

Programming Notes for *FontShow 1.0* by Richard Clark

Introduction

FontShow 1.0 is a simple application designed primarily to demonstrate two of the least understood areas of Macintosh programming – Window Updating and Memory Management. It should run on any Macintosh from the original 128K to a Mac II, but the memory management could use some improvement for it to run its best on the original 128K and 512K Macintoshes. (This, as the saying goes, is left as an exercise to the reader.)

FontShow's Windows

FontShow supports two types of windows: an “About” window (containing a picture which gives some information about the program and a line showing the amount of memory remaining) and a multiple “Font” windows, each of which contains a grid displaying the entire ASCII character set and the Hexadecimal code for each character:

	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$	\$
	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\$0X																
\$1X																
\$2X		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
\$3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
\$4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
\$5X	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
\$6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
\$7X	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
\$8X	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
\$9X	ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
\$AX	†	°	¢	£	§	•	¶	ß	®	©	™	'	¨	≠	Æ	Ø
\$BX	∞	±	≤	≥	¥	μ	∂	Σ	Π	π	∫	ª	º	Ω	æ	ø
\$CX	¿	¡	¬	√	ƒ	≈	Δ	«	»	...		À	Ã	Ö	Œ	œ
\$DX	–	—	“	”	‘	’	÷	◊	ÿ	Ÿ	/	€	<	>	fi	fl
\$EX	‡	·	,	„	‰	Â	Ê	Á	Ë	È	Í	Î	Ï	Ó	Ô	
\$FX	□	Ò	Ú	Û	Ü	ı	ˆ	˜	˘	˙	˚	¸	˝	˛	˜	˘

The font windows are stored as resource 128 in the resource file, and are standard Document windows. Font windows may be resized, though neither Size box nor scroll bars are provided.

When a Font window is created (by the OpenAWindow procedure in the Windows unit), the number 128 is stored in its RefCon field, which is a long integer field provided by Apple for the programmer's own use. (We use the RefCon field to distinguish between the 2 types of windows.) We then create a picture (see procedures AttachWindowPict and DrawFontGrid in the Windows unit)

and place its handle in the WindowPic field of the newly created Font window.

If the WindowPic field of a given window is not **nil**, then the system assumes that it contains the handle to a PICT and, instead of reporting update events to your program as they happen, the System will redraw the window itself using the attached picture. This automatic updating doesn't occur the moment the window needs updating, but waits until the next time you call GetNextEvent or WaitNextEvent. Using the WindowPic field is useful primarily when the contents of the window never change, or change infrequently (as in this case). To change the window, you need to do 4 things: 1) Dispose of the old window picture; 2) Create and attach a new picture; 3) Call InvalidRect for the drawing area of the window; and 4) Erase that area (the system will not erase it before redrawing), or include an EraseRect command in your picture definition.

If you look a little farther down in the Windows unit, you may notice that we have some code which is designed to redraw a Font window if we ever receive an update event for one. No, this isn't because the System will sometimes "forget" to update a window containing a WindowPic (it doesn't forget), but because we might need to throw away one or more of the WindowPics if we begin running low on memory. See the section on "Memory Management and Recovery" below.

When a Font window is closed, we have to remember to dispose of the picture's handle before calling DisposeWindow to close the window and get rid of its storage.

When a Font window is brought to the front, via an activate event, we get its font and font size and store them into two globals (currFont and currSize). These globals are used by the Menu manager to keep the Font and Size menus up to date.

The About window is handled in a slightly different manner. Since the program displays both a picture and the amount of free memory at the bottom, this window cannot be handled with a single PICT; setting it up as a dialog also would be difficult. So, the program uses the "standard" updating technique where it redraws the window each time an update event occurs. To differentiate this window from a Font window, we set its RefCon field to 129. (This isn't done in the program but in the resource definition, since we know that the About window will always have a RefCon of 129.) Only 1 About window is allowed, its pointer is stored in the global variable AboutWindow.

When the About window is opened, we store a pointer to it in AboutWindow, then calculate the location of the memory information string at the bottom of the window. (The string, which contains both the words "Available memory:" and a number, is centered in the window and displayed 4 pixels below the bottom of the picture. The starting coordinates of this string and a rectangle which encloses the numeric part are stored in the variables MemSizeRect, MemSizeLeft, and MemSizeBottom, which are local to the Windows unit.) To calculate the position of this string, we first load in PICT resource 128 which comprises the upper part of the window. Subtracting 16 from the bottom coordinate of this picture gives us the baseline for the memory string. We then load in STR resource 128, which contains the prompt text, and measure it with the StringWidth procedure. (NOTE: Since the string is loaded in as a Handle, and the StringWidth procedure might cause unlocked handles to move around in memory, we have to lock the string down first with HLock and unlock it afterwards. Otherwise, we might get some pretty strange measurements!) Assuming that our memory size number will never get to more than 10 characters (no special reason for this number – it's just an educated guess), we add 10 times the width of the number 0 to get our combined width for the memory string. This string is then centered, and the starting coordinates are stored. We also store the rectangle which encloses the

memory size number.

This rectangle is used to force a window update whenever the available memory changes. See the `UpdateAboutWindow` procedure for more details.

To redraw the About window, we load in the picture and draw it. The memory string is then loaded and drawn (again, locking the string down before calling `QuickDraw`), then the memory size number is converted to a string and drawn. Piece of cake.

Closing the About window is also fairly simple. We get the handle to our PICT resource, then call `ReleaseResource` to remove it from memory. Then we call `DisposeWindow` to get rid of the window, store **nil** in its pointer, and that's all.

FontShow's Menus

The Menus unit contains 3 procedures worth noticing – `AdjustMenus`, `SetFontMenu`, and `SetSizeMenu`. All three of these procedures are concerned with giving the user feedback about the current state of the program, including which commands are available at any given time. This kind of feedback, if done properly, is one of the things that makes Macintosh software especially easy to learn and use.

AdjustMenus concerns itself with showing which options and menus are available at any given time. Notice that any time we enable or disable an entire menu, we need to call `DrawMenuBar` (which unfortunately, causes the whole menu bar to flash. Oh, well.)

SetFontMenu is also fairly simple. Using the global variable `currFont`, we get the current font's name and compare it to each entry in the Font menu. If they match, place a checkmark by the name; if they don't match, clear any mark that may be present.

SetSizeMenu is slightly more complex. This time, we convert each menu entry into a number. Since the `StringToNum` procedure has the nasty habit of treating letters as numbers (by virtue of clearing bits 7 and 8), we have to extract just the numeric portion of the string before calling `StringToNum`. After we have the number, we check to see if the current font exists at that size. If so, set the item's style to `outline`, otherwise, reset it to plain. Also, place a checkmark there if the item's number matches the current size, otherwise remove any mark that may be present.

Memory Management and Recovery

FontShow's memory management was designed with two primary goals in mind – avoiding heap fragmentation (see page I-74 of Inside Macintosh) and avoiding the dreaded System Error 25 (“Out of Memory”). Heap fragmentation is controlled by not locking handles unnecessarily and using a technique called *Preallocation*, while we avoid sudden memory outages using a technique of *Memory Recovery*.

Avoiding Fragmentation

Code segments are often a major cause of heap fragmentation. When the Macintosh runs a procedure in a different code segment than the current one, it loads the segment into memory, *locks the segment down*, and sets up a table containing the addresses of the procedures in the segment. Since the system has no way of knowing when we are finished with the segment, it remains locked down in memory. Beginning with the 512Ke, the code segments are moved clear to the top of the heap before they are locked down, which helps control fragmentation, but if we give the system a little more help, we can all but eliminate fragmentation.

FontShow contains 4 segments – the Main segment (units Globals, MemHandler, and the FontShow program), an Initialization segment (unit Initialization), a Menu handling segment (unit Menus), and a Window handler (unit Windows). Since we use the Initialization segment only once, it can be unlocked (and thrown away) immediately after we call it. The Main segment always has to stay locked down, since we're either executing something in it, or we've called a subroutine which expects to return to the Main segment when it is done. (If you call something in the Menus segment, and it unloads the Main segment, when the subroutine returns it will begin executing random code. This, as the Surgeon General says, could be hazardous to your health.)

Even though the Main segment has to stay locked down at all times, the other segments should be unlocked when we aren't using them. This way, they can be moved up to the top of the heap and won't interfere with other memory allocations. So, at the beginning of the main loop, we call *UnloadSeg* twice – once for the Windows handler and once for the Menus handler. Notice that *UnloadSeg* does two things – it clears the table of procedure addresses it maintains for that segment, and it unlocks the segment's handle. The segment itself is not actually removed from memory until that memory is needed, therefore you don't have to worry about the software's performance degrading because segments are being pulled in from disk every time. This won't happen unless we're running really low on memory.

Generally speaking, we use Handles throughout the program, so if we keep the segments unlocked, then fragmentation shouldn't be a problem. However, each one of our windows uses a Pointer to a block, and these cannot be moved around. Normally, most Macintosh programmers just supply a **nil** to the Window Manager when asking for a window and the Window Manager allocates the storage. *This is not a good idea*. The memory manager always tries to allocate pointer starting at the bottom of the heap and working its way upwards. However, since we might have a nice, big segment sitting in the middle of the heap, the pointer could get allocated just above it. Instant fragmentation. So, what we'll do is to allocate our pointers when all of the segments are unloaded

(and free to move to the top of the heap) – namely, from inside of the main loop. In this way, Handles get allocated from the top down, and pointers get allocated from the bottom up. This leaves the area in the middle (the free memory) as open as possible, and avoids fragmentation. Since we are allocating our pointers before we need them, this is called *Preallocation*.

We store the pre-allocated pointer into the global variable WindowReserve. When we create a window, we pass WindowReserve to the Window Manager, then set WindowReserve to **nil**. This tells the preallocation routine that the pointer has been used and that we should allocate a new pointer.

When we dispose of a window, we can use the opportunity to try and control the fragmentation a little more. Not only do we dispose of the window's pointer, which frees up that memory, but we also dispose of the preallocated block. If the old window's pointer was lower down in memory than the preallocated block's, the preallocation routine will move the allocation down to the old window's block (since the preallocated block is usually the topmost block, this frees up some more memory for pointers). You can see this in the CloseAWindow procedure in the Windows unit.

Memory Recovery

The other memory management technique we use is designed to keep the program from suddenly running out of memory. We base our strategy on the fact that the picture handles attached to our Font windows are really a luxury – we could redraw the windows ourselves, it's just a little slower, that's all. So, when we are about to run out of memory, we can start throwing away window pictures to get a little more space. Of course, when all of the pictures are gone, then we really are out of memory. But by then, we can warn the user that we're about to crash. If he doesn't quit by then, it's his own fault!

The Macintosh was designed with this technique in mind. When you allocate a block of memory, the system looks for the smallest empty block of which is large enough to satisfy your request. If nothing could be found, it *compacts* the heap by moving handles until it can find enough room for your block. If that fails, it looks for resources that are marked *purgeable*, and removes them from memory one by one, compacting each time, until it finds enough room. Still not enough space? Then the system will try to expand the Application heap using the MaxApplZone procedure. If that fails, then you're out of memory. Well, almost. If you supply the address of a "Grow Zone Function", the system will call that and ask you for the memory.

Often, a programmer will load a resource into memory and then, not wanting to load it in from disk each time, will mark it as non-purgeable. This doesn't lock it down, but it does keep the memory manager from throwing it out early in the search for more memory. Or, you can have such "luxuries" as window pictures (as we do). If the program is written in such a way that it doesn't assume that these things are in memory all of the time, then we can mark them as purgeable and/or throw them away when the system asks for more memory. This is the job of the Grow Zone function, and it can make a program much friendlier. That way, the system exhibits a *graceful degradation*, instead of crashing to an outright halt.

When your GrowZone function is called, the system is about to run out of memory. So, you can supply it some memory and notify the user that the system is running low. The best technique is to let go of memory one block at a time until you run out; the system will call you again if that wasn't enough. This way, the user will probably get multiple warnings (and chances to close windows, save files, and quit) before the system dies. If you let go of all of the memory at once, then the next time the system runs out of memory will be the last. At least for this program.

So, how do you write a Grow Zone function? The grow zone function a long integer (the amount of memory the system wants), and returns another long integer which contains a result code. If your function was able to scrounge up any memory, it should return a non-zero value (any one will do). The system will then try compacting the memory and purging resources to find a single block that's large enough. If it can't, you'll get called again and again until it has enough memory or until you return a 0. Zero means "out of memory", and it's a sure ticket to the "Bomb Box".

When the Grow Zone function is called, it should set a flag which indicates that the system is running low on memory. This flag can be tested in your main loop and if set, can put up an alert warning the user of that situation. You don't want to call the alert from inside of the Grow Zone function because, if there isn't enough memory for the alert, the Grow Zone function will get called again and you would have an alert called from inside of an alert. The Dialog Manager doesn't like that, trust me.

You install the Grow Zone function by calling SetGrowZone during your initialization sequence and passing in a pointer to your function. The Grow Zone function should be located in your main segment, as it could get called at any time.

How to die gracefully

If, despite our best efforts, we do get a System Error, there's one more thing we can do. The System Error box has 2 buttons labeled "Restart" and "Resume", but the Resume button is normally dimmed. If, during initialization, we supply the InitDialogs procedure with the address of a procedure of our own, clicking the Resume button will call this "resume" procedure to be called. Our resume procedure is called SafetyNet, and is located in the Globals unit. All SafetyNet does is exit to the Finder, but this is much friendlier than forcing the user to reboot the machine.

Known Bugs

- 1) When the contents of a Font window are copied to the clipboard, they do not appear as they should. If you paste it into the Scrtepbok, the window will appear to be blank. If you paste it into MacWrite, try resizing the picture a little bit to make it appear. Everything seems to be fine if you paste into MacDraw or the equivalent.
- 2) The program sometimes hangs when quitting. I've only noticed this when running with TMON installed on a Mac II.

Possible improvements

- 1) Improve the memory management on Mac 128K's and 512K's. The primary thing to do here is to load in a CODE resource yourself before calling a given segment, call MoveHHI to move it to the top of the heap, then lock it down. You can unlock it after the routine is done, but it should again be moved to the top after locking.
- 2) Increase the amount of memory information in the About box to include the largest free block, and a fragmentation index.
- 3) Implement a call to SysEnvirons preparatory to the next improvements:
- 4) (512K's and later) Implement Zoom boxes on the windows. This is tricky, as the system seems to get confused if you zoom a window with an attached WindowPict – things don't get redrawn correctly).
- 5) Use WaitNextEvent to make this a bit more compatible with MultiFinder.

For more information

You can contact the author, Richard Clark, at
GEnie/DELPHI/MCI Mail: RDCLARK
CompuServe: (Use the MCI gateway)
The MouseHole (Orange County, CA): RDCLARK
The Desktop BBS (Orange County, CA): Box 003

Southern California residents should contact me (Richard) if you are interested in attending my Macintosh Programming in Pascal class.

Some useful references are:

Inside Macintosh Volumes 1-5, Apple Computer Inc.(Addison-Wesley Publishing Company, Reading, Massachusetts, 1985-1988)

Macintosh Technical Notes Apple Computer inc. Developer Technical Support (available from APDA (boo, hiss) or by downloading from GEnie and other on-line sources. Or, you can contact Apple about their Certified Developer program. But if you knew that, you probably already have these)

How to Write Macintosh Software Scott Knaster (Hayden Book Company, Hasbrouck Heights, NJ, 1986) ISBN 0-8104-6564-7
This has especially good sections on memory management and debugging.