Index

Note: Trap names can appear in two different places: under the name of the trap alphabetically and at the end of the index, preceeded by an underscore (_).

#1: Desk Accessories and System Resources

See also: The Resource Manager

Written by: Bryan Stearns
Updated:

February 25, 1985
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 Scott Knaster	January 21, 1986
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.

#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
Updated:

April 4, 1985
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
Updated:

April 4, 1985
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.

#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
Updated:

April 16, 1986
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
Updated:

April 21, 1986
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.

Macintosh
Technical Notes

Developer Technical Support

#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

#10: Pinouts

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

Written by: Mark Baumwell
Modified:
Updated:

April 26, 1985
July 23, 1985
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

#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

#14: The INIT 31 Mechanism

See: The System Resource File
The Start Manager

Written by: Bryan Stearns
Updated:

March 13, 1986
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.

#18: TextEdit Conversion Utility

See also: Macintosh Memory Management: An Introduction
TextEdit

Written by: Harvey Alcabes
Updated:

April 10, 1985
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.
```

Macintosh
Technical Notes

Developer Technical Support

#19: How To Produce Continuous Sound Without Clicking

Revised by: Jim Reekes
Written by: Ginger Jernigan

June 1989
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`
		
		`ATPCloseSkt`
			The server kisses Dave goodbye. After the Response operation completes, the server closes the socket Dave was using. 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.).

#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).

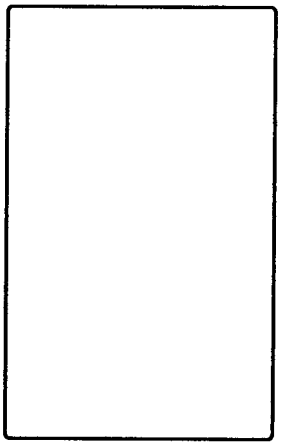
`OpenPicture` and `DrawPicture` set up a default set of port characteristics when they start. When drawing occurs, if the user's settings don't match the defaults, mode opcodes are generated. This is why there is usually a `clipRgn` code after the `picVersion`: the default clip region is an empty rectangle.

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
Updated:

April 21, 1986
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
Updated:

April 25, 1985
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.

#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
Updated:

June 25, 1986
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.

#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

#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
Updated:

May 7, 1984
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
Modified by: Ginger Jernigan

May 7, 1985
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
Updated:

April 25, 1986
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.

#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*).

#33: ImageWriter II Paper Motion

Written by: Ginger Jernigan
Updated:

April 30, 1986
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.

#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
Updated:

June 19, 1986
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
Updated:

June 12, 1985
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.

#38: The ROM Debugger

Written by: Louella Pizzuti
Updated:

June 20, 1986
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

#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

#42: Pascal Routines Passed by Pointer

See also: Macintosh Memory Management: An Introduction

Written by: Scott Knaster
Updated:

July 22, 1985
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
Updated:

October 15, 1985
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:

```
{$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" }
  }
};
```

#48: Bundles

See also: The Finder Interface

Written by: Ginger Jernigan
Updated:

November 1, 1985
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!

#50: Calling SetResLoad

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

Written by: Jim Friedlander
Updated:

October 25, 1985
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
Written by: Jim Friedlander

April 1989
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.

#53: MoreMasters Revisited

See also: The Memory Manager

Written by: Jim Friedlander
Updated:

October 28, 1985
March 1, 1988

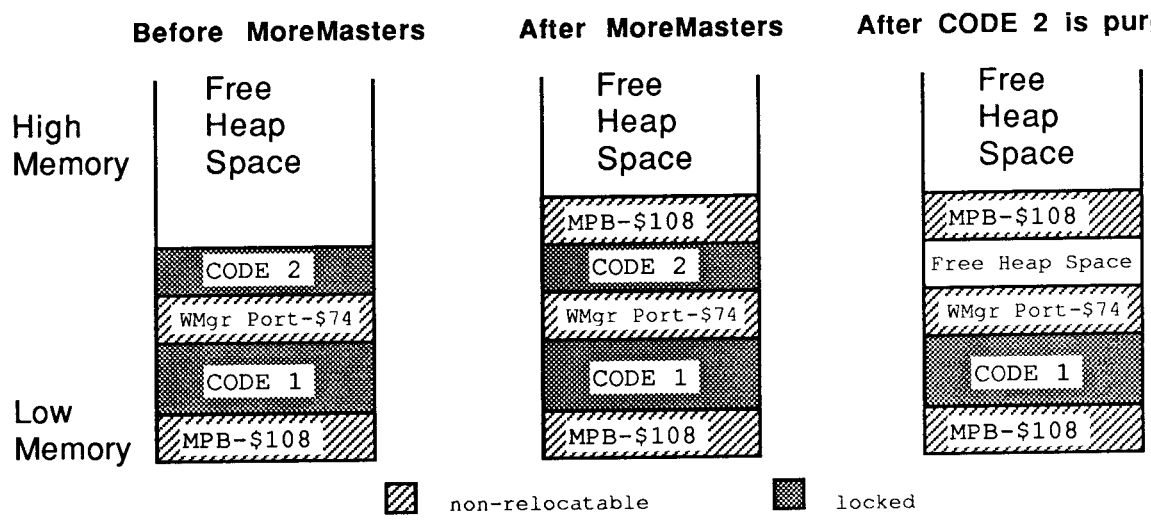
`MoreMasters` should be called from CODE segment 1. The number of master pointers that a program needs can be determined empirically. `MoreMasters` can be tricked into creating the exact number of master pointers desired.

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 */
```

#54: Limit to Size of Resources

Written by: Jim Friedlander
Updated:

October 23, 1985
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
Updated:

October 21, 1985
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
Updated:

December 2, 1986
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
Updated:

January 8, 1986
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
Updated:

January 24, 1986
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
Updated:

January 16, 1986
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
Updated:

January 20, 1986
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
Updated:

January 21, 1986
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.

#62: Don't Use Resource Header Application Bytes

See also: The Resource Manager

Written by: Bryan Stearns
Updated:

January 23, 1986
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.

#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.

#64: IAZNotify

Written by: Jim Friedlander
Modified by: Jim Friedlander
Updated:

January 15, 1986
August 18, 1986
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.

#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

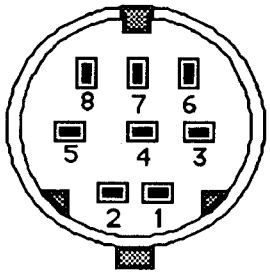
This note gives pinout descriptions for some of the Macintosh Plus ports and Macintosh Plus cables that are different than the Macintosh 128K and 512K.

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

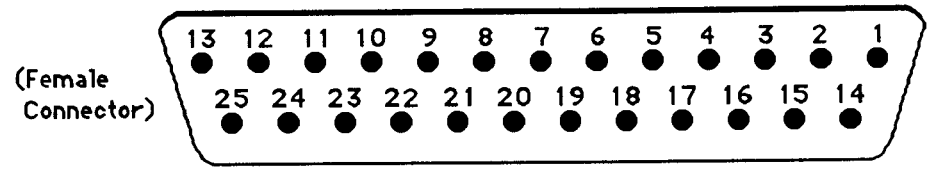
Macintosh Plus Serial Connectors (Mini DIN-8)

(Female Connector)



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)

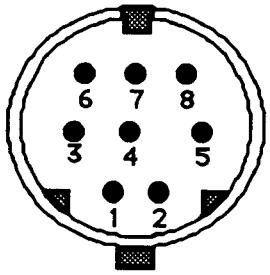


Pin	Name	Description/Notes
1	REQ–	
2	MSG–	
3	I/O–	
4	RST–	
5	ACK–	
6	BSY–	
7	Ground	
8	DB0–	
9	Ground	
10	DB3–	
11	DB5–	
12	DB6–	
13	DB7–	
14	Ground	
15	C/D–	
16	Ground	
17	ATN–	
18	Ground	
19	SEL–	
20	DBP–	
21	DB1–	
22	DB2–	
23	DB4–	
24	Ground	
25	TPWR	Not connected

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).

(Male Connector)



(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
Written by: Jim Friedlander

August 1990
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