and a version number and date.

Icon Masks should match their icons

Your application’s `BNDL` (“bundle”) resource ties the file types that it uses for its documents with the icons to be displayed for those documents. For each icon, a “mask” icon is also provided; this mask is used to punch a hole in the gray desktop before drawing the icon.

Some applications use a cleverly-modified mask to provide an “action icon” that looks different when it’s selected. This causes problems; it is important that the mask be what it’s supposed to be (a solid black copy of the icon).

#148: Suppliers for Macintosh II Board Developers

See also: *Designing Cards and Drivers*
for the Macintosh II and Macintosh SE

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

This note lists suppliers of parts that may be helpful for Macintosh II board developers. If your company supplies these parts, but is not listed here, please send a message to us (at the address on Technical Note #0) and we'll include you in the next revision of this technical note.

This is a list of companies that supply the Macintosh II expansion port cover (p/n 805-5064-05) (Foldout 2 in *Designing Cards and Drivers or the Macintosh II and Macintosh SE*). It is not intended to be an endorsement or an indication of quality; it is just our list of known suppliers.

Galgon Industries, Inc.
37399 Centralmont Place
Fremont, CA 94536
Attn: Ron Haddox—General Sales
(415) 792-8211

Vector Electronics
12460 Gladstone Ave
Sylmar, CA 91342
(818) 365-9661
FAX# 818-356-5718
Attn: Norm Brunell

North American Tool and Die
999 Beecher Street
San Leandro, CA 94577
(415) 632-9263
Attn: Glenn Erikson

In addition to supplying the expansion port cover, Vector Electronics supplies Macintosh II NuBus extender boards and prototyping boards.

Macintosh Technical Notes

#149: Document Names and the Printing Manager

See also: The Printing Manager
Technical Note #122—Device-Independent Printing

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

Our compatibility testing for LaserShare (Apple's LaserWriter spooler) has turned up a number of applications that do not provide the Printing Manager with a document name; although this feature is not required, it is nice for users that share printers.

Some printers (usually those that are shared between many users, like the LaserWriter) can provide the names of the users who are printing and the documents that are being printed to others interested in using the printer.

If the chosen printer uses a document name, the Printing Manager gets the name from the frontmost window's title. If there is no front window, or if the window's title is empty, the Printing Manager defaults to "unknown."

This method was chosen because it works most transparently to applications; however, it won't work if your application doesn't display windows when printing (for instance, many applications that use windows for their documents do not open their documents when printing in response to a Finder "Print" command).

As a general solution to this problem, you can put up a window containing a message like "Press ⌘–. to cancel printing", and give it the document's title. If the window is one that doesn't have a title bar (like `dBoxProc`), this title will not be displayed. MacApp takes this approach. If for some reason you don't want to put up a visible window, you can create a tiny window and hide it behind the menu bar: for instance, global coordinates of (1,1,2,2). Make sure you use a `plainDBox`, so that no title will be drawn (otherwise, in the unlikely case that a user is using a Macintosh II with two stacked screens, main screen on the bottom, the title might be visible on the upper screen).

Since the Printing Manager checks the name at `PrValidate` time, call `PrValidate` after `PrCloseDoc` and before the next `PrOpenDoc`, if you want unique names.

A number of applications set the document name in the print record directly. You should not do this because a) not all printers support this field, and b) none are guaranteed to support it in the future. (Apple does not guarantee that internal fields of the Printing Manager's data structures will remain the same; the Printing Manager is targeted for substantial internal change!)

#150: Macintosh SE Disk Driver Bug

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

A bug in the Macintosh SE ROMs causes the top drive to be slower than the bottom one in two-drive machines. This bug is fixed in System 4.2 and newer.

Macintosh Technical Notes

#151: System Error 33, “zcbFree has gone negative”

See also: The Memory Manager

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

System 3.2 introduced a new system error, ID=33, generated by the Memory Manager when it notices that a heap had been corrupted in a certain way. This error is listed in the file “SysErr.a” as “`negZcbFreeErr`”.

The Memory Manager will trigger an “ID=33” system error when, during some operation which scans the objects in the heap, it sees that its running count of free bytes (`zcbFree`, an internal value) has become negative (an impossible feat in normal operation). This is nearly always caused by writing zeros past the end of one of the Memory Manager’s heap blocks (overwriting and corrupting the next block’s header, making it appear to be a free block).

If you get this error, use a debugger (like Macsbug or TMON) when you attempt to reproduce the error, to check the consistency of the heap up to the point where the error occurs. You may need to do this repeatedly until you isolate the operation that is corrupting the heap.

Note that although the heap may become corrupted during a system call, this doesn’t mean you’ve found a bug in the ROM; your code could be passing incorrect or invalid parameters to this or a previous system call, or could have corrupted a data structure used by a system call. More debugging is usually in order in this case; tools like Discipline (included in TMON; also available from users’ groups and electronic services) can help detect invalid parameters in system calls. Also, there is a Macsbug command, AH, that can check the consistency of the heap on every system call. See the documentation that came with your debugger to see what special features it offers.

A note about “SysErr.a”

Technical Support is often asked for an up-to-date list of error codes. In general, this is provided in “SysErr.a”, the file of error numbers shipped with the most current version of MPW. Admittedly, the documentation value of “SysErr.a” is sometimes low (as in the case of `negZCBFreeErr`), but it may give you a clue as to what the error might mean.

Macintosh
Technical Notes

Developer Technical Support

#67: How to Bless a Folder to Be the System Folder

Rewritten by: Colleen K. Delgadillo May 1992
Updated by: Jim Friedlander March 1988
Written by: Jim Friedlander January 1986

This Technical Note describes how to determine which folder on an HFS volume is the blessed folder, that is, the folder that contains both the System file and the Finder.

Changes since January 1986: The information about how to find the "Blessed Folder" has been deleted from this technical note. The FindFolder function can now be used to find the "Blessed Folder" and is documented in Inside Macintosh Volume VI, pages 9-42 to 9-44. This note now includes information about how to bless a folder to the new system folder.

Note: The following information may be affected by future changes to system software. If you choose to use this information, you must do so at your own risk.

The way to bless a folder is by taking the longword which is the directory ID of the blessed folder and putting it into the Master Directory Block (MDB). This can be accomplished by using the HFS call `PBSetVInfo`. You should not attempt to change this block directly. First call `PBHGetVInfo` and set `ioVFnderInfo[1]` to the directory ID of the the new folder to be blessed. Then call `PBSetVInfo` to save this information. Once you have done this, you will find that the Finder takes a little while to realize that you have blessed the folder. Therefore, the icon will take a little while to change. Exactly how long you will have to wait to see the new icon is unknown.

Forcing the icon to change sooner is not a difficult task. The best way for you to see the icon change more quickly is to change the modification date of the directory into which you are copying the new System Folder. Doing this will cause the Finder to reexamine the window and its contents. When the Finder notices that the volume's modification date has changed, it begins scanning for changes in all the open folders. This scanning process takes place about once every 10 seconds. You can change the last modification date for that volume and the System Folder's directory ID for that volume using `PBSetVInfo`. Changing the file's `FndrInfo` or renaming the file does not change the modification date. When you call `PBSetVInfo` you will need to put the System Folder's directory ID in the longword at `ioVfndrInfo`. This longword will be the first four bytes of this directory ID. (As usual, whenever you make a change to a field of a structure you need to first do a `PBGetCatInfo`, change what you are going to change, and then do `PBSetCatInfo`. This ensures that you change only the portion of the volume that you intended, in this case a longword, and not the whole structure.)

Further Reference:

- Master Directory Block: Inside Macintosh Volume IV on page 166.

Macintosh
Technical Notes

Developer Technical Support

#68: Searching Volumes—Solutions and Problems

Revised by: Jim Luther — January 1992
Written by: Jim Friedlander and Rick Blair — December 1985 – October 1988

This Technical Note discusses the PBCatSearch function and tells why it should be used. It also provides simple algorithms for searching both MFS and HFS volumes and discusses the problems with indexed search routines.

Changes since October 1988: Includes information on PBCatSearch and notes the problems with indexed search routines. Source code examples have been added and revised. Thanks to John Norstad at Northwestern University for pointing out some of the shortcomings of the indexed search routines. Thanks to the System 7 engineering team for adding PBCatSearch.

It may be necessary to search the volume hierarchy for files or directories with specific characteristics. Generally speaking, your application should avoid searching entire volumes because searching can be a very time-consuming process on a large volume. Your application should rely instead on files being in specific directories (the same directory as the application, or in one of the system-related folders that can be found with FindFolder) or on having the user find files with Standard File.

Searching MFS Volumes

Under MFS, indexed calls to PBGetFInfo return information about all files on a given volume. Under HFS, the same technique returns information only about files in the current directory. Here's a short code snippet showing how to use PBGetFInfo to list all files on an MFS volume:

```
PROCEDURE EnumMFS (theVRefNum: Integer);
{ search the MFS volume specified by theVRefNum }
   VAR
      pb: ParamBlockRec;
      itemName: Str255;
      index: Integer;
      err: OSErr;
BEGIN
   WITH pb DO
      BEGIN
         ioNamePtr := @itemName;
         ioVRefNum := theVRefNum;
         ioFVersNum := 0;
      END;
   index := 1;
   REPEAT
      pb.ioFDirIndex := index;
      err := PBGetFInfoSync(@pb);
      IF err = noErr THEN
         BEGIN
            { do something useful with the file information in pb }
         END;
      index := index + 1;
   UNTIL err <> noErr;
END;
```

As noted in Macintosh Technical Note #66, a directory signature of $D2D7 means a volume is an MFS volume, while a directory signature of $4244 means the volume is an HFS volume.

Searching HFS Volumes

Fast, Reliable Searches Using `PBCatSearch`

The fastest and most reliable way to search an HFS volume's catalog is with the File Manager's `PBCatSearch` function. `PBCatSearch` returns a list of `FSSpec` records to files or directories that match the search criteria specified by your application. However, `PBCatSearch` is not available on all volumes or under all versions of the File Manager. Volumes that support `PBCatSearch` can be identified using the `PBHGetVolParms` function. (See the following code.) Versions of the File Manager that support `PBCatSearch` can be identified with the `gestaltFSAttr` Gestalt selector and `gestaltFullExtFSDispatching` bit as shown in the following code:

```
FUNCTION HasCatSearch (vRefNum: Integer): Boolean;
{ See if volume specified by vRefNum supports PBCatSearch }
   VAR
      pb: HParamBlockRec;
      infoBuffer: GetVolParmsInfoBuffer;
      attrib: LongInt;

BEGIN
   HasCatSearch := FALSE; { default to no PBCatSearch support }
   IF GestaltAvailable THEN { See Inside Macintosh Volume VI, Chapter 3 }
      IF Gestalt(gestaltFSAttr, attrib) = noErr THEN
         IF BTst(attrib, gestaltFullExtFSDispatching) THEN
            BEGIN { this version of the File Manager can call PBCatSearch }
               WITH pb DO
                  BEGIN
                     ioNamePtr := NIL;
                     ioVRefNum := vRefNum;
                     ioBuffer := @infoBuffer;
                     ioReqCount := sizeof(infoBuffer);
                  END;
               IF PBHGetVolParmsSync(@pb) = noErr THEN
                  IF BTST(infoBuffer.vMAttrib, bHasCatSearch) THEN
                     HasCatSearch := TRUE; { volume supports PBCatSearch }
            END;
END;
```

Note: File servers that support the AppleTalk Filing Protocol (AFP) version 2.1 support `PBCatSearch`. That includes volumes and directories shared by System 7 File Sharing and by the AppleShare 3.0 file server. Although AFP version 2.1 supports `PBCatSearch`, the `fsSBNegate` bit is not supported in the `ioSearchBits` field. Using `PBCatSearch` to ask the file server to perform the search is usually faster than using the recursive indexed search described in the next section.

`PBCatSearch` should be used if it is available because it is usually *much* faster than a recursive search. For example, the search time for finding all files and directories on a recent Developer CD was around 18 seconds with PBCatSearch. It took 6 minutes and 36 seconds with a recursive indexed search. How long do you want the users of your application to wait?

PBCatSearch can be used to collect a list of FSSpec records to all items on a volume by setting ioSearchBits in the parameter block to 0.

Recursive Indexed Searches Using PBGetCatInfo

When PBCatSearch is not available, an application must resort to a recursive indexed search. There are a couple of potential problems with a recursive indexed search; a recursive indexed search can use up a lot of stack space and the volume directory structure can change in the multi-user/multiprocess Macintosh environment. The example code in this note addresses the stack space problem, but for reasons explained later, does not address problems caused by multiple users or processes changing the volume directory structure during a recursive search.

The default stack space on the Macintosh can be as small as 8K; therefore, the recursive indexed search example shown in this Note encloses the actual recursive routine in a shell that can hold most of the variables needed, which dramatically reduces the size of the stack frame. This example uses only 26 bytes of stack space each time the routine recurses. That is, it could search 100 levels deep (pretty unlikely) and use only 2600 bytes of stack space.

Please notice that when the routine comes back from recursing, it has to clear the nonlocal variable err to noErr, since the reason the routine came back from recursing is that PBGetCatInfo returned an error:

```
EnumerateCatalog(myCPB.ioDrDirID);
err := noErr; {clear error return on way back}
```

Please notice also that you must set myCPB.ioDrDirId each time you call PBGetCatInfo, because if PBGetCatInfo gets information about a file, it returns ioFlNum (the file number) in the same location that ioDrDirID previously occupied.

Be sure to check bit 4, the fifth least significant bit, when you check the file attributes bit to see if you've got a file or a folder. The following routine uses MPW Pascal's BTST function to check that bit. If you use the Toolbox bit manipulation routines (e.g., BitTst), remember to order the bits in reverse order from standard 68000 notation.

Here is the routine in MPW Pascal:

```
PROCEDURE EnumerShell (vRefNumToSearch: Integer; { the vRefNum to search}
                        dirIDToSearch: LongInt);  { the dirID to search }
   VAR
      itemName: Str63;
      myCPB: CInfoPBRec;
      err: OSErr;

   {-----}

   PROCEDURE EnumerateCatalog (dirIDToSearch: LongInt);
      CONST
         ioDirFlgBit = 4;
      VAR
         index: Integer;
   BEGIN { EnumerateCatalog }
      index := 1;
      REPEAT
         WITH myCBP DO
```

```
            BEGIN
               ioFDirIndex := index;
               ioDrDirID := dirIDToSearch; { we need to do this every }
                                           { time through }
               filler2 := 0; { Clear the ioACUser byte if search is  }
                             { interested in it. Nonserver volumes }
                             { won't clear it for you and the value  }
                             { returned is meaningless. }
            END;
         err := PBGetCatInfo(@myCPB, FALSE);
         IF err = noErr THEN
            IF BTST(myCPB.ioFlAttrib, ioDirFlgBit) THEN
               BEGIN { we have a directory }

                  { do something useful with the directory information }
                  { in myCPB }

                  EnumerateCatalog(myCPB.ioDrDirID);
                  err := noErr; {clear error return on way back}
               END
            ELSE
               BEGIN { we have a file }

                  { do something useful with the file information }
                  { in myCPB }

               END;
         index := index + 1;
      UNTIL (err <> noErr);
   END;  { EnumerateCatalog }

   {-----}

BEGIN { EnumerShell }
   WITH myCPB DO
      BEGIN
         ioNamePtr := @itemName;
         ioVRefNum := vRefNumToSearch;
      END;
   EnumerateCatalog(dirIDToSearch);
END; { EnumerShell }
```

In MPW C:

```
/* the following variables are globals */
HFileInfo      gMyCPB;           /* for the PBGetCatInfo call */
Str63          gItemName;        /* place to hold file name */
OSErr          gErr;             /* the usual */

/*--------------------------------------------------------------------*/

void EnumerateCatalog (long int dirIDToSearch)
{  /* EnumerateCatalog */

   short int          index=1;
   do
   {
      gMyCPB.ioFDirIndex= index;
      gMyCPB.ioDirID= dirIDToSearch; /* we need to do this every time    */
                                     /* through, since GetCatInfo        */
                                     /* returns ioFlNum in this field */
      gMyCPB.filler2= 0; /* Clear the ioACUser byte if search is        */
                         /* interested in it. Nonserver volumes won't  */
                         /* clear it for you and the value returned is */
                         /* meaningless. */
```

```
         gErr= PBGetCatInfo(&gMyCPB,false);
         if (gErr == noErr)
         {
            if ((gMyCPB.ioFlAttrib & ioDirMask) != 0)
            {  /* we have a directory */

               /* do something useful with the directory information */
               /* in gMyCPB */

               EnumerateCatalog(gMyCPB.ioDirID); /* recurse */
               gErr = noErr; /* clear error return on way back */
            }
            else
            {  /* we have a file */

               /* do something useful with the file information */
               /* in gMyCPB */

            }
         }
         ++index;
      } while (gErr == noErr);
   }  /* EnumerateCatalog */

   /*-----------------------------------------------------------------------*/

   EnumerShell(short int vRefNumToSearch, long int dirIDToSearch)

   {  /* EnumerShell */
      gMyCPB.ioNamePtr = gItemName;
      gMyCPB.ioVRefNum = vRefNumToSearch;
      EnumerateCatalog(dirIDToSearch);
   }  /* EnumerShell */
```

Please make sure that you are running under HFS before you use this routine (see Technical Note #66). You can search the entire volume by specifying a starting directory ID of `fsRtDirID`, the root directory constant. You can do partial searches of a volume by specifying a starting directory ID other than `fsRtDirID`.

Searching in a Multi-user/Multiprocess Environment

Volumes can be shared by multiple users accessing a file server or multiple processes running on a single Macintosh. Each user or process with access to such a shared volume may be able to make changes to the volume's catalog at any time. Changes in a volume's catalog in the middle of a search can cause two problems:

- Files and directories renamed or moved by another user or process can be entirely missed or found multiple times by a search routine.
- A search routine can easily lose track of its position within the hierarchical directory structure when files or directories are created, deleted, or renamed by another user or process.

A volume searched with a single call to `PBCatSearch` ensures that all parts of the volume are searched without another user or process changing the volume's catalog. However, a single call to `PBCatSearch` may not be possible or practical because of the number of matches you expect, or because you may want to set a time limit on the search so that the user can cancel a long search. `PBCatSearch` returns a `catChangedErr` (–1304) and no matches when the catalog of a volume is changed by another user or process in a way that might affect the current search. The search can be continued with the `CatPositionRec` returned with the `catChangedErr` error, but at the risk of missing catalog entries or finding duplicate catalog entries.

Things aren't so nice for search routines based on indexed File Manager calls. The File Manager won't notify you when a volume's catalog has changed. In fact, there are several ways the catalog can change that are very difficult to detect and correct for. Since methods that attempt to resynchronize an indexed search and find all catalog entries that might be missed or found multiple times when the catalog changes do not work for all cases, those methods are not discussed in this Technical Note. The following paragraphs describe why some changes are very difficult to detect.

There are three changes you can make to the contents of a directory that change the list of files and directories returned by an indexed search: creating, deleting, and renaming. Directories of an HFS volume are always sorted alphabetically, so when a file or subdirectory is deleted from a directory, any directory entries after it bubbles up to fill the vacated entry position; when a file or subdirectory is created, it is inserted into the list and all entries after it bubbles down one position. When a file or subdirectory is renamed, it is removed from its current position and moved into its alphabetically correct position. The first two changes, creating and deleting, can be detected only at the parent directory level. That's because a creation or deletion changes only the modification date of the parent directory but not the modification date of any of the parent directory's ancestors. Renaming a file or subdirectory does not change the modification date of the file or subdirectory renamed or the modification date its parent directory, but it does change the order of files and subdirectories found by an indexed search.

With this in mind, here are a couple of examples that are very difficult to detect.

The first example shows a file, Dashboard, moved (by another user or process) with `PBCatMove` from the CDevs subdirectory to the Control Panels subdirectory. (See figures 1 and 2.) At the time of the move, the search routine has just finished recursively looking through the Development directory and is ready to recursively search the Games directory. After the move, two directories, CDevs and Control Panels, have new modification dates but no change is seen at the root directory of My Disk. There is nothing to immediately tell the search routine something has changed (except for the volume modification date which may or may not mean the directory structure has changed), so the search will see Dashboard twice. If the move were in the opposite direction, from Control Panels to CDevs, Dashboard would be missed by the search routine.

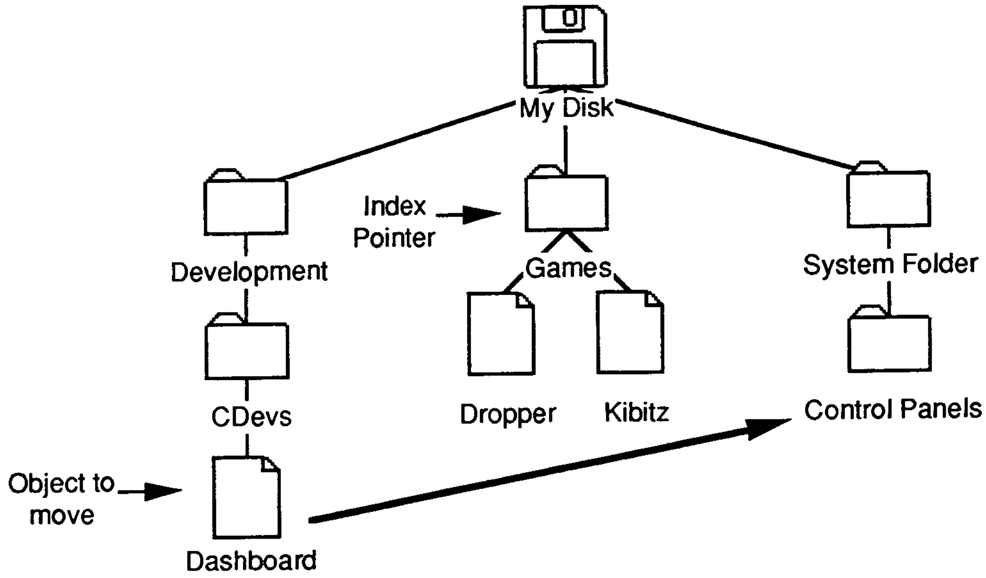


Figure 1 Before Dashboard Is Moved With `PBCatMove`

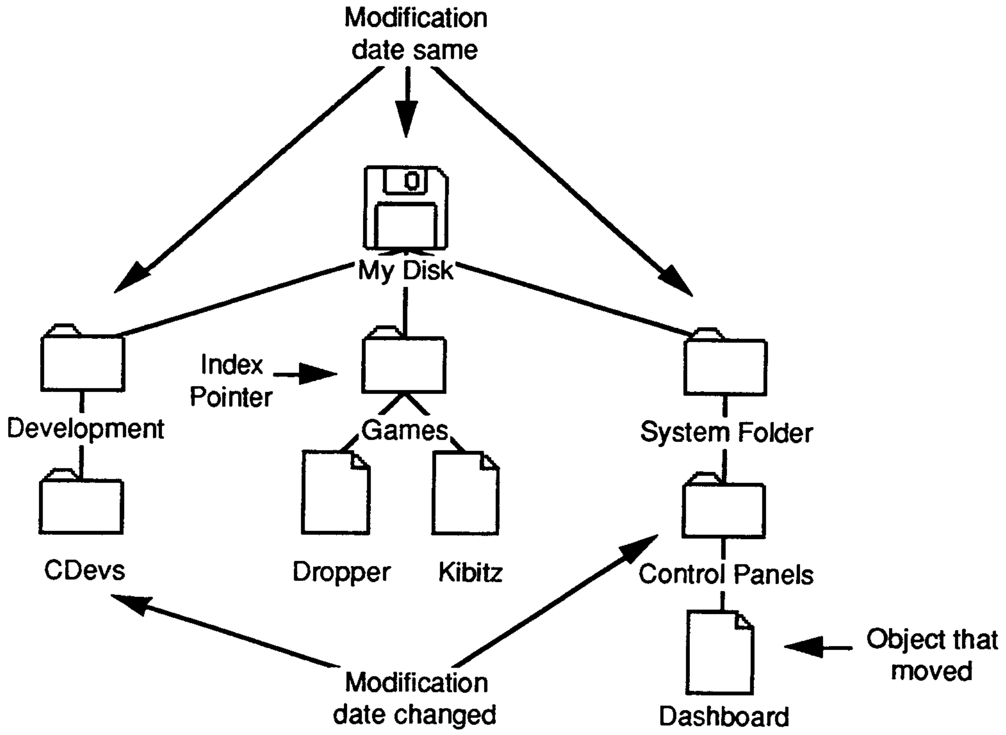


Figure 2 After Dashboard Is Moved With `PBCatMove`

The second example (see Figures 3 and 4) shows a directory, Toys, renamed (by another user or process) with `PBHRename` to Games. At the time of the move, the search routine has seen the files Aardvark and Letter and is looking at the third object in the directory, the file Résumé. After the move, the index pointer is still pointing at the third object but now the third object is the file Letter, a file that has already been seen by the search. This change cannot be detected by looking at the parent directory's modification date because `PBHRename` does not change any modification dates. However, this change can be detected by checking to see if the index pointer still points to the same file or directory. The search routine could re-index through the directory to find the Résumé file again and start searching from there, but what about the directory that was renamed? The search routine either must miss it (and its contents) or it must repeat the search of the entire directory to ensure nothing is missed.

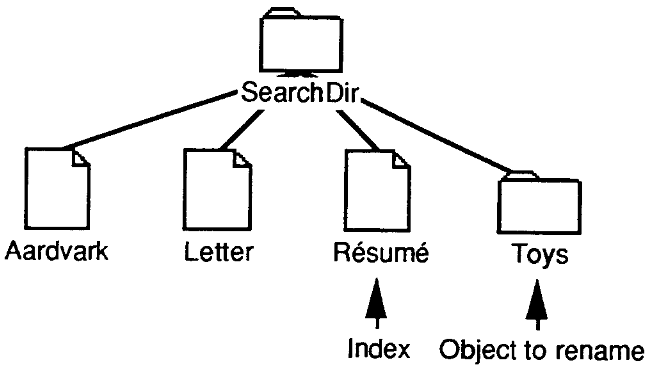


Figure 3 Before Toys Is Renamed With `PBHRename`

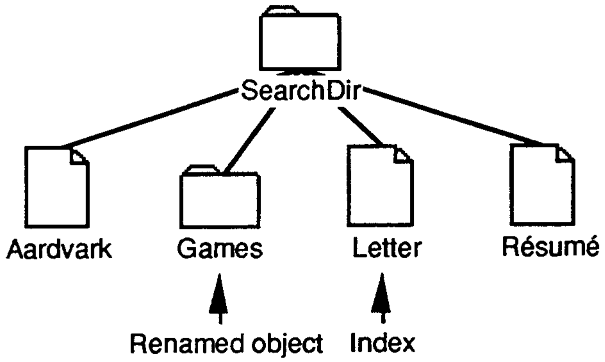


Figure 4 After Toys Is Renamed to Games With `PBHRename`

As these examples show, a change during a search of a hierarchical directory structure with indexed File Manager calls involves the risk of missing catalog entries or finding duplicate catalog entries. If your application depends on seeing all items on a volume at least once and only once, you should make the users of your application aware of the problems associated with indexed searches and suggest to them ways to make sure the volume's catalog is not changed during the indexed search. Here's a good suggestion you could make to the user: do not use other programs during the search. Other programs may create, delete, or rename files during the search.

Conclusion

You should always use `PBCatSearch` to search a volume if it is available. If `PBCatSearch` isn't available and you must use an indexed search, be aware that it is difficult to ensure that you do not miss some catalog entries or see some catalog entries multiple times during your search.

Further Reference:

- *Inside Macintosh*, Volume IV, The File Manager
- *Inside Macintosh*, Volume V, File Manager Extensions in a Shared Environment
- *Inside Macintosh*, Volume VI, The Finder Interface
- *Inside Macintosh*, Volume VI, The File Manager
- Technical Note #66, Determining Which File System Is Active
- Technical Note #305, PBShare, PBUnshare, and PBGetUGEntry

#69: Setting ioFDirIndex in PBGetCatInfo Calls

See also: The File Manager
Technical Note #24—Available Volumes and Files
Technical Note #67—Finding the Blessed Folder

Written by: Jim Friedlander

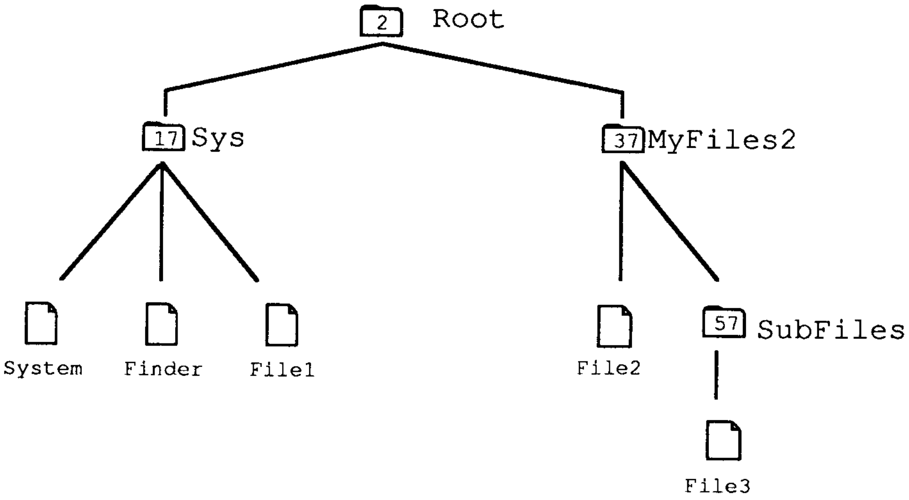
February 15, 1986
Updated: March 1, 1988

This technical note describes how to set ioFDirIndex for PBGetCatInfo.

The File Manager chapter of *Inside Macintosh* volume IV is not very specific in describing how to use ioFDirIndex when calling PBGetCatInfo. It correctly says that ioFDirIndex should be positive if you are making indexed calls to PBGetCatInfo (analogous to making indexed calls to PBGetVInfo as described in Technical Note #24). However, the statement "If ioFDirIndex is negative or 0, the File Manager returns information about the file having the name in ioNamePtr..." is not specific enough.

If ioFDirIndex is 0, you will get information about files or directories, depending on what is specified by ioNamePtr^.

If ioFDirIndex is –1, you will get information about directories only. The name in ioNamePtr^ is ignored. For example, given the following tree structure (with sample DirIDs for the directories):



Calling PBGetCatInfo

We will now make calls to PBGetCatInfo of the form:

```
    err:= PBGetCatInfo(@myCInfoPBRec,FALSE);
```

Note: We will assume that we just have a WDRefnum and a file name—the information that SFGetFile returns.

Setting up the parameter block

We will use the following fields in the parameter block. Before the call, ioCompletion will always be set to NIL, ioNamePtr will always point at a str255, ioVRefNum will always contain a WDRefNum that references the directory 'SubFiles', and offset 48 (dirID/flNum) will always contain a zero:

Offset in parameter block	Variable name(s)
12	ioCompletion
18	ioNamePtr
22	ioVRefNum
28	ioFDirIndex
48	ioDirID/ioFLNum/ioDrDirID
100	ioDrParID/ioFlParID

Sample calls to PBGetCatInfo

The first example will call PBGetCatInfo for the file 'File3'—we will get information about the file (ioFDirIndex = 0):

Before the call		**After the call**	
ioNamePtr^:	'File3'	ioNamePtr^:	'File3'
ioFDirIndex:	0	Offset 48(ioFLNum):	a file number
		Offset 100(parID):	57

Now we will get information about the directory that is specified by the iovRefNum (ioFDirIndex = –1). Notice that ioNamePtr^ is ignored:

Before the call		**After the call**	
ioNamePtr^:	ignored	ioNamePtr^:	'SubFiles'
ioFDirIndex:	–1	Offset 48(dirID):	57
		Offset 100(parID):	37

Notice that, since ioNamePtr^ is ignored, Offset 48 contains the dirID of the directory specified by the iovRefNum that we passed in and that Offset 100 contains the parent ID of that directory.
Notice that if we try to get information about the directory SubFiles by calling PBGetCatInfo with ioFDirIndex set to 0, we will get an error –43 (File not found error) back because there is neither a file nor a directory with the name 'SubFiles' in the directory that ioVRefNum refers to.

If you specify a **full** pathname in ioNamePtr^, then the call returns information about that path, whether it is a directory or a file. The ioVRefNum is ignored:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys'	ioNamePtr^:	'Root:Sys'
ioFDirIndex:	0	Offset 48 (dirID):	17
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	2

Or, if the full pathname specifies a file, the iovRefNum is overridden:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys:Finder'	ioNamePtr^:	'Root:Sys:Finder'
ioFDirIndex:	0	Offset 48 (flNum):	fileNumber
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	17

Or, given an ioVRefNum that refers to MyFiles2 and a partial pathname in ioNamePtr^, we'll get information about the directory 'SubFiles':

Before the call		**After the call**	
ioNamePtr^:	'SubFiles'	ioNamePtr^:	'SubFiles'
ioFDirIndex:	0	Offset 48 (dirID):	57
ioVRefNum:	refers to 'MyFiles2'	Offset 100 (parID):	37

PBGetCatInfo and The Poor Man's Search Path (PMSP)

If no ioDirID is specified (ioDirID is set to zero), calls to PBGetCatInfo will return information about a file in the specified directory, but, if no such file is found, will continue searching down the Poor Man's Search Path. **Note:** the PMSP is not used if ioFDirIndex is non-zero (either –1 or >0). The default PMSP includes the directory specified by ioVRefNum (or, if ioVRefNum is 0, the default directory) and the directory that contains the System File and the Finder—the blessed folder. So for example:

Before the call		**After the call**	
ioNamePtr^:	'System'	ioNamePtr^:	'System'
ioFDirIndex:	0	Offset 48 (ioFLNum):	a file number
		Offset 100 (parID):	17

You must be careful when using PBGetCatInfo in this way to make sure that the file you're getting information about is in the directory that you think it is, and not in a directory further down the Poor Man's Search Path. Of course, this does not present a problem if you are using the fName and the vRefNum that SFGetFile returns.

If you want to specifically look at a file in the blessed folder, please use the technique described in technical note #67 to get the `dirID` of the 'blessed folder' and then use that `dirID` as input in the `ioDirID` field of the parameter block (offset 48).

Summary (`DirID` = 0 in all the following):

If `ioFDirIndex` is set to 0:

1) Information will be returned about files.
2) Information will be returned about directories as follows:
   A) If a partial pathname is specified by `ioNamePtr^` then the volume and directory will be taken from `ioVRefNum`.
   B) If a full pathname is specified by `ioNamePtr^`. In this case, `ioVRefNum` is ignored.

If `ioFDirIndex` is set to −1:

1) Only information about directories will be returned.
2) The name pointed to by `ioNamePtr` is ignored.
3) If `DirID` and `ioVRefNum` are 0, you'll get information about the default directory.

Macintosh Technical Notes

#70: Forcing Disks to be Either 400K or 800K

See also: The Disk Driver
The Disk Initialization Package

Written by: Rick Blair — February 13, 1986
Updated: March 1, 1988

This document explains how to initialize a disk as either single- or double-sided. It only applies to 800K drives, of course.

You can call the disk driver to initialize a disk and determine programmatically whether it should be initialized as single- (MFS) or double- (HFS) sided. All you have to do is call the `.Sony` driver directly to do the formatting then the Disk Initialization Package to write the directory information.

Note: This is not the way you should normally format disks within an application. If the user puts in an unformatted disk, you should let her or him decide whether it becomes single- or double-sided via the Disk Initialization dialog. This automatically happens when you call DIBadMount or the user inserts a disk while in Standard File. The intent of this technical note is to provide a means for specific applications to produce, say, 400K disks. An example might be a production disk copying program.

From MPW Pascal:

```
VAR
  error:      OSErr;
  IPtr:       ^INTEGER;
  paramBlock: ParamBlockRec; {needs OSIntf}
...
WITH paramBlock DO BEGIN
   ioRefNum := -5;                          {.Sony driver}
   ioVRefNum := 1;                          {drive number}
   csCode := 6;                             {format control code}
   IPtr:=@csParam;                          {pretend it's an INTEGER}
   IPtr^:=1;                                {number of sides}
END;
error:=PBControl(@paramBlock, FALSE);       {do the call}
IF error=ControlErr THEN
{you are under MFS, which doesn't support control code 6, but it}
{would always get formatted single-sided anyway.}
{other errors are possible: ioErr, etc.}
END;
```

From MPW C:

```
OSErr             error;
CntrlParam        paramBlock;


paramBlock.ioCRefNum = -5;     /*.Sony driver*/
paramBlock.ioVRefNum = 1;      /*drive number*/
paramBlock.csCode = 6;         /*format control code*/
paramBlock.csParam[0]=1;       /*for single sided,2 for double-sided*/

error=PBControl(&paramBlock, false);/*do the call*/
if (error==controlErr) ;
/*you are under MFS, which doesn't support control code 6, but it*/
/*would always get formatted single-sided anyway.*/
/*other errors are possible: ioErr, etc.*/
```

You then call DIZero to write a standard (MFS or HFS) directory. It will produce MFS if you formatted it single-sided, and HFS if you formatted double-sided.

#71: Finding Drivers in the Unit Table

See also: The Device Manager

Written by: Rick Blair

February 4, 1986
Updated: March 1, 1988

This note will explain how code can be written to determine the reference number of a previously installed driver when only the name is known. **Changes since 2/86:** Since the driver can be purged and the DCE still be allocated, the code now tests for dCtlDriver being NIL as well.

You should already be familiar with The Device Manager chapter of *Inside Macintosh* before reading this technical note.

The Pascal code at the end of this note demonstrates how to obtain the reference number of a driver that has been installed in the Unit Table. The reference number may then be used in subsequent calls to the Device Manager such as `Open`, `Control` and `Prime`.

One thing to note is that the `dRAMBased` bit really only tells you whether `dCtlDriver` is a pointer or a handle, not necessarily whether the driver is in ROM or RAM. SCSI drivers, for instance, are in RAM but not relocatable; their DCE entries contain pointers to them.

From MPW Pascal:

```
PROCEDURE GetDrvrRefNum(driverName: Str255; VAR drvrRefNum: INTEGER);

   TYPE
      WordPtr       = ^INTEGER;

   CONST
      UTableBase    = $11C;      {low memory globals}
      UnitNtryCnt   = $1D2;

      dRAMBased     = 6;         {bit in dCtlFlags that indicates ROM/RAM}
      drvrName      = $12;       {length byte and name of driver [string]}

   VAR
      negCount     : INTEGER;
      DCEH         : DCtlHandle;
      drivePtr     : Ptr;
      s            : Str255;
```

```
BEGIN
   UprString(driverName, FALSE); {force same case for compare}

   negCount := - WordPtr(UnitNtryCnt)^; {get -(table size)}

   {Check to see that driver is installed, obtain refNum.}
   {Assumes that an Open was done previously -- probably by an INIT.}
   {Driver doesn't have to be open now, though.}

   drvrRefNum := - 12 + 1;  {we'll start with driver refnum = -12,
                             right after .ATP entry}

   {Look through unit table until we find the driver or reach the end.}

   REPEAT
      drvrRefNum := drvrRefNum - 1; {bump to next refnum}
      DCEH := GetDCtlEntry(drvrRefNum); {get handle to DCE}

      s := '';              {no driver, no name}

      IF DCEH <> NIL THEN
         WITH DCEH^^ DO BEGIN {this is safe -- no chance of heap moving
                               before dCtlFlags/dCtlDriver references}
            IF (dCtlDriver <> NIL) THEN BEGIN
             IF BTST(dCtlFlags, dRAMBased) THEN
                drivePtr := Handle(dCtlDriver)^ {zee deréference}
             ELSE
                drivePtr := Ptr(dCtlDriver);

             IF drivePtr <> NIL THEN BEGIN
                s := StringPtr(ORD4(drivePtr) + drvrName)^;
                UprString(s,FALSE); {force same case for compare}
             END;
            END;             {IF}
         END;               {WITH}
   UNTIL (s = driverName) OR (drvrRefNum = negCount);

   {Loop until we find it or we've just looked at the last slot.}

   IF s <> driverName THEN drvrRefNum := 0; {can't find driver}
END;
```

From MPW C:

```
short       GetDrvrRefNum(driverName)
char        *driverName[256];

{  /* GetDrvrRefNum */

      #define           UnitNtryCnt 0x1d2

      /*bit in dCtlFlags that indicates ROM/RAM*/
      #define           dRAMBased     6
      /*length byte and name of driver [string]*/
      #define           drvrName      0x12
      short                 negCount,dRef;
      DCtlHandle            DCEH;
      char                  *drivePtr,*s;

      negCount = -*(short *)(UnitNtryCnt); /*get -(table size)*/

      /*Check to see that driver is installed, obtain refNum.*/
      /*Assumes that an Open was done previously -- probably by an INIT.*/
      /*Driver doesn't have to be open now, though.*/

      dRef = -12 + 1;  /*we'll start with driver refnum == -12,
                              right after .ATP entry*/

      /*Look through unit table until we find the driver or reach the
      end.*/

      do
      {
            dRef -= 1; /*bump to next refnum*/
            DCEH = GetDCtlEntry(dRef); /*get handle to DCE*/

            s = "";

            if ((DCEH != nil) && ( (**DCEH).dCtlDriver != nil) )
            {
                  if (((**DCEH).dCtlFlags >> dRAMBased) & 1)
                                                /* test dRamBased bit */
                        drivePtr = *(Handle) (**DCEH).dCtlDriver;
                                                /*zee deréference*/
                  else
                        drivePtr = (**DCEH).dCtlDriver;

                  if (drivePtr != nil)
                        s = drivePtr + drvrName;
            }
      } while (EqualString(s,driverName,0,0) && (dRef != negCount));
      /*Loop until we find it or we've just looked at the last slot.*/

      if (EqualString(s,driverName,0,0))
            return dRef;
      else
            return 0; /*can't find driver*/
}/* GetDrvrRefNum */
```

That's all there is to locating a driver and picking up the reference number.

#72: Optimizing For The LaserWriter—Techniques

Revised by: Pete "Luke" Alexander October 1990
Written by: Ginger Jernigan February 1986

This Technical Note discusses techniques for optimizing code for printing on the LaserWriter.
Changes since March 1988: Updated the "Printable Paper Area" and "Memory Considerations" sections as well as the printer IDs, moved the error messages from the end of the Note to Technical Note #161, A Printing Loop That Cares..., and removed the "Spool-A-Page/Print-A-Page" section because Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, already thoroughly covers this topic.

Introduction

Although the Printing Manager was originally designed to allow application code to be printer independent, there are some things about the LaserWriter that, in some cases, have to be addressed in a printer dependent way. This Note describes what the LaserWriter can and cannot do, memory considerations, speed considerations, as well as other things you need to watch out for if you want to make your printing more efficient on the LaserWriter.

How To Determine The Currently Selected Printer

With the addition of new picture comments and the `PrGeneral` procedure, an application should never need to know the type of device to which it is connected. However, some developers feel their application should be able to take advantage of all of the features provided by a particular device, not just those provided by the Printing Manager, and in doing so, these developers produce device-dependent applications, which can produce unpredictable results third-party and new Apple printing devices. For this reason, Apple strongly recommends that you use only the features provided by the Printing Manager, and do not try to use unsupported device features.

Even though there is no supported method for determining a device's type, there is one method described in the original *Inside Macintosh* that still works for ImageWriter and LaserWriter printer drivers. This method is **not** supported, meaning that at some point in the future it will no longer work. If you use this method in your application, it is up to you to weigh the value of the feature against the compatibility risk. The following method works for all ImageWriter, ImageWriter II, and LaserWriter (original, Plus, IINT, IINTX) drivers. Since all new devices released from Apple and third-party developers have their own unique ID, it is up to you to decide what to do with an ID that your application does not recognize.

If you are using the high-level Printing Manager interface, first call `PrValidate` to make sure you have the correct print record. Look at the high byte of the `wdev` word in the `TPrStl` subrecord of the print record. Note that if you have your own driver and want to have your own number, please let DTS know, and DTS can register it.

Following is the current list of printer IDs:

Printer	wDev
ImageWriter I, ImageWriter II	1
LaserWriter, LaserWriter Plus, LaserWriter IINT, LaserWriter IINTX, and Personal LaserWriter NT	3
LaserWriter IISC, Personal LaserWriter SC	4
ImageWriter LQ	5

If you are using the low-level Printing Manager interface, there is no dependable way of getting the wDev information. You should **not** attempt to determine the device ID when using the low-level Printing Manager interface.

Using QuickDraw With the LaserWriter

When you print to the LaserWriter, all of the QuickDraw calls you make are translated (via QuickDraw bottlenecks) into PostScript®, which is in the LaserWriter ROM. Most of the operations available in QuickDraw are available in PostScript, with a few exceptions. The LaserWriter driver does not support the following:

- `XOR` and `NotXOR` transfer modes.
- The grafverb `invert`.
- `_SetOrigin` calls within `PrOpenPage` and `PrClosePage` calls. Use `_OffsetRect` instead. (This is fixed in version 3.0 and later of the driver.)
- Regions are ignored. You can simulate regions using polygons or bitmaps. Refer to Technical Note #41, Drawing Into An Off-Screen Bitmap, for how to create off-screen bitmaps.
- Clip regions should be limited to rectangles.
- There is a small difference in character widths between screen fonts and printer fonts. Only the end points of text strings are the same.

What You See Is Not Always What You Get

Unfortunately, what you see on the screen is not always what you get. If you are using standard graphic objects, like rectangles, circles, etc., the object is the same size on the LaserWriter as it is on the screen. There are, however, two types of objects where this is not the case: text and bitmaps.

The earlier noted difference between the widths of characters on the screen and the widths of characters on the printer is due to the difference in resolution. However, to maintain the integrity of line breaks, the driver changes the word and character spacing to maintain the end points of the lines as specified. What this all means is that you cannot count on the positions or the widths of printed characters being exactly the same as they are on the screen. This is why in the original MacDraw®, for example, if one carefully places text and a rectangle and prints it, the text sometimes extends beyond the bounds of the rectangle on the printed page. If an application does its own line layout (i.e., positions the words on the line itself), then it may want to disable the LaserWriter's line layout routines. To disable these routines, use the `LineLayoutOff` picture comment described in the LaserWriter Reference Manual and Technical Note #91, Optimizing for the LaserWriter—Picture Comments.

The sole exception to this rule is if an application is running on 128K ROMs or later. The 128K ROM Font Manager supports the specification of fractional pixel widths for screen fonts, increasing the screen to printer accuracy. This fractional width feature is disabled by default. To enable it, an application can use `_SetFractEnable`, after calling `_InitFonts`.

Applications can use picture comments to left-, right-, or center-justify text. Only the left, right, or center end points are accurate. If the text is fully justified, both end points are accurate. Technical Note #91, Optimizing for the LaserWriter—Picture Comments, discusses these picture comments.

Memory Considerations

To print to the LaserWriter, you need to make sure that you have enough memory available to load the driver's code. The best way to do this is to have all the code you need for printing in a separate segment and unload everything else. When you print to the LaserWriter you are only able to print in Draft mode. You are not able to spool (as the ImageWriter does in the standard or high-quality settings), and your print code, data, and the driver code have to be resident in memory.

In terms of memory requirements, there is not any magic number that always works with all printer drivers (including third-party printer drivers) that are available for the Macintosh. To make sure there is enough memory available during print time, you should make your printing code a separate segment and swap out all unwanted code and data before you call `_PrOpen`.

Printable Paper Area

On the LaserWriter there is a 0.45-inch border that surrounds the printable area of the paper (this is assuming an 8.5" x 11" paper). If you select the "Larger Print Area" option in the Page Setup dialog box, the border changes to 0.25 of an inch. This printable area is different than the available print area of the ImageWriter. An application cannot print a larger area because of the memory PostScript needs to image a page. PostScript takes the amount of memory available in the printer and centers it on the paper, and there is not enough RAM in the LaserWriter to image an entire sheet of paper.

Page Sizes

Many developers have expressed a desire to support page sizes other than those provided by the Apple printer drivers. Even though some devices can physically support other page sizes, there is no way for an application to tell the driver to use this size. With the ImageWriter driver, it is possible to modify certain fields in the print record and expand the printable area of the page. However, each of the Apple drivers implements the page sizes in a different way. No one method works for all drivers. Because of this difference, it is strongly recommended that applications do not attempt to change the page sizes provided in the "Style" dialog box. If your application currently supports page sizes other than those provided by the printer driver, you are taking a serious compatibility risk with future Apple and third-party printer drivers.

Speed Considerations

Although the LaserWriter is relatively fast, there are some techniques an application can use to ensure its maximum performance.

- Try to avoid using the QuickDraw Erase calls (e.g., `_EraseRect`, `_EraseOval`, etc.). It takes a lot of time to handle the erase function because every bit (90,000 bits per square inch) has to be cleared. Erasing is unnecessary because the paper does not need to be erased the way the screen does.

- Printing patterns takes time, since the bitmap for the pattern has to be built. The patterns `black`, `white`, and all the gray patterns have been optimized to use the PostScript gray scales. If you use a different pattern it works, but it just takes longer than usual. In addition, the patterns in driver version 3.0 are rotated; they are not rotated in version 1.0.

- Try to avoid frequently changing fonts. PostScript has to build each character it needs either by using the drawing commands for the built-in LaserWriter fonts or by resizing bitmaps downloaded from screen fonts on the Macintosh. As each character is built, it is cached (if there's room), so if that character is needed again PostScript gets if from the cache. When the font changes, the characters have to be built from scratch in the new font, which takes time. If the font is not in the LaserWriter, it takes time to download it from the Macintosh. If the user has the option of choosing fonts, you have no control over this variable; however, if you control which fonts to use, keep this in mind.

- Avoid using `_TextBox`. It makes calls to `_EraseRect`, which slows the printer, for every line of text it draws. You might want to use a different method of displaying text (e.g., `_DrawString` or `_DrawText`) or write your own version of `_TextBox`. If an application is currently calling `_TextBox`, changing to another method of displaying text can improve speed on the order of five to one.

- Because of the way rectangle intersections are determined, if your clip region falls outside of the `rPage` rectangle, you slow down the printer substantially. By making sure your clip region is entirely within the `rPage` rectangle, you can get a speed improvement of approximately four to one.

- Do not use spool-a-page/print-a-page as some applications do when printing on the ImageWriter. It slows things down considerably because of all of the preparation that has to be done when a job is initiated. Refer to Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, for more information.

- Using `_DrawChar` to place every character to print can take a lot of time. One reason, of course, is because it has to go through the bottlenecks for every character that is drawn. The other is that the printer driver does its best to do line layout, making the character spacing just right. If you are trying to position characters and the driver is trying to position characters too, there is conflict, and printing takes much longer than necessary. In version 3.0 of the driver, there are picture comments that turn off the line layout optimization, alleviating some of the problem. Refer to Technical Note #91, Optimizing for the LaserWriter—Picture Comments, for more information.

Clipping Within Text Strings

When clipping characters out of a string, make sure that the clipping rectangle or region is greater than the bounding box of the text you want to clip. The reason is that if you clip part of a character (e.g., a descender), the clipped character has to be rebuilt, which takes time. In addition, because of the difference between screen fonts and printer fonts, chances are that you cannot accurately clip the right characters unless you are running on the 128K ROMs and have fractional pixel widths enabled.

When to Validate the Print Record

To validate the print record, call `PrValidate`. You need validation to check to see if all of the fields are accurate according to the current printer selected and the current version of the driver. You should call `PrValidate` when you have allocated a new print record or whenever you need to access information from the print record (i.e., when you get `rPage`). The routines `PrStlDialog` and `PrJobDialog` call `PrValidate` when they are called, so you do not have to worry about it if you use these calls.

Empty QuickDraw Objects

QuickDraw objects that are empty (i.e., they have no pixels in them) and are filled but not framed, do not print on the ImageWriter and do not show up on the screen; however, on the LaserWriter they are real objects and do print.

Further Reference:

- *Inside Macintosh*, Volume I, QuickDraw
- *Inside Macintosh*, Volume II, The Printing Manager
- *LaserWriter Reference Manual*
- Technical Note #41, Drawing Into An Off-Screen Bitmap
- Technical Note #91, Optimizing for the LaserWriter—Picture Comments
- Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers
- Technical Note #161, A Printing Loop That Cares…
- PostScript Language Reference, Adobe Systems, Incorporated
- PostScript Language Tutorial and Cookbook, Adobe Systems, Incorporated

MacDraw is a registered trademark of Claris Corporation.
PostScript is a registered trademark of Adobe Systems, Incorporated.

#73: Color Printing

See also: QuickDraw
The Printing Manager
PostScript Language Reference Manual,
Adobe Systems

Written by: Ginger Jernigan February 3, 1986
Modified by: Scott "ZZ" Zimmerman January 1, 1988
Updated: March 1, 1988

This discusses color printing in a Macintosh application.

Whereas the original eight-color model of QuickDraw was sufficient for printing in color on the ImageWriter II, the introduction of Color QuickDraw has created the need for more sophisticated printing methods.

The first section describes using the eight-color QuickDraw model with the ImageWriter II and ImageWriter LQ drivers. Since the current Print Manager does not support Color GrafPorts, the eight-color model is the only method available for the ImageWriters.

The next section describes a technique that can be used for printing halftone images using PostScript (when it is available). Also described is a device independent technique for sending the PostScript data. This technique can be used on any LaserWriter driver 3.0 or later. It will work with all LaserWriters except the the LaserWriter IISC.

It is very likely that better color support will be added to the Print Manager in the future. Until then, these are the best methods available.

Part 1, ImageWriters

The ImageWriter drivers are capable of generating each of the eight standard colors defined in QuickDraw by the following constants:

```
    whiteColor
    blackColor
    redColor
    greenColor
    blueColor
    cyanColor
    magentaColor
    yellowColor
```

To generate color all you need to do is set the foreground and background colors before you begin drawing (initially they are set to blackColor foreground and whiteColor background). To do this you call the QuickDraw routines `ForeColor` and `BackColor` as described in *Inside Macintosh*. If you are using QuickDraw pictures, make sure you set the foreground and background colors before you call `ClosePicture` so that they are recorded in the picture. Setting the colors before calling `DrawPicture` doesn't work.

The drivers also recognize two of the transfer modes: `srcCopy` and `srcOr`. The effect of the other transfer modes is not well defined and has not been tested. It may be best to stay away from them.

Caveats

When printing a large area of more than one color you will encounter a problem with the ribbon. When you print a large area of one color, the printer's pins pick up the color from the back of the ribbon. When another large area of color is printed, the pins deposit the previous color onto the back of the ribbon. Eventually the first color will come through to the front of the ribbon, contaminating the second color. You can get the same kind of effect if you set, for example, a foreground color of yellow and a background color of blue. The ribbon will pick up the blue as it tries to print yellow on top of it. This problem is partially alleviated in the 2.3 version of the ImageWriter driver by using a different printing technique.

The ribbon goes through the printer rather quickly when printing large areas. When the ribbon comes through the second time the colors don't look too great.

Part 2, LaserWriters

Using the PostScript 'image' Operator to Print Halftones

About 'image'

The PostScript image operator is used to send Bitmaps or Pixmaps to the LaserWriter. The image operator can handle depths from 1 to 8 bits per pixel. Our current LaserWriters can only image about twenty shades of gray, but the printed page will look like there's more. Being that the image operator is still a PostScript operator, it expects its data in the form of hexidecimal bytes. The bytes are represented by two ASCII characters(0-9,A-F). The image operator takes these parameters:

```
width  height  depth  matrix  image-data
```

The first three are the width, height, and depth of the image, and the matrix is the transformation matrix to be applied to the current matrix. See the *PostScript Language Reference Manual* for more information. The image data is where the actual hex data should go. Instead of inserting the data between the first parameters and the image operator itself, it is better to use a small, PostScript procedure to read the data starting from right after the image operator. For example:

```
640 480 8 [640 0 0 480 0 0]
{currentfile picstr readhexstring pop}
image
FF 00 FF 00 FF 00 FF 00 ...
```

In the above example, the width of the image is 640, the height is 480, and the depth is 8. The matrix (enclosed in brackets) is setup to draw the image starting at QuickDraw's 0,0 (top left of page), and with no scaling. The PostScript code (enclosed in braces) is not executed. Instead, it is passed to the image operator, and the image operator will call it repeatedly until it has enough data to draw the image. In this case, it will be expecting 640*480 bytes. When the image operator calls the procedure, it does the following:

1. Pushes the current file which in this case is the stream of data coming to the LaserWriter over AppleTalk. This is the first parameter to readhexstring.

2. Next picstr is pushed. picstr is a string variable defined to hold one row of hex data. The PostScript to create the picstr is:

   ```
   /picstr 640 def
   ```

3. Now readhexstring is called to fill picstr with data from the current file. It begins reading bytes which are the characters following the image operator.

4. Since readhexstring leaves both the string we want, and a boolean that we don't want on the stack, we do one pop to kill of the boolean. Now the string is left behind for the image operator to use.

So using the above PostScript code you can easily print an image. Just fill in the width height and depth, and send the hex data immediately following the PostScript code.

Setting Up for 'image'

Most of the users of this technique are going to want to print a Color QuickDraw PixMap. Although the image command does a lot of the work for you, there are still a couple of tricks that are recommended for performance.

Assume the Maximum Depth

Since the current version of the image operator has a maximum depth of 8 bits/pixel, it is wise to convert the source image to the same depth before imaging. This can be done very simply by using an offscreen GrafPort that is set to 8 bits/pixel, and then using CopyBits to do the depth conversion for you. This will do a nice job of converting lower resolution images to 8 bits/pixel.

Build a Color Table

An 8 bit deep image can only use 256 colors. Since the image that you are starting with is probably color, and the image you get will be grayscale, you need to convert the colors in the source color table into PostScript grayscale values. This is actually easy to do using the Color Manager. First create a table that can hold 512 bytes. This is 2 bytes for each color value from 0 to 255. Since PostScript wants the values in ASCII, you need two characters for each pixel. Now loop through the colors in the color table. Call Index2Color to get the real RGB color for that index, and then call RGB2HSL to convert the RGB color into a luminance value. This value will be expressed as a SmallFract which can then be scaled into a value from 0 to 255. This value should then be converted to ASCII, and stored at the appropriate location in the table. When you are done, you should be able to use a pixel value as an index into your table of PostScript color values. For each pixel in the image, send two characters to the LaserWriter.

Sending the Data

Once you have set up the color table, all that left to do is to loop through all of the pixels, and send their PostScript representation to the LaserWriter. There are a couple of ways to do this. First is to use the low-level Print Manager interface and stream the PostScript using the stdBuf PrCtlCall. Although this seems like it would be the fastest way, the latest version of the LaserWriter driver (5.0) converts all low-level calls to their high level equivalent before executing them. Because of this, the low-level interface is no longer faster than the high level. In an FKEY I have written, I use the high-level Print Manager interface, and send the data via the PostScriptHandle PicComment. This way, I can buffer a large amount of data, before actually sending it. Using this technique, I have been able to image a Mac II screen in about 5 minutes on a LaserWriter Plus, and about 1.5 minutes on a LaserWriter II NTX.

#74: Don't Use the Resource Fork for Data

See also: The Resource Manager
Technical Note #62—Resource Header Application Bytes

Written by: Bryan Stearns March 13, 1986
Updated: March 1, 1988

Don't use the resource fork of a file for non-resource data. Parts of the system (including the File Manager and the Finder) assume that if this fork exists, it will contain valid Resource Manager information.

`PBOpenRF` was provided to allow copying of the resource fork of a file in its entirety, without Resource Manager interpretation. Do not use it to open "another data fork."

The File Manager assumes that the first block of the resource fork of a file will be part of the resource header, and puts information there to aid in scavenging. Note that this means that if you copy a resource file (opened with `PBOpenRF`), the duplicate may not be exactly like the original.

#75: Apple's Multidisk Installer

Revised by: Rich Kubota — January 1992
Written by: scott douglass — March 1986

This Technical Note documents Apple's Multidisk Installer, and it is in addition to separate Installer documentation which provides the details of writing scripts.
Changes since September 1991: Revised information on the use of Installer version 3.1 to version 3.2. Revised information on the use of ScriptCheck version 3.2.1 with Installer version 3.2. Added Common Questions and Answers relating to the use of the Installer.

Apple's Multidisk Installer is intended to make it easy for Macintosh users to add or update software. It is a very useful tool for adding third-party software, and Apple recommends that you use the Installer unless your software installation is simple. Apple also recommends that you use version 3.2 of the Installer.

The Multidisk Installer has the following features, as of version 3.2:

- "Easy Install" mode where the Installer script writer can determine the appropriate installation based upon examination of the target environment and provide the user with "One-Button Installation"
- An optional "Custom Install" mode where power users can customize their installation
- "Live" installation to the currently booted and active system; thus it is no longer necessary to ship the Installer on a bootable disk with a System Folder
- Ability to install from an AppleShare server ("Network Install")
- Ability to install from multiple source disks
- Installation of software to folders other than the System Folder as well as creation of new folders as necessary
- Runs under System 4.2 and later versions
- "User Function" support; this feature provides linkage to developer-defined code segments during Easy Install, so script writers can customize the process of determining what software to install and how to install it
- "Action Atom" support; this feature provides linkage to developer-defined code segments that are called before or after the installation takes place; script writers can use this feature to extend the capabilities of the Installer
- Audit Records; this feature provides the script writer with the ability to record details about an installation so that future installations can be more intelligent

The `'indm'` (default map) resource of Installer 3.0.1 is no longer supported in Installer 3.1 and later versions. This was used by script writers to implement Easy Install. It is replaced by `'infr'` (framework) and `'inrl'` (rule) resources.

Note: If the user opens the Installer document rather than the Installer, the wrong Installer may be launched (depending upon the contents of their mounted volumes). This is only a problem between versions 3.1 and 3.0.x. If you are developing a 3.1 script, you may want to add an `'indm'` resource that puts up a warning dialog box. If you are developing a 3.0.x script, you may want to add an `'infr'` and `'inrl'` resource that puts up a `reportSysError` dialog box. This problem is resolved in Installer 3.2. With version 3.2, the file type and creator are both `'bjbc'` as opposed to the use of `'cfbj'` with versions 3.0.1 and 3.1.

Installer version 3.2 is available as a complete reference suite which includes the following:

- *Installer 3.2 Scripting Guide* (dated December 1, 1991, on the cover)
- *Installer ScriptCheck 3.2b7 User's Manual*
- Installer 3.2 application
- ScriptCheck 3.2.1 (MPW Tool)
- InstallerTypes.r (MPW Rez interface file)
- ActionAtomIntf.a, .h, .p (Action Atom interface files for Assembler, C, and Pascal)

The reference suite for Multidisk Installer 3.2 is available on the latest Developer CD and on AppleLink in the Developer Services Bulletin Board. The Multidisk Installer was also provided on the System 7 Golden Master CD-ROM: however, that package included the b7 release of the MPW ScriptCheck tool.

Multidisk Installer version 3.2 contains a few minor improvements that will make it easier to write scripts that work on both System 6.0.x and 7.0. Installer 3.1 had minimal testing with System 7.0. If you are expecting to install software onto machines running System 7.0, you should consider upgrading. Script changes should be minimal.

Common Question and Answers

Q How can I check for a minimum system version?

A Use the `checkFileVersion` clause as part of the `'inrl'` Rules Framework resource. The format of the minimal-version parameter is shown in the InstallerTypes.r file as '`#define Version`'. The most common difficulties are in remembering that BCD values are required and in dealing with two-digit version numbers. Some samples follow.

Assuming that the target-filespec resource, `'infs'`, for the System file is 1000, use the following clause to check for System version 6.0.5:

```
checkFileVersion{1000, 6, 5, release, 0};
```

Assuming that the target-filespec resource, `'infs'`, for the Finder file is 1001, use the following clause to check for Finder version 6.1.5:

```
checkFileVersion{1001, 6, 0x15, release, 0};
```

Assuming that the target-filespec resource, `'infs'`, for the AppleTalk resource file is 1002, use the following clause to check for AppleTalk version 53:

```
checkFileVersion{1002, 0x53, 0, release, 0};
```

Q My Installer script installs a desk accessory. Under System 6, each time I run the script, a new copy of the DA appears as a DRVR resource in the System file. Why?

A Unfortunately, this is a symptom when the `'deleteWhenInstalling'` flag is used in conjunction with the `'updateExisting'` flag. The *Installer 3.1 & 3.2 Scripting Guide* indicates that resources marked with the `'dontDeleteWhenInstalling'` flag can be replaced with a new resource. The guide also indicates that the Installer will overwrite a preexisting resource in the target file if the 'updateExisting' flag is set. Given these two flag settings, and the use of the replace 'byName' (noByID) flag, the Installer does not delete the DA. Instead a new DRVR resource is created with the same name but a new resource ID.

The correct Installer action is accomplished by setting the 'deleteWhenInstalling' flag in conjunction with the 'updateExisting' flag. Alternatively, use the 'dontDeleteWhenInstalling' flag with the 'keepExisting' flag.

Q How can I include the current volume name in a reportVolError alert as many of the installation scripts from Apple do?

A The volume name can be included by inserting "^0" in the desired location of the Pascal string passed to the reportVolError error reporting clause.

Q. I set the searchForFile flag in my 'infs' resource, however, the Installer acts as if it's unable to find the file. Why?

A. The likely reason for this problem is that the desired file is within a folder by the same name. When the searchForFile flag is set, the Installer will also find a match on a folder. The Installer will not replace a folder with a file, nor will it add a resource to a folder. The Installer continues as if the search failed.

Q What is the 'incd' resource about?

A When the MPW ScriptCheck tool is used, it reads the script's file creation date/time stamp and converts it into a long word with the Date2Secs procedure. ScriptCheck stores this long word in the 'incd' resource for use with verifying files when a network installation is performed. See the following questions for a discussion of this resource.

Q What checks are made by the Installer when preflighting an installation? Occasionally the alert "Could not find a required file . . ." occurs and the installation is aborted.

A The Installer compiles a list of the source file specifications from each of the resource 'inra' and file 'infa' atoms specified among the package 'inpk' atoms included for installation. Each source file specification includes a complete path name. As each source file is accessed, a check is made of the file's creation date/time stamp with the date/time stamp recorded in the corresponding 'infs' resource. If the date/time stamps do not match, the alert results and the installation is aborted. The creation date/time stamp in the 'infs' resource can be

- entered manually into the script file so long as the value is not 1 or 0,
- filled in by ScriptCheck automatically, if a value of 1 is entered in the date field,
- forced to be updated, if the -d switch is used with ScriptCheck.

Q What are some of the considerations when configuring a network installation setup?

A Under Installer 3.1/3.2, network software installations are made possible by setting up an installation folder on the server volume. This folder will contain the Installer application, the Script file, and a folder(s) matching the names of the required disk(s). Within the disk folder(s) are the corresponding contents of the disk(s).

A problem can occur when a workstation is used to create the server installation folder and the system date and time differ significantly between the two systems. Under such condition, files copied from the workstation to the server may have their creation and modification date time/stamps altered. If a modification is made, the "delta" is constant for both the creation and modification date/time stamp and for all files copied at that time.

As indicated in the previous question, the Installer preflights a file by comparing its creation date/time stamp with the value stored in the corresponding 'infs' resource in the script file. To compensate for the fact that a server may alter a file's creation date/time stamp, the Installer implements the 'incd' resource. After the user selects the Install button, the Installer reads the 'incd' resource and compares it with the script file's creation date/time stamp. The difference is stored as the "delta." On a normal disk installation, the "delta" is always zero. As the Installer finds each required source file, the file's creation date/time stamp is converted to a long word and adjusted by the "delta." The modified date/time stamp is then compared with that stored in the script file. If the values match, the file is considered found and the installation proceeds. On network installations, the delta may be nonzero. If so, it indicates that the file's creation date/time stamps were modified when copied to the server. Thus the 'incd' resource gives the Installer a way to maintain file verification even though the date/time stamp may be altered.

A specific problem can occur when an installation is set up on some systems running older versions of Novell Server software. Under specific conditions, files copied to some Novell servers have their creation time stamp altered to 12:00 A.M. regardless of the original time stamp. This includes the creation time stamp of the script file. This condition wreaks havoc with the Installer's preflight mechanism. The "delta" determined between the 'incd' resource and the Script file's creation date/time stamp may not be consistent with the creation date/time stamp stored in the infs resource and the corresponding file's time stamp now at 12:00 A.M.

A work-around solution for this problem is to set the Creation time stamp for all files on the installation disk to 12:00 A.M. , BEFORE running the ScriptCheck tool. Use the MPW tool SetFile to perform this function. Here's a sample MPW script for performing this function:

```
SetFile   -d "1/1/92 12:00AM" `files -r -s -f ≈`
```

This script assumes that the current directory is set to the root of the Installation disk. For multiple disks, run this script on each disk. Use the '-f' switch with ScriptCheck to ensure that the date/time stamps are updated on scripts previously checked.

Installation of software is a nontrivial process. Apple recommends that you reserve time for development and testing to ensure that the installation process proceeds smoothly on all target machine configurations.

To ship the Installer with your product you must contact Apple's Software Licensing Department (AppleLink: SW.LICENSE) and license the Installer alone or with the system software package that includes the version of the Installer you intend to use. Software Licensing also supplies you with a copy of the Installer that you may ship.

Further Reference:

- Installer 3.2 Reference Suite

#76: The Macintosh Plus Update Installation Script

Written by:	scott douglass	February 24, 1986
Updated:		March 1, 1988

Earlier versions of this note described the Macintosh Plus Update installation script, because it was the first script created for the Installer. Since then, many versions of this script have been created which no longer match what was described here. In addition, many other scripts now exist.

#77: HFS Ruminations

See also: The File Manager
Technical Note #66—
Determining Which File System is Active
Technical Note #67—Finding the "Blessed Folder"
Technical Note #68—
Searching All Directories on an HFS Volume

Written by: Jim Friedlander June 7, 1986
Updated: March 1, 1988

This technical note contains some thoughts concerning HFS.

HFS numbers

A drive number is a small positive word (e.g. 3).

A `VRefNum` (as opposed to a `WDRefNum`) is a small negative word (e.g. `$FFFE`).

A `WDRefNum` is a large negative word (e.g. `$8033`).

A `DirID` is a long word (e.g. 38). The root directory of an HFS volume always has a `dirID` of 2.

Working Directories

Normally an application doesn't need to open working directories (henceforth `WDs`) using `PBOpenWD`, since `SFGetFile` returns a `WDRefnum` if the selected file is in a directory on a hierarchical volume and you are running HFS. There are times, however, when opening a `WD` is desirable (see the discussion about `BootDrive` below).

If you do open a `WD`, it should be created with an `ioWDProcID` of 'ERIK' (`$4552494B`) and it will be deallocated by the Finder. Note that under MultiFinder, ioWDProcID will be ignored, so you should only use 'ERIK'.

`SFGetFile` also creates `WDs` with an `ioWDProcID` of 'ERIK'. If `SFGetFile` opens two files from the same directory (during the same application), it will only create one working directory.

There are no WDRefnums that refer to the root—the root directory of a volume is always referred to by a vRefNum.

When you can use HFS calls

All of the HFS 'H' calls, except for PBHSetVInfo, can be made without regard to file system as long as you pass in a pointer to an HFS parameter block. PBHGetVol, PBHSetVol (see the warnings in the File Manager chapter of *Inside Macintosh*), PBHOpen, PBHOpenRF, PBHCreate, PBHDelete, PBHGetFInfo, PBHSetFInfo, PBHSetFLock, PBHRstFLock and PBHRename differ from their MFS counterparts only in that a dirID can be passed in at offset $30.

The only difference between, for example, PBOpen and PBHOpen is that bit 9 of the trap word is set, which tells HFS to use a larger parameter block. MFS ignores this bit, so it will use the smaller parameter block (not including the dirID). Remember that all of these calls will accept a WDRefNum in the ioVRefNum field of the parameter block.

PBHGetVInfo returns more information than PBGetVInfo, so, if you're counting on getting information that **is** returned in the HFS parameter block, but not in the MFS parameter block, you should check to see which file system is active.

HFS-specific calls can only be made if HFS is active. These calls are: PBGetCatInfo, PBSetCatInfo, PBOpenWD, PBCloseWD, PBGetFCBInfo,PBGetWDInfo, PBCatMove and PBDirCreate. PBHSetVInfo has no MFS equivalent. If any of these calls are made when MFS is running, a system error will be generated. If PBCatMove or PBDirCreate are called for an MFS volume, the function will return the error code –123 (wrong volume type). If PBGetCatInfo or PBSetCatInfo are called on MFS volumes, it's just as if PBGetFInfo and PBSetFInfo were called.

Default volume

If HFS is running, a call to GetVol (before you've made any SetVol calls) will return the WDRefNum of your application's parent directory in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the WDRefNums of those documents' parent directories are available in the vRefNum field of the AppFile record returned from GetAppFiles.

If MFS is running, a call to GetVol (before you've made any SetVol calls) will return the vRefNum of the volume your application is on in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the vRefNum of those documents' volume are available in the vRefNum field of the AppFile record returned from GetAppFiles.

BootDrive

If your application or desk accessory needs to get the WDRefNum of the "blessed folder" of the boot drive (for example, you might want to store a configuration file there), it can not rely on the low-memory global BootDrive (a word at $210) to contain the correct value. If your application is the startup application, BootDrive will contain the WDRefNum of the directory/volume that your application is in (not the WDRefNum of the "blessed folder"); Your application could have been _Launched from an application that has modified BootDrive; if you are a desk accessory, you might find that some applications alter BootDrive.

To get the "real" WDRefNum of the "blessed folder" that contains the currently open System file, you should call SysEnvirons (discussed in Technical Note #129). If that is impossible, you can do something like this (**Note:** if you are running under MFS, BootDrive always contains the vRefNum of the volume on which the currently open System file is located):

```
...
CONST
        SysWDProcID   = $4552494B; {"ERIK"}
        BootDrive     = $210;      {address of Low-Mem global BootDrive}
        FSFCBLen      = $3F6;      {address of Low-Mem global to
                                    distinguish file systems }
        SysMap        = $A58;      {address of Low-Mem global that contains
                                    system map reference number}

TYPE
        WordPtr = ^Integer;                 {Pointer to a word(2 bytes)}
...

FUNCTION HFSExists: BOOLEAN;

Begin {HFSExists}
        HFSExists := WordPtr(FSFCBLen)^ > 0;
End;   {HFSExists}


FUNCTION GetRealBootDrive: INTEGER;

VAR
        MyHPB         : HParamBlockRec;
        MyWDPB        : WDPBRec;
        err           : OSErr;
        sysVRef       : integer; {will be the vRefNum of open system's vol}

Begin {GetRealBootDrive}
        if HFSExists then Begin    {If we're running under HFS... }

                {get the VRefNum of the volume that }
                {contains the open System File      }
                err:= GetVRefNum(WordPtr(SysMap)^,sysVRef);
```

```
              with MyHPB do Begin
              {Get the "System" vRefNum and "Blessed" dirID}
                     ioNamePtr    := NIL;
                     ioVRefNum    := sysVRef; {from the GetVrefNum call}
                     ioVolIndex   := 0;
              End; {with}
              err := PBHGetVInfo(@MyHPB, FALSE);

              with myWDPB do Begin      {Open a working directory there}
                     ioNamePtr   := NIL;
                     ioVRefNum   := sysVRef;
                     ioWDProcID  := SysWDProcID; {Using the system proc ID}
                     ioWDDirID  := myHPB.ioVFndrInfo[1];{ see TechNote 67}
              End; {with}
              err := PBOpenWD(@myWDPB, FALSE);

              GetRealBootDrive := myWDPB.ioVRefNum;
              {We've got the real WD}
       End Else {we're running MFS}
              GetRealBootDrive := WordPtr(BootDrive)^;
              {BootDrive is valid under MFS}
End;  {GetRealBootDrive}
```

From MPW C:

```
/*"ERIK"*/
#define     SysWDProcID 0x4552494B
#define     BootDrive   0x210
/*address of Low-Mem global that contains system map reference number*/
#define     SysMap      0xA58
#define     FSFCBLen    0x3F6
#define     HFSIsRunning ((*(short int *)(FSFCBLen)) > 0)

OSErr GetRealBootDrive(BDrive)
short int *BDrive;
{ /*GetRealBootDrive*/

      /*three different parameter blocks are used here for clarity*/
      HVolumeParam      myHPB;
      FCBPBRec          myFCBRec;
      WDPBRec           myWDPB;
      OSErr             err;
      short int         sysVRef; /*will be the vRefNum of open system's
                                       vol*/

      if (HFSIsRunning)

      { /*if we're running under HFS... */

      /*get the vRefNum of the volume that contains the open System File*/
            myFCBRec.ioNamePtr= nil;
            myFCBRec.ioVRefNum = 0;
            myFCBRec.ioRefNum = *(short int *)(SysMap);
            myFCBRec.ioFCBIndx = 0;

            err = PBGetFCBInfo(&myFCBRec,false);
            if (err != noErr) return(err);
      /*now we need the dirID of the "Blessed Folder" on this volume*/
```

```
                myHPB.ioNamePtr = nil;
                myHPB.ioVRefNum = myFCBRec.ioFCBVRefNum;
                myHPB.ioVolIndex = 0;

                err = PBHGetVInfo(&myHPB,false);
                if (err != noErr) return(err);

          /*we can now open a WD for the directory that contains the open
          system file one will most likely already be open, so PBOpenWD will
          just return that WDRefNum*/
                myWDPB.ioNamePtr = nil;
                myWDPB.ioVRefNum = myHPB.ioVRefNum;
                myWDPB.ioWDProcID = SysWDProcID; /*'ERIK'*/
                myWDPB.ioWDDirID = myHPB.ioVFndrInfo[0]; /* see Technote # 67
                                                     [c has 0-based arrays]*/

                err = PBOpenWD(&myWDPB,false);
                if (err != noErr) return err;

                *BDrive = myWDPB.ioVRefNum; /*that's all!*/
          } /* if (HFSIsRunning) */
          else
                *BDrive = *(short int *)(BootDrive);
                /*BootDrive is valid under MFS*/

          return noErr;
    }                                            /*GetRealBootDrive*/
```

The Poor Man's Search Path (PMSP)

If HFS is running, the PMSP is used for any file system call that can return a file-not-found error, such as PBOpen, PBClose, PBDelete, PBGetCatInfo, etc. It is **not** used for indexed calls (that is, where ioFDirIndex is positive) or when a file is created (PBCreate) or when a file is being moved between directories (PBCatMove). The PMSP is also **not** used when a non-zero dirID is specified.

Here's a brief description of how the default PMSP works.

1) The directory that you specify (specified by WDRefNum or pathname) is searched; if the specified file is not found, then

2) the volume/directory specified by BootDrive (low-memory global at $210) is searched IF it is on the same volume as the directory you specified (see #1 above); if the specified file is not found, or the directory specified by BootDrive is not on the same volume as the directory that you specified, then

3) if there is a "blessed folder" on the same volume as the directory you specified (see #1 above), it is searched. Please note that if #2 above specifies the same directory as #3, then that directory is **not** searched twice. If no file is found, then

4) fnfErr is returned.

ioDirId and ioFlNum

Two fields of the HParamBlockRec record share the same location. ioDirID and ioFlNum are both at offset $30 from the start of the parameter block. This causes a problem, since, in some calls (e.g. PBGetCatInfo), a dirID is passed in and a file number is returned in the same field.

Future versions of Apple's HFS interfaces will omit the ioFlNum designator, so, if you need to get the file number of a file, it will be in the ioDirID of the parameter block **after** you have made the call. If you are making successive calls that depend on ioDirID being set correctly, you must "reset" the ioDirID field before each call. The program fragment in Technical Note #68 does this.

PBHGetVInfo

Normally, PBHGetVInfo will be called specifying a vRefNum. There are times, however, when you may make the call and only specify a volume name. If this is so, there are a couple of things to look out for.

Let's say that we have two volumes mounted: "Vol1:" (the default volume) and "Vol2:". We also have a variable of type HParamBlockRec called MyHPB. We want to get information about Vol2:, so we put a pointer to a string (let's call it fName) in MyHPB.ioNamePtr. The string fName is equal to "Vol2" (Please note the missing colon). We also initialize MyHPB.ioVRefNum to 0. Then we make the call. We are very surprised to find out that we are returned an error of 0 (noErr) and that the ioVRefNum that we get back is **not** the vRefNum of Vol2:, but rather that of Vol1:.

Here's what's happening: PBHGetVInfo looks at the volume name, and sees that it is improper (it is missing a colon). So, being a forgiving sort of call, it goes on to look at the ioVRefNum field that you passed it (see pp. 99 of *Inside Macintosh*, vol. II). It sees a 0 there, so it returns information about the default volume.

If you want to get information about a volume, and you just have its name and you are not sure that the name is a proper one, you should set MyHPB.ioVRefNum to –32768 ($8000). No vRefNum or WDRefNum can be equal to $8000. By doing this, you are forcing PBHGetVInfo to use the volume name and, if that name is invalid, to return a –35 error (nsvErr), "No such volume."

PBGetWDInfo and Register D1

There was a problem with PBGetWDInfo that sometimes caused the call to inaccurately report the dirID of a directory. It is fixed in System 3.2 and later. To be absolutely sure that you won't get stung by this, clear register D1 (CLR.L D1) before a call to PBGetWDInfo. You can do this either with an INLINE (Lisa Pascal and most C's) or with a short assembly-language routine before the call to PBGetWDInfo.

#78: Resource Manager Tips

See also: The Resource Manager
The Memory Manager
The Menu Manager
Technical Note #129—SysEnvirons

Written by: Jim Friedlander June 8, 1986
Updated: March 1, 1988

This note discusses some problems with the Resource Manager and how to work around them.

OpenResFile Bug

This section of the note formerly described a bug in `OpenResFile` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

GetMenu and ResErrProc

If your application makes use of `ResErrProc` (a pointer to a procedure stored in low-memory global `$AF2`) to detect resource errors, you will get unexpected calls to your `ResErrProc` procedure when calling `GetMenu` on 128K ROMs. The Menu Manager call `GetMenu` makes a call to `GetResInfo`, requesting resource information about MDEF 0. Unfortunately, `ROMMapInsert` is set to `FALSE`, so this call fails, setting `ResErr` to –192 (`resNotFound`). This in turn will cause a call to your `ResErrProc`, procedure even though the `GetMenu` call has worked correctly. This is **only** a problem if you are using `ResErrProc`.

The workaround is to:

1) save the address of your `ResErrProc` procedure
2) clear `ResErrProc`
3) do a `GetResource` call on the MENU resource you want to get
4) check to see if you get a nil handle back, if you do, you can handle the error in whatever way is appropriate for your application
5) call `GetMenu`, and
6) when you are done calling `GetMenu`, restore `ResErrProc`

SetResAttrs on read-only resource maps

SetResAttrs does not return an error if you are setting the resource attributes of a resource in a resource file that has a read-only resource map. The workaround is to check to see if the map is read-only and proceed from there:

```
CONST
    MapROBit = 8; {Toolbox bit ordering for bit 7 of low-order byte}

BEGIN
    ...
    attrs:= GetResFileAttrs(refNum);
    IF BitTst(@attrs,MapROBit) THEN ... {write-protected map}
```

Macintosh
Technical Notes

Developer Technical Support

#79: _ZoomWindow

Revised by: Craig Prouse April 1990
Written by: Jim Friedlander June 1986

This Technical Note contains some hints about using _ZoomWindow.
Changes since February 1990: Fixed a bug in DoWZoom which caused crashes if the content of a window did not intersect with any device's gdRect. Also made DoWZoom more robust by making savePort a local variable and checking for off-screen and inactive GDevice records. (One variable name has changed.) Additional minor changes: Corrected original sample code to use _EraseRect before zooming and added references to Human Interface Note #7, Who's Zooming Whom? for more subtle and application-specific considerations.

Basics

_ZoomWindow allows a window to be toggled between two states (where "state" means size and location): a default state and a user-selectable state. The default state stays the same unless the application changes it, while the user-selectable state is altered when the user changes the size or location of a zoomable window. The code to handle zoomable windows in a main event loop would look something like the examples which follow.

Note: _ZoomWindow assumes that the window that you are zooming is the current GrafPort. If thePort is not set to the window that is being zoomed, an address error is generated.

MPW Pascal

```
CASE myEvent.what OF
  mouseDown: BEGIN
    partCode:= FindWindow(myEvent.where, whichWindow);
      CASE partCode OF
        inZoomIn, InZoomOut:
          IF TrackBox(whichWindow, myEvent.where, partCode) THEN
            BEGIN
              GetPort(oldPort);
              SetPort(whichWindow);
              EraseRect(whichWindow^.portRect);
              ZoomWindow(whichWindow, partCode, TRUE);
              SetPort(oldPort);
            END; {IF}
        ...{and so on}
      END; {CASE}
    END; {mouseDown}
  ...{and so on}
END; {CASE}
```

MPW C

```
switch (myEvent.what) {
  case mouseDown:
    partCode = FindWindow(myEvent.where, &whichWindow);
    switch (partCode) (
      case inZoomIn:
      case inZoomOut:
        if (TrackBox(whichWindow, myEvent.where, partCode)) (
          GetPort(&oldPort);
          SetPort(whichWindow);
          EraseRect(whichWindow->portRect);
          ZoomWindow(whichWindow, partCode, true);
          SetPort(oldPort);
          } /* if */
        break;
      ... /* and so on */
    } /* switch */
  ... /* and so on */
} /* switch */
```

If a window is zoomable, that is, if it has a window definition `ID = 8` (using the standard `'WDEF'`), `WindowRecord.dataHandle` points to a structure that consists of two rectangles. The user-selectable state is stored in the first rectangle, and the default state is stored in the second rectangle. An application can modify either of these states, though modifying the user-selectable state might present a surprise to the user when the window is zoomed from the default state. An application should also be careful to not change either rectangle so that the title bar of the window is hidden by the menu bar.

Before modifying these rectangles, an application must make sure that `DataHandle` is not `NIL`. If it is `NIL` for a window with window definition `ID = 8`, that means that the program is not executing on a system or machine that supports zooming windows.

One need not be concerned about the use of a window with window definition `ID = 8` making an application machine-specific—if the system or machine that the application is running on doesn't support zooming windows, `_FindWindow` never returns `inZoomIn` or `inZoomOut`, so neither `_TrackBox` nor `_ZoomWindow` are called.

If `DataHandle` is not `NIL`, an application can set the coordinates of either rectangle. For example, the Finder sets the second rectangle (default state) so that a zoomed-out window does not cover the disk and trash icons.

For the More Adventurous (or Seeing Double)

Developers should long have been aware that they should make no assumptions about the screen size and use `screenBits.bounds` to avoid limiting utilization of large video displays. Modular Macintoshes and Color QuickDraw support multiple display devices, which invalidates the use of `screenBits.bounds` unless the boundary of only the primary display (the one with the menu bar) is desired. When dragging and growing windows in a **multi-screen environment**, developers are now urged to use the bounding rectangle of the `GrayRgn`. In most cases, this is not a major modification and does not add a significant amount of code. Simply define a variable

```
desktopExtent := GetGrayRgn^^.rgnBBox;
```

and use this in place of screenBits.bounds. When zooming a document window, however, additional work is required to implement a window-zooming strategy which fully conforms with Apple's Human Interface Guidelines.

One difficulty is that when a new window is created with _NewWindow or _GetNewWindow, its default stdState rectangle (the rectangle determining the size and position of the zoomed window) is set by the Window Manager to be the gray region of the main display device inset by three pixels on each side. If a window has been moved to reflect a position on a secondary display, that window still zooms onto the main device, requiring the user to pan across the desktop to follow the window. The preferred behavior is to zoom the window onto the device containing the **largest portion** of the unzoomed window. This is a perfect example of a case where it is necessary for the application to modify the default state rectangle before zooming.

DoWZoom is a Pascal procedure which implements this functionality. It is a good example of how to manipulate both a WStateData record and the Color QuickDraw device list. On machines without Color QuickDraw (e.g., Macintosh Plus, Macintosh SE, Macintosh Portable) the stdState rectangle is left unmodified and the procedure reduces to five instructions, just like it is illustrated under "Basics." If Color QuickDraw is present, a sequence of calculations determines which display device contains most of the window prior to zooming. That device is considered dominant and is the device onto which the window is zoomed. A new stdState rectangle is computed based on the gdRect of the dominant GDevice. Allowances are made for the window's title bar, the menu bar if necessary, and for the standard three-pixel margin. (Please note that DoWZoom only mimics the behavior of the default _ZoomWindow trap as if it were implemented to support multiple displays. It does not account for the "natural size" of a window for a particular purpose. See Human Interface Note #7, Who's Zooming Whom?, for details on what constitutes the natural size of a window.) It is not necessary to set stdState prior to calling _ZoomWindow when zooming back to userState, so the extra code is not executed in this case.

DoWZoom is too complex to execute within the main event loop as shown in "Basics," but if an application is already using a similar scheme, it can simply add the DoWZoom procedure and replace the conditional block of code following

```
IF TrackBox…
```

with

```
DoWZoom(whichWindow, partCode);.
```

Happy Zooming.

```
PROCEDURE DoWZoom (theWindow: WindowPtr; zoomDir: INTEGER);
VAR
  windRect, theSect, zoomRect : Rect;
  nthDevice, dominantGDevice : GDHandle;
  sectArea, greatestArea : LONGINT;
  bias : INTEGER;
  sectFlag : BOOLEAN;
  savePort : GrafPtr;
BEGIN
  { theEvent is a global EventRecord from the main event loop }
  IF TrackBox(theWindow,theEvent.where,zoomDir) THEN
    BEGIN
      GetPort(savePort);
      SetPort(theWindow);
      EraseRect(theWindow^.portRect);    {recommended for cosmetic reasons}

      { If there is the possibility of multiple gDevices, then we  }
      { must check them to make sure we are zooming onto the right }
      { display device when zooming out. }
      { sysConfig is a global SysEnvRec set up during initialization  }
      IF (zoomDir = inZoomOut) AND sysConfig.hasColorQD THEN
        BEGIN
          { window's portRect must be converted to global coordinates }
          windRect := theWindow^.portRect;
          LocalToGlobal(windRect.topLeft);
          LocalToGlobal(windRect.botRight);
          { must calculate height of window's title bar }
          bias :=   windRect.top - 1
              -  WindowPeek(theWindow)^.strucRgn^^.rgnBBox.top;
          windRect.top := windRect.top - bias; {Thanks, Wayne!}
          nthDevice := GetDeviceList;
          greatestArea := 0;
          { This loop checks the window against all the gdRects in the   }
          { gDevice list and remembers which gdRect contains the largest }
          { portion of the window being zoomed. }
          WHILE nthDevice <> NIL DO
            IF TestDeviceAttribute(nthDevice,screenDevice) THEN
              IF TestDeviceAttribute(nthDevice,screenActive) THEN
                BEGIN
                  sectFlag := SectRect(windRect,nthDevice^^.gdRect,theSect);
                  WITH theSect DO
                    sectArea := LONGINT(right - left) * (bottom - top);
                  IF sectArea > greatestArea THEN
                    BEGIN
                      greatestArea := sectArea;
                      dominantGDevice := nthDevice;
                    END;
                  nthDevice := GetNextDevice(nthDevice);
                END; {of WHILE}
          { We must create a zoom rectangle manually in this case. }
          { account for menu bar height as well, if on main device }
          IF dominantGDevice = GetMainDevice THEN
            bias := bias + GetMBarHeight;
          WITH dominantGDevice^^.gdRect DO
            SetRect(zoomRect,left+3,top+bias+3,right-3,bottom-3);
          { Set up the WStateData record for this window. }
          WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
        END; {of Color QuickDraw conditional stuff}

      ZoomWindow(theWindow,zoomDir,TRUE);
      SetPort(savePort);
    END;
END;
```

In an attempt to avoid declaring additional variables, the original version of this document was flawed. In addition, the assignment statement responsible for setting the `stdState` rectangle is relatively complex and involves two type-casts. The following may look like C, but it really is Pascal. Trust me.

```
WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
```

It could be expanded into a more readable form such as:

```
VAR
   theWRec : WindowPeek;
   zbRec   : WStateDataHandle;

theWRec := WindowPeek(theWindow);
zbRec := WStateDataHandle(theWRec^.dataHandle);
zbRec^^.stdState := zoomRect;
```

Further Reference:

- *Inside Macintosh*, Volume IV, The Window Manager (pp. 49–52)
- *Inside Macintosh*, Volume V, Graphics Devices (p. 124), The Window Manager (p. 210)
- Human Interface Note #7, Who's Zooming Whom?

#80: Standard File Tips

See also: The Standard File Package

Written by: Jim Friedlander June 7, 1986
Updated: March 1, 1988

SFSaveDisk and CurDirStore

Low-memory location $214 (SFSaveDisk—a word) contains –1* the vRefNum of the volume that SF is displaying (MFS and HFS). It never contains –1* a WDRefNum.

Low-memory location $398 (CurDirStore—a long word) contains the dirID of the directory that SF is displaying (HFS only).

This information can be particularly useful at hook time, when the vRefNum field of the reply record has not yet been filled in. **Note:** reply.fName is filled in correctly at hook time if a file has been selected. If a directory has been selected, reply.fType is non-zero (it contains the dirID of the selected directory). If neither a file nor a directory is selected, both reply.fName[0] and reply.fType are 0.

Setting Standard File's default volume and directory

If you want SFGetFile or SFPutFile to display a certain volume when it draws its dialog, you can put –1 * the vRefNum of the volume you wish it to display into the low-memory global SFSaveDisk (a word at $214).

In Pascal, you would use something like:

```
...
TYPE
   WordPtr = ^INTEGER;                {pointer to a two-byte location}
CONST
   SFSaveDisk = $214;                 {location of low-memory global}
VAR
   SFSaveVRef : WordPtr;
   myVRef     : INTEGER;
BEGIN
...
{myVRef gets assigned here}
...
SFSaveVRef := WordPtr(SFSaveDisk);   {point to SFSaveDisk}
SFSaveVRef^:= -1 * myVRef;           {"stuff" the value in}
SFGetFile(...
```

In C you would use something like this (where a variable of type "short" occupies 2 bytes):

```
#define SFSaveDisk (*(short *)0x214)

short myVRef;
...
/* myVRef gets assigned here */
...
SFSaveDisk = -1 * myVRef; /* "stuff" the value in */
SFGetFile(...
```

If you are running HFS and would like to have Standard File display a particular directory as well as a particular volume, you can't just put a `WDRefNum` into `SFSaveDisk`. If you do put a `WDRefNum` into `SFSaveDisk`, Standard File will display the root directory of the default volume. Instead, you must put –1 * the `vRefNum` into `SFSaveDisk` (see above) and put the `dirID` of the directory that you wish to have displayed in `CurDirStore`. If you put an invalid `dirID` into `CurDirStore`, Standard File will display the root level of the volume referred to by `SFSaveDisk`. To change `CurDirStore` you can use a technique similar to the above, but remember that `CurDirStore` is a four-byte value. If your application is running under MFS, Standard File ignores `CurDirStore`, so you can use the same code regardless of file system.

#81: Caching

See also: The File Manager
The Device Manager
Technical Note #14—The INIT 31 Mechanism

Written by: Rick Blair June 17, 1986
Updated: March 1, 1988

This technical note describes disk and File System caching on the Macintosh, with particular emphasis on the high-level File System cache. Of the three caches used for file I/O, this is the one which could have the most impact on your program. **Note**: This big File System cache is not available on 64K ROM machines.

A term

In this note I will use the term “HFS” to mean the Hierarchical File System **and** the Sony driver which can access the 800K drives. Both RAM-based HFS (Hard Disk 20 file) and the 128K ROM version include the second-generation Sony driver.

There’s always a cache (type 1)

The first type of cache used by the File System has been around since the days of the Macintosh File System. Under MFS, each volume has a one-block buffer for all file/volume data. This prevents a read of two bytes followed by a read (at the next file position) of 4 bytes from causing actual disk I/O. The volume allocation map also gets saved in the system heap but it’s not really part of the cache.

This type of caching is still used by HFS, which includes MFS-format volumes which may be mounted while running HFS. With HFS, the cache is a little bigger: each volume gets 1 block of buffering for the bitmap, 2 blocks for volume (including file) data, and 16 blocks for HFS B*-tree control buffering.

This cache lives in the system heap (unless HFS is using the new File System caching mechanism, in which case things become more complicated. See “type 3” below).

Cache track fever (type 2)

The track cache, only present with the enhanced Sony driver, will cache the current track (up to twelve blocks) so that subsequent reads to that track may use the cache. The track cache is "write through"; all writes go to both the cache and the Sony disk so flushing is never required.

Track caching only takes place for synchronous I/O calls; when an application makes asynchronous calls it expects to use the time while the disk is seeking, etc. to execute other code.

The track cache gets its storage space from the system heap.

Cache me if you can (type 3)

The last type of cache to be discussed is only available under the 128K and greater ROMs. This user-controlled cache is **not** "write-through".

Based on how much space the user has allocated via the control panel, the File System will set up a cache which can accommodate a certain number of blocks. This storage will come from the application heap in the space above `BufPtr` (see technical note #14 and below). This is really the space above the jump table and the "`A5` world", not technically part of the application heap. However, moving `BufPtr` down will cause a corresponding reduction in the space available to the application heap.

The installation code will also grab the space used by the old File System cache (type 1) since all types of disk blocks can be accommodated by this new cache.

The bulk of the caching **code** used for this RAM cache is also loaded above `BufPtr` at application launch time. This is accomplished by the INIT 35 resource which is installed in the system heap and initialized at boot time. At application launch time, INIT 35 checks the amount of cache allocated via the control panel and moves `BufPtr` down accordingly before bringing in the balance of the caching code. The RAM caching code is in the 'CACH' 1 resource in the System File.

The caching code always makes sure there is room for 128K of application heap and 32K of cache. If the user-requested amount would reduce the heap/cache below these values then the cache space is readjusted accordingly.

Up to 36 separate files may be buffered by the cache. Each queue is a list of blocks cached for that file. Information is kept about the "age" of each block and the blocks are also kept in a list in the order in which they occur in the file. The aging information tells which blocks were least recently used; these are the first to be released when new blocks become eligible for caching. The file order information is useful for flushing the cache to the disk in an efficient manner, i.e. the file order approximates disk order.

Assuming this cache has been enabled by the user, all files which are read from *or* written to by File System (HFS) calls are subject to caching under the current implementation. The cache is not "write through" like the track cache. When a File System write (`PBWrite`, `WriteResource`, etc.) is done, the block is buffered until the block is released (age discrimination), a volume flush is done or the application terminates.

It may be useful to an application to prevent this process of reading and writing "in place". The Finder disables caching of newly read/written blocks while doing file copies since it would be silly to cache files that the Finder was reading into memory anyway. Copy protection schemes may also need this capability. Disabling reading and writing in place is accomplished by setting a bit in a low memory flag byte, `CacheCom` (see below). When you set this flag, no new candidates for caching will be accepted. Blocks already saved may still be read from the cache, of course.

`CacheCom` is at `$39C`. Bit 7 is the bit to set to disable subsequent caching, as follows:

```
        MOVE.B CacheCom,saveTemp ;save away the old value
        BSET.B #7,CacheCom       ;tell caching code to stop R/W I.P.
        ...
        BTST.B #7,saveTemp       ;check saved value
        BNE.S  @69
        BCLR.B #7,CacheCom       ;clear it if it was cleared before
    @69
```

Bit 6 contains another flag which can force **all** I/O to go to the disk. If that flag is set then every time even one byte is requested from the File System the disk will be hit. I can think of no good reason to use this except to test the system code itself. The other bits should likewise be left alone.

Please don't use this feature unnecessarily; the user should retain control over caching. **Important:** if your program doesn't have enough space to run due to caching you should ask the user to disable (or reduce) it with the control panel and then relaunch your application. This may be the subject of a future technical note.

BufPtr

The RAM-resident caching software arbitrates `BufPtr` in the friendliest manner possible. It saves the old value away before changing it, and then when it is time to release its space it looks at it again. If `BufPtr` has been moved again, it knows that it can't restore the old value it saved until `BufPtr` is put back to where it left it. In this manner any subsequent code or data put up under `BufPtr` is assured of not being obliterated by the caching routines.

A final note

To avoid problems with data in the cache not getting written out to disk, call `FlushVol` after each time you write a file to disk. This ensures that the cache is written, in case a crash occurs soon thereafter.

#82: TextEdit: Advice & Descent

See also: TextEdit
Technical Note #22—TEScroll Bug
Technical Note #127—TextEdit EOL Ambiguity
Technical Note #131—TextEdit Bugs

Written by: Rick Blair June 21, 1986
Updated: March 1, 1988

This technical note will point out some bugs (and possible workarounds), and other items of interest for the TextEdit programmer.

TESelRect

Multiple line selections are often more complex shapes than simple rectangles. If this is the case, the `teSelRect` field of the `TERec` is set to the **last** (bottommost) rectangle in the selection. The `teHiHook` is called to invert each line of the selection.

The ROM limits the selection range (i.e. the lines that get set into `teSelRect`) to only those lines which will fit into the `viewRect`. This means that `teSelRect` will be left at the last **visible** line. (The old 64K ROMs made all the calls for the complete selection and just let clipping take care of the rest.)

TEDoText

The parameters of this special hook into TextEdit need a little additional explanation. `D3` and `D4` are described on page 391 of *Inside Macintosh Volume I* as being the first and last characters to be redrawn. This is true but specific to the –1 "`DoDraw`" case. In fact, all the calls to `TEDoText` are interested in these first and last character positions. They determine the selection for a (1) highlight call, the caret position for a (–2) `DoCaret` call (where `D4` is ignored as it's assumed to equal `D3`), etc.

Note that the `DoCaret` (–2) call behaves differently than described in *Inside Macintosh*, as well. Good old page 391 says it sets up the pen position for caret drawing. Since an `InvertRect` call is used to draw the caret if you use the default `teCarHook`, the ROMs just set up `teSelRect`, they don't bother with the QuickDraw pen.

TEScrpLength

Inside Macintosh describes `TEScrpLength` as a long integer; indeed, four bytes are reserved for this value with the intent of someday using that range of values. However, the ROMs use word operations in their accesses to `TEScrpLength` and make word calculations with it. This means that the high word of `TEScrpLength` is used for calculations. This is something to watch out for.

CharWidth

Inside Macintosh says that `CharWidth` takes stylistic variations into account when determining the width of a character. In fact, for *italic* and outlined styles the extra width is not taken into account. TextEdit relies on `CharWidth` for positioning of the caret, etc. If you have chosen to use, for instance, italic style in your TE record you will find that as you type the caret actually overlaps the character to the left and so when the caret is erased some of that character will get erased, too. This is somewhat disconcerting to the user but the program will still function correctly.

Clikloops

If you add your own click loop and try to do something like update scroll bars you may run into trouble. Before your routine gets called, TextEdit will have set clipping down to just the `viewRect`. You will have to save away the old clipping region, set it out to sufficient size (–32767, –32767, 32767, 32767 is probably OK), do your drawing, then restore TextEdit's clipping area so that it can function properly.

#83: System Heap Size Warning

See also: The Memory Manager

Written by: Jim Friedlander June 21, 1986
Updated: March 1, 1988

Earlier versions of this note pointed out that, due to varying system heap sizes, the application heap does not always start at `$CB00`. The start of the application heap has not been fixed for some time now; programs that depend on it never work on the Macintosh SE or the Macintosh II.

#84: Edit File Format

Written by: Harvey Alcabes April 11, 1985
Modified by: Bryan Johnson August 15, 1986
Updated: March 1, 1988

This technical note describes the format of the files created by Edit. It has been verified for versions 1.x and 2.0.

Edit, a text editor licensed by Apple and included in the Consulair 68000 Development System, can read any text-only file whose file type is TEXT. Files created by Edit have a creator ID of EDIT. Edit is a disk-based editor so the file length is not limited by available memory. Files created or modified by Edit, have the format described below; if they are not too long they can be read by any application which can read TEXT files (eg: MacWrite, Microsoft Word, or the APDA example program File).

> The data fork contains text (ASCII characters). Carriage return characters indicate line breaks; tab characters are displayed as described below. No other characters have special significance.
>
> The resource fork contains resources of type ETAB and EFNT. If Edit opens a text-only file that does not have these resources it will add them.
>
> The ETAB (Editor TAB) resource, resource ID 1004, contains two integers. The first is the number of pixels to display for each space within a tab (not necessarily the same as for the space character). The second integer is the number of these spaces which will be displayed for each tab character.
>
> The EFNT (Editor FoNT) resource, resource ID 1003, contains an integer followed by a Pascal string (length byte followed by characters). The integer is the point size of the document's font. The string contains the font name. If the string size (including the length byte) is odd, an extra byte is added so that the resource size is even.

For more information about Edit, contact:

Consulair Corp.
140 Campo Drive
Portola Valley, CA 94025
(415) 851-3272

Macintosh Technical Notes

#85: GetNextEvent; Blinking Apple Menu

See also: The Menu Manager
The Toolbox Event Manager
The Desk Manager

Written by: Rick Blair August 14, 1986
Updated: March 1, 1988

Wherein arcane mysteries are unraveled so you can make the Alarm Clock (or a similar desk accessory) blink the Apple menu at the appointed second. Also, why `GetNextEvent` is a good thing.

The obvious

Don't disable interrupts within an application! There will almost certainly come a time (or Macintosh) where you won't be able to change the interrupt mask because the processor is running in user mode. The one-second interrupt is used to blink the apple.

The not-so-obvious

You must call `GetNextEvent` periodically. `GetNextEvent` uses a filter (GNE filter) which allows for a routine to be installed which overrides (or augments) the behavior of the system. The GNE filter is installed by pointing the low-memory global `jGNEFilter` (a long word at `$29A`) to the routine. After all other GNE processing is complete, the `routine` will be called with `A1` pointing to the event record and `D0` containing the boolean result. The filter may then modify the event record or change the function result by altering the word on the stack at `4(A7)`. This word will match `D0` initially, of course.

A GNE filter is used to do the blinking when the interrupt handler has announced that the moment is at hand. `GetOSEvent` won't do. If you don't have a standard main event loop, it is generally a good idea to give `GetNextEvent` (and `SystemTask`, too) a call whenever you have any idle time. `GetNextEvent` "extra" services include, but aren't limited to, the following:

1. Calling the GNE filter.
2. Removing lingering disk-switched windows (uncommon unless memory is tight).
3. Making Window Manager activate, deactivate and update events happen.
4. Getting various events from a journaling driver when one is playing.
5. Giving `SystemEvent` a chance at each event.
6. Running command-shift function key routines (e.g. command-shift-4 to print the screen to an ImageWriter).

The more subtle

When the (default) GNE filter sees that the interrupt handler has set the "time to blink" flag, it looks at the first menu in `MenuList`. The title of that menu must consist solely of the "apple" character or no blinking will occur. It really just looks at the first word of the string to see if it is `$0114`. This is a Pascal string which has only the `$14` "apple" character in it. So you musn't have any spaces or any other characters in the title of your first menu or you'll get no blinkin' results.

#86: MacPaint Document Format

Revised by: Jim Reekes June 1989
Written by: Bill Atkinson 1983

This Technical Note describes the internal format of a MacPaint® document, which is a standard used by many other programs. This description is the same as that found in the "Macintosh Miscellaneous" section of early *Inside Macintosh* versions.
Changes since October 1988: Fixed bugs in the example code.

MacPaint documents are easy to read and write, and they have become a standard interchange format for full–page images on the Macintosh. This Note describes the MacPaint internal document format to help developers generate and interpret files in this format.

MacPaint documents have a file type of "PNTG," and since they use only the data fork, you can ignore the resource fork. The data fork contains a 512–byte header followed by compressed data which represents a single bitmap (576 pixels wide by 720 pixels tall). At a resolution of 72 pixels per inch, this bitmap occupies the full 8 inch by 10 inch printable area of a standard ImageWriter printer page.

Header

The first 512 bytes of the document form a header of the following format:

- 4–byte version number (default = 2)
- 38*8 = 304 bytes of patterns
- 204 unused bytes (reserved for future expansion)

As a Pascal record, the document format could look like the following:

```
MPHeader = RECORD
        Version:        LONGINT;
        PatArray:       ARRAY [1..38] of Pattern;
        Future:         PACKED ARRAY [1..204] of SignedByte;
END;
```

If the version number is zero, the document uses default patterns, so you can ignore the rest of the header block, and if your program generates MacPaint documents, you can write 512 bytes of zero for the document header. Most programs which read MacPaint documents can skip the header when reading.

Bitmap

Following the header are 720 compressed scan lines of data which form the 576 pixel wide by 720 pixel tall bitmap. Without compression, this bitmap would occupy 51,840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10K using the `_PackBits` procedure to compress runs of equal bytes within each scan line. The bitmap part of a MacPaint document is simply the output of `_PackBits` called 720 times, with 72 bytes of input each time.

To determine the maximum size of a MacPaint file, it is worth noting what *Inside Macintosh* says about `_PackBits`:

> *"The worst case would be when `_PackBits` adds one byte to the row of bytes when packing."*

If we include an extra 512 bytes for the file header information to the size of an uncompressed bitmap (51,840), then the total number of bytes would be 52,352. If we take into account the extra 720 "potential" bytes (one for each row) to the previous total, the maximum size of a MacPaint file becomes 53,072 bytes.

Reading Sample

```
PROCEDURE ReadMPFile;
{ This is a small example procedure written in Pascal that demonstrates
  how to read MacPaint files. As a final step, it takes the data that
  was read and displays it on the screen to show that it worked.
  Caveat: This is not intended to be an example of good programming
  practice, in that the possible errors merely cause the program to exit.
  This is VERY uninformative, and there should be some sort of error handler
  to explain what happened. For simplicity, and thus clarity, those types
  of things were deliberately not included. This example will not work
  on a 128K Macintosh, since memory allocation is done too simplistically.
}

CONST
       DefaultVolume = 0;
       HeaderSize = 512;                   { size of MacPaint header in bytes }
       MaxUnPackedSize = 51840;            { maximum MacPaint size in bytes }
                                           { 720 lines * 72 bytes/line }

VAR
       srcPtr:         Ptr;
       dstPtr:         Ptr;
       saveDstPtr:     Ptr;
       lastDestPtr:    Ptr;
       srcFile:        INTEGER;
       srcSize:        LONGINT;
       errCode:        INTEGER;
       scanLine:       INTEGER;
       aPort:          GrafPort;
       theBitMap:      BitMap;

BEGIN
       errCode := FSOpen('MP TestFile', DefaultVolume, srcFile); { Open the file. }
       IF errCode <> noErr THEN ExitToShell;

       errcode := SetFPos(srcFile, fsFromStart, HeaderSize);     { Skip the header. }
       IF errCode <> noErr THEN ExitToShell;

       errCode := GetEOF(srcFile, srcSize);        { Find out how big the file is, }
       IF errCode <> noErr THEN ExitToShell;       { and figure out source size. }

       srcSize := srcSize - HeaderSize ;           { Remove the header from count. }
       srcPtr := NewPtr(srcSize);                  { Make buffer just the right size. }
       IF srcPtr = NIL THEN ExitToShell;

       errCode := FSRead(srcFile, srcSize, srcPtr); { Read the data into the buffer. }
       IF errCode <> noErr THEN ExitToShell;       { File marker is past header. }
```

```
        errCode := FSClose(srcFile);              { Close the file we just read. }
        IF errCode <> noErr THEN ExitToShell;

        { Create a buffer that will be used for the Destination BitMap. }
        dstPtr := NewPtrClear(MaxUnPackedSize);    {MPW library routine, see TN 219}
        IF dstPtr = NIL THEN ExitToShell;
        saveDstPtr := dstPtr;

        { Unpack each scan line into the buffer. Note that 720 scan lines are
          guaranteed to be in the file. (They may be blank lines.) In the
          UnPackBits call, the 72 is the count of bytes done when the file was
          created.  MacPaint does one scan line at a time when creating the file.
          The destination pointer is tested each time through the scan loop.
          UnPackBits should increment this pointer by 72, but in the case where
          the packed file is corrupted UnPackBits may end up sending bits into
          uncharted territory.  A temporary pointer "lastDstPtr" is used for testing
          the result.}

        FOR scanLine := 1 TO 720 DO BEGIN
             lastDstPtr := dstPtr;
             UnPackBits(srcPtr, dstPtr, 72);        { bumps both pointers }
             IF ORD4(lastDstPtr) + 72 <> ORD4(dstPtr) THEN ExitToShell;
        END;

        { The buffer has been fully unpacked. Create a port that we can draw into.
          You should save and restore the current port.  }
        OpenPort(@aPort);

        { Create a BitMap out of our saveDstPtr that can be copied to the screen. }
        theBitMap.baseAddr := saveDstPtr;
        theBitMap.rowBytes := 72;                 { width of MacPaint picture }
        SetPt(theBitMap.bounds.topLeft, 0, 0);
        SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

        { Now use that BitMap and draw the piece of it to the screen.
          Only draw the piece that is full screen size (portRect). }
        CopyBits(theBitMap, aPort.portBits, aPort.portRect,
                      aPort.portRect, srcCopy, NIL);

        { We need to dispose of the memory we've allocated.  You would not
          dispose of the destPtr if you wish to edit the data.  }
        DisposPtr(srcPtr);                  { dispose of the source buffer }
        DisposPtr(dstPtr);                  { dispose of the destination buffer }
END;
```

Writing Sample

```
PROCEDURE WriteMPFile;
{ This is a small example procedure written in Pascal that demonstrates how
  to write MacPaint files. It will use the screen as a handy BitMap to be
  written to a file.
}

CONST
       DefaultVolume = 0;
       HeaderSize = 512;                     { size of MacPaint header in bytes }
       MaxFileSize = 53072;                  { maximum MacPaint file size. }

VAR
       srcPtr:        Ptr;
       dstPtr:        Ptr;
       dstFile:       INTEGER;
       dstSize:       LONGINT;
       errCode:       INTEGER;
       scanLine:      INTEGER;
       aPort:         GrafPort;
       dstBuffer:     PACKED ARRAY[1..HeaderSize] OF BYTE;
       I:             LONGINT;
       picturePtr:    Ptr;
       tempPtr:       BigPtr;
       theBitMap:     BitMap;

BEGIN
       { Make an empty buffer that is the picture size. }
       picturePtr := NewPtrClear(MaxFileSize);    {MPW library routine, see TN 219}
       IF picturePtr = NIL THEN ExitToShell;

       { Open a port so we can get to the screen's BitMap easily.  You should save
         and restore the current port. }
       OpenPort(@aPort);

       { Create a BitMap out of our dstPtr that can be copied to the screen. }
       theBitMap.baseAddr := picturePtr;
       theBitMap.rowBytes := 72;                  { width of MacPaint picture }
       SetPt(theBitMap.bounds.topLeft, 0, 0);
       SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

       { Draw the screen over into our picture buffer. }
       CopyBits(aPort.portBits, theBitMap, aPort.portRect,
                     aPort.portRect, srcCopy, NIL);

       { Create the file, giving it the right Creator and File type.}
       errCode := Create('MP TestFile', DefaultVolume, 'MPNT', 'PNTG');
       IF errCode <> noErr THEN ExitToShell;

       { Open the data file to be written. }
       errCode := FSOpen(dstFileName, DefaultVolume, dstFile);
       IF errCode <> noErr THEN ExitToShell;

       FOR I := 1 to HeaderSize DO                { Write the header as all zeros. }
            dstBuffer[I] := 0;
       errCode := FSWrite(dstFile, HeaderSize, @dstBuffer);
       IF errCode <> noErr THEN ExitToShell;
```

```
        { Now go into a loop where we pack each line of data into the buffer,
          then write that data to the file. We are using the line count of 72
          in order to make the file readable by MacPaint. Note that the
          Pack/UnPackBits can be used for other purposes. }
        srcPtr := theBitMap.baseAddr;            { point at our picture BitMap }
        FOR scanLine := 1 to 720 DO
             BEGIN
                dstPtr := @dstBuffer;            { reset the pointer to bottom }
                PackBits(srcPtr, dstPtr, 72);            { bumps both ptrs }
                dstSize := ORD(dstPtr)-ORD(@dstBuffer);    { calc packed size }
                errCode := FSWrite(dstFile, dstSize, @dstBuffer);
                IF errCode <> noErr THEN ExitToShell;
             END;

        errCode := FSClose(dstFile);             { Close the file we just wrote. }
        IF errCode <> noErr THEN ExitToShell;
END;
```

Further Reference:

- *Inside Macintosh*, Volume I-135, QuickDraw
- *Inside Macintosh*, Volume I-465, Toolbox Utilities
- *Inside Macintosh*, Volume II-77, The File Manager
- Technical Note #219, New Memory Manager Glue Routines

MacPaint is a registered trademark of Claris Corporation.

#87: Error in FCBPBRec

See also: The File Manager

Written by: Jim Friedlander August 18, 1986
Updated: March 1, 1988

The declaration of a FCBPBRec is wrong in *Inside Macintosh Volume IV* and early versions of MPW. This has been fixed in MPW 1.0 and newer.

An error was made in the declaration of an FCBPBRec parameter block that is used in PBGetFCBInfo calls. The field ioFCBIndx was incorrectly listed as a LONGINT. The following declaration (found in *Inside Macintosh*):

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      LONGINT;
ioFCBFlNm:      LONGINT;
...
```

should be changed to:

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      INTEGER;
ioFCBFiller1:   INTEGER;
ioFCBFlNm:      LONGINT;
...
```

Macintosh Technical Notes

#88: Signals

See also: Using Assembly Language (Mixing Pascal & Assembly)

Written by: Rick Blair August 1, 1986
Updated: March 1, 1988

Signals are a form of intra-program interrupt which can greatly aid clean, inexpensive error trapping in stack frame intensive languages. A program may invoke the `Signal` procedure and immediately return to the last invocation of `CatchSignal`, including the complete stack frame state at that point.

Signals allow a program to leave off execution at one point and return control to a convenient error trap location, regardless of how many levels of procedure nesting are in between.

The example is provided with a Pascal interface, but it is easily adapted to other languages. The only qualification is that the language **must** bracket its procedures (or functions) with `LINK` and `UNLK` instructions. This will allow the signal code to clean up at procedure exit time by removing `CatchSignal` entries from its internal queue. **Note:** only procedures and/or functions that call `CatchSignal` need to be bracketed with `LINK` and `UNLK` instructions.

Important: `InitSignals` must be called from the main program so that `A6` can be set up properly.

Note that there is no limit to the number of local `CatchSignals` which may occur within a single routine. Only the last one executed will apply, of course, unless you call `FreeSignal`. `FreeSignal` will "pop" off the last `CatchSignal`. If you attempt to `Signal` with no `CatchSignals` pending, `Signal` will halt the program with a debugger trap.

`InitSignals` creates a small relocatable block in the application heap to hold the signal queue. If `CatchSignal` is unable to expand this block (which it does 5 elements at a time), then it will signal back to the last successful `CatchSignal` with `code` = 200. A `Signal(0)` acts as a `NOP`, so you may pass `OSErrs`, for instance, after making File System type calls, and, if the `OSErr` is equal to `NoErr`, nothing will happen.

CatchSignal may not be used in an expression if the stack is used to evaluate that expression. For example, you can't write:

```
c:= 3*CatchSignal;
```

"Gotcha" summary

1. Routines which call CatchSignal must have stack frames.
2. InitSignals must be called from the outermost (main) level.
3. Don't put the CatchSignal function in an expression. Assign the result to an INTEGER variable; i.e. i:=CatchSignal.
4. It's safest to call a procedure to do the processing after CatchSignal returns. See the Pascal example TestSignals below. This will prevent the use of a variable which may be held in a register.

Below are three separate source files. First is the Pascal interface to the signaling unit, then the assembly language which implements it in MPW Assembler format. Finally, there is an example program which demonstrates the use of the routines in the unit.

```
{File ErrSignal.p}
UNIT ErrSignal;

INTERFACE

{Call this right after your other initializations (InitGraf, etc.)--in other
words as early as you can in the application}
PROCEDURE InitSignals;

{Until the procedure which encloses this call returns, it will catch
subsequent Signal calls, returning the code passed to Signal. When
CatchSignal is encountered initially, it returns a code of zero. These calls
may "nest"; i.e. you may have multiple CatchSignals in one procedure.
Each nested CatchSignal call uses 12 bytes of heap space }
FUNCTION CatchSignal:INTEGER;

{This undoes the effect of the last CatchSignal. A Signal will then invoke
the CatchSignal prior to the last one.}
PROCEDURE FreeSignal;

{Returns control to the point of the last CatchSignal. The program will then
behave as though that CatchSignal had returned with the code parameter
supplied to Signal.}
PROCEDURE Signal(code:INTEGER);

END.
{End of ErrSignal.p}
```

Here's the assembly source for the routines themselves:

```
; ErrSignal code w. InitSignal, CatchSignal,FreeSignal, Signal
; defined
;
;                Version 1.0 by Rick Blair

           PRINT     OFF
           INCLUDE   'Traps.a'
           INCLUDE   'ToolEqu.a'
           INCLUDE   'QuickEqu.a'
           INCLUDE   'SysEqu.a'
           PRINT     ON


CatchSigErr EQU      200              ;"insufficient heap" message
SigChunks   EQU      5                ;number of elements to expand by
FrameRet    EQU      4                ;return addr. for frame (off A6)
SigBigA6    EQU      $FFFFFFFF        ;maximum positive A6 value


; A template in MPW Assembler describes the layout of a collection of data
; without actually allocating any memory space. A template definition starts ;
with a RECORD directive and ends with an ENDR directive.

; To illustrate how the template type feature works, the following template
; is declared and used. By using this, the assembler source appromixates very
; closely Pascal source for referencing the corresponding information.

;template for our table elements
  SigElement  RECORD    0     ;the zero is the template origin
  SigSP       DS.L      1     ;the SP at the CatchSignal-(DS.L just like EQU)
  SigRetAddr  DS.L      1     ;the address where the CatchSignal returned
  SigFRet     DS.L      1     ;return addr. for encl. procedure
  SigElSize   EQU       *     ;just like EQU 12
              ENDR


; The global data used by these routines follows. It is in the form of a
; RECORD, but, unlike above, no origin is specified, which means that memory
; space *will* be allocated.
; This data is referenced through a WITH statement at the beginning of the
; procs that need to get at this data. Since the Assembler knows when it is
; referencing data in a data module (since they must be declared before they
; are accessed), and since such data can only be accessed based on A5, there
; is no need to explicitly specify A5 in any code which references the data
; (unless indexing is used). Thus, in this program we have omitted all A5
; references when referencing the data.

  SigGlobals RECORD                   ;no origin means this is a data record
                                      ;not a template(as above)
  SigEnd     DS.L      1              ;current end of table
  SigNow     DS.L      1              ;the MRU element
  SigHandle  DC.L      0              ;handle to the table
             ENDR
```

```
InitSignals PROC     EXPORT          ;PROCEDURE InitSignals;

           IMPORT    CatchSignal
           WITH      SigElement,SigGlobals

;the above statement makes the template SigElement and the global data
;record SigGlobals available to this procedure

           MOVE.L    #SigChunks*SigElSize,D0
           _NewHandle                ;try to get a table
           BNE.S     forgetit        ;we couldn't get that!?

           MOVE.L    A0,SigHandle    ;save it
           MOVE.L    #-SigElSize,SigNow ;point "now" before start
           MOVE.L    #SigChunks*SigElSize,SigEnd ;save the end
           MOVE.L    #SigBigA6,A6    ;make A6 valid for Signal
forgetit   RTS
           ENDP

CatchSignal PROC     EXPORT          ;FUNCTION CatchSignal:INTEGER;
           IMPORT    SiggySetup,Signal,SigDeath
           WITH      SigElement,SigGlobals

           MOVE.L    (SP)+,A1        ;grab return address
           MOVE.L    SigHandle,D0    ;handle to table
           BEQ       SigDeath        ;if NIL then croak
           MOVE.L    D0,A0           ;put handle in A-register
           MOVE.L    SigNow,D0
           ADD.L     #SigElSize,D0
           MOVE.L    D0,SigNow       ;save new position
           CMP.L     SigEnd,D0       ;have we reached the end?
           BNE.S     catchit         ;no, proceed

           ADD.L     #SigChunks*SigElSize,D0 ;we'll try to expand
           MOVE.L    D0,SigEnd       ;save new (potential) end
           _SetHandleSize
           BEQ.S     @0              ;jump around if it worked!

;signals, we use 'em ourselves
           MOVE.L    SigNow,SigEnd   ;restore old ending offset
           MOVE.L    #SigElSize,D0
           SUB.L     D0,SigNow       ;ditto for current position
           MOVE.W    #catchSigErr,(SP);we'll signal a "couldn't
                                      ;                catch" error
           JSR       Signal          ;never returns of course


@0         MOVE.L    SigNow,D0

catchit    MOVE.L    (A0),A0         ;deref.
           ADD.L     D0,A0           ;point to new entry
           MOVE.L    SP,SigSP(A0)    ;save SP in entry
           MOVE.L    A1,SigRetAddr(A0) ;save return address there
           CMP.L     #SigBigA6,A6    ;are we at the outer level?
           BEQ.S     @0              ;yes, no frame or cleanup needed
           MOVE.L    FrameRet(A6),SigFRet(A0);save old frame return
                                             ;               address
```

```
            LEA         SiggyPop,A0
            MOVE.L      A0,FrameRet(A6) ;set cleanup code address
@0          CLR.W       (SP)            ;no error code (before its time)
            JMP         (A1)            ;done setting the trap

SiggyPop    JSR         SiggySetup      ;get pointer to element
            MOVE.L      SigFRet(A0),A0 ;get proc's real return address
            SUB.L       #SigElSize,D0
            MOVE.L      D0,SigNow       ;"pop" the entry
            JMP         (A0)            ;gone
            ENDP

FreeSignal  PROC        EXPORT          ;PROCEDURE FreeSignal;
            IMPORT      SiggySetup
            WITH        SigElement,SigGlobals
            JSR         SiggySetup      ;get pointer to current entry
            MOVE.L      SigFRet(A0),FrameRet(A6) ;"pop" cleanup code
            SUB.L       #SigElSize,D0
            MOVE.L      D0,SigNow       ;"pop" the entry
            RTS
            ENDP

Signal      PROC        EXPORT          ;PROCEDURE Signal(code:INTEGER);
            EXPORT      SiggySetup,SigDeath
            WITH        SigElement,SigGlobals
            MOVE.W      4(SP),D1        ;get code
            BNE.S       @0              ;process the signal if code is non-zero
            MOVE.L      (SP),A0         ;save return address
            ADDQ.L      #6,SP           ;adjust stack pointer
            JMP         (A0)            ;return to caller(code was 0)

@0          JSR         SiggySetup      ;get pointer to entry
            BRA.S       SigLoop1

SigLoop     UNLK        A6              ;unlink stack by one frame
SigLoop1    CMP.L       SigSP(A0),A6    ;is A6 beyond the saved stack?
            BLO.S       SigLoop         ;yes, keep unlinking
            MOVE.L      SigSP(A0),SP    ;bring back our SP
            MOVE.L      SigRetAddr(A0),A0 ;get return address
            MOVE.W      D1,(SP)         ;return code to CatchSignal
            JMP         (A0)            ;Houston, boost the Signal!
                        ;(or Hooston if you're from the Negative Zone)

SiggySetup  MOVE.L      SigHandle,A0
            MOVE.L      (A0),A0         ;deref.
            MOVE.L      A0,D0           ;to set CCR
            BEQ.S       SigDeath        ;nil handle means trouble
            MOVE.L      SigNow,D0       ;grab table offset to entry
            BMI.S       SigDeath        ;if no entries then give up
            ADD.L       D0,A0           ;point to current element
            RTS

SigDeath    _Debugger                   ;a signal sans catch is bad news

            ENDP
            END
```

Now for the example Pascal program:

```
PROGRAM TestSignals;
USES ErrSignal;

VAR i:INTEGER;

PROCEDURE DoCatch(s:STR255; code:INTEGER);
BEGIN
  IF code<>0 THEN BEGIN
    Writeln(s,code);
    Exit(TestSignals);
  END;
END; {DoCatch}

PROCEDURE Easy;
  PROCEDURE Never;
    PROCEDURE DoCatch(s:STR255; code:INTEGER);
    BEGIN
      IF code<>0 THEN BEGIN
        Writeln(s,code);
        Exit(Never);
      END;
    END; {DoCatch}

  BEGIN {Never}
  i:=CatchSignal;
  DoCatch('Signal caught from Never, code = ', i );

  i:=CatchSignal;
  IF i<>0 THEN DoCatch('Should never get here!',i);

  FreeSignal; {"free" the last CatchSignal}
  Signal(7); {Signal a 7 to the last CatchSignal}
  END;{Never}
BEGIN {Easy}
Never;
Signal(69);       {this won't be caught in Never}
END;{Easy}        {all local CatchSignals are freed when a procedure exits.}

BEGIN {PROGRAM}
InitSignals; {You must call this early on!}

{catch Signals not otherwise caught by the program}
i:=CatchSignal;
IF i<>0 THEN
 DoCatch('Signal caught from main, code = ',i);

Easy;
END.
```

The example program produces the following two lines of output:

```
Signal caught from Never, code = 7
Signal caught from main, code = 69
```

#89: DrawPicture Bug

Written by: Ginger Jernigan August 16, 1986
Updated: March 1, 1988

Earlier versions of this note described a bug in `DrawPicture`. This bug never occurred on 64K ROM machines, and has been fixed in System 3.2 and newer. Use of Systems older than 3.2 on non-64K ROM machines is no longer recommended.

#90: SANE Incompatibilities

Written by:	Mark Baumwell	August 14, 1986
Updated:		March 1, 1988

Earlier versions of this note described a problem with SANE and System 2.0. Use of System 2.0 is only recommended for Macintosh 128 machines, which contain the 64K ROMs. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#91: Optimizing for the LaserWriter—Picture Comments

See also: The Print Manager
QuickDraw
Technical Note #72—
Optimizing for the LaserWriter—Techniques
Technical Note #27—MacDraw Picture Comments
PostScript Language Reference Manual, Adobe Systems
PostScript Language Tutorial and Cookbook,
Adobe Systems
LaserWriter Reference Manual

Written by: Ginger Jernigan November 15, 1986
Modified by: Ginger Jernigan March 2, 1987
Updated: March 1, 1988

This technical note is a continuation of Technical Note #72. This technical note discusses the picture comments that the LaserWriter driver recognizes.

This technical note has been modified to include corrected descriptions of the `SetLineWidth`, `PostScriptFile` and `ResourcePS` comments and to include some additional warnings.

The implementation of QuickDraw's `picComment` facility by the LaserWriter driver allows you to take advantage of features (like rotated text) which are available in PostScript but may not be available in QuickDraw.

Warning: Using PostScript-specific comments will make your code printer-dependent and may cause compatibility problems with non-PostScript devices, so don't use them unless you absolutely have to.

Some of the picture comments below are designed to be issued along with QuickDraw commands that simulate the commented commands on the Macintosh screen. When the comments are used, the accompanying QuickDraw comments are ignored. If you are designing a picture to be printed by the LaserWriter, the structure and use of these comments must be precise, otherwise nothing will print. If another printer driver (like the ImageWriter I/II driver) has not implemented these comments, the comments are ignored and the accompanying QuickDraw commands are used.

Below are the picture comments that the LaserWriter driver recognizes:

Type	Kind	Data Size	Data	Description
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NIL	End text function
StringBegin	152	0	NIL	Begin pieces of original string
StringEnd	153	0	NIL	End pieces of original string
TextCenter	154	8	TTxtCenter	Offset to center of rotation
* LineLayoutOff	155	0	NIL	Turns LaserWriter line layout off
* LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following poly data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
picPlyClo	165	0	NIL	Close the poly
* DashedLine	180	-	TDashedLine	Draw following lines as dashed
* DashedStop	181	0	NIL	End dashed lines
* SetLineWidth	182	4	Point	Set fractional line widths
* PostScriptBegin	190	0	NIL	Set driver state to PostScript
* PostScriptEnd	191	0	NIL	Restore QuickDraw state
* PostScriptHandle	192	-	PSData	PostScript data in handle
*† PostScriptFile	193	-	FileName	FileName in data handle
* TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
*† ResourcePS	195	8	Type/ID/Index	PostScript data in a resource file
**RotateBegin	200	4	TRotation	Begin rotated port
**RotateEnd	201	0	NIL	End rotation
**RotateCenter	202	8	Center	Offset to center of rotation
**FormsPrinting	210	0	NIL	Don't clear print buffer after each page
**EndFormsPrinting	211	0	NIL	End forms printing after PrClosePage

* These comments are only implemented in LaserWriter driver 3.0 or later.
** These comments are only implemented in LaserWriter driver 3.1 or later.
† These comments are not available when background printing is enabled.

Each of these comments are discussed below in six groups: Text, Polygons, Lines, PostScript, Rotation, and Forms. Code examples are given where appropriate. For other examples of how to use picture comments for printing please see the Print example program in the Software Supplement (currently available through APDA as "Macintosh Example Applications and Sources 1.0").

Note: The examples used in the *LaserWriter Reference Manual* are incorrect. Please use the examples presented here instead.

Text

In order to support the What-You-See-Is-What-You-Get paradigm, the LaserWriter driver uses a line layout algorithm to assure that the placement of the line on the printer closely approximates the placement of the line on the screen. This means that the printer driver gets the width of the line from QuickDraw, then tells PostScript to place the text in exactly the same place with the same width.

The `TextBegin` comment allows the application to specify the layout and the orientation of the text that follows it by specifying the following information:

```
TTxtPicRec = PACKED RECORD
   tJus:  Byte;     {0,1,2,3,4 or greater => none, left, center, right, full
                    justification }
   tFlip: Byte;     {0,1,2 => none, horizontal, vertical coordinate flip }
   tRot:  INTEGER;  {0..360 => clockwise rotation in degrees }
   tLine: Byte;     {1,2,3.. => single, 1-1/2, double.. spacing }
   tCmnt: Byte;     {Reserved }
END; { TTxtPicRec }
```

Left, right or center justification, specified by `tJust`, tells the driver to maintain only the left, right or center point, without recalculating the interword spacing. Full justification specifies that both endpoints be maintained and interword spacing be recalculated. This means that the driver makes sure that the specified points are maintained on the printer without caring whether the overall width has changed. Full justification means that the overall width of the line has been maintained. `tFlip` and `tRot` specify the orientation of the text, allowing the application to take advantage of the rotation features of PostScript. `tLine` specifies the interline spacing. When no `TextBegin` comment is used, the defaults are full justification, no rotation and single-spaced lines.

String Reconstruction

The `StringBegin` and `StringEnd` comments are used to bracket short strings of text that are actually sections of an original long string. MacDraw, for instance, breaks long strings into shorter pieces to avoid stack overflow problems with QuickDraw in the 64K ROM. When these smaller strings are bracketed by `StringBegin` and `StringEnd`, the LaserWriter driver assumes that the enclosed strings are parts of one long string and will perform its line layout accordingly. Erasing or filling of background rectangles should take place before the `StringBegin` comment to avoid confusing the process of putting the smaller strings back together.

Text Rotation

In order to rotate a text object, PostScript needs to have information concerning the center of rotation. The `TextCenter` comment provides this information when a rotation is specified in the `TextBegin` comment. This comment contains the offset from the present pen location to the center of rotation. The offset is given as the y-component, then the x-component, which are declared as fixed-point numbers. This allows the center to be in the middle of a pixel. This comment should appear after the `TextBegin` comment and before the first following `StringBegin` comment.

The associated comment data looks like this:

```
TTxtCenter = RECORD
   y,x: Fixed;      {offset from current pen location to center of rotation}
END; { TTxtCenter }
```

Right after a `TextBegin` comment, the LaserWriter driver expects to see a `TextCenter` comment specifying the center of rotation for any text enclosed within the text comment calls. It will ignore all further `CopyBits` calls, and print all standard text calls in the rotation specified by the information in `TTxtPicRec`. The center of rotation is the offset from the beginning position of the first string following the `TextCenter` comment. The printer driver also expects the string locations to be in the coordinate system of the current QuickDraw port. The printer driver rotates the entire port to draw the text so it can draw several strings with one rotation comment and one center comment. It is good practice to enclose an entire paragraph or paragraphs of text in a single rotation comment so that the driver makes the fewest number of rotations.

The printer driver can draw non-textual objects within the bounds of the text rotation comments but it must unrotate to draw the object, then re-rotate to draw the next string of text. To do this the printer driver must receive another `TextCenter` comment before each new rotation. So, rotated text and unrotated objects can be drawn inter-mixed within one `TextBegin/TextEnd` comment pair, but performance is slowed.

Note that all bit maps and all clip regions are ignored during text rotation so that clip regions can be used to clip out the strings on printers that can't take advantage of these comments. This has the unfortunate side effect of not allowing rotated text to be clipped.

Rotated text comments are not associated with landscape and portrait orientation of the printer paper as selected by the Page Setup dialog. These are rotations with reference to the current QuickDraw port only.

All of the above text comments are terminated by a `TextEnd` comment.

Turning Off Line Layout

If your application is using its own line layout algorithm (it uses its own character widths or does its own character or word placement), the printer driver doesn't need to do it too. To turn off line layout, you can use the `LineLayoutOff` comment. `LineLayoutOn` turns it on again.

Turning on `FractEnable` for the 128K ROMs has the same effect as `LineLayoutOff`. When the driver detects that `FractEnable` has been turned on, line layout is not performed. The driver assumes that all text being printed is already spaced correctly for the LaserWriter and just sends it as is.

Polygons

The polygon comments are recognized by the LaserWriter driver because they are used by MacDraw as an alternate method of defining polygons.

The PolyBegin and PolyEnd comments bracket polygon line segments, giving an alternate way to specify a polygon. All StdLine calls between these two comments are part of the polygon. The endpoints of the lines are the vertices of the polygon.

The picPlyClo comment specifies that the current polygon should be closed. This comes immediately after PolyBegin, if at all. It is not sufficient to simply check for begPt = endPt, since MacDraw allows you to create a “closed” polygon that isn’t really closed. This comment is especially critical for smooth curves because it can make the difference between having a sharp corner or not in the curve.

These comments also work with the StdPoly call. If a FillRgn is encountered before the PolyEnd comment, then the polygon is filled. Unlike QuickDraw polygons, comment polygons do not require an initial MoveTo call within the scope of the polygon comment. The polygon will be drawn using the current pen location at the time the polygon comment is received. The pen must be set before the polygon comment is called.

Splines

A spline is a method used to determine the smallest number of points that define a curve. In MacDraw, splines are used as a method for smoothing polygons. The vertices of the underlying unsmoothed polygon are the control nodes for the quadratic B-spline curve which is drawn. PostScript has a direct facility for cubic B-splines and the LaserWriter translates the quadratic B-spline nodes it gets into the appropriate nodes for a cubic B-spline that will exactly emulate the original quadratic B-spline.

The PolySmooth comment specifies that the current polygon should be smoothed. This comment also contains data that provides a means of specifying which verbs to use on the smoothed polygon (bits 7 through 3 are not currently assigned):

```
TPolyVerb = PACKED RECORD
   f7, f6, f5, f4, f3, fPolyClose, fPolyFill, fPolyframe : Boolean;
END; { TPolyVerb }
```

Although the closing information is redundant with the picPlyClo comment, it is included for the convenience of the LaserWriter.

The LaserWriter uses the pen size at the time the PolyBegin comment is received to frame the smoothed polygon if framing is called for by the TPolyVerb information. When the PolyIgnore comment is received by the LaserWriter driver, all further StdLine calls are ignored until the PolyEnd comment is encountered. For polygons that are to be smoothed, set the initial pen width to zero after the PolyBegin comment so that the unsmoothed polygon will not be drawn by other printers not equipped to handle polygon comments. To fill the polygon, call StdRgn with the fill verb and the appropriate pattern set, as well as specifying fill in the PolySmooth comment.

Lines

The DashedLine and DashedLineStop comments are used to communicate PostScript information for drawing dashed lines.

The DashedLine comment contains the following additional data:

```
TDashedLine = PACKED RECORD
  offset:   SignedByte;                   {Offset as specified by PostScript}
  centered: SignedByte;                   {Whether dashed line should be
                                           centered to begin and end points}
  dashed:   Array[0..1] of SignedByte;    {1st byte is # bytes following}
END; { TDashedLine }
```

The printer driver sets up the PostScript dashed line command, as defined on page 214 of Adobe's *PostScript Language Reference Manual*, using the parameters specified in the comment. You can specify that the dashed line be centered between the begin and end points of the lines by making the centered field nonzero.

The SetLineWidth comment allows you to set the pen width of all subsequent objects drawn. The additional data is a point. The vertical portion of the point is the numerator and the horizontal portion is the denominator of the scaling factor that the horizontal and vertical components of the pen are then multiplied by to obtain the new pen width. For example, if you have a pen size of 1,2 and in your line width comment you use 2 for the horizontal of the point and 7 for the vertical, the pen size will then be (7/2)*1 pixels wide and (7/2)*2 pixels high.

Below is an example of how to use the line comments:

```
PROCEDURE LineTest;
{This procedure shows how to do dashed lines and how to change the line width}
CONST
  DashedLine = 180;
  DashedStop = 181;
  SetLineWidth = 182;

TYPE
  DashedHdl = ^DashedPtr;
  DashedPtr = ^TDashedLine;
  TDashedLine = PACKED RECORD
    offset: SignedByte;
    Centered: SignedByte;
    dashed: Array[0..1] of SignedByte;   { the 0th element is the length }
  END; { TDashedLine }
  widhdl = ^widptr;
  widptr = ^widpt;
  widpt = Point;


VAR
  arect    : rect;
  Width    : Widhdl;
  dashedln : DashedHdl;
```

```
 BEGIN {LineTest}
   Dashedln := dashedhdl(NewHandle(sizeof(tdashedline)));
   Dashedln^^.offset := 0;        { No offset}
   Dashedln^^.centered := 0;      { don't center}
   Dashedln^^.dashed[0] := 1;     { this is the length }
   Dashedln^^.dashed[1] := 8;     { this means 8 points on, 8 points off }

   Width := widhdl(NewHandle(sizeof(widpt)));
   Width^^.h := 2;                { denominator is 2}
   Width^^.v := 7;                { numerator is 7}

   myPic := OpenPicture(theWorld);
     SetPen(1,2);                 { Set the pen size to 1 wide x 2 high }
     ClipRect(theWorld);
     MoveTo(20,20);
     DrawString('Do line test');
     PicComment(DashedLine,GetHandleSize(Handle(dashedln)),Handle(dashedln));
     PicComment(SetLineWidth,4,Handle(width));   {SetLineWidth}
     SetRect(arect,100,100,500,500);
     FrameRect(aRect);
     MoveTo(500,500);
     Lineto(100,100);
     PicComment(DashedStop,0,nil);              {DashedStop}
  ClosePicture;
  DisposHandle(handle(width));                  {Clean up}
  DisposHandle(handle(dashedln));
  PrintThePicture;                              {print it please}
  KillPicture(MyPic);
END; {LineTest}
```

PostScript

The PostScript comments tell the printer driver that the application is going to be communicating with the LaserWriter directly using PostScript commands instead of QuickDraw. The driver sends the accompanying PostScript to the printer with no preprocessing and no error checking. The application can specify data in the comment handle itself or point to another file which contains text to send to the printer. When the application is finished sending PostScript, the `PostScriptEnd` comment tells the printer driver to resume normal QuickDraw mode.

Any Quickdraw drawing commands made by the application between the `PostScriptBegin` and `PostScriptEnd` comments will be ignored by PostScript printers. In order to use PostScript in a device independent way, you should always include two representations of your document. The first representation should be a series of Quickdraw drawing commands. The second representation of your document should be a series of PostScript commands, sent to the Printing Manager via picture comments. This way, when you are printing to a PostScript device, the picture comments will be executed, and the Quickdraw commands ignored. When printing to a non-PostScript device, the picture comments will be ignored, and the Quickdraw commands will be executed. This method allows you to use PostScript, without having to ask the device if it supports it. This allows your application to get the best results with any printer, without being device dependent.

Here are some guidelines you need to remember:

- The graphic state set up during QuickDraw calls is maintained and is not affected by PostScript calls made with these comments.

- The header has changed a number of parameters so sometimes you won't get the results you expect. You may want to take a look at the header listed in *The LaserWriter Reference Manual* available through APDA.

- The header changes the PostScript coordinate system so that the origin is at the top-left corner of the page instead of at the bottom-left corner. This is done so that the QuickDraw coordinates that are used don't have to be remapped into the standard PostScript coordinate system. If you don't allow for this, all drawing is printed upside down. Please see the *PostScript Language Reference Manual* for details about transformation matrices.

- Don't call `showpage`. This is done for you by the driver. If you do, you won't be able to switch back to QuickDraw mode and an additional page will be printed when you call `PrClosePage`.
- Don't call `exitserver`. You may get very strange results.
- Don't call `initgraphics`. Graphics states are already set up by the header.

- Don't do anything that you expect to live across jobs.

- You won't be able to interrogate the printer to get information back through the driver.

The PostScriptBegin comment sets the driver state to prepare for the generation of PostScript by the application by calling gsave to save the current state. PostScript is then sent to the printer by using comments 192 through 195. The QuickDraw state of the driver is then restored by the PostScriptEnd comment. All QuickDraw operations that occur outside of these comments are performed; no clipping occurs as with the text rotation comments.

PostScript From a Text Handle

When the PostScriptHandle comment is used, the handle PSData points to the PostScript commands which are sent. PSData is a generic handle that points to text, without a length byte. The text is terminated by a carriage return. This comment is terminated by a PostScriptEnd comment.

Note: Due to a bug in the 3.1 LaserWriter driver, PostScriptEnd will not restore the QuickDraw state after the use of a PostScriptHandle comment. The workaround is to only use this comment at the end of your drawing, after you have made all the QuickDraw calls you need. This problem is fixed in more recent versions of the driver.

Here's an example of how to use this comment:

```
PROCEDURE PostHdl;
{this procedure shows how to use PostScript from a text Handle}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  PostScriptHandle = 192;

VAR
  MyString  : Str255;
  tempstr   : String[1];
  MyHandle  : Handle;
  err       : OSErr;

BEGIN { PostHdl }
  MyString := '/Times-Roman findfont 12 scalefont setfont 230 600 moveto
              (Hello World) show';
  tempstr:=' ';
  tempstr[1] := chr(13); {has to be terminated by a carriage return }
  MyString := Concat(MyString, tempstr); { in order for it to execute}
  err := PtrToHand (Pointer(ord(@myString)+1), MyHandle, length(MyString));
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('PostScript from a Handle');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(PostScriptHandle,length(mystring),MyHandle);
    PicComment(PostScriptEnd,0,nil);        {PostScript End}
  ClosePicture;
  DisposHandle(MyHandle);                   {Clean up}
  PrintThePicture;                          {print it please}
  KillPicture(MyPic);
END; { PostHdl }
```

Defining PostScript as QuickDraw Text

All QuickDraw text following the `TextIsPostScript` comment is sent as PostScript. No error checking is performed. This comment is terminated by a `PostScriptEnd` comment.

Here is an example:

```
PROCEDURE PostText;
{Shows how to use PostScript in strings in a QuickDraw picture}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  TextIsPostScript = 194;

BEGIN { PostTest }
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('TextIsPostScript Comment');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(TextIsPostScript,0,nil);     {following text is PostScript}
      DrawString('0 728 translate');        {move the origin and rotate the}
      DrawString('1 -1 scale');             {coordinate system}

      DrawString('newpath');
      DrawString('100 470 moveto');
      DrawString('500 470 lineto');
      DrawString('100 330 moveto');
      DrawString('500 330 lineto');
      DrawString('230 600 moveto');
      DrawString('230 200 lineto');
      DrawString('370 600 moveto');
      DrawString('370 200 lineto');
      DrawString('10 setlinewidth');
      DrawString('stroke');
      DrawString('/Times-Roman findfont 12 scalefont setfont');
      DrawString('230 600 moveto');
      DrawString('(Hello World) show');
    PicComment(PostScriptEnd,0,nil);        {PostScriptEnd}
  ClosePicture;
  PrintThePicture;                                     {print it please}
  KillPicture(MyPic);
END; { PostText }
```

PostScript From a File

The PostScriptFile and ResourcePS comments allow you to send PostScript to the printer from a resource file. Before these comments are described there are some restrictions you need to follow:

- Don't ever copy a picture containing these comments to the clipboard. If it is pasted into another application and the specified file or resource is not available, printing will be aborted and the user won't know what went wrong. This could be very confusing to a user. If you want the PostScript information to be available when printed from another application, use one of the other comments and include the information in the picture.
- Don't keep the PostScript in a separate file from the actual data file. If the data file ever gets moved without the PostScript file, when the picture is printed the data file may not be found and the print job will be aborted, again without the user knowing what went wrong. Keeping the data and PostScript in the same file will forestall many headaches for you and the user.

Now, a description of the comments:

The PostScriptFile comment tells the driver to use the POST type resources contained in the file FileNameString. FileNameString is declared as a Str255.

When this comment is encountered, the driver calls OpenResFile using the file name specified in FileNameString. It then calls GetResource('POST',theID); repeatedly, where theID begins at 501 and is incremented by one for each GetResource call. If the driver gets a ResNotFound error, it closes the specified resource file. If the first byte of the resource is a 3, 4, or 5 then the remaining data is sent and the file is closed.

The format of the POST resource is as follows: The IDs of the resources start at 501 and are incremented by one for each resource. Each resource begins with a 2 byte data field containing the data type in the first byte and a zero in the second. The possible values for the first byte are:

0 ignore the rest of this resource (a comment)
1 data is ASCII text
2 data is binary and is first converted to ASCII before being sent
3 AppleTalk end of file. The rest of the data, if there is any, is interpreted as ASCII text and will be sent after the EOF.
4 open the data fork of the current resource file and send the ASCII text there
5 end of the resource file

The second byte of the field must always be zero. Resources should be kept small, around 2K. Text and binary should not be mixed in the same resource. Make sure you include either a space or a return at the end of each PostScript string to separate it from the following command.

Here's an example:

```
PROCEDURE PostFile;
{This procedure shows how to use PostScript from a specified FILE}
CONST
  PostScriptBegin = 190;
  PostScriptFile = 193;
  PostScriptEnd = 191;

VAR
  MyString          : Str255;
  MyHandle          : Handle;
  err               : OSErr;

BEGIN   { PostFile }
  {You should never do this in a real program. This is only a test.}
  MyString := 'HardDisk:MPW:Print Examples:PSTestDoc';
  err := PtrToHand(pointer(MyString),MyHandle,length(MyString) + 1);
  MyPic := OpenPicture(theWorld);
  ClipRect(theWorld);
  MoveTo(20,20);
  DrawString('PostScriptFile Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(PostScriptFile,GetHandleSize(MyHandle),MyHandle);
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  MoveTo(50,50);
  DrawString('PostScriptEnd has terminated');
  ClosePicture;
  DisposHandle(MyHandle); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;    { PostFile }
```

Here are the resources:

```
type 'POST' {
  switch {
    case Comment:           /* this is a comment */
          key bitstring[8] = 0;
          fill byte;
          string;

    case ASCII:            /* this is just ASCII text */
          key bitstring[8] = 1;
          fill byte;
          string;

    case Bin:              /* this is binary */
          key bitstring[8] = 2;
          fill byte;
          string;

    case ATEOF:            /* this is an AppleTalk EOF */
          key bitstring[8] = 3;
          fill byte;
          string;
          case DataFork:         /* send the text in the data fork */
               key bitstring[8] = 4;
               fill byte;

          case EOF:              /* no more */
               key bitstring[8] = 5;
               fill byte;
          };
     };

     resource 'POST' (501) {
     ASCII{"0 728 translate "}};

     resource 'POST' (502) {
     ASCII{"1 -1 scale "}};

     resource 'POST' (503) {
     ASCII{"newpath "}};

     resource 'POST' (504) {
     ASCII{"100 470 moveto "}};

     resource 'POST' (505) {
     ASCII{"500 470 lineto "}};

     resource 'POST' (506) {
     ASCII{"100 330 moveto "}};

     resource 'POST' (507) {
     ASCII{"500 330 lineto "}};

     resource 'POST' (508) {
     ASCII{"230 600 moveto "}};

     resource 'POST' (509) {
     ASCII{"230 200 lineto "}};

     resource 'POST' (510) {
     ASCII{"370 600 moveto "}};

     resource 'POST' (511) {
     ASCII{"370 200 lineto "}};

     resource 'POST' (512) {
     ASCII{"10 setlinewidth "}};

     resource 'POST' (513) {
     ASCII{"stroke "}};

     resource 'POST' (514) {
     ASCII{"/Times-Roman findfont 12 scalefont setfont "}};

     resource 'POST' (515) {
     ASCII{"230 600 moveto "}};

     resource 'POST' (516) {
     ASCII{"(Hello World) show "}};
```

```
/* It will stop reading and close the file after 517 */
 resource 'POST' (517) {
 EOF
 {}};

/* it never gets here */
 resource 'POST' (518) {
 DataFork
 {}};
```

When the `ResourcePS` comment is encountered, the LaserWriter driver sends the text contained in the specified resource as PostScript to the printer. The additional data is defined as

```
PSRsrc = RECORD
           PSType : ResType;
           PSID   : INTEGER;
           PSIndex: INTEGER;
         END;
```

The resource can be of type STR or STR#. If the Type is STR then the index should be 0. Otherwise an index should be given.

This comment is essentially the same as the PrintF control call to the driver. The imbedded command string it uses is `'^r^n'`, which basically tells the driver to send the string specified by the additional data, then send a newline. For more information about printer control calls see the *LaserWriter Reference Manual*.

Here's an example:

```
PROCEDURE PostRSRC;
{This procedure shows how to get PostScript from a resource FILE}
  CONST
    PostScriptBegin = 190;
    PostScriptEnd = 191;
    ResourcePS = 195;

  TYPE
    theRSRChdl = ^theRSRCptr;
    theRSRCptr = ^theRSRC;
    theRSRC = RECORD
       theType: ResType;
       theID: INTEGER;
       Index: INTEGER;
    END;

  VAR
    temp         : Rect;
    TheResource  : theRSRChdl;
    i,j          : INTEGER;
    myport       : GrafPtr;
    err          : INTEGER;
    atemp        : Boolean;
```

```
BEGIN          { PostRSRC }
   TheResource := theRSRChdl(NewHandle(SizeOf(theRSRC)));
   TheResource^^.theID := 500;
   TheResource^^.Index := 0;
   TheResource^^.theType := 'STR ';
   HLock(Handle(TheResource));
   MyPic := OpenPicture(theWorld);
   DrawString('ResourcePS Comment');
   PicComment(PostScriptBegin,0,nil); {Begin PostScript}
   PicComment(ResourcePS,8,Handle(TheResource)); {Send postscript}
   PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
   ClosePicture;
   DisposHandle(Handle(TheResource)); {Clean up}
   PrintthePicture; {print it please}
   KillPicture(MyPic);
END;           { PostRSRC }
```

Here's the resource:

```
resource 'STR ' (500)
{"0 728 translate 1 -1 scale newpath 100 470 moveto 500 470 lineto 100 330
moveto 500 330 lineto 230 600 moveto 230 200 lineto 370 600 moveto 370 200
lineto 10 setlinewidth stroke /Times-Roman findfont 12 scalefont setfont 230
600 moveto (Hello World) show"
};
```

Rotation

The concept of rotation doesn't apply to text alone. PostScript can rotate any object. The rotation comments work exactly like text rotation except that all objects drawn between the two comments are drawn in the rotated coordinate system specified by the center of rotation comment, not just text. Also, no clipping of CopyBits calls occurs. These comments only work on the 3.1 and newer LaserWriter drivers.

The RotateBegin comment tells the driver that the following objects will be drawn in a rotated plane. This comment contains the following data structure:

```
Rotation = RECORD
  Flip:  INTEGER; {0,1,2 => none, horizontal, vertical coordinate flip }
  Angle: INTEGER; {0..360 => clockwise rotation in degrees }
END; { Rotation }
```

When you are finished, the RotateEnd comment returns the coordinate system to normal, terminating the rotation.

The relative center of rotation is specified by the RotateCenter comment in exactly the same manner as the TextCenter comments. The difference, however, is that this comment **must** appear **before** the RotateBegin comment. The data structure of the accompanying handle is exactly like that for the TextCenter comment.

Here's an example of how to use rotation comments:

```
PROCEDURE Test;
{This procedure shows how to do rotations}
CONST
  RotateBegin = 200;
  RotateEnd = 201;
  RotateCenter = 202;

TYPE
    rothdl = ^rotptr;
    rotptr = ^trot;
    trot = RECORD
      flip : INTEGER;
      Angle : INTEGER;
    END; { trot }
    centhdl = ^centptr;
    centptr = ^cent;
    Cent = PACKED RECORD
             yInt: INTEGER;
             yFrac: INTEGER;
             xInt: INTEGER;
             xFrac: INTEGER;
    END; { Cent }

VAR
  arect    : Rect;
  rotation : rothdl;
  center   : centhdl;
```

```
BEGIN { Test }
  rotation := rothdl(NewHandle(sizeof(trot)));
  rotation^^.flip := 0;                              {no flip}
  rotation^^.angle := 15;         {15 degree rotation}

  center := centhdl(NewHandle(sizeof(cent)));
  center^^.xInt := 50;                               {center at 50,50}
  center^^.yInt := 50;
  center^^.xFrac := 0;                               {no fractional part}
  center^^.yFrac := 0;

  myPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('Begin Rotation');

    {set the center of Rotation}
    PicComment(RotateCenter,GetHandleSize(Handle(center)),Handle(center));
    {Begin Rotation}
   PicComment(RotateBegin,GetHandleSize(Handle(rotation)),Handle(rotation));
    SetRect(arect,100,100,500,500);
    FrameRect(aRect);
    MoveTo(500,500);
    Lineto(100,100);
    PicComment(RotateEnd,0,nil);             {RotateEnd}
  ClosePicture;
  DisposHandle(handle(rotation));            {Clean up}
  DisposHandle(handle(center));
  PrintThePicture;                                                    {print
it please}
  KillPicture(MyPic);
END; { Test }
```

Forms

The two form printing comments allow you to prepare a template to use for printing. When the `FormsBegin` comment is used, the LaserWriter's buffer is not cleared after `PrClosePage`. This allows you to download a form then change it for each subsequent page, inserting the information you want. `FormsEnd` allows the buffer to be cleared at the next `PrClosePage`.

#92: The Appearance of Text

See also:
The Printing Manager
The Font Manager
Technical Note #91—
Optimizing for the LaserWriter—Picture Comments

Written by: Ginger Jernigan November 15, 1986
Updated: March 1, 1988

This technical note describes why text doesn't always look the way you expect depending on the environment you are in.

There are a number of Macintosh text editing applications where layout is critical. Unfortunately, text on a newer machine sometimes prints differently than text on a 64K ROM Macintosh. Let's examine some differences you should expect and why.

The differences we will consider here are only differences in the layout of text lines (line layout), not differences in the appearance of fonts or the differences between different printers. Differences in line layout may affect the position of line, paragraph and page breaks. The four variables that can affect line layout are fonts, the printer driver, the font manager mode, and ROMs.

Fonts

Every font on a Macintosh contains its own table of widths which tells QuickDraw how wide characters are on the screen. For every style point size there is a separate table which may contain widths that vary from face to face and from point size to point size. Character widths can vary between point sizes of characters even in the same face. In other words, fonts on the screen are not necessarily linearly scalable.

Non-linearity is not normally a problem since most fonts are designed to be as close to linear as possible. A font face in 6 point has very nearly the same scaled widths of the same font face in 24 point (plus or minus round-off or truncation differences). QuickDraw, however, requires only one face of any particular font to be in the System file to use it in any point size. If only a 10 point face actually exists, QuickDraw may scale that face to 9, 18, 24 (or whatever point size) by performing a linear scale of the 10 point face.

This can cause problems. Suppose a document is created on one Macintosh containing a font that only exists in that System file in one point size, say 9 point. The document is then taken to another Macintosh with a System file containing that same font but only in 24 point. The document may, in fact, appear differently on the two screens, and when it is printed, will have line breaks (and thus paragraph and page breaks) occurring in different places simply because of the differences in character widths that exist between the 9 point and 24 point faces.

The Printer Driver

Even when the printer you are using has a much higher resolution than what the screen can show, printer drivers perform line layout to match the screen layout as closely as possible.

The line layout performed by printer drivers is limited to single lines of text and does not change line break positions within multiple lines. The driver supplies metric information to the application about the page size and printable area to allow the application to determine the best place to make line and page breaks.

Printer driver line layout does affect word spacing, character spacing and even word positioning within a line. This may affect the overall appearance of text, particularly when font substitutions are made or various forms of page or text scaling are involved. But print drivers NEVER change line, paragraph or page break positions from what the application or screen specified. This means that where line breaks appear on the screen, they will always appear in the same place on the printer regardless of how the line layout may affect the appearance within the line.

Operating System and ROMs

In this context, operating system refers to the ROM trap routines which handle fonts and QuickDraw. Changes have occurred between the ROMs in the handling of fonts. Fonts in the 64K ROMs contain width tables (as described above) which are limited to integer values. Several new tables, however, have been added to fonts for the newer ROMs. The newer ROMs add an optional global width table containing fractional or fixed point decimal values. In addition, there is another optional table containing fractional values which can be scaled for the entire range of point sizes for any one face. There is also an optional table which provides for the addition (or removal) of width to a font when its style is changed to another value such as bold, outline or condensed. It is also possible, under the 128K ROMs, to add fonts to the system with inherent style properties containing their own width tables that produce different character widths from derived style widths.

One or all of the above tables may or may not be invoked depending on, first, their presence, and second, the mode of the operating system. The Font Manager in the newer ROMs allows the application to arbitrarily operate in either the fractional mode or integer mode (determined, in most cases, by the setting of `FractEnable`) as it chooses, with the default being integer. There is one case where fractional widths will be used if they exist even though fractional mode is disabled. When `FScaleDisable` is used fractional widths are always used if they exist regardless of the setting of `FractEnable`.

Differences in line layout (and thus line breaks) may be affected by any combination of the presence or absence of the optional tables, and the operating mode, either fractional or integer, of the application. Any of the combinations can produce different results from the original ROMs (and from each other).

The integer mode on the newer ROMs is very similar to, but not exactly the same as, the original 64K ROMs. When fonts with the optional tables present are used on Macintoshes with 64K ROMs, they continue to work in the old way with the integer widths. However, on newer ROMs, even in the integer mode, there may be variations in line width from what is seen on the old ROMs. In the plain text style there is very little if any difference (except if the global width table is present), but as various type styles are selected, line widths may vary more between ROMs.

Variations in the above options, by far, account for the greatest variation in the appearance of lines when a document is transported between one Macintosh and another. Line breaks may change position when documents created on one system (say a Macintosh) are moved to another system (like a Macintosh Plus). Variations are more pronounced as the number and sizes of various type styles increase within a document.

In all cases, however, a printer driver will produce exactly the same line breaks as appear on the screen with any given system combination.

Macintosh Technical Notes

#93: MPW: {$LOAD}; _DataInit;%_MethTables

See also: MPW Reference Manuals

Written by: Jim Friedlander November 15, 1986
Modified by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note discusses the Pascal `{$LOAD}` directive as well as how to unload the `_DataInit` and `%_MethTables` segments.

{$LOAD}

MPW Pascal has a `{$LOAD}` directive that can dramatically speed up compiles.

```
{$LOAD HD:MPW:PLibraries:PasSymDump}
```

will combine symbol tables of all units following this directive (until another `{$LOAD}` directive is encountered), and dump them out to `HD:MPW:PLibraries:PasSymDump`. In order to avoid using fully specified pathnames, you can use `{$LOAD}` in conjunction with the `-k` option for Pascal:

```
Pascal -k "{PLibraries}" myfile
```

combined with the following lines in `myfile`:

```
USES
    {$LOAD PasSymDump}
        MemTypes,QuickDraw, OSIntf, ToolIntf, PackIntf,
    {$LOAD} {This “turns off” $LOAD for the next unit}
        NonOptimized,
    {$LOAD MyLibDump}
        MyLib;
```

will do the following: the first time a program containing these lines is compiled, two symbol table dump files (in this case `PasSymDump` and `MyLibDump`) will be created in the directory specified by the `-k` option (in this case `{PLibraries}`). No dump file will be generated for the unit `NonOptimized`. The compiler will compile `MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf` (quite time consuming) and dump those units' symbols to `PasSymDump` and it will compile the interface to `MyLib` and dump its symbols to `MyLib`. For subsequent compiles of this program (or any program that uses the same dump file(s)), the interface files won't be recompiled, the compiler will simply read in the symbol table.

Compiling a sample five line program on a Macintosh Plus/HD20SC takes 62 seconds without using the `{$LOAD}` directive. The same program takes 10 seconds to compile using the `{$LOAD}` directive (once the dump file exists). For further details about this topic, please see the *MPW Pascal Reference Manual*.

Note: If any of the units that are dumped into a dump file change, you need to make sure that the dump file is deleted, so that it can be regenerated by the Pascal compiler with the correct information. The best way to do this is to use a makefile to check the dump file against the files it depends on, and delete the dump file if it is out of date with respect to any of the units that it contains. An excellent (and well commented) example of doing this is in the *MPW Workshop Manual.*

The _DataInit Segment

The Linker will generate a segment whose resource name is `%A5Init` for any program compiled by the C or Pascal compilers. This segment is called by a program's main segment. This segment is loaded into the application heap and locked in place. It is up to your program to unload this segment (otherwise, it will remain locked in memory, possibly causing heap fragmentation). To do this from Pascal, use the following lines:

```
PROCEDURE _DataInit;EXTERNAL;
...

BEGIN         {main PROGRAM}
UnloadSeg(@_DataInit);
{remove data initialization code before any allocations}
...
```

From C, use the following lines:

```
extern _DataInit();
...
{ /* main */
                UnloadSeg(_DataInit);
                /*remove data initialization code before any allocations*/
...
```

For further details about Data Initialization, see the *MPW Reference Manual.*

%_MethTables and %_SelProcs

Object use in Pascal produces two segments which can cause heap problems. These are `%_MethTables` and `%_SelProcs` which are used when method calls are made. MacApp deals with them correctly, so this only applies to Object Pascal programs that don't use MacApp. You can make the segments locked and preloaded (probably the easiest route), so they will be loaded low in the heap, or you can unload them temporarily while you are doing heap initialization. In the latter case, make sure there are no method calls while they are unloaded. To reload `%_MethTables` and `%_SelProcs`, call the dummy procedure `%_InitObj`. `%_InitObj` loads `%MethTables` —calling any method will then load `%_SelProcs`.

Reminder: The linker is case sensitive when dealing with module names. Pascal converts all module names to upper-case (unless a routine is declared to be a C routine). The Assembler default is the same as the Pascal default, though it can be changed with the CASE directive. C preserves the case of module names (unless a routine is declared to be `pascal`, in which case the module name is converted to upper-case letters).

Make sure that any external routines that you reference are capitalized the same in both the external routine and the external declaration (especially in C). If the capitalization differs, you will get the following link error (library routine = `findme`, program declaration = `extern FindMe();`):

```
### Link: Error  Undefined entry, name: FindMe
```

#94: Tags

See also: The File Manager

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

Apple has decided to eliminate support for file-system tags on its future products; this technical note explains this decision.

Some of Apple's disk products (and some third-party products) have the ability to store 532 bytes per sector, instead of the normal 512. Twelve of the extra bytes are used to store redundant file system information, known as "tags", to be used by a scavenging utility to reconstruct damaged disks.

Apple has decided to eliminate support for these tags on its products; this was decided for several reasons:

1) Tags were implemented back when we had to deal with "Twiggy" drives on Lisa. These drives were less reliable than current drives, and it was expected that tags would be needed for data integrity.

2) We're working on a scavenging utility (Disk First Aid), and we've found that tags don't help us in reconstructing damaged disks (ie, if we can't fix it without using tags, tags wouldn't help us fix it). So, at least the first two versions of our scavenging utility will not use tags, and a third version (which we've planned for, but will probably never implement) can probably work without them.

3) 532-byte-per-sector drives and controllers tend to cost more, even at Apple's volumes. Thus, the demise of tags saves us (and our customers) money. The Apple Hard Disk 20SC currently supports tags; this may not always be the case, however; we'll probably drop the large sectors when we run out of our current stock of drives.

The Hierarchical File System (HFS) documentation didn't talk about tags because the writer had no information available about how they worked under HFS. Because of this decision, it is unlikely that we'll ever have documentation on how to correctly implement them under HFS.

Macintosh Technical Notes

#95: How To Add Items to the Print Dialogs

See also: The Printing Manager
The Dialog Manager

Written by: Ginger Jernigan
Lew Rollins

November 15, 1986

Updated: March 1, 1988

This technical note discusses how to add your own items to the Printing Manager's dialogs.

When the Printing Manager was initially designed, great care was taken to make the interface to the printer drivers as generic as possible in order to allow applications to print without being device-specific. There are times, however, when this type of non-specific interface interferes with the flexibility of an application. An application may require additional information before printing which is not part of the general Printing Manager interface. This technical note describes a method that an application can use to add its own items to the existing style and job dialogs.

Before continuing, you need to be aware of some guidelines that will increase your chances of being compatible with the printing architecture in the future:

- Only add items to the dialogs as described in this technical note. Any other methods will decrease your chances of survival in the future.
- Do not change the position of any item in the current dialogs. This means don't delete items from the existing item list or add items in the middle. Add items **only at the end** of the list.
- Don't count on an item retaining its current position in the list. If you depend on the Draft button being a particular number in the ImageWriter's style dialog item list, and we change the Draft button's item number for some reason, your program may no longer function correctly.
- Don't use more than half the screen height for your items. Apple reserves the right to expand the items in the standard print dialogs to fill the top half of the screen.
- If you are adding lots of items to the dialogs (which may confuse users), you should consider having your own separate dialog in addition to the existing Printing Manager dialogs.

The Heart

Before we talk about how the dialogs work, you need to know this: at the heart of the printer dialogs is a little-known data structure partially documented in the MacPrint interface file. It's a record called `TPrDlg` and it looks like this:

```
TPrDlg = RECORD    {Print Dialog: The Dialog Stream object.}
   dlg        : DialogRecord;    {dialog window}
   pFltrProc  : ProcPtr;         {filter proc.}
   pItemProc  : ProcPtr;         {item evaluating proc.}
   hPrintUsr  : THPrint;         {user's print record.}
   fDoIt      : BOOLEAN;
   fDone      : BOOLEAN;
   lUser1     : LONGINT;         {four longs reserved by Apple}
   lUser2     : LONGINT;
   lUser3     : LONGINT;
   lUser4     : LONGINT;
   iNumFst    : INTEGER;         {numeric edit items for std filter}
   iNumLst    : INTEGER;
{... plus more stuff needed by the particular printing dialog.}
END;
TPPrDlg = ^TPrDlg;               {== a dialog ptr}
```

All of the information pertaining to a print dialog is kept in the `TPrDlg` record. This record will be referred to frequently in the discussion below.

How the Dialogs Work

When your application calls `PrStlDialog` and `PrJobDialog`, the printer driver actually calls a routine called `PrDlgMain`. This function is declared as follows:

```
FUNCTION PrDlgMain (hprint: THPrint; pDlgInit: ProcPtr): BOOLEAN;
```

`PrDlgMain` first calls the `pDlgInit` routine to set up the appropriate dialog (in `Dlg`), dialog hook (`pItemProc`) and dialog event filter (`pFilterProc`) in the `TPrDlg` record (shown above). For the job dialog, the address of `PrJobInit` is passed to `PrDlgMain`. For the style dialog, the address of `PrStlInit` is passed. These routines are declared as follows:

```
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
```

After the initialization routine sets up the `TPrDlg` record, `PrDlgMain` calls `ShowWindow` (the window is initially invisible), then it calls `ModalDialog`, using the dialog event filter pointed to by the `pFltrProc` field. When an item is hit, the routine pointed to by the `pItemProc` field is called and the items are handled appropriately. When the OK button is hit (this includes pressing Return or Enter) the print record is validated. The print record is not validated if the Cancel button is hit.

How to Add Your Own Items

To modify the print dialogs, you need to change the TPrDlg record before the dialog is drawn on the screen. You can add your own items to the item list, replace the addresses of the standard dialog hook and event filter with the addresses of your own routines and then let the dialog code continue on its merry way.

For example, to modify the job dialog, first call PrJobInit. PrJobInit will fill in the TPrDlg record for you and return a pointer to that record. Then call PrDlgMain directly, passing in the address of your own initialization function. The example code's initialization function adds items to the dialog item list, saves the address of the standard dialog hook (in our global variable prPItemProc) and puts the address of our dialog hook into the pItemProc field of the TPrDlg record. Please note that your dialog hook **must** call the standard dialog hook to handle all of the standard dialog's items.

Note: If you wish to have an event filter, handle it the same way that you do a dialog hook.

Now, here is an example (written in MPW Pascal) that modifies the job dialog. The same code works for the style dialog if you globally replace 'Job' with 'Stl'. Also included is a function (AppendDITL) provided by Lew Rollins (originally written in C, translated for this technical note to MPW Pascal) which demonstrates a method of adding items to the item list, placing them in an appropriate place, and expanding the dialog window's rectangle.

The MPW Pascal Example Program

```
PROGRAM ModifyDialogs;

 USES
  {$LOAD PasDump.dump}
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint;

 CONST
  MyDITL    = 256;
  MyDFirstBox  = 1;     {Item number of first box in my DITL}
  MyDSecondBox = 2;

 VAR
  PrtJobDialog: TPPrDlg;  { pointer to job dialog }
  hPrintRec  : THPrint;  { Handle to print record }
  FirstBoxValue,       { value of our first additional box }
  SecondBoxValue: Integer; { value of our second addtl. box }
  prFirstItem,       { save our first item here }
  prPItemProc : LongInt;  { we need to store the old itemProc here }
  itemType   : Integer;  { needed for GetDItem/SetDItem calls }
  itemH    : Handle;
  itemBox   : Rect;
  err    : OSErr;
  {-------------------------------------------------------------------------}

 PROCEDURE _DataInit;
  EXTERNAL;
 {-----------------------------------------------------------------------}

 PROCEDURE CallItemHandler(theDialog: DialogPtr; theItem: Integer; theProc:
LongInt);
   INLINE $205F,$4E90;     { MOVE.L (A7)+,A0
                              JSR (A0)          }

{ this code pops off theProc and then does a JSR to it, which puts the
 real return address on the stack. }

 {-----------------------------------------------------------------------}

 FUNCTION AppendDITL(theDialog: DialogPtr; theDITLID: Integer): Integer;
 { version 0.1 9/11/86 Lew Rollins of Human-Systems Interface Group}
 { this routine still needs some error checking }

{ This routine appends all of the items of a specified DITL
onto the end of a specified DLOG — We don't even need to know the format
of the DLOG }

{ this will be done in 3 steps:
 1. append the items of the specified DITL onto the existing DLOG
 2. expand the original dialog window as required
 3. return the adjusted number of the first new user item
}
   TYPE
    DITLItem   = RECORD { First, a single item }
            itmHndl: Handle; { Handle or procedure pointer for this item }
            itmRect: Rect; { Display rectangle for this item }
            itmType: SignedByte; { Item type for this item — 1 byte }
            itmData: ARRAY [0..0] OF SignedByte; { Length byte of data }
          END;  {DITLItem}

    pDITLItem  = ^DITLItem;
    hDITLItem  = ^pDITLItem;

    ItemList   = RECORD { Then, the list of items }
            dlgMaxIndex: Integer; { Number of items minus 1 }
            DITLItems: ARRAY [0..0] OF DITLItem; { Array of items }
          END;  {ItemList}

    pItemList  = ^ItemList;
    hItemList  = ^pItemList;

    IntPtr    = ^Integer;

   VAR
    offset   : Point;  { Used to offset rectangles of items being appended }
    maxRect   : Rect;  { Used to track increases in window size }
    hDITL    : hItemList; { Handle to DITL being appended }
    pItem    : pDITLItem; { Pointer to current item being appended }
    hItems   : hItemList; { Handle to DLOG's item list }
    firstItem : Integer; { Number of where first item is to be appended }
    newItems,       { Count of new items }
    dataSize,       { Size of data for current item }
    i      : Integer; { Working index }
    USB     : RECORD  {we need this because itmData[0] is unsigned}
            CASE Integer OF
```

```
              1:
                (SBArray: ARRAY [0..1] OF SignedByte);
              2:
                (Int: Integer);
           END;   {USB}

   BEGIN          {AppendDITL}
{
 Using the original DLOG

 1. Remember the original window Size.
 2. Set the offset Point to be the bottom of the original window.
 3. Subtract 5 pixels from bottom and right, to be added
    back later after we have possibly expanded window.
 4. Get working Handle to original item list.
 5. Calculate our first item number to be returned to caller.
 6. Get locked Handle to DITL to be appended.
 7. Calculate count of new items.
}
    maxRect := DialogPeek(theDialog)^.window.port.portRect;
    offset.v := maxRect.bottom;
    offset.h := 0;
    maxRect.bottom := maxRect.bottom - 5;
    maxRect.right := maxRect.right - 5;
    hItems := hItemList(DialogPeek(theDialog)^.items);
    firstItem := hItems^^.dlgMaxIndex + 2;
    hDITL := hItemList(GetResource('DITL',theDITLID));
    HLock(Handle(hDITL));
    newItems := hDITL^^.dlgMaxIndex + 1;
{
 For each item,
  1. Offset the rectangle to follow the original window.
  2. Make the original window larger if necessary.
  3. fill in item Handle according to type.
}

    pItem := @hDITL^^.DITLItems;
    FOR i := 1 TO newItems DO BEGIN
     OffsetRect(pItem^.itmRect,offset.h,offset.v);
     UnionRect(pItem^.itmRect,maxRect,maxRect);

     USB.Int := 0;    {zero things out}
     USB.SBArray[1] := pItem^.itmData[0];

     { Strip enable bit since it doesn't matter here. }
     WITH pItem^ DO
      CASE BAND(itmType,$7F) OF
       userItem:  { Can't do anything meaningful with user items. }
        itmHndl := NIL;
       ctrlItem + btnCtrl,ctrlItem + chkCtrl,ctrlItem + radCtrl:{build Control }
        itmHndl := Handle(NewControl(theDialog, { theWindow }
                      itmRect, { boundsRect }
                      StringPtr(@itmData[0])^, { title }
                      true, { visible }
                      0,0,1, { value, min, max }
                      BAND(itmType,$03), { procID }
                      0)); { refCon }
       ctrlItem + resCtrl: BEGIN { Get resource based Control }
```

```
        itmHndl := Handle(GetNewControl(IntPtr(@itmData[1])^, { controlID }
                        theDialog)); { theWindow }
        ControlHandle(itmHndl)^^.contrlRect := itmRect; {give it the right
                               rectangle}
        {An actionProc for a Control should be installed here}
       END;     {Case ctrlItem + resCtrl}
       statText,editText: { Both need Handle to a copy of their text. }
        err := PtrToHand(@itmData[1], { Start of data }
                 itmHndl, { Address of new Handle }
                 USB.Int); { Length of text }
       iconItem:  { Icon needs resource Handle. }
        pItem^.itmHndl := GetIcon(IntPtr(@itmData[1])^); { ICON resID }
       picItem:   { Picture needs resource Handle. }
        pItem^.itmHndl := Handle(GetPicture(IntPtr(@itmData[1])^));{PICT resID}
       OTHERWISE
        itmHndl := NIL;
      END;      {Case}

    dataSize := BAND(USB.Int + 1,$FFFE);
    {now advance to next item}
    pItem := pDITLItem(Ptr(ord4(@pItem^) + dataSize + sizeof(DITLItem)));
   END;         {for}
   err := PtrAndHand
    (@hDITL^^.DITLItems,Handle(hItems),GetHandleSize(Handle(hDITL)));
   hItems^^.dlgMaxIndex := hItems^^.dlgMaxIndex + newItems;
   HUnlock(Handle(hDITL));
   ReleaseResource(Handle(hDITL));
   maxRect.bottom := maxRect.bottom + 5;
   maxRect.right := maxRect.right + 5;
   SizeWindow(theDialog,maxRect.right,maxRect.bottom,true);
   AppendDITL := firstItem;
  END;          {AppendDITL}

{-------------------------------------------------------------------------}

PROCEDURE MyJobItems(theDialog: DialogPtr; itemNo: Integer);
{
This routine replaces the routine in the pItemProc field in the
TPPrDlg record.  The steps it takes are:
1. Check to see if the item hit was one of ours. This is done by "localizing"
   the number, assuming that our items are numbered from 0..n
2. If it's one of ours  then case it and Handle appropriately
3. If it isn't one of ours then call the old item handler
}

  VAR
   MyItem,firstItem: Integer;
   thePt    : Point;
   thePart   : Integer;
   theValue   : Integer;
   debugPart  : Integer;

  BEGIN         {MyJobItems}
   firstItem := prFirstItem; { remember, we saved this in myJobDlgInit }
   MyItem := itemNo - firstItem + 1; { "localize" current item No }
   IF MyItem > 0 THEN BEGIN { if localized item > 0, it's one of ours }
    { find out which of our items was hit }
    GetDItem(theDialog,itemNo,itemType,itemH,itemBox);
```

```
      CASE MyItem OF
       MyDFirstBox: BEGIN
         { invert value of FirstBoxValue and redraw it }
         FirstBoxValue := 1 - FirstBoxValue;
         SetCtlValue(ControlHandle(itemH),FirstBoxValue);
       END;       {case MyDFirstBox}
       MyDSecondBox: BEGIN
         { invert value of SecondBoxValue and redraw it }
         SecondBoxValue := 1 - SecondBoxValue;
         SetCtlValue(ControlHandle(itemH),SecondBoxValue);
       END;       {case MyDSecondBox}
       OTHERWISE
         Debug;    { OH OH — We got an item we didn't expect }
      END;        {Case}
     END         { if MyItem > 0 }
     ELSE       { chain to standard item handler, whose address is saved
                 in prPItemProc }
      CallItemHandler(theDialog,itemNo,prPItemProc);
   END;           { MyJobItems }

  {-------------------------------------------------------------------------}

  FUNCTION MyJobDlgInit(hPrint: THPrint): TPPrDlg;
{
 This routine appends items to the standard job dialog and sets up the
 user fields of the printing dialog record TPRDlg
 This routine will be called by PrDlgMain
 This is what it does:
 1. First call PrJobInit to fill in the TPPrDlg record.
 2. Append our items onto the old DITL. Set them up appropriately.
 3. Save the address of the old item handler and replace it with ours.
 4. Return the Fixed dialog to PrDlgMain.
 }

   VAR
    firstItem : Integer; { first new item number }

   BEGIN          {MyJobDlgInit}
    firstItem := AppendDITL(DialogPtr(PrtJobDialog),MyDITL);

    prFirstItem := firstItem; { save this so MyJobItems can find it }

    { now we'll set up our DITL items — The "First Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),FirstBoxValue);

    { now we'll set up the second of our DITL items — The "Second Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem + 1,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),SecondBoxValue);

{ Now comes the part where we patch in our item handler.  We have to save
 the old item handler address, so we can call it if one of the standard
 items is hit, and put our item handler's address
 in pItemProc field of the TPrDlg struct}

    prPItemProc := LongInt(PrtJobDialog^.pItemProc);

    { Now we'll tell the modal item handler where our routine is }
  PrtJobDialog^.pItemProc := ProcPtr(@MyJobItems);

  { PrDlgMain expects a pointer to the modified dialog to be returned.... }
  MyJobDlgInit := PrtJobDialog;

 END;           {myJobDlgInit}

{-------------------------------------------------------------------------}

FUNCTION Print: OSErr;

 VAR
  bool    : BOOLEAN;

 BEGIN          {Print}
  hPrintRec := THPrint(NewHandle(sizeof(TPrint)));
  PrintDefault(hPrintRec);
  bool := PrValidate(hPrintRec);
  IF (PrError <> noErr) THEN BEGIN
   Print := PrError;
   Exit(Print);
  END;          {If}

  { call PrJobInit to get pointer to the invisible job dialog }
  PrtJobDialog := PrJobInit(hPrintRec);
  IF (PrError <> noErr) THEN BEGIN
   Print := PrError;
   Exit(Print);
  END;          {If}

{Here's the line that does it all!}
  IF NOT (PrDlgMain(hPrintRec,@MyJobDlgInit)) THEN BEGIN
   Print := cancel;
   Exit(Print);
  END;          {If}

  IF PrError <> noErr THEN Print := PrError;

  { that's all for now }

 END;           { Print }

{-------------------------------------------------------------------------}

BEGIN           {PROGRAM}

  UnloadSeg(@_DataInit);  {remove data initialization code before any
allocations}
  InitGraf(@thePort);
  InitFonts;
  FlushEvents(everyEvent,0);
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(NIL);
  InitCursor;

  { call the routine that does printing }
```

```
   FirstBoxValue := 0;    { value of our first additional box }
   SecondBoxValue := 0;    { value of our second addtl. box }
   PrOpen;    { Open the Print Manager }
   IF PrError = noErr THEN
    err := Print      { This actually brings up the modified Job dialog }
   ELSE BEGIN
    {tell the user that PrOpen failed}
   END;

   PrClose;    { Close the Print Manager and leave }
  END.
```

The Lightspeed C Example Program

```
/* NOTE: Apple reserves the top half of the screen (where the current DITL
      items are located). Applications may use the bottom half of the
      screen to add items, but should not change any items in the top half
      of the screen.  An application should expand the print dialogs only
      as much as is absolutely necessary.
*/

/* Note: A global search and replace of 'Job' with 'Stl' will produce
      code that modifies the style dialogs */
#include <DialogMgr.h>
#include <MacTypes.h>
#include <Quickdraw.h>
#include <ResourceMgr.h>
#include <WindowMgr.h>
#include <pascal.h>
#include <printmgr.h>
#define nil 0L


static TPPrDlg PrtJobDialog;         /* pointer to job dialog */

/*    This points to the following structure:

        struct {
              DialogRecord    Dlg;         (The Dialog window)
              ProcPtr         pFltrProc;   (The Filter Proc.)
              ProcPtr         pItemProc;   (The Item evaluating proc. --
                                           we'll change this)
              THPrint         hPrintUsr;   (The user's print record.)
              Boolean         fDoIt;
              Boolean         fDone;
                 (Four longs -- reserved by Apple Computer)
              long                  lUser1;
              long                  lUser2;
              long                  lUser3;
              long                  lUser4;
   } TPrDlg; *TPPrDlg;
*/



/*    Declare 'pascal' functions and procedures */
pascal Boolean PrDlgMain();          /* Print manager's dialog handler */
pascal TPPrDlg PrJobInit();          /* Gets standard print job dialog. */
pascal TPPrDlg MyJobDlgInit();       /* Our extention to PrJobInit */
pascal void MyJobItems();            /* Our modal item handler */

#define MyDITL 256                   /* resource ID of my DITL to be spliced
                                           on to job dialog */


THPrint hPrintRec;                   /* handle to print record */
short FirstBoxValue = 0;             /* value of our first additional box */
short SecondBoxValue = 0;            /* value of our second addtl. box */
long prFirstItem;                    /* save our first item here */
long prPItemProc;              /* we need to store the old itemProc here */
```

```
/*-----------------------------------------------------------------------*/
      WindowPtr   MyWindow;
      OSErr       err;
      Str255      myStr;
main()
{
      Rect        myWRect;

      InitGraf(&thePort);
      InitFonts();
      InitWindows();
      InitMenus();
      InitDialogs(nil);
      InitCursor();
      SetRect(&myWRect,50,260,350,340);

      /* call the routine that does printing */
      PrOpen();
      err = Print();

      PrClose();
} /* main */

/*-----------------------------------------------------------------------*
/

OSErr Print()

{
      /* call PrJobInit to get pointer to the invisible job dialog */
      hPrintRec = (THPrint)(NewHandle(sizeof(TPrint)));
      PrintDefault(hPrintRec);
      PrValidate(hPrintRec);
      if (PrError() != noErr)
            return PrError();

      PrtJobDialog = PrJobInit(hPrintRec);
      if (PrError() != noErr)
            return PrError();


      if (!PrDlgMain(hPrintRec, &MyJobDlgInit)) /* this line does all the
                                                   stuff */
            return Cancel;

      if (PrError() != noErr)
            return PrError();

/* that's all for now */

} /* Print */

/*-----------------------------------------------------------------------*
/

pascal TPPrDlg MyJobDlgInit (hPrint)
THPrint hPrint;
```

```
/* this routine appends items to the standard job dialog and sets up the
      user fields of the printing dialog record TPRDlg
      This routine will be called by PrDlgMain */
{
      short        firstItem;        /* first new item number */

      short        itemType;         /* needed for GetDItem/SetDItem call */
      Handle       itemH;
      Rect         itemBox;

      firstItem = AppendDITL (PrtJobDialog, MyDITL); /*call routine to do
                                                       this */

      prFirstItem = firstItem; /* save this so MyJobItems can find it */

/* now we'll set up our DITL items -- The "First Box" */
      GetDItem(PrtJobDialog,firstItem,&itemType,&itemH,&itemBox);
      SetCtlValue(itemH,FirstBoxValue);

/* now we'll set up the second of our DITL items  -- The "Second Box" */
      GetDItem(PrtJobDialog,firstItem+1,&itemType,&itemH,&itemBox);
      SetCtlValue(itemH,SecondBoxValue);

/* Now comes the part where we patch in our item handler.  We have to save
      the old item handler address, so we can call it if one of the
      standard items is hit, and put our item handler's address
      in pItemProc field of the TPrDlg struct
*/

      prPItemProc = (long)PrtJobDialog->pItemProc;

/* Now we'll tell the modal item handler where our routine is */
      PrtJobDialog->pItemProc = (ProcPtr)&MyJobItems;

/* PrDlgMain expects a pointer to the modified dialog to be returned.... */
      return PrtJobDialog;

} /*myJobDlgInit*/


/*------------------------------------------------------------------------*/

/* here's the analogue to the SF dialog hook */

pascal void MyJobItems(theDialog,itemNo)
TPPrDlg     theDialog;
short       itemNo;

{ /* MyJobItems */
      short        myItem;
      short        firstItem;

      short        itemType;         /* needed for GetDItem/SetDItem call */
      Handle       itemH;
      Rect         itemBox;

      firstItem = prFirstItem; /* remember, we saved this in myJobDlgInit
*/
```

```
      myItem = itemNo-firstItem+1;  /* "localize" current item No */
      if (myItem > 0)   /* if localized item > 0, it's one of ours */
      {
            /* find out which of our items was hit */
            GetDItem(theDialog,itemNo,&itemType,&itemH,&itemBox);
            switch (myItem)
            {
                  case 1:
                        /* invert value of FirstBoxValue and redraw it */
                        FirstBoxValue ^= 1;
                        SetCtlValue(itemH,FirstBoxValue);
                        break;

                  case 2:
                        /* invert value of SecondBoxValue and redraw it */
                        SecondBoxValue ^= 1;
                        SetCtlValue(itemH,SecondBoxValue);
                        break;
                  default: Debugger(); /* OH OH */
            } /* switch */
      } /* if (myItem > 0) */
      else /* chain to standard item handler, whose address is saved in
                  prPItemProc */
      {
            CallPascal(theDialog,itemNo,prPItemProc);
      }
} /* MyJobItems */
```

The Rez Source

```
#include "types.r"

resource 'DITL' (256) {
 { /* array DITLarray: 2 elements */
   /* [1] */
   {8, 0, 24, 112},
   CheckBox {
    enabled,
    "First Box"
   };
   /* [2] */
   {8, 175, 24, 287},
   CheckBox {
    enabled,
    "Second Box"
   }
 }
};
```

Macintosh Technical Notes

#96: SCSI Bugs

See also: The SCSI Manager
SCSI Developer's Package

Written by:	Steve Flowers	October 1, 1986
Modified by:	Bryan Stearns	November 15, 1986
Modified by:	Bo3b Johnson	July 1, 1987
Updated:		March 1, 1988

There are a number of problems in the SCSI Manager; this note lists the ones we know about, along with an explanation of what we're doing about them. Changes made for the 2/88 release are made to more accurately reflect the state of the SCSI Manager. System 4.1 and 4.2 are very similar; one bug was fixed in System 4.2.

There are several categories of SCSI Manager problems:

1. Those in the ROM boot code
(Before the System file has been opened, and hence, before any patches could possibly fix them.)
2. Those that have been fixed in System 3.2
3. Those that have been fixed in System 4.1/4.2
4. Those that are new in System 4.1/4.2
5. Those that have not yet been fixed.

The problems in the ROM boot code can only be fixed by changing the ROMs. Most of the bugs in the SCSI Manager itself have been fixed by the patch code in the System 3.2 file. There are a few problems, though, that are not fixed with System 3.2—most of these bugs have been corrected in System 4.1/4.2. Any that are not fixed will be detailed here. ROM code for future machines will, of course, include the corrections.

ROM boot code problems

- In the process of looking for a bootable SCSI device, the boot code issues a SCSI bus reset before each attempt to read block 0 from a device. If the read fails for any reason, the boot code goes on to the next device. SCSI devices which implement the `Unit Attention` condition as defined by the Revision 17B SCSI standard will fail to boot in this case. The read will fail because the drive is attempting to report the `Unit Attention` condition for the first command it receives after the SCSI bus reset. The boot code does not read the sense bytes and does not retry the failed command; it simply resets the SCSI bus and goes on to the next device.

If no other device is bootable, the boot code will eventually cycle back to the same SCSI device ID, reset the bus (causing `Unit Attention` in the drive again), and try to read block 0 (which fails for the same reason).

The 'new' Macintosh Plus ROMs that are included in the platinum Macintosh Plus have only one change. The change was to simply do a single SCSI Bus Reset after power up instead of a Reset each time through the SCSI boot loop. This was done to allow `Unit Attention` drives to be bootable. It was an object code patch (affecting approximately 30 bytes) and no other bugs were fixed. For details on the three versions of Macintosh Plus ROMs, see Technical Note #154.

We recommend that you choose an SCSI controller which does not require the `Unit Attention` feature—either an older controller (most of the SCSI controllers currently available were designed before Revision 17B), or one of the newer Revision-17B-compatible controllers which can enable/disable `Unit Attention` as a formatting option (such as those from Seagate, Rodime, et al). Since the vast majority of Macintosh Plus computers have the ROMs which cannot use `Unit Attention` drives, we still recommend that you choose an SCSI controller that does not require the `Unit Attention` feature.

- If an SCSI device goes into the `Status` phase after being selected by the boot code, this leads to the SCSI bus being left in the `Status` phase indefinitely, and no SCSI devices can be accessed. The current Macintosh Plus boot code does not handle this change to `Status` phase, which means that the presence of an SCSI device with this behavior (as in some tape controllers we've seen) will prevent any SCSI devices from being accessed by the SCSI Manager, even if they already had drivers loaded from them. The result is that any SCSI peripheral that is turned on at boot time must not go into `Status` phase immediately after selection; otherwise, the Macintosh Plus SCSI bus will be left hanging. Unless substantially revised ROMs are released for the Macintosh Plus (highly unlikely within the next year or so), this problem will never be fixed on the Macintosh Plus, so you should design for old ROMs.

- The Macintosh Plus would try to read 256 bytes of blocks 0 and 1, ignoring the extra data. The Macintosh SE and Macintosh II try to read 512 bytes from blocks 0 and 1, ignoring errors if the sector size is larger (but not smaller) than 512 bytes. Random access devices (disks, tapes, CD ROMS, etc.) can be booted as long as the blocks are at least 512 bytes, blocks 0, 1 and other partition blocks are correctly set up, and there is a driver on it. With the new partition layout (documented in *Inside Macintosh* volume V), more than 256 bytes per sector may be required in some partition map entries. This is why we dropped support for 256-byte sectors. Disks with tag bytes (532-byte sectors) or larger block sizes (1K, 2K, etc.) can be booted on any Macintosh with an SCSI port. Of course, the driver has to take care of data blocking and de-blocking, since HFS likes to work with 512-byte sectors.

Problems with ROM SCSI Manager routines

Note that the following problems are fixed after the System file has been opened; for a device to boot properly, it must not depend on these fixes. The sample SCSI driver, available from APDA, contains an example of how to find out if the fixes are in place.

- **Prior to System file 3.2,** blind transfers (both reads and writes) would not work properly with many SCSI controllers. Since blind operation depends on the drive's ability to transfer data fast enough, it is the responsibility of the driver writer to make sure blind operation is safe for a particular device.

- **Prior to System file 3.2,** the SCSI Manager dropped a byte when the driver did two or more `SCSIReads` or `SCSIRBlinds` in a row. (Each `Read` or `RBlind` has to have a Transfer Information Block (TIB) pointer passed in.) The TIB itself can be as big and complex as you want—it is the process of returning from one `SCSIRead` or `SCSIRBlind` and entering another one (while still on the same SCSI command) that causes the first byte for the other `SCSIReads` to be lost.

  Note that this precludes use of file-system tags. Apple no longer recommends that you support tags; see Technical Note #94 for more information.

- **Prior to System file 3.2,** `SCSIStat` didn't work; the new version works correctly.

- **Running under System file 3.2,** the SCSI Manager does not check to make sure that the last byte of a write operation (to the peripheral) was handshaked while operating in pseudo-DMA mode. The SCSI Manager writes the final byte to the NCR 5380's one-byte buffer and then turns pseudo-DMA mode off shortly thereafter (reported to be 10-15 microseconds). If the peripheral is somewhat slow in actually reading the last byte of data, it asserts `REQ` after the Macintosh has already turned off pseudo-DMA mode and never gets an `ACK`. The CPU then expects to go into the `Status` phase since it thinks everything went OK, but the peripheral is still waiting for `ACK`. Unless the driver can recover from this somehow, the SCSI bus is 'hung' in the `Data Out` phase. In this case, all successive SCSI Manager calls will fail until the bus is reset.

- **Running under System file 4.1/4.2,** the SCSI Manager waits for the last byte of a write operation to be handshaked while operating in pseudo-DMA mode; it checks for a final `DRQ` (or a phase change) at the end of a `SCSIWrite` or `SCSIWBlind` before turning off the pseudo-DMA mode. Drivers that could recover from this problem by writing the last byte again if the bus was still in a `Data Out` phase will still work correctly, as long as they were checking the bus state.

- **Running under System file 3.2,** the SCSI Manager does not time out if the peripheral fails to finish transferring the expected number of bytes for polled reads and writes. (Blind operation does poll for the first byte of each requested data transfer in the Transfer Information Block.)

- **Running under System file 4.1/4.2,** SCSIRead and SCSIWrite return an error to the caller if the peripheral changes the bus phase in the middle of a transfer, as might happen if the peripheral fails to transfer the expected number of bytes. The computer is no longer left in a hung state.

- **Running under System file 3.2,** the Selection timeout value is very short (900 microseconds). Patches to the SCSI Manager **in System 4.1/4.2** ensure that this value is the recommended 250 milliseconds.

- **Running under System file 3.2,** the SCSI Manager routine SCSIGet (which arbitrates for the bus) will fail if the BSY line is still asserted. Some devices are a bit slow in releasing BSY after the completion of an SCSI operation, meaning that BSY may not have been released before the driver issues a SCSIGet call to start the next SCSI operation. A work-around for this is to call SCSIGet again if it failed the first time. (Rarely has it been necessary to try it a third time.) This assumes, of course, that the bus has not been left 'hanging' by an improperly terminated SCSI operation before calling SCSIGet.

- **Running under System file 4.1/4.2,** the SCSIGet function has been made more tolerant of devices that are slow to release the BSY line after a SCSI operation. The SCSI Manager now waits up to 200 milliseconds before returning an error.

Problems with the SCSI Manager that haven't been fixed yet

These problems currently exist in the Macintosh Plus, SE, and II SCSI Manager. We plan to fix these problems in a future release of the System Tools disk, but in the mean time, you should try to work around the problems (but don't "require" the problems!).

- Multiple calls to SCSIRead or SCSIRBlind after issuing a command and before calling SCSIComplete may not work. Suppose you want to read some mode sense data from the drive. After sending the command with SCSICmd, you might want to call SCSIRead with a TIB that reads four bytes (typically a header). After reading the field (in the four-byte header) that tells how many remaining bytes are available, you might call SCSIRead again with a TIB to read the remaining bytes. The problem is that the first byte of the second SCSIRead data will be lost because of the way the SCSI Manager handles reads in pseudo-DMA mode. The work-around is to issue two separate SCSI commands: the first to read only the four-byte header, the second to read the four-byte header plus the remaining bytes. We recommend that you **not** use a clever TIB that contains two data transfers, the second of which gets the transfer length from the first transfer's received data (the header). These two step TIBs will not work in the future. This bug will probably not be fixed.

- On read operations, some devices may be slow in deasserting REQ after sending the last byte to the CPU. The current SCSI Manager (all machines) will return to the caller without waiting for REQ to be deasserted. Usually the next call that the driver would make is SCSIComplete. On the Macintosh SE and II, the SCSIComplete call will check the bus to be sure that it is in Status phase. If not, the SCSI Manager will return a new error code that indicates the bus was in Data In/Data Out phase when SCSIComplete was called. The combination of the speed of the Macintosh II and a

slow peripheral can cause `SCSIComplete` to detect that the bus is still in Data In phase before the peripheral has finally changed the bus to `Status` phase. This results in a false error being passed back by `SCSIComplete`.

- The `scComp` (compare) TIB opcode does not work in System 4.1 on the Macintosh Plus only. It returns an error code of 4 (bad parameters). This has been fixed in System 4.2.

Other SCSI Manager Issues

- At least one third-party SCSI peripheral driver used to issue SCSI commands from a VBL task. It didn't check to see if the bus was in the free state before sending the command! This is guaranteed to wipe out any other SCSI command that may have been in progress, since the SCSI Manager on the Macintosh Plus does not mask out (or use) interrupts.

  We strongly recommend that you avoid calling the SCSI Manager from interrupt handlers (such as VBL tasks). If you must send SCSI commands from a VBL task (like for a removable media system), do a `SCSIStat` call first to see if the bus is currently busy. If it's free (`BSY` is not asserted), then it's probably safe; otherwise the VBL task should not send the command. Note that you can't call `SCSIStat` before the System file fixes are in place. Since SCSI operations during VBL are not guaranteed, you should check all errors from SCSI Manager calls.

- A new SCSI Manager call will be added in the future. This will be a high-level call; it will have some kind of parameter block in which you give a pointer to a command buffer, a pointer to your TIB, a pointer to a sense data buffer (in case something goes wrong, the SCSI Manager will automatically read the sense bytes into the buffer for you), and a few other fields. The SCSI Manager will take care of arbitration, selection, sending the command, interpreting the TIB for the data transfer, and getting the status and message bytes (and the sense bytes, if there was an error). It should make SCSI device drivers much easier to write, since the driver will no longer have to worry about unexpected phase changes, getting the sense bytes, and so on. In the future, this will be the recommended way to use the SCSI Manager.

- The SCSI Manager (all machines) does not currently support interrupt-driven (asynchronous) operations. The Macintosh Plus can never support it since there is no interrupt capability, although a polled scheme may be implemented by the SCSI Manager. The Macintosh SE has a maskable interrupt for `IRQ`, and the Macintosh II has maskable interrupts for both `IRQ` and `DRQ`. Apple is working on an implementation of the SCSI Manager that will support asynchronous operations on the Macintosh II and probably on the SE as well. Because the interrupt hardware will interact adversely with any asynchronous schemes that are polled, it is strongly recommended that third parties do not attempt asynchronous operations until the new SCSI Manager is released. Apple will not attempt to be compatible with any products that bypass some or all of the SCSI Manager. In order to implement software-based (polled) asynchronous operations it is necessary to bypass the SCSI Manager.

The SCSI Manager section of the alpha draft of *Inside Macintosh* volume V documented the Disconnect and Reselect routines which were intended to be used for asynchronous I/O. Those routines cannot be used. Those routines have been removed from the manual. Any software that uses those routines will have to be revised when the SCSI Manager becomes interrupt-driven. Drivers which send SCSI commands from VBL tasks may also have to be modified.

Hardware in the SCSI

There is some confusion on how many terminators can be used on the bus, and the best way to use them. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of the wire that goes through all of the devices. One terminator should be on the end of the system cable that comes out of the Macintosh. The other terminator would be on the very end of the last device on the chain. If you have an SE or II with an internal hard disk, there is already one terminator on the front of the chain, inside the computer.

On the Macintosh SE and II, there is additional hardware support for the SCSI bus transfers in pseudo-DMA mode. The hardware makes it possible to handshake the data in Blind mode so that the Blind mode is safe for all transfers. On the Macintosh Plus, the Blind transfers are heavily timing dependent and can overrun or underrun during the transfer with no error generated. Assuring that Blind mode is safe on the Macintosh Plus depends upon the peripheral being used. On the SE and II, the transfer is hardware assisted to prevent overruns or underruns.

Changes in SCSI for SE and II

The changes made to the SCSI Manager found in the Macintosh SE and Macintosh II are primarily bug fixes. No new functionality was added. The newer SCSI Manager is more robust and has more error checking. Since the Macintosh Plus SCSI Manager only did limited error checking, it is possible to have code that would function (with bugs) on the Macintosh Plus, but will not work correctly on the SE or II. The Macintosh Plus could mask some bugs in the caller by not checking errors. An example of this is sending or receiving the wrong number of bytes in a blind transfer. On the Macintosh Plus, no error would be generated since there was no way to be sure how many bytes were sent or received. On the SE and II, if the wrong number of bytes are transferred an error will be returned to the caller. The exact timing of transfers has changed on the SE and II as well, since the computers run at different speeds. Devices that are unwittingly dependent upon specific timing in transfers may have problems on the newer computers. To find problems of this sort it is usually only necessary to examine the error codes that are passed back by the SCSI Manager routines. The error codes will generally point out where the updated SCSI Manager found errors.

To report other bugs or make suggestions

Please send additional bug reports and suggestions to us at the address in Technical Note #0. Let us know what SCSI controller you're using in your peripheral, and whether you've had any particularly good or bad experiences with it. We'll add to this note as more information becomes available.

#97: PrSetError Problem

Written by: Mark Baumwell November 15, 1986
Updated: March 1, 1988

This note formerly described a problem in Lisa Pascal glue for the `PrSetError` routine. The glue in MPW (and most, if not all, third party compilers) does not have this problem.

#98: Short-Circuit Booleans in Lisa Pascal

Written by: Mark Baumwell November 15, 1986
Updated: March 1, 1988

This note formerly described problems with the Lisa Pascal compiler. These problems have been fixed in the MPW Pascal compiler.

#99: Standard File Bug in System 3.2

See also: The Standard File Package

Written by: Jim Friedlander November 15, 1986
Updated: March 1, 1988

This note formerly described a bug in Standard File in System 3.2. This bug has been fixed in more recent Systems.

#100: Compatibility with Large-Screen Displays

See also: Technical Note #2—Macintosh Compatibility Guidelines

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

A number of third-party developers have announced large-screen display peripherals for Macintosh. One of them, Radius Inc., has issued a set of guidelines for developers who wish to remain compatible with their Radius FPD; unfortunately, one of their recommendations can cause system crashes. This note suggests a more correct approach.

On the first page of the appendix to their guidelines, "How to be FPD Aware," Radius recommends the following:

> "First, to detect the presence of a Radius FPD, you should check address `$C00008...`"

Unfortunately, this assumes that you're running on a Macintosh or Macintosh Plus; this test will not work on Macintosh XL, nor on a Macintosh II. Since these displays weren't designed to work with systems other than Macintosh and Macintosh Plus, you should make sure you're running on one of these systems before addressing I/O locations (such as those for an add-on display).

Before testing for the presence of any large-screen display, you should first check the machine ID; it's the byte located at `(ROMBASE)+8` (that is, take the long integer at the low-memory location `ROMBASE` [`$2AE`], and add 8 to get the address of the machine ID byte. On a Macintosh or Macintosh Plus, this address will work out to be `$400008`; however, use the low-memory location, to be compatible with future systems that may have the ROM at a different address!).

The machine ID byte will be `$00` for all current Macintosh systems. If the value isn't `$00`, you can assume that no large-screen display is present, but don't forget to follow Technical Note #2's guidelines for screen size independence!

Note: If you are a developer of an add-on large-screen display, we'd be happy to review your guidelines for developers in advance of distribution; please send them to us at the address for comments in Technical Note #0. Future versions of this note may recommend general guidelines for dealing with add-on large-screen displays.

#101: CreateResFile and the Poor Man's Search Path

See also: The File Manager
The Resource Manager
Technical Note #77—HFS Ruminations

Written by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

CreateResFile checks to see if a resource file with a given name exists, and if it does, returns a dupFNErr (–48) error. Unfortunately, to do this check, CreateResFile uses a call that follows the Poor Man's Search Path (PMSP).

CreateResFile checks to see if a resource file with a given name exists, and if it does, returns a dupFNErr (–48) error. Unfortunately, to do the check, CreateResFile calls PBOpenRF, which uses the Poor Man's Search Path (PMSP). For example, if we have a resource file in the System folder named 'MyFile' (and no file with that name in the current directory) and we call CreateResFile('MyFile'), ResError will return a dupFNErr, since PBOpenRF will search the current directory first, then search the blessed folder on the same volume. This makes it impossible to use CreateResFile to create the resource file 'MyFile' in the current directory if a file with the same name already exists in a directory that's in the PMSP.

To make sure that CreateResFile will create a resource file in the current directory whether or not a resource file with the same name already exists further down the PMSP, call _Create (PBCreate or Create) before calling CreateResFile:

```
err := Create('MyFile',0,myCreator,myType);
            {0 for VRefNum means current volume/directory}
CreateResFile('MyFile');
err := ResError; {check for error}
```

In MPW C:

```
err = Create("\pMyFile",0,myCreator,myType);
CreateResFile("\pMyFile");
err = ResError();
```

This works because _Create does **not** use the PMSP. If we already have 'MyFile' in the current directory, _Create will fail with a dupFNErr, then, if 'MyFile' has an empty resource fork, CreateResFile will write a resource map, otherwise, CreateResFile will return dupFNErr. If there is no file named 'MyFile' in the current directory, _Create will create one and then CreateResFile will write the resource map.
Notice that we are intentionally ignoring the error from _Create, since we are calling it

only to assure that a file named 'MyFile' does exist in the current directory.

Please note that SFPutFile does **not** use the PMSP, but that FSDelete does. SFPutFile returns the vRefNum/WDRefNum of the volume/folder that the user selected. If your program deletes a resource file before creating one with the same name based on information returned from SFPutFile, you can use the following strategy to avoid deleting the wrong file, that is, a file that is not in the directory specified by the vRefNum/WDRefNum returned by SFPutFile, but in some other directory in the PMSP:

```
VAR
   wher   : Point;
   reply  : SFReply;
   err    : OSErr;
   oldVol : Integer;

...

   wher.h := 80; wher.v := 90;
   SFPutFile(wher,'','',NIL,reply);
   IF reply.good THEN BEGIN
       err := GetVol(NIL,oldVol); {So we can restore it later}
       err := SetVol(NIL,reply.vRefNum);{for the CreateResFile call}

   {Now for the Create/CreateResFile calls to create a resource file that
    we know is in the current directory}

       err := Create(reply.fName,reply.vRefNum,myCreator,myType);
       CreateResFile(reply.fName); {we'll use the ResError from this ...}

       CASE ResError OF
          noErr:{the create succeeded, go ahead and work with the new
                 resource file -- NOTE: at this point, we don't know
                 what's in the data fork of the file!!} ;
          dupFNErr: BEGIN {duplicate file name error}
              {the file already existed, so, let's delete it. We're now
               sure that we're deleting the file in the current directory}

                err:= FSDelete(reply.fName,reply.vRefNum);

              {now that we've deleted the file, let's create the new one,
               again, we know this will be in the current directory}

                err:= Create(reply.fName,reply.vRefNum,myCreator,myType);
                CreateResFile(reply.fName);
          END; {CASE dupFNErr}
          OTHERWISE     {handle other errors} ;
      END;              {Case ResError}
      err := SetVol(NIL,oldVol);{restore the default directory}
   END;                 {If reply.good}
...
```

In MPW C:

```
Point         wher;
SFReply       reply;
OSErr         err;
short         oldVol;


wher.h = 80; wher.v = 90;
SFPutFile(wher,"","",nil,&reply);
if (reply.good )
{
    err = GetVol(nil,&oldVol);
    /*So we can restore it later*/
    err = SetVol(nil,reply.vRefNum);/*for the CreateResFile call*/

    /*Now for the Create/CreateResFile calls to create a resource file
    that we know is in the current directory*/

    err = Create(&reply.fName,reply.vRefNum,myCreator,myType);
    CreateResFile(&reply.fName);
    /*we'll use the ResError from this ...*/

    switch (ResError())
    {
       case noErr:;/*the create succeeded, go ahead and work with the
                    new resource file -- NOTE: at this point, we don't
                    know what's in the data fork of the file!!*/
            break; /* case noErr*/
       case dupFNErr: /*duplicate file name error*/
                 /*the file already existed, so, let's delete it.
                 We're now sure that we're deleting the file in the
                 current directory*/

                 err= FSDelete(&reply.fName,reply.vRefNum);

                 /*now that we've deleted the file, let's create the
                 new one, again, we know this will be in the current
                 directory*/

                 err= Create(&reply.fName,reply.vRefNum,
                                          myCreator,myType);
                 CreateResFile(&reply.fName);
          break; /*case dupFNErr*/
       default:;    /*handle other errors*/
   }  /* switch */
   err = SetVol(nil,oldVol);/*restore the default directory*/
}                /*if reply.good*/
```

Note: `OpenResFile` uses the PMSP too, so you may have to adopt similar strategies to make sure that you are opening the desired resource file and not some other file further down the PMSP. This is normally not a problem if you use `SFGetFile`, since `SFGetFile` does not use the PMSP, in fact, `SFGetFile` does not open or close files, so it doesn't run into this problem.

#102: HFS Elucidations

See also: The File Manager
Technical Note #77—HFS Ruminations

Written by: Bryan “Bo3b” Johnson January 12, 1987
Updated: March 1, 1988

This technical note will describe a few problems that can occur while using HFS. It will also describe ways to avoid these problems.

This technical note will discuss the following problems:

1) It is very important to be careful about how files are opened and closed. There must be no more than one close for every open.

2) Don't use Driver names, like `.Bout`, `.Print` or `.Sony`, in place of file names or the file system will become confused.

3) Be aware of the `ioFlVersNum` byte in all file calls. A number of pieces of the Macintosh system do not use, and may in fact ignore, files created with non-zero `ioFlVersNums`.

Each of these can lead to strange occurrences, as well as problems for the users. Doing any or all of these marginally illegal operations will not necessarily lead to a System Error. In some cases the confusion generated may be worse than a System Error.

One Close is always enough

If a file is closed twice, it is possible to corrupt the file system on a disk. If a program has been creating unreadable disks, this may be the cause.

One aspect of the file system that is not well documented is how it allocates access paths to files that are currently open. As a result of this, it is possible to get a rather cavalier attitude about opening and closing files. This discussion will explain why it is necessary to be very careful about opening and closing files.

When the File Manager receives an `Open` call, it will look at the parameters passed in the parameter block and create a new access path for the file that is being opened. The access path is how the File Manager keeps track of where to send data that is written, and where to get data that is read from that file. An access path is nothing more than: 1) a buffer that the file system uses to read and write data, and 2) a File Control Block that describes how the file is stored on a disk.

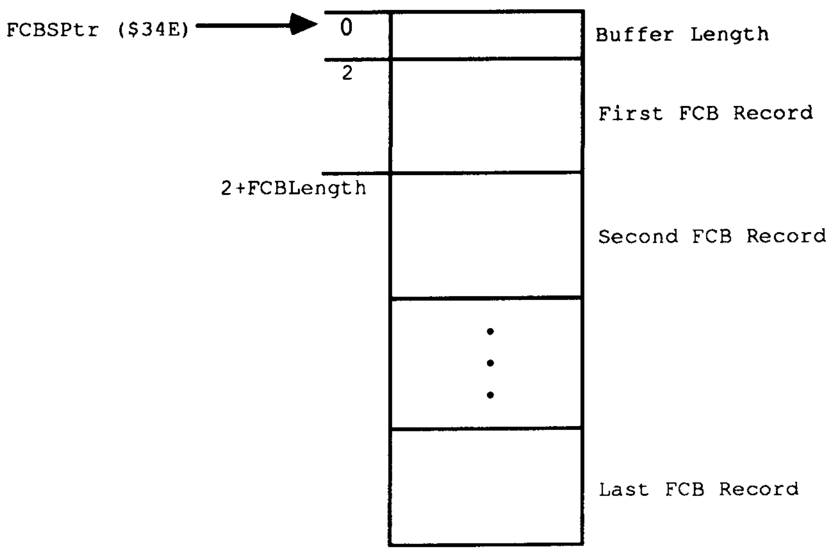
A call like:

```
     ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
```

will create the access path as a buffer and a File Control Block (FCB) in the FCB queue.

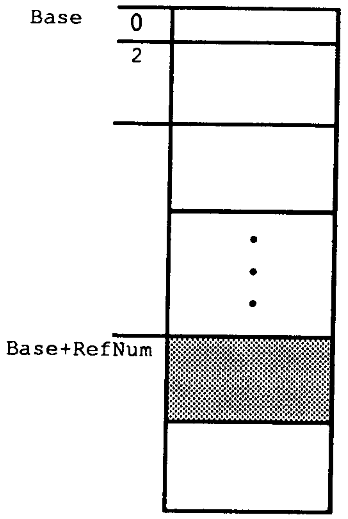
Note: The following information is here for illustrative purposes only; dependence on it may cause compatibility problems with future system software.

The structure of the queue can be visualized as:



where `FCBSPtr` is a low-memory global (at `$34E`) that holds the address of a nonrelocatable block. That block is the File Control Block buffer, and is composed of the two byte header which gives the length of the block, followed by the FCB records themselves. The records are of fixed length, and give detailed information about an open file. As depicted, any given record can be found by adding the length of the previous FCB records to the start of the block, adding 2 for the two byte header; giving an offset to the record itself. The size of the block, and hence the number of files that can be open at any given time, is determined at startup time. The call to open '`FirstFile`' above will pass back the File Reference Number to that file in `FirstRefNum`. This is the number that will be used to access that file from that point on. The File Manager passes back an offset into the FCB queue as the `RefNum`. This offset is the number of bytes past the beginning of the queue to that FCB record in the queue. That FCB record will describe the file that was opened. An example of a number that might get passed back as a `RefNum` is `$1D8`. That also means that the FCB record is `$1D8` bytes into the FCB block.

A visual example of a record being in use, and how the RefNum is related is:

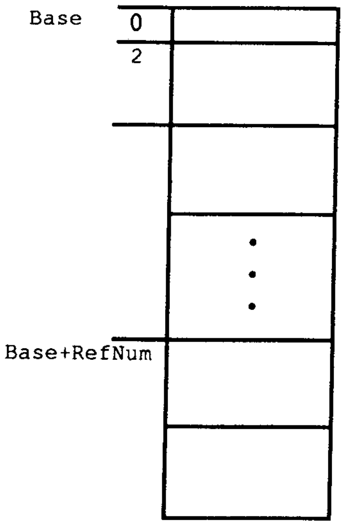


Base is merely the address of the nonrelocatable block that is the FCB buffer. FCBSPtr points to it. The RefNum (a number like $1D8) is added to Base, to give an address in the block. That address is what the file system will use to read and write to an open file, which is why you are required to pass the RefNum to the PBRead and PBWrite calls.

Since that RefNum is merely an offset into the queue, let's step through a dangerous imaginary sequence and see what happens to a given record in the FCB Buffer. Here's the sequence we will step through:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose ( FirstRefNum );
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose ( FirstRefNum ); {the wrong file gets closed!!!}
{the above line will close 'SecondFile', not 'FirstFile', which is already
 closed}
```

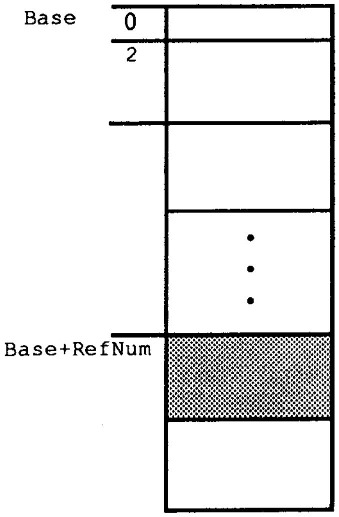
Before any operations:
the record at $1D8 is not used.



After the call:

`ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);`

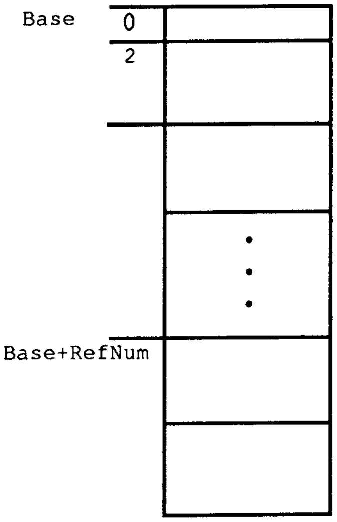
`FirstRefNum = $1D8` and the record is in use.



After the call:

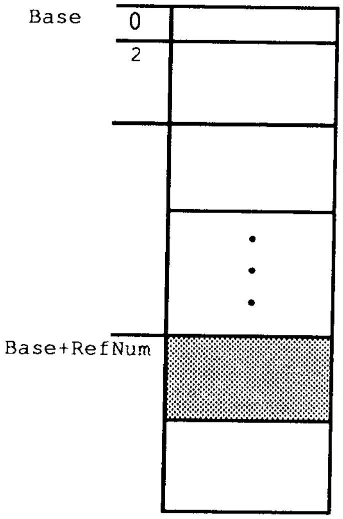
`ErrStuff := FSClose (FirstRefNum);`

`FirstRefNum is still equal to $1D8, but the FCB record is unused.`



After the call:

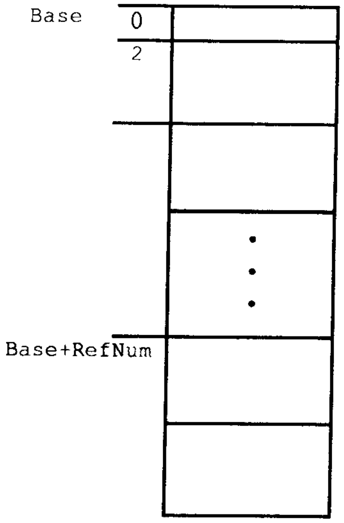
`ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);`
`SecondRefNum = $1D8, FirstRefNum = $1D8, and the record is reused.`



After the call:

`ErrStuff := FSClose (FirstRefNum);`
`The FirstRefNum = $1D8, SecondRefNum = $1D8,`

the queue element is cleared. This happens, even though `FirstFile` was already closed. Actually, `SecondFile` was closed:



Note that the second close is using the old `RefNum`. The second close will still close a file, and in fact will return `noErr` as its result. Any subsequent accesses to the `SecondRefNum` will return an error, since the file '`SecondFile`' was closed. The File Control Blocks are reused, and since they are just offsets, it is possible to get the same file `RefNum` back for two different files. In this case, `FirstRefNum = SecondRefNum` since '`FirstFile`' was closed before opening '`SecondFile`' and the same FCB record was reused for '`SecondFile`'.

There are worse cases than this, however. As an example, think of what can happen if a program were to close a file, then the user inserted an HFS disk. The FCB could be reused for the Catalog File on that HFS disk. If the program had a generic error handler that closed all of its files, it could inadvertently close "its" file again. If it thought "its" file was still open it would do the close, which could close the Catalog file on the HFS disk. This is catastrophic for the disk since the file could easily be closed in an inconsistent state. The result is a bad disk that needs to be reformatted.

There are any number of nasty cases that can arise if a file is closed twice, reusing an old `RefNum`. A common programming practice is to have an error handler or cleanup routine that goes through the files that a program creates and closes them all, even if some may already be closed. If an FCB element was not reused, the `Close` will return the expected `fnOpnErr`. If the FCB had been reused, then the `Close` could be closing the wrong file. This can be very dangerous, particularly for all those paranoid hard disk users.

How to avoid the problem:

A very simple technique is to merely clear the `RefNum` after each close. If the variable that the program uses is cleared after each close, then there is no way of reusing a `RefNum` in the program. An example of this technique would be:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose (FirstRefNum);
FirstRefNum := 0; { We just closed it, so clear our refnum }
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose (FirstRefNum); { returns an error }
```

This makes the second `Close` pass back an error. In this case, the second close will try to close `RefNum` = 0, which will pass back a `fnOpnErr` and do no damage. **Note:** Be sure to use 0, which will never be a valid `RefNum`, since the first FCB entry is beyond the FCB queue length word. Don't confuse this with the 0 that the Resource Manager uses to represent the System file.

Thus, if an error handler were cleaning up possibly open files, it could blithely close all the files it knew about, since it would legitimately get an error back on files that are already closed. This is not done automatically, however. The programmer must be careful about the opening and closing of files. The problem can get quite complex if an error is received halfway through opening a sequence of ten files, for example. By merely clearing the `RefNum` that is stored after each close, it is possible to avoid the complexities of trying to track which files are open and which are closed.

This .file name looks outrageous.

There is a potential conflict between file names and driver names. If a file name is named something like `.Bout`, `.Print` or `.Sony`, then the file system will open the driver instead of the file. Drivers have priority on the 128K ROMs, and will always be opened before a file of the same name. This may mean that an application will get an error back

when opening these types of files, or worse, it will get back a driver RefNum from the call. What the application thought was a file open call was actually a driver open call. If the program uses that access path as a file RefNum, it is possible to get all kinds of strange things to happen. For example, if .Sony is opened, the Sony driver's RefNum would be passed back, instead of a file RefNum. If the application does a Write call using that RefNum, it will actually be a driver call, using whatever parameters happen to be in the parameter block. Disks may be searching for new life after this type of operation. If a program creates files, it should not allow a file to be created whose name begins with '.'.

This file's not my type.

This has been discussed in other places, but another aspect of the File Manager that can cause confusion is the ioFlVersNum byte that is passed to the low-level File Manager calls. This is called ioFileType from Assembly, and should not be confused with ioFVersNum. This byte must be set to zero for normal Macintosh files. There are a number of parts of the system that will not deal correctly with files that have the wrong versions: the Standard File package will not display any file with a non-zero ioFlVersNum; the Segment Loader and Resource Manager cannot open files that have non-zero ioFlVersNums. It is not sufficient to ignore this byte when a file is created. The byte must be cleared in order to avoid this type of problem. Strictly speaking, it is not a problem unless a file is being created on an MFS disk. The current system will easily allow the user to access 400K disks however, so it is better to be safe than confused.

Macintosh Technical Notes

#103: Using MaxApplZone and MoveHHi from Assembly Language

See also: Using Assembly Language
The Memory Manager
Technical Note #129—SysEnvirons

Written by: Bryan "Bo3b" Johnson January 12, 1987
Updated: March 1, 1988

When calling MaxApplZone and MoveHHi from assembly language, be sure to get the correct code.

MaxApplZone and MoveHHi were marked [Not in ROM] in *Inside Macintosh, Volumes I-III* . They are ROM calls in the 128K ROM. Since they are not in the 64K ROM, if you want your program to work on 64K ROM routines it is necessary to call the routines by a JSR to a glue (library) routine instead of using the actual trap macro. The glue calls the ROM routines if they are available, or executes its copy of them (linked into your program) if not.

How to do it:

Whenever you need to use these calls, just call the library routine. It will check ROM85 to determine which ROMs are running, and do the appropriate thing.

For MDS, include the Memory.Rel library in your link file and use:

```
XREF   MoveHHi     ; we need to use this 'ROM' routine
...
JSR   MoveHHi      ; jump to the glue routine that will check ROM85 for us
```

For MPW link with Interface.o and use:

```
IMPORT  MoveHHi    ; we need to use this
...
JSR   MoveHHi      ; jump to the glue routine that will check ROM85 for us
```

Avoid calling _MaxApplZone or _MoveHHi directly if you want your software to work on the 64K ROMs, since that will assemble to an actual trap, not to a JSR to the library.

If your program is going to be run **only** on machines with the 128K ROM or newer, you can call the traps directly. Be sure to check for the 64K ROMs, and report an error to the user. You can check for old ROMs using the SysEnvirons trap as described in Technical Note #129.

Macintosh Technical Notes

#104: MPW: Accessing Globals From Assembly Language

See also: MPW Reference Manual

Written by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note demonstrates how to access MPW Pascal and MPW C globals from the MPW Assembler.

To allow access of MPW Pascal globals from the MPW Assembler, you need to identify the variables that you wish to access as external. To do this, use the `{$Z+}` compiler option. Using the `{$Z+}` option can substantially increase the size of the object file due to the additional symbol information (no additional code is generated and the symbol information is stripped by the linker). If you are concerned about object file size, you can "bracket" the variables you wish to access as external variables with `{$Z+}` and `{$Z-}`. Here's a trivial example:

Pascal Source

```
PROGRAM MyPascal;
USES
   MemTypes,QuickDraw,OSIntf,ToolIntf;

VAR
   myWRect: Rect;
{$Z+} {make the following external}
   myInt: Integer;
{$Z-} {make the following local to this file (not lexically local)}
   err: Integer;

PROCEDURE MyAsm; EXTERNAL; {routine doubles the value of myInt}

BEGIN {PROGRAM}
   myInt:= 5;
   MyAsm; {call the routine, myInt will be 10 now}
   writeln('The value of myInt after calling myAsm is ', myInt:1);
END. {PROGRAM}
```

Assembly Source for Pascal

```
        CASE   OFF          ;treat upper and lower case identically
MyAsm   PROC   EXPORT       ;CASE OFF is the assembler's default
        IMPORT myInt:DATA   ;we need :DATA, the assembler assumes CODE
        ASL.W  #1,myInt     ;multiply by two
        RTS                 ;all done with this extensive routine, whew!
        END
```

The variable myInt is accessible from assembler. Neither myWRect nor err are accessible. If you try to access myWRect, for example, from assembler, you will get the following linker error:

```
### Link: Error   Undefined entry name:   MYWRECT.
```

C Source

In an MPW C program, one need only make sure that MyAsm is declared as an external function, that myInt is a global variable (capitalizations must match) and that the CASE ON directive is used in the Assembler:

```
#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <windows.h>
#include <events.h>
#include <textedit.h>
#include <dialogs.h>
#include <stdio.h>

extern MyAsm();     /* assembly routine that doubles the value of myInt */
short myInt;       /* we'll change the value of this variable from MyAsm */

main()
{
WindowPtr MyWindow;
Rect myWRect;

myInt = 5;
MyAsm();
printf(" The value of myInt after calling myAsm is %d\n",myInt);
} /*main*/
```

Assembly source for C

```
        CASE   ON            ;treat upper and lower case distinct
MyAsm   PROC   EXPORT        ;this is how C treats upper and lower case
        IMPORT myInt:DATA    ;we need :DATA, the assembler assumes CODE
        ASL.W  #1,myInt      ;multiply by two
        RTS                  ;all done with this extensive routine, whew!
        END
```

Macintosh Technical Notes

#105: MPW Object Pascal Without MacApp

See also: Technical Note #93—{$LOAD};_DataInit;%_MethTables

Written by: Rick Blair — January 12, 1987
Updated: March 1, 1988

Object Pascal must have a CODE segment named `%_MethTables` in order to access object methods. In MacApp this is taken care of "behind the scenes" so you don't have to worry about it . However, if you are doing a straight Object Pascal program, you must make sure that `%_MethTables` is around when you need it. If it's unloaded when you call a method, your Macintosh will begin executing wild noncode and die a gruesome and horrible death.

The MPW Pascal compiler must see some declaration of an object in order to produce a reference to the magic segment. You can achieve this cheaply by simply including `ObjIntf.p` in your `Uses` declaration. This must be in the main program, by the way. The compiler will produce a call to `%_InitObj` which is in `%_MethTables`.

If you're a more adventurous soul, you can call `%_InitObj` explicitly from the initialization section of your main program (you must use the `{$%+}` compiler directive to allow the use of "`%`" in identifiers). This will load the `%_MethTables` segment. See Technical Note #93 for ideas about locking down segments that are needed forever without fragmenting the heap.

#106: The Real Story: VCBs and Drive Numbers

See also: The File Manager
Technical Note #36—Drive Queue Element Format

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

The top of page IV-178 in The File Manager chapter of *Inside Macintosh* in attempts to explain the behavior of two fields in a volume control block when the corresponding disk is offline or ejected. Due to the fact that a little bit is left unsaid, this paragraph is rather misleading. The two fields in question are `vcbDrvNum` and `vcbDRefNum` (referred to as `ioVDrvInfo` and `ioVDRefNum` in C and Pascal). `PBHGetVInfo` can be used to access these fields.

Offline

When a mounted volume is placed offline, `vcbDrvNum` is cleared **and** `vcbDRefNum` is set to the two's complement of the drive number. Since drive numbers are assigned positive values (starting with one), this will be a negative number. If `vcbDrvNum` is zero **and** `vcbDRefNum` is negative, you know that the volume is offline.

Ejected

When a volume is ejected, `vcbDrvNum` is cleared and `vcbDRefNum` is set to the positive drive number. If `vcbDrvNum` is zero and `vcbDRefNum` is positive, you know that the volume is ejected. Ejection implies being offline. There is no such thing as "premature ejection".

Summary

	online	offline	ejected
vcbDrvNum	>0 (DrvNum)	0	0
vcbDRefNum	<0 (DRefNum)	<0 (-DrvNum)	>0 (DrvNum)

Please refrain from assuming anything about a VCB queue element beyond what is documented in *Inside Macintosh*, and don't expect it to always be 178 bytes in size. It grew when we went from MFS to HFS, and it may grow again. It's safest to use calls like `PBHGetVInfo` to get the information that you need.

Macintosh Technical Notes

#107: Nulls in Filenames

See also: The File Manager

Written by: Rick Blair — March 2, 1987
Updated: March 1, 1988

Some applications (loosely speaking so as to include Desk Accessories, INITs, and what-have-you) generate or rename special files on the fly so that they are not explicitly named by the user via `SFPutFile`. Since the Macintosh file system is very liberal about filenames and only excludes colons from the list of acceptable characters, this can lead to some difficulties, both for the end user and for writers of other programs which may see these files.

Other programs which might be backing up your disk or something similar may get confused. A program written in C will think it has found the end of a string when it hits a null (ASCII code 0) character, so nulls in filenames are especially risky.

As a rule, filenames should only include characters which the user can see and edit. The only reasonable exception might be invisible files, but it can be argued that they are of dubious value anyway. You can argue "but what about my help file, I don't want it renamed" but we already have what we think is the best approach for that situation. If you can't find a configuration or other file because the user has renamed or moved it, then call `SFGetFile` and let the user find it. If the user cancels, and you can't run without the file, then quit with an appropriate message.

Please consider carefully before you put non-displaying characters in filenames!

Macintosh Technical Notes

#108: _AddDrive, _DrvrInstall, and _DrvrRemove

See also: Technical Note #36, Drive Queue Elements
SCSI Development Package (APDA)

Written by: Jim Friedlander March 2, 1987
Revised by: Pete Helme December 1988

_AddDrive, _DrvrInstall, and _DrvrRemove are used in the sample SCSI driver in the SCSI Development Package, which is available from APDA. This Technical Note documents the parameters for these calls.
Changes since March 1, 1988: Updated the _DrvrInstall text to reflect the use of register A0, which should contain a pointer to the driver when called. Also added simple glue code for _DrvrInstall and _DrvrRemove since none is available in the MPW interfaces.

_AddDrive

_AddDrive adds a drive to the drive queue, and is discussed in more detail in Technical Note #36, Drive Queue Elements:

```
FUNCTION AddDrive(DQE:DrvQEl;driveNum,refNum:INTEGER):OSErr;
```

A0 (input)	→	pointer to DQE
D0 high word(input)	→	drive number
D0 low word(input)	→	driver RefNum
D0 (output)	←	error code
		noErr (always returned)

_DrvrInstall

_DrvrInstall is used to install a driver. A DCE for the driver is created and its handle entered into the specified Unit Table position (–1 through –64). If the unit number is –4 through –9, the corresponding ROM-based driver will be replaced:

```
FUNCTION DrvrInstall(drvrHandle:Handle; refNum: INTEGER): OSErr;
```

A0 (input)	→	pointer to driver
D0 (input)	→	driver RefNum (–1 through –64)
D0 (output)	←	error code
		noErr
		badUnitErr

_DrvrRemove

_DrvrRemove is used to remove a driver. A RAM-based driver is purged from the system heap (using _ReleaseResource). Memory for the DCE is disposed:

```
FUNCTION DrvrRemove(refNum: INTEGER):OSErr;
```

D0 (input)	→	Driver RefNum
D0 (output)	←	error code noErr qErr

Interfaces

Through a sequence of cataclysmic events, the glue code for _DrvrInstall and _DrvrRemove was never actually added to the MPW interfaces (i.e., "We forgot."), so we will include simple glue here at no extra expense to you.

It would be advisable to first lock the handle to your driver with _HLock before making either of these calls since memory may be moved.

```
;-------------------------------------------------------------
; FUNCTION DRVRInstall(drvrHandle:Handle; refNum:INTEGER):OSErr;
;-------------------------------------------------------------

DRVRInstall    PROC    EXPORT
        MOVEA.L        (SP)+, A1        ; pop return address
        MOVE.W         (SP)+, D0        ; driver reference number
        MOVEA.L        (SP)+, A0        ; handle to driver
        MOVEA.L        (A0), A0         ; pointer to driver
        _DrvrInstall                    ; $A03D
        MOVE.W         D0, (SP)         ; get error
        JMP            (A1)             ; & split
        ENDPPROC


;-------------------------------------------------------------
; FUNCTION DRVRRemove(refNum:INTEGER):OSErr;
;-------------------------------------------------------------

DRVRRemove     PROC    EXPORT
        MOVEA.L        (SP)+, A1        ; pop return address
        MOVE.W         (SP)+, D0        ; driver reference number
        _DrvrRemove                     ; $A03E
        MOVE.W         D0, (SP)         ; get error
        JMP            (A1)             ; & split
        ENDPPROC
```

#109: Bug in MPW 1.0 Language Libraries

See also: MPW Reference Manual

Written by: Scott Knaster March 2, 1987
Updated: March 1, 1988

This note formerly described a problem in the language libraries for MPW 1.0. This bug is fixed in MPW 1.0.2, available from APDA.

#110: MPW: Writing Stand-Alone Code

Revised by: Keith Rollin August 1990
Written by: Jim Friedlander March 1987

This Technical Note formerly discussed using MPW Pascal and C to write stand-alone code, such as `'WDEF'`, `'LDEF'`, `'INIT'`, and `'FKEY'` resources.
Changes since February 1990: Merged the contents of this Note into Technical Note #256, Stand-Alone Code, *ad nauseam.*

This Note formerly discussed using MPW Pascal and C to write stand-alone code. This information has been expanded and is now contained in Technical Note #256, Stand-Alone Code, *ad nauseam.*

Macintosh Technical Notes

#111: MoveHHi and SetResPurge

See also: The Memory Manager
The Resource Manager

Written by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

`SetResPurge(TRUE)` is called to make the Memory Manager call the Resource Manager before purging a block specified by a handle. If the handle is a handle to a resource, and its `resChanged` bit is set, the resource data will be written out (using `WriteResource`).

When `MoveHHi` is called, even though the handle's block is not actually being purged, the resource data specified by the handle will be written out. An application can prevent this by calling `SetResPurge(FALSE)` before calling `MoveHHi` (and then calling `SetResPurge(TRUE)` after the `MoveHHi` call).

#112: FindDItem

See also: The Dialog Manager

Written by: Rick Blair March 2, 1987
Updated: March 1, 1988

`FindDItem` is a potentially useful call which returns the number of a dialog item given a point in local coordinates and a dialog handle. It returns an item number of –1 if no item's rectangle overlaps the point. This is all well and good, except you don't get back quite what you would expect.

The item number returned is zero-based, so you have to add one to the result:

```
     theitem := FindDItem(theDialog, thePoint) + 1;
```

Macintosh Technical Notes

#113: Boot Blocks

See also: The Segment Loader

Written by: Bo3b Johnson March 2, 1987
Updated: March 1, 1988

There are two undocumented features of the Boot Blocks. This note will describe how they currently work.

Warning: The format and functionality of the Boot Blocks will change in the future; dependence on this information may cause your program to fail on future hardware or with future System software.

The first two sectors of a bootable Macintosh disk are used to store information on how to start up the computer. The blocks contain various parameters that the system uses to startup such as the name of the system file, the name of the Finder, the first application to run at boot time, the number of events to allow, etc.

Changing System Heap Size

The boot blocks dictate what size the system heap will be after booting. Any common sector editing program will allow you to change the data in the boot blocks. Changing the system heap size is accomplished by changing two parameters in the boot blocks: the long word value at location `$86` in Block 0 indicates the size of the system heap; the word value at location `$6` is the version number of the boot blocks. Changing the version number to be greater than `$14` (`$15` is recommended) tells the ROM to use the value at `$86` for the system heap size, otherwise the value at `$86` is ignored. The `$86` location only applies to computers with more than 128K of RAM.

Secondary Sound and Video Pages

Another occasionally useful feature of the boot blocks is the ability to specify that the secondary sound and video pages be allocated at boot time. This is done before a debugger is loaded, so the debugger will load below the alternate screen. This is useful for debugging software that uses the alternate video page, like page-flipping demos or games. To allocate the second video and sound buffers, change the two bytes starting at location `$8` in the boot blocks. Change the value (normally 0) to a negative number (`$FFFF`) to allocate both video and sound buffers. Change the value to a positive number (`$0001`) to allocate only the secondary sound buffer.

Warning: MacsBug may not work properly if you allocate additional pages for sound and video.

#114: AppleShare and Old Finders

See also: *AppleShare User's Guide*

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

A rumor has been spread that if you use a pre-AppleShare Finder on a workstation to access AppleShare volumes, you can bypass AppleShare's "access privilege" mechanisms.

This is not true. Access controls are enforced by the server, **not** by the Finder. If you use an older Finder, you are still prevented (by the server) from gaining access to protected files and folders; however, you will not get the proper user-interface feedback that you would if you were using the correct Finder: for instance, folders on the server will always appear plain white (that is, without the permission feedback you'd normally get), and error messages would not be as explanatory as those from Finders that "know" about AppleShare servers.

#115: Application Configuration with Stationery Pads

See also: The File Manager
Technical Note #116—AppleShare-able Applications
Technical Note #47—Customizing SFGetFile
Technical Note #48—Bundles
"Application Development in a Shared Environment"

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

With the introduction of AppleShare (Apple's file server) there are restrictions on self-modification of application resource files and the placement of configuration files. This note describes one way to get around the necessity for configuration files.

Configuration Files

Some applications need to store information about configuration; others could benefit simply from allowing users to customize default ruler settings, window placement, fonts, etc.

There are applications which store this information as additional resources in the application's resource file; when the user changes the configuration, the application writes to itself to change the saved information.

AppleShare, however, requires that if an application is to be used by more than one user at a time, it must not need write access to itself. This means that the above method of storing configuration information cannot be used. (For more information about making your application sharable, see Technical Note #116.)

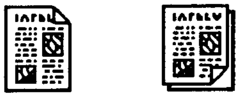
Storing configuration in a special configuration file can be a problem; the user must keep the file in the system folder or the application must search for it. This process has design issues of its own.

An alternative to configuration files: Stationery Pads

A basis for one solution to this problem was a user-interface feature of the Lisa Office System architecture. Lisa introduced the concept of "stationery pads", special documents that created copies of themselves to allow users to save a pre-set-up document for future use. On Lisa, this was the way Untitled documents were created.

Your Macintosh application can provide the option of saving a document as a stationery pad, to provide similar functionality. Here's how:

- You'll need to add a checkbox to your `SFPutFile` dialog box (if you don't know how to do this, check out Technical Note #47); if the user checks this box, save the document as you normally would, but use a different file type (the file type of a document is usually set when the document is created, using the File Manager `Create` procedure, or later using `SetFileInfo`).



A Document and its Stationery pad

- Be sure to use a different but similar icon for the stationery pad file. This is easy if you differentiate between stationery and normal files solely by file type—the Finder uses the type to determine which icon to display, see Technical Note #48 for help with the "bundle" mechanism used to associate a file type with an icon.

- When opening a stationery pad file, the window should come up named "Untitled", with the contents of the stationery pad file.

- "Revert" should re-read the stationery pad file.

- Don't forget to add the stationery pad's file type to the file-types list that you pass to Standard File, so that the new files will appear in the list when the user chooses Open. This file type should be registered with Macintosh Developer Technical Support.

#116: AppleShare-able Applications and the Resource Manager

See also: The Resource Manager
"Application Development in a Shared Environment"
Technical Note #40—Finder Flags

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

Normally, applications on an AppleShare server volume cannot be executed by more than one user at a time. This technical note explains why, and tells how you can enable your application to be shared.

The Resource Manager versus Shared Files

Part of the explanation of why applications are not automatically sharable is based on the design of the Resource Manager. The Resource Manager is a great little database. It was originally conceived as a way to keep applications localizable (a task it has performed admirably), and was found to be an excellent foundation for the Segment Loader, Font Manager, and a large part of the rest of the Macintosh operating system.

However, it was never designed to be a multi-user database. When the Resource Manager opens a resource file (such as an application), it reads the file's resource map into memory. This map remains in memory until the resource file is closed by the Segment Loader, which regains control when the application exits. Sometimes it is necessary to write the map out to disk; normally, this is only done by `UpdateResFile` and `CloseResFile`.

If two users opened the same resource file at the same time, and one of them had write access to the file and added a resource to it, the other user's Resource Manager wouldn't know about it; this would make the other user's copy of the file's original resource map invalid. This could cause (at least) a crash; if both users had write access, it's not unlikely that the resource file involved would become corrupted. Also, although you can tell the Resource Manager to write out an updated resource map, there's no way for another user to tell it to refresh the copy of the map in memory if the file changes.

What does all this have to do with running my application twice?

Your application is stored as a resource file; code segments, alert and dialog templates, etc., are resources. If you write to your application's resource file (for instance, to add configuration information, like print records), your application can't be shared.

In Apple's compatibility testing of existing applications (during development of AppleShare), we found quite a few applications, some of them quite popular, that wrote to their own resource files. So we decided, to improve the safety of using AppleShare, to always launch applications using a combination of access privileges such that only one user at a time could use a given application (these privileges will be discussed in a future Technical Note). In fact, AppleShare opens all resource files this way, unless the resource file is opened with `OpenRFPerm` and read-only permission is specified.

But my application doesn't write to itself!

We realize that many applications do not. However, there are other considerations (covered in detail, with suggestions for fixes, in "Application Development in a Shared Environment", available from APDA). In brief, here are the big ones we know about:

- Does your application create temporary files with fixed names in a fixed place (such as the directory containing the application)? Without AppleShare's protection, two applications trying to use the same temporary file could be disastrous.

- Is your application at least "conscious" of the fact that it may be in a multi-user environment? For instance, does it work correctly if a volume containing an existing document is on a locked volume? Does it check all result codes returned from File Manager calls, and `ResError` after relevant Resource Manager calls?

OK, I follow the rules. What do I do to make my application sharable?

There is a flag in each file's Finder information (stored in the file's directory entry) known as the "shared" bit. If you set this bit on your application's resource file, the Finder will launch your application using read-only permissions; if anyone else launches your application, they'll also get it read-only (their Finder will see the same "shared" bit set.).

Three important warnings accompany this information:

- The definition of the "shared" bit was incorrect in previous releases of information and software from Apple. This includes the June 16, 1986 version of Technical Note #40 (fixed in the March 2, 1987 version), as well as all versions of ResEdit before and including 1.1b3 (included with MPW 2.0). For now, the most reliable way to set this bit is to get the 1.1b3 version of ResEdit, use it to Get Info on your application, and check the box labeled "cached" (the incorrect documentation upon which ResEdit [et al.] was based called the real shared bit "cached"; the bit labeled as "shared" is the real cached bit [a currently unused but reserved bit which should be left clear]).

- By checking this bit, you're promising (to your users) that your application will work entirely correctly if launched by more than one user. This means that you follow the other rules, in addition to simply not writing to your application's own resource file. See "Application Development for a Shared Environment," and test carefully!

- Setting this bit has nothing to do with allowing your application's documents to be shared; you must design this feature into your application (it's not something that Apple system software can take care of behind your application's back.). You should realize from reading this note, however, that if you store your document's data in resource files, you won't be able to allow multiple users to access them simultaneously.

#117: Compatibility: Why & How

See Also: Technical Note #2—Compatibility Guidelines
Technical Note #7—A Few Quick Debugging Tips

Written by: Bo3b Johnson February 9, 1987
Updated: March 1, 1988

While creating or revising any program for the Macintosh, you should be aware of the most common reasons why programs fail on various versions of the Macintosh. This note will detail some common failure modes, why they occur, and how to avoid them.

We've tried to explain the issues in depth, but recognize that not everyone is interested in every issue. For example, if your application is not copy protected, you're probably not very interested in the section on copy protection. That's why we've included the outline form of the technical note. The first two pages outline the problems and the solutions that are detailed later. Feel free to skip around at will, but remember that we're sending this enormous technical note because the suggestions it provides may save you hasty compatibility revisions when we announce a new machine.

We know it's a lot, and we're here to help you if you need it. Our address (electronic and physical) is on page three—contact us with **any** questions—that's what we're here for!

Compatibility: the outline

Don't assume the screen is a fixed size

To get the screen size:

- check the QuickDraw global `screenBits.bounds`

Don't assume the screen is in a fixed location

To get the screen location:

- check the QuickDraw global `screenBits.baseAddr`

Don't assume that `rowBytes` is equal to the width of the screen

To get the number of bytes on a line:

- check the QuickDraw global `screenBits.rowBytes`

To get the screen width:

- check the QuickDraw global `screenBits.bounds.right`

To do screen-size calculations:

- Use `LongInts`

Don't write to or read from `nil` Handles or `nil` Pointers

Don't create or Use Fake Handles

To avoid creating or using fake handles:

- Always let the Memory Manager perform operations with handles
- Never write code that assigns something to a master pointer

Don't write code that modifies itself

Self modifying code will not live across incarnations of the 68000

Think carefully about code designed strictly as copy protection

To avoid copy protection-related incompatibilities:

- Avoid copy protection altogether
- Rely on schemes that don't require specific hardware
- Make sure your scheme doesn't perform illegal operations

Don't ignore errors

To get valuable information:

- Check all pertinent calls for errors
- Always write defensive code

Don't access hardware directly

To avoid hardware-related incompatibilities:

- Don't read or write the hardware
- If you can't get the support from the ROM, ask the system where the hardware is
- Use low-memory globals

Don't use bits that are reserved

To avoid compatibility problems when bit status changes:

- Don't use undocumented stuff
- When using low-memory globals, check only what you want to know

Summary

Minor bugs are getting harder and harder to get away with:

- Good luck
- We'll help
- AppleLink: MacDTS, MCI: MacDTS
- U.S. Mail: 20525 Mariani Ave.; M/S 27-T; Cupertino, CA 95014

What it Is

The basic idea is to make sure that your programs will run, regardless of which Macintosh they are being run on. The current systems to be concerned with include:

- Macintosh 128K
- Macintosh 512K
- Macintosh XL
- Macintosh 512Ke
- Macintosh Plus
- Macintosh SE
- Macintosh II

If you perform operations in a generic fashion, there is rarely any reason to know what machine is running. This means that you should avoid writing code to determine which version of the machine you are running on, unless it is absolutely necessary.

For the purposes of this discussion, the term "programs" will be used to describe any code that runs on a Macintosh. This includes applications, INITs, FKEYs, Desk Accessories and Drivers.

What the "Rules" mean

Compatibility across all Macintosh computers (which may sound like it involves more work for you) may actually mean that you have less work to do, since it may not be necessary to revise your program each time Apple brings out a new computer or System file. Users, as a group, do not understand compatibility problems; all they see is that the program does not run on their system.

The benefits of being compatible are many-fold: your customers/users stay happy, you have less programming to do, you can devote your time to more valuable goals, there are fewer versions to deal with, your code will probably be more efficient, your users will not curse you under their breath, and your outlook on life will be much merrier.

Now that we know what being compatible is all about, recognize that nobody is requiring you to be compatible with anything. Apple does not employ roving gangs of thought police to be sure that developers are following the recommended guidelines. Furthermore, when the guidelines comprise 1200 pages of turgid prose (*Inside Macintosh*), you can be expected to miss one or two of the "rules." It is no sin to be incompatible, nor is it a punishable offense. If it were, there would be no Macintosh programs, since virtually all developers would be incarcerated. What it does mean, however, is that your program will be unfavorably viewed until it steps in line with the current system (which is a moving target). If a program becomes incompatible with a new Macintosh, it usually requires rethinking the offending code, and releasing a new version. You may read something like "If the developers followed Apple guidelines, they would be compatible with the transverse-hinged diatomic quark realignment system." This means that if you made any mistakes (you read all 1200 pages carefully, right?), you will not be compatible. It is extremely difficult to remain completely compatible, particularly in a system as complex as the Macintosh. The rules haven't changed, but what you can get away with has. There are, however, a number of things that you can do to improve your odds—some of which will be explained here.

It's your choice

It is still your choice whether you will be concerned with compatibility or not. Apple will not put out a warrant for your arrest. However, if you are doing things that are specifically illegal, Apple will also not worry about "breaking" your program.

Bad Things

The following list is not intended to be comprehensive, but these are the primary reasons why programs break from one version of the system to the next. These are the current top ten commandments:

I Thou shalt not assume the screen is a fixed size.
II Thou shalt not assume the screen is at a fixed location.
III Thou shalt not assume that `rowBytes` is equal to the width of the screen.
IV Thou shalt not use `nil` handles or `nil` pointers.
V Thou shalt not create or use fake handles.
VI Thou shalt not write code that modifies itself.
VII Thou shalt think twice about code designed strictly as copy protection.
VIII Thou shalt check errors returned as function results.
IX Thou shalt not access hardware directly.
X Thou shalt not use any of the bits that are reserved (unused means reserved).

This has been determined from extensive testing of our diverse software base.

Assuming the screen is a fixed size

Do not assume that the Macintosh screen is 512 x 342 pixels. Programs that do generally have problems on (or special case for) the Macintosh XL, which has a wider screen. Most applications have to create the bounding rectangle where a window can be dragged. This is the `boundsRect` that is passed to the call:

```
    DragWindow (myWindowPtr, theEvent.where, boundsRect);
```

Some ill-advised programs create the `boundsRect` by something like:

```
    SetRect (boundsRect, 0,0,342,512);  { oops, this is hard-coded…}
```

Why it's Bad

This is bad because it is **never** necessary to specifically put in the bounding rectangle for the screen. On a Macintosh XL for example, the screen size is 760x364 (and sometimes 608x431 with alternate hardware). If a program uses the hard-coded 0,0,342,512 as a bounding rectangle, end users will not be able to move their windows past the fictitious boundary of 512. If something similar were done to the `GrowWindow` call, it would make it impossible for users to grow their window to fill the entire screen. (Always a saddening waste of valuable screen real-estate.)

Assuming screen size makes it more difficult to use the program on Macintoshes with big screens, by making it difficult to grow or move windows, or by drawing in strange places where they should not be drawing (outside of windows). Consider the case of running on a Macintosh equipped with one of the full page displays, or Ultra-Large screens. No one who paid for a big screen wants to be restricted to using only the upper-left corner of it.

How to avoid becoming a screening fascist

Never hard code the numbers 512 and 342 for screen dimensions. You should avoid using constants for system values that can change. Parameters like these are nearly always available in a dynamic fashion. Programs should read the appropriate variables while the program is running (at run-time, not at compile time).

Here's how smart programs get the screen dimensions:

```
    InitGraf(@thePort); { QuickDraw global variables have to be initialized.}
    …
    boundsRect := screenBits.bounds;    { The Real way to get screen size }
                                          { Use QuickDraw global variable. }
```

This is smart, because the program never has to know specifically what the numbers are. All references to rectangles that need to be related to the screen (like the drag and grow areas of windows) should use `screenBits.bounds` to avoid worrying about the screen size.

Note that this does not do anything remotely like assume that "if the computer is not a standard Macintosh, then it must be an XL." Special casing for the various versions of the Macintosh has always been suspicious at best; it is now grounds for breaking. (At least with respect to screen dimensions.)

By the way, remember to take into account the menu bar height when using this rectangle. On 128K ROMs (and later) you can use the low-memory global `mBarHeight` (a word at `$BAA`). But since we didn't provide a low-memory global for the menu bar height in the 64K ROMs, you'll have to hard code it to 20 (`$14`). (You're not the only ones to forget the future holds changes.)

How to find fascist screenism in current programs

The easiest way is to exercise your program on one of the Ultra-Large screen Macintoshes. There should be no restrictions on sizing or moving the windows, and all drawing should have no problems. If there are any anomalies in the program's usage, there is probably a lurking problem. Also, do a global find in the source code to see if the numbers 512 or 342 occur in the program. If so, and if they are in reference to the screen, excise them.

Assuming the screen is at a fixed location

Some programs use a fixed screen address, assuming that the screen location will be the same on various incarnations of the Macintosh. This is not the case. For example, the screen is located at memory location $1A700 on a 128K Macintosh, at $7A700 on a 512K Macintosh, at $F8000 on the Macintosh XL, and at $FA700 on the Macintosh Plus.

Why it's Bad

When a program relies upon the screen being in a fixed location, Murphy's Law dictates that an unknowing user will run it upon a computer with the screen in a different location. This usually causes the system to crash, since the offending program will write to memory that was used for something important. Programs that crash have been proven to be less useful than those that don't.

How to avoid being a base screener

Suffice it to say that there is no way that the address of the screen will remain static, but there are rare occasions where it is necessary to go directly to the screen memory. On these occasions, there are bad ways and not-as-bad ways to do it. A bad way:

```
    myScreenBase := Pointer ($7A700);  { not good.  Hard-coded number. }
```

A not-as-bad way:

```
    InitGraf(@thePort);   { do this only once in a program. }
    ...
    myScreenBase := screenBits.baseAddr;  { Good.  Always works. }
                                       {Yet another QuickDraw global variable}
```

Using the latter approach is guaranteed to work, since QuickDraw has to know where to draw, and the operating system tells QuickDraw where the screen can be found. When in doubt, ask QuickDraw. This will work on Macintosh computers from now until forever, so if you use this approach you won't have to revise your program just because the screen moved in memory.

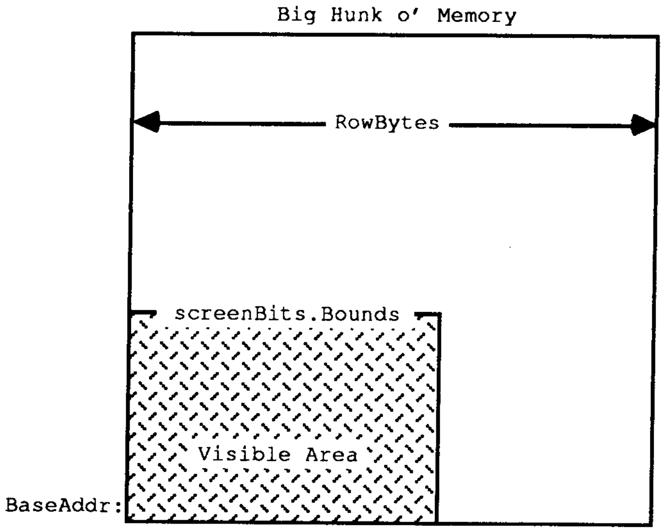
If you have a program (such as an INIT) that cannot rely upon QuickDraw being initialized (via InitGraf), then it is possible to use the ScrnBase low-memory global variable (a long word at $824). This method runs a distant second to asking QuickDraw, but is sometimes necessary.

How to find base screeners

The easiest way to find base screeners is to run the offending program on machines that have different screen addresses. If any addresses are being used in a base manner, the system will usually crash. The offending program may also occasionally refuse to draw. Some programs afflicted with this problem may also hang the computer (sometimes known as accessing funny space). Also, do a global find on the source code to look for numbers like $7A700 or $1A700. When found, exercise caution while altering the offending lines.

Assuming that rowbytes is equal to the width of the screen

According to the definition of a `bitMap` found in *Inside Macintosh* (p I-144), you can see that `rowBytes` is the number of actual bytes in memory that are used to determine the `bitMap`. We know the screen is just a big hunk of memory, and we know that QuickDraw uses that memory as a `bitMap`. `rowBytes` accomplishes the translation of a big hunk of memory into a `bitMap`. To do this, `rowBytes` tells the system how long a given row is in memory and, more importantly, where in memory the next row starts. For conventional Macintoshes, `rowBytes` (bytes per Row) * 8 (Pixels per Byte) gives the final horizontal width of the screen as Pixels per Row. This does not have to be the case. It is possible to have a Macintosh screen where the `rowBytes` extends beyond what is actually visible on the screen. You can think of it as having the screen looking in on a larger `bitMap`. Diagrammatically, it might look like:



With an Ultra-Large screen, the number of bytes used for screen memory may be in the 500,000 byte range. Whenever calculations are being made to find various locations in the screen, the variables used should be able to handle larger screen sizes. For example, a 16 bit `Integer` will not be able to hold the 500,000 number, so a `LongInt` would be required. Do **not** assume that the screen size is 21,888 bytes long. `bitMaps` **can** be larger than 32K or 64K.

Why it's Bad

Programs that assume that all of the bytes in a row are visible may make bad calculations, causing drawing routines to produce unusual, and unreadable, results. Also, programs that use the `rowBytes` to figure out the width of the screen rectangle will find that their calculated rectangle is not the real `screenBits.Bounds`. Drawing into areas that are not visible will not necessarily crash the computer, but it will probably give erroneous results, and displays that don't match the normal output of the program.

Programs that assume that the number of bytes in the screen memory will be less than 32768 may have problems drawing into Ultra-Large screens, since those screens will often have more memory than a normal Macintosh screen. These particular problems do not evidence themselves by crashing the system. They generally appear as loss of

functionality (not being able to move a window to the bottom of the screen), or as drawing routines that no longer look correct. These problems can prevent an otherwise wonderful program from being used.

How to avoid being a row byter

In any calculations, the `rowBytes` variable should be thought of as the way to get to the next row on the screen. This is distinct from thinking of it as the width of the screen. The width should always be found from `screenBits.bounds.right-screenBits.bounds.left`.

It is also inappropriate to use the rectangle to decide how many bytes there are on a row. Programs that do something like:

```
    bytesLine := screenBits.bounds.right DIV 8;  { bad use of bounds }
    rightSide := screenBits.rowBytes * 8;     { bad use of rowBytes }
```

will find that the screen may have more `rowBytes` than previously thought. The best way to avoid being a row byter is to use the proper variables for the proper things. Without the proper mathematical basis to the screen, life becomes much more difficult. Always do things like:

```
    bytesLine := screenBits.rowBytes;  { always the correct number }
    rightSide := screenBits.bounds.right;  { always the correct screen size }
```

It is sometimes necessary to do calculations involving the screen. If so, be sure to use `LongInts` for all the math, and be sure to use the right variables (i.e. use `LongInts`). For example, if we need to find the address of the 500th row in the screen (500 lines from the top):

```
VAR   myAddress:   LongInt;
      myRow:       LongInt;      { so the calculations don't round off. }
      myOffset:    LongInt;      { could easily be over 32768 ... }
      bytesLine:   LongInt;

      ...
      myAddress := ord4(screenBits.baseAddr); {start w/the real base address }
      myRow := 500;                          {the row we want to address }
      bytesLine := screenBits.rowBytes;      {the real bytes per line }
      myOffset := myRow * bytesLine;         {lines * bytes per lines gives bytes }
      myAddress := myAddress + myOffset;     {final address of the 500th line }
```

This is not something you want to do if you can possibly avoid it, but if you simply must go directly to the screen, be careful. The big-screen machines (Ultra-Large screens) will thank you for it. If QuickDraw cannot be initialized, there is also the low-memory global `screenRow` (a word at `$106`) that will give you the current `rowBytes`.

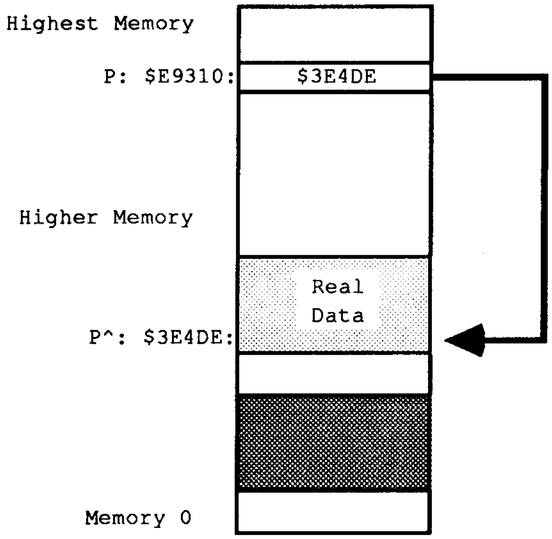
How to find row byters

To find current problems with row byter programs, run them on a machine equipped with Ultra-Large screens and see if any anomalies crop up. Look for drawing sequences that don't work right, and for drawing that clips to an imaginary edge. For source-level

inspection, look for uses of the `rowBytes` variables and be sure that they are being used in a mathematically sound fashion. Be highly suspicious of any code that uses `rowBytes` for the screen width. Any calculations involving those system variables should be closely inspected for round-off errors and improper use. Search for the number 8. If it is being used in a calculation where it is the number of bits per byte, then watch that code closely for improper conceptualization. This is code that could leap out and grab you by the throat at anytime. Be careful!

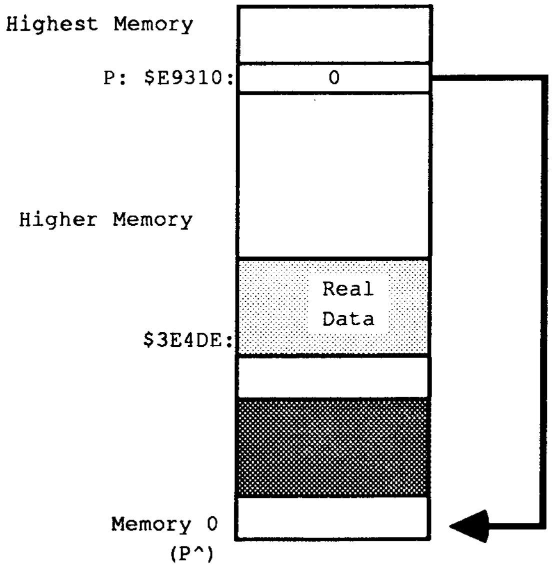
Using `nil` Handles or `nil` Pointers

A `nil` pointer is a pointer that has a value of 0. Recognize that pointers are merely addresses in memory. This means that a `nil` pointer is pointing to memory location 0. Any use of memory location 0 is strictly forbidden, since it is owned by Motorola. Trespassers may be shot on sight, but they may not die until much later. Sometimes trespassers are only wounded and act strangely. Any use of memory location 0 can be considered a bug, since there are **no** valid reasons for Macintosh programs to read or write to that memory. However, `nil` pointers themselves are not necessarily bad. It is occasionally necessary to pass `nil` pointers to ROM routines. This should not be confused with reading or writing to memory location 0. A pointer normally points to (contains the address of) a location in memory. It could look like this:



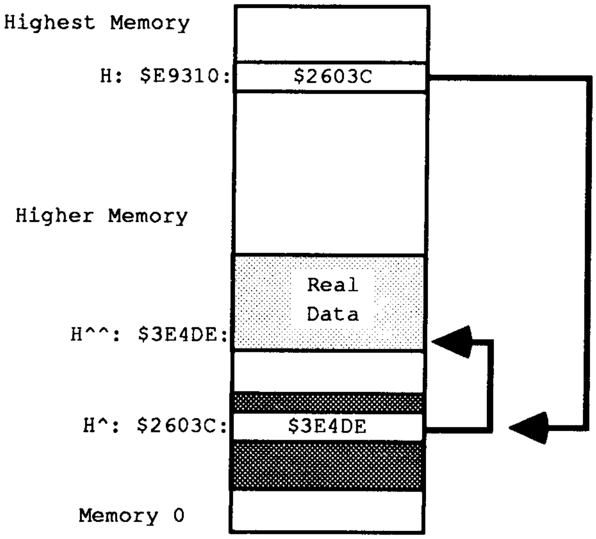
This is how a Pointer works. The address of the pointer variable itself is $E9310 (@P) and is four bytes long. The pointer points to (contains the address of) the block at $3E4DE (P). That memory location is where the actual data resides (P^).

If a pointer has been cleared to `nil`, it will point to memory location 0. This is OK as long as the program does not try to read from or write to that pointer. An example of a `nil` pointer could look like:



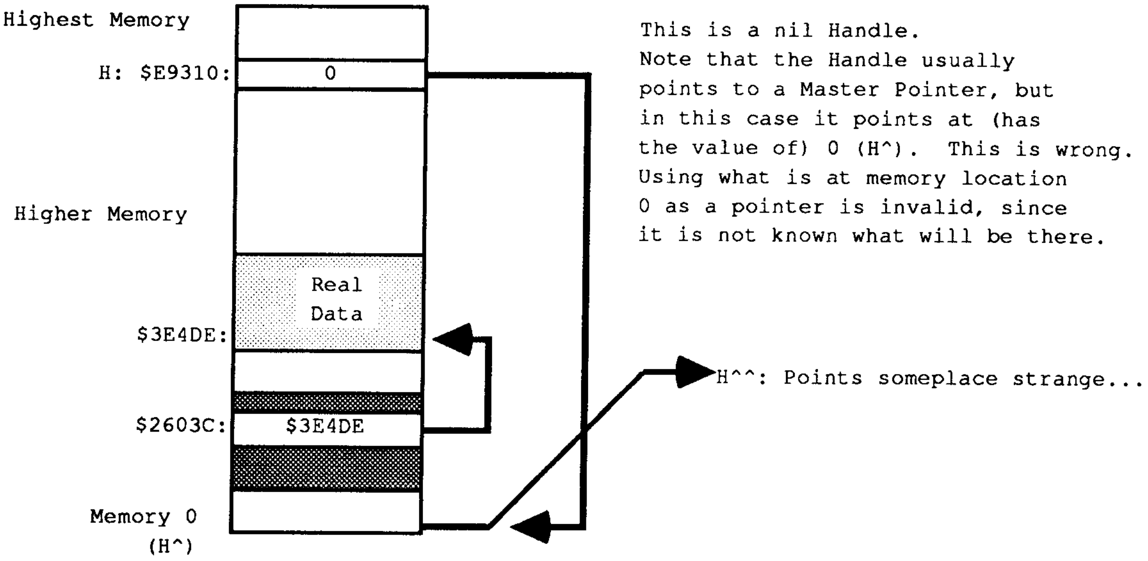
This is a nil Pointer. Note that the memory that it points to (the address) is 0 (P^). This is wrong. There is no valid data at memory location 0. Any writing to or reading from this pointer is a bug.

nil handles are related to the problem, since a handle is merely the address of a pointer (or a pointer to a pointer). An example of what a normal handle might look like is:



This is how a Handle works. The address of the handle variable itself (H) is $E9310. That variable points (has the address) to the master pointer at location $2603C (H). That variable is a pointer also, and points to the real data found at $3E4DE (H^^). The dark grey block is a Master pointer block. It is a group (usually 64) of Master Pointers. One of them is the Master Pointer at address $2603C (H^).

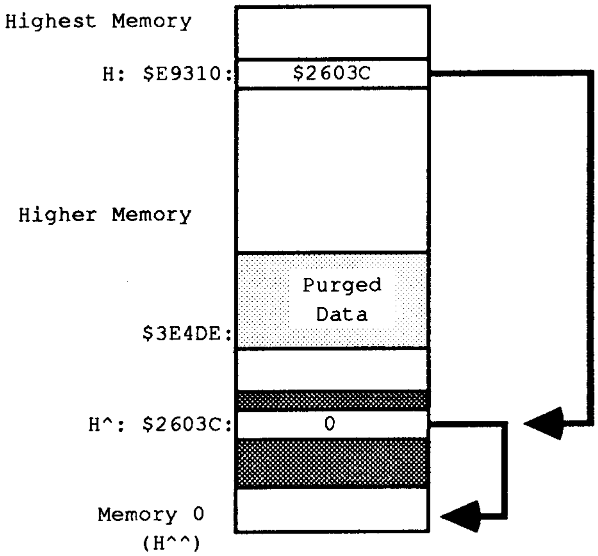
When the first pointer (h) becomes nil, that implies that memory location 0 can be used as a pointer. This is strictly illegal. There are **no** cases where it is valid to read from or write to a nil handle. A pictorial representation of what a nil handle could look like:



This is a nil Handle. Note that the Handle usually points to a Master Pointer, but in this case it points at (has the value of) 0 (H^). This is wrong. Using what is at memory location 0 as a pointer is invalid, since it is not known what will be there.

If the memory at 0 contains an odd number (numerically odd), then using it as a pointer will cause a system error with ID=2. This can be very useful, since that tells you exactly where the program is using this illegal handle, making it easy to fix. Unfortunately, there are cases where it is appropriate to pass a nil handle to ROM routines (such as GetScrap). These cases are rare, and it is **never** legal to read from or write to a nil handle.

There is also the case of an empty handle. An empty handle is one where the handle itself (the first pointer) points to a valid place in memory; that place in memory is also a pointer, and if it is `nil` the entire handle is termed empty. There are occasions where it is necessary to use the handle itself, but using the `nil` pointer that it contains is not valid. An example of an empty handle could be:



This is an Empty Handle. Note that the handle itself has a valid Master Pointer address in it $2603C (H^). The Master Pointer is nil however, which is the address of location 0 in memory. It is wrong to use the Master Pointer in this case, although there are cases where using the Handle itself is valid.

Fundamentally, any reading or writing to memory using a pointer or handle that is `nil` is punishable by death (of your program).

Why it's Bad

The use of `nil` pointers can lead to the use of make-believe data. This make-believe data often changes for different versions of the computer. This changing data makes it difficult to predict what will happen when a program uses `nil` pointers. Programs may not crash as a result of using a `nil` pointer, and they may behave in a consistent fashion. This does not mean that there isn't a bug. This merely means that the program is lucky, and that it should be playing the lottery, not running on a Macintosh. If a program acts differently on different versions of the Macintosh, you should think "could there be a nasty `nil` pointer problem here?" Use of a `nil` handle usually culminates in reading or writing to obscure places in memory. As an example:

```
VAR   myHandle:  TEHandle;

myHandle := nil;
```

That's pretty straightforward, so what's the problem? If you do something like:

```
myHandle^^.viewRect := myRect;  { very bad idea with myHandle = nil }
```

memory location zero will be used as a pointer to give the address of a TextEdit record. What if that memory location points to something in the system heap? What if it points to the sound buffer? In cases like these, eight bytes of rectangle data will be written to wherever memory location 0 points.

Use of a nil handle will never be useful. This memory is reserved and used by the 68000 for various interrupt vectors and Valuable Stuff. This Valuable Stuff is composed of things that you definitely do not want to change. When changed, the 68000 finds out, and decides to get back at your program in the most strange and wonderful ways. These strange results can range from a System Error all the way to erasing hard disks and destroying files. There really is no limit to the havoc that can be wreaked. This tends to keep the users on the edge of their seat, but this is not really the desired effect. As noted above, it won't necessarily cause traumatic results. A program can be doing naughty things and not get caught. This is still a bug that needs to be fixed, since it is nearly guaranteed to give different results on different versions of the Macintosh. Programs exhibiting schizophrenia have been proven to be less enjoyable to use.

How to avoid being a Niller

Whenever a program uses pointers and handles, it should ensure that the pointer or handle will not be nil. This could be termed defensive programming, since it assumes that everyone is out to get the program (which is not far from the truth on the Macintosh). You should always check the result of routines that claim to pass back a handle. If they pass you back a nil handle, you could get in trouble if you use them. Don't trust the ROM. The following example of a defensive use of a handle involves the Resource Manager. The Resource Manager passes back a handle to the resource data. There are any number of places where it may be forced to pass back a nil handle. For example:

```
VAR   myRezzie:  MyHandle;

myRezzie := MyHandle(GetResource(myResType, myResNumber)); { could be missing…}
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;                { We know it is OK }
```

As another example, think of how handles can be purged from memory in tight memory conditions. If a block is marked purgeable, the Memory Manager may throw it away at any time. This creates an empty handle. The defensive programmer will always make sure that the handles being used are not empty.

```
VAR   myRezzie:  myHandle;

myRezzie := myHandle(GetResource(myResType, myResNumber));  { could be
                                                               missing… }
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;     { We know it is OK }
tempHandle := NewHandle (largeBlock);  {might dispose a purgeable myRezzie}
IF myRezzie^ = nil  THEN LoadResource(Handle(myRezzie)); {Re-load empty
                                                          handle}
IF ResError = noErr  THEN
    myRezzie^^.StatusField := OK;       { guaranteed not empty, and actually
                                        gets read back in, if necessary }
```

Be especially careful of places where memory is being allocated. The NewHandle and NewPtr calls will return a nil handle or pointer if there is not enough memory. If you use that handle or pointer without checking, you will be guilty of being a Niller.

How to find Nillers

The best way to find these nasty `nil` pointer problems is to set memory location zero to be an **odd** number (a good choice is 'NIL!' = `$4E494C21`, which is numerically odd, as well as personality-wise). Please see Technical Note #7 for details on how to do this.

If you use TMON, you can use the extended user area with Discipline. Discipline will set memory location 0 to 'NIL!' to help catch those nasty pointer problems. If you use Macsbug, just type `SM 0 'NIL!` and go. Realize of course, that if a program has made a transgression and is actually using `nil` pointers, this may make the program crash with an ID=2 system error. This is good! This means that you have found a bug that may have been causing you untold grief. Once you know where a program crashes, it is usually very easy to use a debugger to find where the error is in the source code. When the program is compiled, turn on the debugging labels (usually a $D+ option). Set memory location 0 to be 'NIL!'. When the program crashes, look at where the program is executing and see what routine it was in (from a disassembly). Go back to that routine in the source code and remove the offending code with a grim smile on your face. Another scurvy bug has been vanquished. The intoxicating smell of victory wafts around your head.

Another way to find problems is to use a debugger to do a checksum on the first four bytes in memory (from 0 to 3 inclusive). If the program ever traps into the debugger claiming that the memory changed, see which part of the program altered memory location 0. Any code that writes to memory location zero is guilty of high treason against the state and must be removed. Remember to say, “bugs are not my friends.”

Creating or Using Fake Handles

A fake handle is one that was not manufactured by the system, but was created by the program itself. An example of a fake handle is:

```
CONST aMem = $100;
VAR   myHandle: Handle;
      myPointer: Ptr;

myPointer := Ptr (aMem);      { the address of some memory }
myHandle := @myPointer;       {the address of the pointer variable. Very bad.}
```

The normal way to create and use handles is to call the Memory Manager `NewHandle` function.

Why it's Bad

A handle that is manufactured by the program is not a legitimate handle as far as the operating system is concerned. Passing a fake handle to routines that use handles is a good way to discover the meaning of "Death by ROM." For example, think how confused the operating system would get if the fake handle were passed to `DisposHandle`. What would it dispose? It never allocated the memory, so how can it release it? Programs that manufacture handles may find that the operating system is no longer their friend.

When handles are passed to various ROM routines, there is no telling what sorts of things will be done to the handle. There are any number of normal handle manipulation calls that the ROM may use, such as `SetHandleSize`, `HLock`, `HNoPurge`, `MoveHHi` and so on. Since a program cannot guarantee that the ROM will not be doing things like this to handles that the program passes in, it is wise to make sure that a real handle is being used, so that all these type of operations will work as the ROM expects. For fake handles, the calls like `HLock` and `SetHandleSize` have no bearing. Fake handles are very easy to create, and they are very bad for the health of otherwise upstanding programs. Whenever you need a handle, get one from the Memory Manager.

As a particularly bad use of a fake handle:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

myHandle := NewHandle (SIZEOF(myStuff));     { create a new normal handle }
myHandle^ := @myStuff;  {YOW!  Intended to make myHandle a handle to
                           the myStuff record.  What it really does is
                           blow up a Master Pointer block, Heap corruption,
                           and death by Bad Heap.  Never do this. }
```

This can be a little confusing, since it is fine to use your own pointers, but very bad to use your own handles. The difference is that handles can move in memory, and pointers cannot, hence the pointers are not dangerous. This does not mean you should use pointers for everything since that causes other problems. It merely means that you have to be careful how you use the handles.

The use of fake handles usually causes system errors, but can be somewhat mysterious in its effects. Fake handles can be particularly hard to track down since they often cause damage that is not uncovered for many minutes of use. Any use of fake handles that causes the heap to be altered will usually crash the system. Heap corruption is a common failure mode. In clinical studies, 9 out of 10 programmers recommend uncorrupted heaps to their users who use heaps.

How to avoid being a fakir

The correct way to make a handle to some data is to make a copy of the data:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

errCode := PtrToHand (@myStuff, myHandle, SIZEOF(myStuff));
IF  errCode <> noErr  THEN ErrorHandler ('Out of memory');
```

Always, always, let the Memory Manager perform operations with handles. Never write code that assigns something to a master pointer, like:

```
VAR   myDeath:  Handle;
myDeath^ := stuff;  { Don't change the Master pointer. }
```

If there is code like this, it usually means the heap is being corrupted, or a fake handle is being used. It is, however, OK to pass around the handle itself, like:

```
myCopyHandle := myHandle;   { perfectly OK, nobody will yell about this. }
```

This is far different than using the ^ operator to accidentally modify things in the system. Whenever it is necessary to write code to use handles, be careful. Watch things carefully as they are being written. It is much easier to be careful on the way in than it is to try to find out why something is crashing. Be very careful of the @ operator. This operator can unleash untold problems upon unsuspecting programs. If at all possible, try to avoid using it, but if it is necessary, be absolutely sure you know what it is doing. It is particularly dangerous since it turns off the normal type checking that can help you find errors (in Pascal). In short, don't get crazy with pointer and handle manipulations, and they won't get crazy with you.

How to find fakirs

Problems of this form are particularly insidious because it can be very difficult to find them after they have been created. They tend to not crash immediately, but rather to crash sometime long after the real damage has been done. The best way to find these problems is to run the program with Discipline. (Discipline is a programmer's tool that will check all parameters passed to the ROM to see if they are legitimate. Discipline can be found as a stand-alone tool, but the most up-to-date version will be found in the Extended User Area for the TMON debugger. The User Area is public domain, but TMON itself is not. TMON has a number of other useful features, and is well worth the price.) Discipline will check handles that are passed to the ROM to see if they are real handles or not, and if not, will stop the program at the offending call. This can lead you back to the source at a point that may be close to where the bad handle was created. If a program passes the Discipline test, it will be a healthy, robust program with drastically improved odds for compatibility. Programs that do not pass Discipline can sleep poorly at night, knowing that they have broken at least one or two of the "rules."

A way to find programs that are damaging the heap is to use a debugger (TMON or Macsbug) and turn on the Heap Check operation. This will check the heap for errors at each trap call, and if the heap is corrupted will break into the debugger. Hopefully this will be close to where the code is that caused the damage. Unfortunately, it may not be close enough; this will force you to look further back.

Looking in the source code, look for all uses of the `@` operator, and examine the code carefully to see if it is breaking the rules. If it is, change it to step in line with the rest of the happy programs here in happy valley. Also, look for any code that changes a master pointer like the `myHandle^ := stuff.` Any code of this form is highly suspect, and probably a member of the Anti-Productivity League. The APL has been accused of preventing software sales and the rise of the Yen. These problems can be quite difficult to find at times, but don't give up. These fake handles are high on the list of guilty parties, and should never be trusted.

Writing code that modifies itself

Self-modifying code is software that changes itself. Code that alters itself runs into two main groupings: code that modifies the code itself and code that changes the block the code is stored in. Copy protection code often modifies the code itself, to change the way it operates (concealing the meaning of what the code does). Changing the code itself is very tricky, and also prone to having problems, particularly when the microprocessor itself changes. There are third-party upgrades available that add a 68020 to a Macintosh. Because of the 68020's cache, programs that modify themselves stand a good chance of having problems when run on a 68020. This is a compatibility point that should not be missed (nudge, nudge, wink, wink). Code that changes other code (or itself) is prone to be incompatible when the microprocessor changes.

The second group is code that changes the block that the code is stored in. Keeping variables in the CODE segment itself is an example of this. This is uncommon with high-level languages, but it is easy to do in assembly language (using the DC directive). Variables defined in the code itself should be read-only (constants). Code that modifies itself has signed a tacit agreement that says "I'm being tricky, if I die, I'll revise it."

Why it's Bad

There are now three different versions of the microprocessor, the 68000, 68010, and the 68020. They are intended to be compatible with each other, but may not be compatible with code that modifies itself. As the Macintosh evolves, the system may have compatibility problems with programs that try to "push the envelope."

How to avoid being an abuser

Well, the obvious answer is to avoid writing self-modifying code. If you feel obliged to write self-modifying code, then you are taking an oath to not complain when you break in the future. But don't worry about accidentally taking the oath: you won't do it without knowing it. If you choose to abuse, you also agree to personal visits from the Apple thought police, who will be hired as soon as we find out.

How to find abusers

Run the program on a 68020 system. If it fails, it could be related to this problem, but since there are other bugs that might cause failures, it is not guaranteed to be a self-modifying code problem. Self-modifying code is often used in copy protection, which brings us to the next big topic.

Code designed strictly as copy protection

Copy protection is used to make it difficult to make copies of a program. The basic premise is to make it impossible to copy a program with the Finder. This will not be a discussion as to the pros and cons of copy protection. Everyone has an opinion. This will be a description of reality, as it relates to compatibility.

Why it's Bad

System changes will never be made merely to cause copy protection schemes to fail, but given the choice between improving the system and making a copy protection scheme remain compatible, the system improvement will always be chosen.

- Copy protection is number one on the list of why programs fail the compatibility test.
- Copy protection by its very nature tends to do the most "illegal" things.
- Programs that are copy protected are assumed to have signed a tacit agreement to revise the program when the system changes.

Copy protection itself is not necessarily bad. What is bad is when programs that would otherwise be fully compatible do not work due only to the copy protection. This is very sad, since it requires extra work, revisions to the software, and time lost while the revision is being produced. The users are not generally humored when they can no longer use their programs. Copy protection schemes that fail generally cause system errors when they are run. They also can refuse to run when they should.

How to avoid being a protectionist

The simple answer is to do without copy protection altogether. If you think of compatibility as a probability game, if you leave out the copy protection, your odds of winning skyrocket. As noted above, copy protection is the single biggest reason why programs fail on the various versions of the Macintosh. For those who are required to use copy protection, try to rely on schemes that do not require specific hardware and make sure that the scheme used is not performing illegal operations. If a program runs, an experienced Macintosh programmer armed with a debugger can probably make a copy of it, (no matter how sophisticated the copy protection scheme) so a moderate scheme that does not break the rules is probably a better compatibility bet. The trickier and more devious the scheme, the higher the chance of breaking a rule. Tread lightly.

How to find protectionists

The easiest way to see if a scheme is being overly tricky is to run it on a Macintosh XL. Since the floppy disk hardware is different this will usually demonstrate an unwanted hardware dependency. Be wary of schemes that don't allow installation on a hard disk. If the program cannot be installed on a hard disk, it may be relying upon things that are prone to change. Don't use schemes that access the hardware directly. All Macintosh software should go through the various managers in the ROM to maintain compatibility. Any code that sidesteps the ROM will be viewed as having said "It's OK to make me revise myself."

Check errors returned as function results

All of the Operating System functions, as well as some of the Toolbox functions, will return result codes as the value of the function. Don't ignore these result codes. If a program ignores the result codes, it is possible to have any number of bad things happen to the program. The result code is there to tell the program that something went wrong; if the program ignores the fact that something is wrong, that program will probably be killed by whatever went wrong. (Bugs do not like to be ignored.) If a program checks errors, an anomaly can be nipped in the bud, before something really bizarre happens.

Why it's Bad

A program that ignores result codes is skipping valuable information. This information can often prevent a program from crashing and keep it from losing data.

How to avoid becoming a skipper

Always write code that is defensive. Assume that everyone and everything is out to kill you. Trust no one. An example of error checking is:

```
myRezzie := GetResource (myResType, myResId);
IF  myRezzie = nil  THEN  ErrorHandler ('Who stole my resource...');
```

Another example:

```
fsErrCode := FSOpen ('MyFile', myVRefNum, myFileRefNum);
IF fsErrCode <> noErr  THEN ErrorHandler (fsErrCode, 'File error');
```

And another:

```
myTPPrPort := PrOpenDoc (myTHPrint, nil, nil);
IF  PRError <> noErr  THEN  ErrorHandler (PRError, 'Printing error');
```

Any use of Operating System functions should presume that something nasty can happen, and have code to handle the nasty situations. Printing calls, File Manager calls, Resource Manager calls, and Memory Manager calls are all examples of Operating System functions that should be watched for returning errors. Always, always check the result codes from Memory Manager calls. Big memory machines are pretty common now, and it is easy to get cavalier about memory, but realize that someone will always want to run the program under Switcher, or on smaller Macintoshes. It never hurts to check, and always hurts to ignore it.

How to find skippers

This is easy: just do weird things while the program is running. Put in locked or unformatted disks while the program is running. Use unconventional command sequences. Run out of disk space. Run on 128K Macintoshes to see how the program deals with running out of memory. Run under Switcher for the same reason. (Programs that die while running under Switcher are often not Switcher's fault, and are in fact due

to faulty memory management.) Print with no printer connected to the Macintosh. Pop disks out of the drives with the Command-Shift sequence, and see if the program can deal with no disk. When a disk-switch dialog comes up, press Command-period to pass back an error to the requesting program (128K ROMs only). Torturing otherwise well-behaved programs can be quite enjoyable, and a number of users enjoy torturing the program as much as the program enjoys torturing them. For the truly malicious, run the debugger and alter error codes as they come back from various routines. Sure it's a dirty low-down rotten thing to do to a program, but we want to see how far we can push the program. (This is also a good way to check your error handling.) It's one thing to be an optimist, but it's quite another to assume that nothing will go wrong while a program is running.

Accessing hardware directly

Sometimes it is necessary to go directly to the Macintosh hardware to accomplish a specific task for which there is no ROM support. Early hard disks that used the serial ports had no ROM support. Those disks needed to use the SCC chip (the 8530 communication chip) in a high-speed clocked fashion. Although it is a valid function, it is not something that is supported in the ROM. It was therefore necessary to go play with the SCC chip directly, setting and testing various hardware registers in the chip itself. Another example of a valid function that has no ROM support is the use of the alternate video page for page-flipping animation. Since there is no ROM call to flip pages, it is necessary to go play with the right bit in the VIA chip (6522 Versatile Interface Adapter). Going directly to the hardware does not automatically throw a program into the incompatible group, but it certainly lowers its odds.

Why it's bad

Going directly to the hardware poses any number of problems for enlightened programs that are trying to maintain compatibility across the various versions of the Macintosh. On the Macintosh XL for example, a lot of the hardware is found in different locations, and in some cases the hardware doesn't exist. On the XL there is no sound chip. Programs that go directly to the sound hardware will find they don't work correctly on an XL. If the same program were to go through the Sound Manager, it would work fine, although the sound would not be the same as expected. Since the Macintosh is heavily oriented to the software side of things, expecting various hardware to always be available is not a safe bet. Choosy programmers choose to leave the hardware to the ROM.

How to avoid having a hard attack

Don't read or write the hardware. Exhaust every possible conventional approach before deciding to really get down and dirty. If there is a Manager in the ROM for the operation you wish to perform, it is far better to use the Manager than to go directly to the hardware. Compatibility at the hardware level can very rarely be maintained, but compatibility at the Manager level is a prime consideration. If a program is down to the last ditch effort, and cannot get the support from the ROM that is desired, then access the hardware in an enlightened approach. The really bad way to do it:

```
VIA := Pointer ($EFE1FE);  { sure it's the base address today...}
                             { This is bad.  Hard-coded number. }
```

The with-it, inspired programmer of the eighties does something like:

```
TYPE LongPointer = ^LongInt;

VAR  VIA: LongPointer;
     VIABase: LongInt;

VIA := Pointer ($1D4);  { the address of the low-memory global. }
VIABase := VIA^;        { get the low-memory variable's value }
                       { Now VIABase has the address of the chip }
```

The point here is that the best way to get the address of a hardware chip is to ask the system where it currently is to be found. The system always knows where the pieces of the system are, and will always know for every incarnation of the Macintosh. There are low-memory global variables for all of the pieces of hardware currently found in the Macintosh. This includes the VIA, the SCC, the Sound Chip, the IWM, and the video display. Whenever you are stuck with going to the hardware, use the low-memory globals. The fact that a program goes directly to the hardware means that it is risking imminent incompatibility, but using the low-memory global will ensure that the program has the best odds. It's like going to Las Vegas: if you don't gamble at all, you don't lose any money; if you have to gamble, play the game that you lose the least on.

How to find hard attacks

Run the suspicious program on the Macintosh XL. Nearly all of the hardware is in a different memory location on the XL. If a program has a hard-coded hardware address in it, it will fail. It may crash, or it might not perform the desired task, but it won't work as advertised. This unfortunately, is not a completely legitimate test, since the XL does not have some of the hardware of other Macintoshes, and some of the hardware that is there has the register mapping different. This means that it is possible to play by the rule of using the low-memory global and still be incompatible.

Don't use bits that are reserved

Occasionally during the life of a Macintosh programmer, there comes a time when it is necessary to bite the bullet and use a low-memory global. These are very sad days, since it has been demonstrated (by history) that low-memory global variables are a mysterious lot, and not altogether friendly. One fellow in particular is known as `ROM85`, a word located at `$28E`. This particular variable has been documented as the way to determine if a program is running on the 128K ROMs or not. Notably, the top most bit of that word is the determining bit. This means that the rest of the bits in that word are reserved, since nothing is described about any further bits. Remember, if it doesn't say, assume it's reserved. If it's reserved, don't depend upon it. Take the cautious way out and assume that the other bits that aren't documented are used for Switcher local variables, or something equally wild. An example of a bad way to do the comparison is:

```
VAR  Rom85Ptr: WordPtr;
     RomsAre64: Boolean;

Rom85Ptr := Pointer ($28E);    { point at the low-memory global }
IF  Rom85Ptr^ = $7FFF  THEN  RomsAre64 := False  { Bad test. }
ELSE  RomsAre64 := True;
```

This is a bad test since the comparison is testing the value of **all** of the bits, not only the one that is valid. Since the other bits are undocumented, it is impossible to know what they are used for. Assume they are used for something that is arbitrarily random, and take the safe way out.

How to avoid being bitten

```
VAR     ROM85Ptr: Ptr

Rom85Ptr := Pointer ($28E);    { point at the low-memory global }
IF BitTst(ROM85Ptr,0) THEN RomsAre64 := True {Good--tests only hi-bit}
ELSE  RomsAre64 := False;
```

This technique will ensure that when those bits are documented, your program won't be using them for the wrong things. Beware of trojan bits.

Don't use undocumented stuff. Be very careful when you use anything out of the ordinary stream of a high-level language. For instance, in the `ROM85` case, it is very easy to make the mistake of checking for an absolute value instead of testing the actual bit that encodes the information. Whenever a program is using low-memory globals, be sure that only the information desired is being used, and not some undocumented (and hence reserved) bits. It's not always easy to determine what is reserved and what isn't, so conservative programmers always use as little as possible. Be wary of the strange bits, and accept rides from none of them. The ride you take might cause you to revise your program.

How to find those bitten

Since there are such a multitude of possible places to get killed, there is no simple way to see what programs are using illegal bits. As time goes by it will be possible to find more of these cases by running on various versions of the Macintosh, but there will probably never be a comprehensive way of finding out who is accepting strange rides, and who is not. Whenever the use of a bit changes from reserved status to active, it will be possible to find those bugs via extensive testing. From a source level, it would be advisable to look over **any** use of low-memory globals, and eye them closely for inappropriate bit usage. Do a global search for the $ (which describes those ubiquitous hexadecimal numbers), and when found see if the use of the number is appropriate. Trust no one that is not known. If they are documented, they will stay where they are, and have the same meaning. Be very careful in realms that are undocumented. Bits that suddenly jump from reserved to active status have been known to cause more than one program to have a sudden anxiety attack. It is very unnerving to watch a program go from calm and reassuring to rabid status. Users have been known to drop their keyboards in sudden shock (which is bad on the keyboards).

Summary

So what does all this mean? It means that it is getting harder and harder to get away with minor bugs in programs. The minor bugs of yesterday are the major ones of today. No one will yell at you for having bugs in your program, since all programs have bugs of one form or another. The goal should be to make the programs run as smoothly and effortlessly as possible. The end-users will never object to bug-reduced programs.

What is the best way to test a program? A reasonably comprehensive test is to exercise all of the program's functions under the following situations:

- Use Discipline to be sure the program does not pass illegal things to the ROM.
- Use heap scramble and heap purge to be sure that handles are being used correctly, and that the memory management of the program is correct.
- Run with a checksum on memory locations 0...3 to see if the program writes to these locations.
- Run on a 128K Macintosh, or under Switcher with a small partition, to see how the program deals with memory-critical situations.
- Run on a 68020 system to see if the program is 68020-compatible and to make sure that changing system speed won't confuse the program.
- Run on a Macintosh XL to be sure that the program does not assume too much about the operating system, and to test screen handling.
- Run on an Ultra-Large screen to be sure that the screen handling is correct, and that there are no hard-coded screen dimensions.
- Run on 64K ROM machines to be sure new traps are not being used when they don't exist.
- Run under both HFS and MFS to be sure that the program deals with the file system correctly. (400K floppies are usually MFS.)

If a program can live through all of this with no Discipline traps, no checksum breaks, no system errors, no anomalies, no data loss and still get useful work done, then you deserve a gold medal for programming excellence. Maybe even an extra medal for conduct above and beyond the call of duty. In any case, you will know that you have done your job about as well as it can be done, with today's version of the rules, and today's programming tools.

Sounds like a foreboding task, doesn't it? The engineers in Macintosh Technical Support are available to help you with compatibility issues (we won't always be able to talk about new products, since we love our jobs, but we can give you some hints about compatibility with what the future holds).

Good luck.

#127: TextEdit EOL Ambiguity

See also: TextEdit

Written by: Rick Blair May 4, 1987
Updated: March 1, 1988

`TESetSelect` may be used to position the insertion point at the end of a line. There is an ambiguity, though; should the insertion point appear at the end of the preceding line or the start of the following one? It is possible to determine what will happen, as you are about to see.

There is an internal flag used by TextEdit to determine where the insertion point at the end of a line appears. This flag is part of the `clikStuff` field in the `TERec`. It is there mainly for the use of `TEClick`, but it is also used by `TESetSelect` (although it defaults to the right side of the previous line).

The following code can be used to force the insertion point to appear at the left of the following line when it is positioned at the end of a line; in MPW Pascal:

```
TEDeactivate(tH);
tH^^.clikStuff := 255;                          {position caret on left}
TESetSelect(eolcharpos, eolcharpos, tH);        {ambiguous point}
TEActivate(tH);
```

In MPW C:

```
TEDeactivate(tH);
(**tH).clikStuff = 255;                         /*position caret on left*/
TESetSelect(eolcharpos, eolcharpos, tH);        /*ambiguous point*/
TEActivate(tH);
```

If you want to ensure that the caret is on the right side (to which it normally defaults) then substitute a zero for the 255.

#128: PrGeneral

See also: The Printing Manager
Technical Note #118—
How to Check and Handle Printing Errors

Written by: Ginger Jernigan
Updated:

May 4, 1987
March 1, 1988

The Printing Manager architecture has been expanded to include a new procedure called `PrGeneral`. The features described here are advanced, special-purpose features, intended to solve specific problems for those applications that need them. The calls to determine printer resolution introduce a good deal of complexity into the application's code, and should be used only when necessary.

Version 2.5 (and later) of the ImageWriter driver and version 4.0 (and later) of the LaserWriter driver implement a generic Printing Manager procedure called `PrGeneral`. This procedure allows the Print Manager to expand in functionality, by allowing printer drivers to implement various new functions. The Pascal declaration of `PrGeneral` is:

```
PROCEDURE PrGeneral (pData: Ptr);
```

The `pData` parameter is a pointer to a data block. The structure of the data block is declared as follows:

```
TGnlData = RECORD {1st 8 bytes are common for all PrGeneral calls)
   iOpCode    : INTEGER;  {input}
   iError     : INTEGER;  {output}
   lReserved : LONGINT;  {reserved for future use}
   {more fields here, depending on particular call}
END;
```

The first field is a 2-byte opcode, `iOpCode`, which acts like a routine selector. The currently available opcodes are described below.

The second field is the error result, `iError`, which is returned by the print code. This error only reflects error conditions that occur during the `PrGeneral` call. For example, if you use an opcode that isn't implemented in a particular printer driver then you will get a `OpNotImpl` error.

Here are the errors currently defined:

```
CONST
   noErr = 0;                {everything's hunky}
   NoSuchRsl = 1;            {the resolution you chose isn't available}
   OpNotImpl = 2;            {the driver doesn't support this opcode}
```

After calling `PrGeneral` you should always check `PrError`. If `noErr` is returned, then you can proceed. If `ResNotFound` is returned, then the current printer driver doesn't support `PrGeneral` and you should proceed appropriately. See Technical Note #118 for details on checking errors returned by the Printing Manager.

`IError` is followed by a four byte reserved field (that means don't use it). The contents of the rest of the data block depends on the opcode that the application uses. There are currently five opcodes used by the ImageWriter and LaserWriter drivers.

The Opcodes

Initially, the following calls are implemented via `PrGeneral`:

- `GetRslData` (get resolution data): `iOpCode = 4`
- `SetRsl` (set resolution): `iOpCode = 5`
- `DraftBits` (bitmaps in draft mode): `iOpCode = 6`
- `noDraftBits` (no bitmaps in draft mode): `iOpCode = 7`
- `GetRotn` (get rotation): `iOpCode = 8`

The `GetRslData` and `SetRsl` allow the application to find out what physical resolutions the printer supports, and then specify a supported resolution. `DraftBits` and `noDraftBits` invoke a new feature of the ImageWriter, allowing bitmaps (imaged via `CopyBits`) to be printed in draft mode. `GetRotn` lets an application know whether landscape has been selected. Below is a detailed description of how each routine works.

The GetRslData Call

`GetRslData (iOpCode = 4)` returns a record that lets the application know what resolutions are supported by the current printer. The application can then use `SetRsl` (description follows) to tell the printer driver which one it will use. This is the format of the input data block for the `GetRslData` call:

```
TRslRg = RECORD            {used in TGetRslBlk}
   iMin, iMax: Integer;  {0 if printer only supports discrete resolutions}
END;

TRslRec = RECORD           {used in TGetRslBlk}
   iXRsl, iYRsl: Integer; {a discrete, physical resolution}
END;
```

```
TGetRslBlk = RECORD        {data block for GetRslData call}
   iOpCode:     Integer;   {input; = getRslDataOp}
   iError:      Integer;   {output}
   lReserved:   LongInt;   {reserved for future use}
   iRgType:     Integer;   {output; version number}
   XRslRg:      TRslRg;    {output; range of X resolutions}
   YRslRg:      TRslRg;    {output; range of Y resolutions}
   iRslRecCnt:  Integer;   {output; how many RslRecs follow}
   rgRslRec:    ARRAY[1..27] OF TRslRec;  {output; number filled depends on
                                            printer type}
END;
```

The `iRgType` field is much like a version number; it determines the interpretation of the data that follows. At present, a `iRgType` value of 1 applies both to the LaserWriter and to the ImageWriter.

For variable-resolution printers like the LaserWriter, the resolution range fields `XRslRg` and `YRslRg` express the ranges of values to which the X and Y resolutions can be set. For discrete-resolution printers like the ImageWriter, the values in the resolution range fields are zero.

Note: In general, X and Y in these records are the horizontal and vertical directions of the **printer**, not the document! In landscape orientation, X is horizontal on the printer but vertical on the document.

After the resolution range information there is a word which gives the number of resolution records that contain information. These records indicate the physical resolutions at which the printer can actually print dots. Each resolution record gives an X value and a Y value.

When you call `PrGeneral` you pass in a data block that looks like this:

OpCode = 4	1 word
Error Code	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 0, max = 0	2 words
Y Resolution Range: min =0, max = 0	2 words
Resolution Record Count =0	1 word
Resolution Record #1: X = 0, Y = 0	2 words
Resolution Record #2..27	

Below is the data block returned for the LaserWriter:

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 72, max = 1500	2 words
Y Resolution Range: min = 72, max = 1500	2 words
Resolution Record Count = 1	1 word
Resolution Record #1: X = 300, Y = 300	2 words

Note that all the resolution range numbers happen to be the same for this printer. There is only one resolution record, which gives the physical X and Y resolutions of the printer (300x300).

Below is the data block returned for the ImageWriter.

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min =0, max = 0	2 words
Y Resolution Range: min = 0, max = 0	2 words
Resolution Record Count = 4	1 word
Resolution Record #1: X = 72, Y = 72	2 words
Resolution Record #2: X =144, Y = 144	2 words
Resolution Record #3: X = 80, Y = 72	2 words
Resolution Record #4: X = 160, Y = 144	2 words

All the resolution range values are zero, because only discrete resolutions can be specified for this printer. There are four resolution records giving these discrete physical resolutions.

Note that `GetRslData` always returns the same information for a particular printer type—it is **not** dependent on what the user does or on printer configuration information.

The SetRsl Call

SetRsl (iOpCode = 5) is used to specify the desired imaging resolution, after using GetRslData to determine a workable pair of values. Below is the format of the data block:

```
TSetRslBlk =    RECORD      {data block for SetRsl call}
   iOpCode:     Integer;    {input; = setRslOp}
   iError:      Integer;    {output}
   lReserved:   LongInt;    {reserved for future use}
   hPrint:      THPrint;    {input; handle to a valid print record}
   iXRsl:       Integer;    {input; desired X resolution}
   iYRsl:       Integer;    {input; desired Y resolution}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate. If the call executes successfully, the print record is updated with the new resolution; the data block comes back with 0 for the error and is otherwise unchanged.

However, if the desired resolution is not supported, the error is set to noSuchRsl and the resolution fields are set to the printer's default resolution

Note that you can undo the effect of a previous call to SetRsl by making another call that specifies an unsupported resolution (such as 0x0), forcing the default resolution.

The DraftBits Call

DraftBits (iOpCode = 6) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) Below is the format of the data block:

```
TDftBitsBlk =   RECORD      {data block for DraftBits and NoDraftBits calls}
   iOpCode:     Integer;    {input; = draftBitsOp or noDraftBitsOp}
   iError:      Integer;    {output}
   lReserved:   LongInt;    {reserved for future use}
   hPrint:      THPrint;    {input; handle to a valid print record}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate.

This call forces draft-mode (i.e., immediate) printing, and will allow bitmaps to be printed via CopyBits calls. The virtue of this is that you avoid spooling large masses of bitmap data onto the disk, and you also get better performance.

The following restrictions apply:

- This call should be made before bringing up the print dialogs because it affects their appearance. On the ImageWriter, calling DraftBits disables the landscape icon in the Style dialog, and the Best, Faster, and Draft buttons in the Job dialog.

- If the printer does not support draft mode, already prints bitmaps in draft mode, or does not print bitmaps at all, this call does nothing.

- Only text and bitmaps can be printed.

- As in the normal draft mode, landscape format is not allowed.

- Everything on the page must be strictly Y-sorted, i.e. no reverse paper motion between one string or bitmap and the next. Note that this means you can't have two or more objects (text or bitmaps) side by side; the top boundary of each object must be no higher than the bottom of the preceding object.

The last restriction is important. If you violate it, you will not like the results. But note that if you want two or more bitmaps side by side, you can combine them into one before calling `CopyBits` to print the result. Similarly, if you are just printing bitmaps you can rotate them yourself to achieve landscape printing.

The NoDraftBits Call

`NoDraftBits (iOpCode = 7)` is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) The format of the data block is the same as that for the `DraftBits` call.

This call cancels the effect of any preceding `DraftBits` call. If there was no preceding `DraftBits` call, or the printer does not support draft-mode printing anyway, this call does nothing.

The GetRotn Call

`GetRotn (iOpCode = 8)` is implemented on the ImageWriter and LaserWriter. Here is the format of the data block:

```
TGetRotnBlk =  RECORD          {data block for GetRotn call}
   iOpCode:    Integer;        {input; = getRotnOp}
   iError:     Integer;        {output}
   lReserved:  LongInt;        {reserved for future use}
   hPrint:     THPrint;        {input; handle to a valid print record}
   fLandscape: Boolean;        {output; Boolean flag}
   bXtra:      SignedByte;     {reserved}
END;
```

`hPrint` should be the handle to a print record that has previously been passed to `PrValidate`.

If landscape orientation is selected in the print record, then `fLandscape` is true.

How To Use The PrGeneral Opcodes

The `SetRsl` and `DraftBits` calls may require the print code to suppress certain options in the Style and/or Job dialogs, therefore they should always be called before any call to the Style or Job dialogs. An application might use these calls as follows:

- Get a new print record by calling `PrintDefault`, or take an existing one from a document and call `PrValidate` on it.
- Call `GetRslData` to find out what the printer is capable of, and decide what resolution to use. Check `PrError` to be sure the `PrGeneral` call is supported on this version of the print code; if the error is `ResNotFound`, you have older print code and must print accordingly. But if the PrError return is 0, proceed:
- Call `SetRsl` with the print record and the desired resolution if you wish.
- Call `DraftBits` to invoke the printing of bitmaps in draft mode if you wish.

Note that if you call either `SetRsl` or `DraftBits`, you should do so before the user sees either of the printing dialogs.

Macintosh
Technical Notes

Developer Technical Support

#129: _Gestalt & _SysEnvirons—a Never-Ending Story

Revised by: Dave Radcliffe May 1992
Written by: Jim Friedlander May 1987

This Technical Note discusses the latest changes and enhancements in the `_Gestalt` and `_SysEnvirons` calls.

Changes since October 1991: Clarified information on Gestalt information for Macintosh PowerBook computers and added information on the Macintosh LC II and the `gestaltHardwareAttr` selector.

Introduction

Previous versions of this Note provided the latest documentation on new information the `_SysEnvirons` trap could return. DTS will continue to revise this Note to provide this information; however, as the `_Gestalt` trap is now the preferred method for determining information about a machine environment, this Note will also provide up-to-date information on `_Gestalt` selectors.

_Gestalt

This Note now documents `_Gestalt` selectors and return values added since the release of *Inside Macintosh* Volume VI. Please note that this is supplemental information; for the complete description of `_Gestalt` and its use, please refer to *Inside Macintosh* Volume VI.

The Macintosh LC II is identical to the Macintosh LC, except for the presence of an MC68030 processor, so it returns the *same* gestaltMachineType response as the Macintosh LC (i.e. 19). Developers are reminded that the gestaltMachineType selector is for informational purposes only and should not be used as a basis for programmatic decisions. As always, developers are encouraged to test for the specific features they need and not to rely on any particular machine having a particular set of features.

Note: The *Macintosh PowerBook 100 Developer Notes* and the *Macintosh PowerBook 140/170 Developer Notes*, available from APDA and on the *Developer CD Series* disc and AppleLink, incorrectly document `gestaltMachineType` response values for the Macintosh PowerBook computers. The following values are, and have always been, the correct values.

Additional Gestalt Response Values

```
{ gestaltMachineType response values }
        gestaltQuadra900            = 20;    { Macintosh Quadra 900 }
        gestaltPowerBook170         = 21;    { Macintosh PowerBook 170 }
        gestaltQuadra700            = 22;    { Macintosh Quadra 700 }
        gestaltClassicII            = 23;    { Macintosh Classic II }
        gestaltPowerBook100         = 24;    { Macintosh PowerBook 100 }
        gestaltPowerBook140         = 25;    { Macintosh PowerBook 140 }

{ gestaltKeyboardType response values }
        gestaltPwrBookADBKbd        = 12;    { PowerBook Keyboard }
        gestaltPwrBookISOADBKbd     = 13;    { PowerBook Keyboard (ISO) }
```

gestaltHardwareAttr Selector

The gestaltHardwareAttr selector has been a source of confusion for developers since originally documented in *Inside Macintosh* Volume VI . This section will try to reduce that confusion and also introduce additional information returned by the selector. But be warned that use of this selector for anything other than informational purposes should be deemed a compatibility risk. In other words, if you are dependent on the information returned by this selector to function on existing computers, you will almost certainly have problems on future systems.

The reason for this is that gestaltHardwareAttr returns very low-level hardware information. If you need to use this information, it implies you are too hardware dependent. So be very careful about using this information.

The principal source of confusion is bit 7, described as gestaltHasSCSI. What this bit really means is the machine is equipped with SCSI based on the 53C80 chip, which was introduced in the Macintosh Plus. This bit will be zero on the Macintosh IIfx and the Macintosh Quadra computers because they have a different low-level SCSI implementation. The Macintosh IIfx has a 53C80 compatible chip that also supports SCSI DMA. It reports this information using bit 6 of the gestaltHardwareAttr response. The Macintosh Quadra computers have yet another SCSI implementation based on the 53C96 chip and so report different information (see below).

Another source of confusion is bit 4 (gestaltHasSCC). The Macintosh IIfx and Macintosh Quadra 900 have intelligent I/O processors (IOPs) that normally isolate the hardware and make direct access to the SCC impossible. Normally, these machines will report that they do not have an SCC implying, correctly, that were you to attempt to access it directly, you would fail. However, if the user has used the Compatibility Switch control panel to enable compatibility mode, gestaltHasSCC will report true indicating you may access the SCC directly. But remember that doing so means you are doing direct hardware access and that there may be a day when you can't access the SCC under any circumstances.

New gestaltHardwareAttr Values for Macintosh Quadra Computers

Below are the new bits supported by the Macintosh Quadra computers. Any other bits remain undocumented and subject to change.

```
        gestaltHasSCSI961                     = 21; { 53C96 SCSI controller on internal bus
}
```

```
        gestaltHasSCSI962                  = 22; { 53C96 SCSI controller on external bus
}
```

_SysEnvirons

_SysEnvirons was the standard way to determine the features available on a given machine. The preferred method to get this information is now _Gestalt; information on _SysEnvirons is now provided only for backward compatibility.

As originally conceived, _SysEnvirons would check the versionRequested parameter to determine what level of information you were prepared to handle, but this technique means updating _SysEnvirons for every new hardware product Apple produces. With system software version 6.0, _SysEnvirons introduced version 2 of environsVersion to provide information about new hardware as we introduce it; this new version returns the same SysEnvRec as version 1.

Beginning with system software version 6.0.1, Apple releases a new version of _SysEnvirons only when engineering makes changes to its structure (that is, when they add new fields to SysEnvRec); all existing versions return accurate information about the machine environment even if part of that information was not originally defined for the version you request. For example, if you call _SysEnvirons with versionRequested = 1 on a Macintosh IIfx, it returns a machineType of envMacIIfx even though this machine type originally was not defined for version 1 of the call.

You should use version 2 of _SysEnvirons until Apple releases a newer version. MPW 3.0 defines a constant curSysEnvVers, which can be used to minimize the need for source code revisions when _SysEnvirons evolves. Regardless of the version used, however, your software should be prepared to handle unexpected values and should not make assumptions about functionality based on current expectations. For example, if your software currently requires a Macintosh II, testing for machineType >= envMacII may result in your software trying to run on a machine that does not support the features it requires, so test for specific functionality (that is, hasFPU, hasColorQD, and so on).

Warning: This test for specific functionality is particularly true of FPUs (floating-point units). Some CPUs, such as the Macintosh IIsi, may have optional, user-installed FPUs; therefore, an application should not assume that any Macintosh with a microprocessor greater than a 68000 (for example, 68020, 68030 or 68040) has an FPU (68881/68882 or built-in for the 68040). If an application makes a conditional branch to execute floating-point instructions directly, then it should first explicitly check for the presence of the FPU.

You should always check the environsVersion when returning from _SysEnvirons since the glue always returns as much information as possible, with environsVersion indicating the highest version available, even if the call returns an envSelTooBig (–5502) error.

Calling _SysEnvirons From a High-Level Language

Due to a documentation error in *Inside Macintosh* Volume V, DTS still receives questions about how to call _SysEnvirons properly from Pascal and C. *Inside Macintosh* defines the Pascal interface to _SysEnvirons as follows:

```
FUNCTION SysEnvirons (versRequested: INTEGER; VAR theWorld: SysEnvRecPtr) : OSErr;
```

Because `theWorld` is passed by reference (as a `VAR` parameter), it is not correct to pass a `SysEnvRecPtr` in the second argument. Pascal would then generate a pointer to this pointer and pass that to the `_SysEnvirons` trap in `A0`. (The assembly-language information is essentially correct; `_SysEnvirons` really does want a pointer to a `SysEnvRec` in `A0`.) The correct Pascal interface to `_SysEnvirons` is therefore:

```
FUNCTION SysEnvirons (versionRequested: INTEGER; VAR theWorld: SysEnvRec) : OSErr;
```

In this case, Pascal pushes a pointer to `theWorld` on the stack. The Pascal interface glue then pops this pointer off the stack directly into `A0` and calls `_SysEnvirons`. Everything is copacetic.

C programmers should recognize their corresponding interface:

```
pascal OSErr SysEnvirons (short versionRequested, SysEnvRec *theWorld);
```

Inside Macintosh defines the type `SysEnvPtr = ^SysEnvRec`. It also sometimes refers to this type as `SysEnvRecPtr`. The inconsistency is insignificant because in reality MPW does not define any such type, under either name; therefore, it is never needed.

Inside Macintosh also states that "all of the Toolbox Managers must be initialized before calling SysEnvirons." This statement is not necessarily true. Startup documents (INITs), for instance, may wish to call `_SysEnvirons` without initializing any of the Toolbox Managers. Keep in mind that the `atDrvrVersNum` field returns a zero result if the AppleTalk drivers are not initialized. The system version, machine type, processor type, and other key data return normally.

Additional _SysEnvirons Constants

The following are new `_SysEnvirons` constants which are not documented in *Inside Macintosh*; however, you should refer to *Inside Macintosh* Volume V-1, Compatibility Guidelines, for the rest of the story.

machineType

```
envMacIIx                = 5;     { Macintosh IIx }
envMacIIcx               = 6;     { Macintosh IIcx }
envSE30                  = 7;     { Macintosh SE/30 }
envPortable              = 8;     { Macintosh Portable }
envMacIIci               = 9;     { Macintosh IIci }
envMacIIfx               = 11;    { Macintosh IIfx }
envMacClassic            = 15;    { Macintosh Classic }
envMacIIsi               = 16;    { Macintosh IIsi }
envMacLC                 = 17;    { Macintosh LC }
envMacQuadra900          = 18;    { Macintosh Quadra 900 }
envMacPowerBook170       = 19;    { Macintosh PowerBook 170 }
envMacQuadra700          = 20;    { Macintosh Quadra 700 }
envMacClassicII          = 21;    { Macintosh Classic II }
envMacPowerBook100       = 22;    { Macintosh PowerBook 100 }
envMacPowerBook140       = 23;    { Macintosh PowerBook 140 }
```

processor

```
env68030                       = 4;     { MC68030 processor }
env68040                       = 5;     { MC68040 processor }
```

keyBoardType

```
envPrtblADBKbd                 = 6;     { Portable Keyboard }
envPrtblISOKbd                 = 7;     { Portable Keyboard (ISO) }
envStdISOADBKbd                = 8;     { Apple Standard Keyboard (ISO) }
envExtISOADBKbd                = 9;     { Apple Extended Keyboard (ISO)

envADBKbdII                    = 10;    { Apple Keyboard II }
envADBISOKbdII                 = 11;    { Apple Keyboard II (ISO) }
envPwrBkADBKbd                 = 12;    { PowerBook Keyboard }
envPwrBkISOKbd                 = 13;    { PowerBook Keyboard (ISO) }
```

Further Reference:

- *Inside Macintosh*, Volumes V and VI, Compatibility Guidelines

#130: Clearing ioCompletion

See also: The File Manager

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

When making synchronous calls to the File Manager, it is not necessary to clear `ioCompletion` field of the parameter block, since that is done for you.

Some earlier technotes explicitly cleared `ioCompletion`, with the knowledge that this was unnecessary, to try to encourage developers to fill in all fields of parameter blocks as indicated in *Inside Macintosh*.

By the way, this is true of all parameter calls—you only have to set fields that are explicitly required.

#131: TextEdit Bugs in System 4.2

Written by: Chris Derossi June 1, 1987
Updated: March 1, 1988

This note formerly described the known bugs with the version of Styled TextEdit that was provided with System 4.1. Many of these bugs were fixed in System 4.2. This updated Technical Note describes the remaining known problems.

TEStylInsert

Calling `TEStylInsert` while the TextEdit record is deactivated causes unpredictable results, so make sure to only call `TEStylInsert` when the TextEdit record is active.

TESetStyle

When using the `doFace` mode with `TESetStyle`, the style that you pass as a parameter is `ORed` into the style of the currently selected text. If you pass the empty set (no styles) though, `TESetStyle` is supposed to remove all styles from the selected text. But `TESetStyle` checks an entire word instead of just the high-order byte of the `tsFace` field. The style information is contained completely in the high-order byte, and the low-order byte may contain garbage.

If the low-order byte isn't zero, `TESetStyle` thinks that the `tsFace` field isn't empty, so it goes ahead and ORs it with the selected text's style. Since the actual style portion of the `tsFace` field is zero, no change occurs with the text. If you want to have `TESetStyle` remove all styles from the text, you can explicitly set the `tsFace` field to zero like this:

```
VAR
   myStyle   : TextStyle;
   anIntPtr  : ^Integer;

BEGIN
   ...
   anIntPtr := @myStyle.tsFace;
   anIntPtr^ := 0;
   TESetStyle(doFace, myStyle, TRUE, textH);
   ...
END;
```

TEStylNew

The line heights array does not get initialized when TEStylNew is called. Because of this, the caret is initially drawn in a random height. This is easily solved by calling TECalText immediately after calling TEStylNew. Extra calls to TECalText don't hurt anything anyway, so this will be compatible with future Systems.

An extra character run is placed at the beginning of the text which corresponds to the font, size, and style which were in the grafPort when TEStylNew was called. This can cause the line height for the first line to be too large. To avoid this, call TextSize with the desired text size before calling TEStylNew. If the text's style information cannot be determined in advance, then call TextSize with a small value (like 9) before calling TEStylNew.

TEScroll

The bug documented in Technical Note #22 remains in the new TextEdit. TEScroll called with zero for both vertical and horizontal displacements causes the insertion point to disappear. The workaround is the same as before; check to make sure that dV and dH are not both zero before calling TEScroll.

Growing TextEdit Record

TextEdit is supposed to dynamically grow and shrink the LineStarts array in the TERec so that it has one entry per line. Instead, when lines are added, TextEdit expands the array without first checking to see if it's already big enough. In addition, TextEdit never reduces the size of this array.

Because of this, the longer a particular TextEdit record is used, the larger it will get. This can be particularly nasty in programs that use a single TERec for many operations during the program's execution.

Restoring Saved TextEdit Records

Applications have used a technique for saving and restoring styled text which involves saving the contents of all of the TextEdit record handles. When restoring, TEStylNew is called and the TextEdit record's handles are disposed. The saved handles are then loaded and put into the TextEdit record. This technique should not be used for the nullStyle handle in the style record.

Instead, when TEStylNew is called, the nullStyle handle from the style record should be copied into the saved style record. This will ensure that the fields in the null-style record point to valid data.