Being "MultiFinder Aware"

Introduction

MultiFinder is Apple's newest System Software wrinkle for the Macintosh. The major new functionality available under MultiFinder is the ability to run multiple applications at once, all of them sharing the same screen or display world. Multifinder thus provides the closest thing to multitasking available yet on the Mac, except perhaps for things such as UNIX which are just getting off the ground. Apple seems to have done a great job in introducing this new functionality without greatly altering either the familiar Macintosh environment for the user nor the programming model for the developer. Indeed, many applications run correctly under MultiFinder, but many do not. Moreover as of this writing, very few applications take full advantage of MultiFinder's background support and are fully "MultiFinder Aware".

If an application's developer has been good and followed the rules laid down at various times and in various fashions by Apple, chances are that the application will run correctly under MultiFinder and that it may be modified relatively easily to take full advantage of MultiFinder. The most common features which an application developer may want to utilize include proper window display when suspended, background processing, temporary memory allocation, and new goodies such as the Notification Manager and Inter-Applications Communications. These last two will be discussed briefly at the end of this paper, being relatively new (in fact, not yet released at the time of this writing.) Temporary memory allocation has not been utilized by this author, and so it will be mentioned only in passing. However, the first two items mentioned, which are in themselves "noble aspirations", are relatively easy to accomplish, and should be considered minimum requirements for "MultiFinder Awareness". This statement assumes an application is already MultiFinder friendly.

But sometimes a programmer will take an undocumented and/or unapproved shortcut. Some examples of these include fussing with various low memory variables, using CopyBits to save and restore a portion of the screen while a dialog or alert is up, drawing directly to the screen rather than in windows or other grafports, messing too much with the trap table, messing with memory blocks without going through the proper Memory Manager calls, and assuming the size and location of the application and system heaps. Now, if you've done some of these so-called "bad things", you may still be OK. In fact, if your application is already compatible with Switcher (which is no longer supported by Apple), it is probably reasonably compatible with MultiFinder, since the application environment differences between Switcher and MultiFinder are minimal.

Some General Rules of the Game

First, let's discuss some of the do's and don'ts of being "MultiFinder friendly". Then, I'll describe some of the new things (WaitNextEvent, temporary memory allocation, and the new version of the SIZE -1 resource), after which I'll present some of the techniques I used in Pro-Cite as an example of how one might provide some of the functionality needed to be "MultiFinder Aware". The rules fall into one of four categories: General or Global (pun intended) rules, Window Manager rules, Event Manager rules, and Memory Manager rules. I will not present an exhaustive list, but just hit the big ones. For a complete set of guidelines, you may want to acquire the *MultiFinder Development Package*.

General/Global Rules

- Don't access low memory more than necessary. Things which are clearly documented in *Inside Macintosh* or in Technical Notes are probably OK, but Apple could decide to change them in the future, too!

- Don't change the Apple menu except through the proper ToolBox calls. MultiFinder basically feels it owns the Apple menu and could get nasty if you change it directly rather than using more "normal" methods.

- I've always felt that patching traps was an ignoble thing, but maybe that's just due to my own limited experience! Anyway, if you're going to engage in this activity, use the SetTrapAddress calls rather than writing into the dispatch table in low memory and place your patch receiving routines in your application heap, not in the System heap. You'll be glad you did. Also, remember to unplug all your patched traps before exiting your application. Although I haven't seen words to this effect in any Apple documentation, it may be a good idea to unplug your patches on suspend events, too. I've had some bad experiences with not doing this when I tried to patch TextEdit for tabs. Better to be safe than sorry.

- Don't access your global data from within an interrupt handler, a patch receiving routine, an I/O completion routine, or a VBL task. It isn't even safe to presume that the value in CURRENTA5 is correct! Instead, save a copy of your A5 in some data structure you're sure you can find (such as a parameter block that's passed in) when your routine is called. See Technical Note #180 for an example of how best to do this.

- Use the Scrap Manager to access the scrap, rather than manipulating either the low memory scrap data or the Clipboard file itself directly.

- Don't assume that at exit time (about when you're ready to call ExitToShell, or otherwise fall off the end of your program) anything goes. Don't clear the screen with a second call to InitWindows or something like PaintRgn(GrayRgn, DeskPat). Don't blow away system data structures like the WindowList before exiting.

Window Manager Rules

- Don't EVER modify Window Manager data structures (including the window record and GrafPort fields, plus any low memory goodies) directly, rather use the proper Window Manager ToolBox calls. You can do whatever you want to with off-screen GrafPorts.

- Do all drawing to the screen within the bounds of a window with appropriate ToolBox (QuickDraw) calls. Always draw only into GrafPorts (this of course includes windows) that your application has created. Most game programs are notorious for not doing this (Try Crystal Quest under MultiFinder…first click, and you're switched out with only "some" of the Finder display surviving!)

- Don't bypass the proper techniques for updating windows obscurred by a dialog or alert after it's dismissed. Some applications use CopyBits to do this, but under MultiFinder they might just end up putting back "old" bits from the Finder display or from another running application.

- Consider the Window Manager port to be off-limits, or at least read-only. Don't draw on the desktop (at least, not in a destructive manner…zoom effect lines and such seem to be OK, for now.)

Event Manager Rules (it certainly does!)

- The most important thing to do is treat all update events seriously. When an application is running in the background (or at least switched out/suspended: even those applications which do not support background processing will often find themselves in this state), it will receive update events whenever the foreground application has obscurred and then exposed one of its windows. Applications should not implement deferred window updating schemes but should respond (in other words, DRAW) directly upon receiving an update event.

- Null events should be treated more seriously as well. Null events will be used by MultiFinder to provide time for background applications. Hence, time consuming activities such as garbage collection should not be performed on every null event received. Use the TickCount to arbitrate use of null events, except for cursor tracking.

- Don't call SystemTask on every (null) event, but rather call GetNextEvent (or preferably WaitNextEvent, see the Pro-Cite discussion later). Functions previously supported by SystemTask are now handled properly by GetNextEvent/WaitNextEvent.

- Support the suspend/resume events. They can eliminate most of the time required for a switch, since MultiFinder also does the same desk accessory charade that Switcher did, if the application doesn't indicate (via the SIZE -1 resource) that it supports suspend/resume events. Besides, supporting suspend/resume events is the only sure way you can keep track of whether or not your application is suspended (and you WILL want to do that!)

Memory Manager Rules

- Don't make assumptions regarding application or system heap size or location. In fact, your application should be written such that it doesn't care where the heap is or what size it is (except when it runs out of memory, of course!) Use GetApplLimit to get at the size of your heap and use SetApplLimit to resize your stack.

- Allocate additional heaps within your original heap as non-relocatable blocks or else within your stack. Consider your application's available memory to be the application heap and stack ONLY.

- Don't assume which heap a particular resource is loaded, unless it is a resource you loaded from a file you opened directly. MultiFinder dynamically loads resources from the System file and from printer resource files into a number of different heaps that it maintains.

- Try to use MultiFinder's temporary block allocation calls for unusual needs such as copy buffers, and thus keep the normal memory requirements of your application smaller.

- Support the SIZE -1 resource to describe your application's memory requirements and capabilities to MultiFinder. The minimum size should be sufficient for the application to perform some useful work while the maximum size should be no larger than than which the application uses when exercised under normal circumstances. Let the user set the size larger if need be. A preferred size of 2 Megabytes is probably excessive.

New Wrinkles from MultiFinder

MultiFinder introduces some new system functionality in the form of temporary memory allocation and enhances existing functionality with the WaitNextEvent ToolBox call. This section is also a good time to cover the new version of the SIZE -1 resource.

WaitNextEvent

By far the most important change, and the focus of providing real MultiFinder support in an application, is the WaitNextEvent trap. An application can perform properly under MultiFinder by just continuing to use GetNextEvent, but true background task support can only really be achieved with WaitNextEvent. The prototype for this trap is

```
FUNCTION WaitNextEvent(VAR theEvent : EventRecord;
                       theMask : EventMask;
                       YieldTime : INTEGER;
                       MouseRgn : RgnHandle) :       BOOLEAN; INLINE $A860;
```

The first two parameters (the event record and event mask) are identical to those in the familiar GetNextEvent call. The YieldTime parameter (also called the "sleep" parameter) indicates how much time to "give up" to any background applications. It may be thought of as how many ticks for the Event Manager to "go away and visit other applications" before returning a null event to allow this application to do garbage collection type activities or possibly background processing. A value of zero will cause the Event Manager to "return" immediately after still providing some minimal time to any other processes currently active; this is essentially equivalent to the GetNextEvent case. When the time specified by YieldTime elapses, WaitNextEvent returns a null event and a return value of FALSE. If a "real" event (e.g., update event if in the background or anything else including an update event if foreground) occurs before the YieldTime value elapses, WaitNextEvent returns the event immediately, along with a return value of TRUE.

The last parameter, MouseRgn, is a handle to a region which describes the area in which the cursor may maintain its current setting, as desired by the application. If this parameter is not NIL, MultiFinder will generate a special type of an event, called a "mouse-moved" event, whenever the cursor has been moved outside of the given region. The application then may change the cursor and generate a new MouseRgn to pass to WaitNextEvent. I found that some of the regions I needed to describe were rather complex and thus I passed NIL for the MouseRgn parameter. This meant that no mouse-moved would be generated. While the purpose of the mouse-moved events is to improve performance, I found that Pro-Cite's cursor-tracking routine was simple enough that it could be executed on each event provided by MultiFinder (except when Pro-Cite is suspended, of course) and still provide adequate performance to other processes.

Temporary Memory Allocation

MultiFinder provides a temporary memory allocation service to help reduce the memory requirements for an application's heap. It provides the ability to allocate and release handles, lock and unlock handles to blocks within a special MultiFinder heap zone. This service would be particularly useful for graphics or animation buffers or for disk/file or resource copying buffers. In fact, the Finder now uses these temporary memory calls for copy buffer space during file copy operations. In Pro-Cite, I found little need for this service and so did not make use of it. See the *MultiFinder Development Package* for full details on the temporary memory allocation features of MultiFinder.

The New SIZE -1 Resource

To be truly MultiFinder aware, an application must include a SIZE -1 resource, as introduced in the time of Switcher. The SIZE resource both indicates an application's memory requirements as well as its degree of MultiFinder compatiblity. The format of this resource (in MPW Rez terms) is:

```
resource 'SIZE' (-1, purgeable) {
        saveScreen ("reserved"),
        acceptSuspendResumeEvents/dontacceptSuspendResumeEvents,
        enableOptionSwitch ("reserved"),
        canBackground/cannotBackground,
        multiFinderAware/notmultiFinderAware,
        (11 bits "reserved")
        Preferred Size,
        Minimum Size
};
```

We'll just ignore the bits marked "reserved". Bit 11 indicates that the application is aware of MultiFinder, if set. Most applications will have this bit on, but some may function properly only when this bit is off. If this bit is set, MultiFinder will not generate activate/deactivate events for the frontmost window at resume/suspend event times, but will expect the application to do this itself, which is the most efficient way. Bit 14 indicates support for suspend/resume events if set (an application will NOT receive suspend/resume events unless this bit is set), while Bit 12 indicates the capability of supporting background null events for background processing (remember, ALL applications should be capable of supporting update events at all times!) Most applications should eventually provide support for suspend/resume events while only a few initially will be capable of background processing under MultiFinder (Pro-Cite is one of these, else why would I be writing this paper?)

The preferred memory size is no easier to determine under MultiFinder than it was for Switcher. It is something best determined by "inspection" (a euphemism for "trial and error.") The minimum size should be chosen such that the application could provide the user with a minimal amount of useful work and should never (well, almost never) give a "system error". The perferred size should be chosen such that 90% of the application's functionality may be utilized without memory problems. The application in any case should never be too greedy with its memory requirement. Remember that the application may have to exist in harmony with a number of other applications at the same time. An application with a large preferred size such as 1024K will be looked down upon by users and developers of other applications alike, unless it's a development system (or perhaps it's just FullWrite.)

Pro-Cite has the three options all set to TRUE (multiFinderAware, acceptSuspendResume, and canBackground), a preferred memory size of 384K (which actually can be considered an operational minimum) and a minimum memory size of 224K (which is really left over from PBS support for Switcher.)

Pro-Cite®--A (Mostly) MultiFinder Aware Application

Pro-Cite®, the new replacement for the Professional Bibliographic System for the Macintosh, is a vertical market application and a bibliographic database program which has aspirations of being a word-processor besides. When I was well along in the design and implementation of Pro-Cite in the spring of 1987, Apple made PBS a beta test site for a new version of the Macintosh System then called "Juggler" (rumor had been rampant about it anyway). Fortunately, the Professional Bibliographic System had been designed and implemented in a Macintosh standard and Switcher-friendly way so that only a few problems had to be ironed out to make it compatible with MultiFinder. As it turned out, most of the PBS processing tasks had also been written in loops which regularly called SystemTask to support desk accessories, so that MultiFinder background processing support was not all that difficult to provide. For the remainder of this paper, I shall try to describe some of the code and techniques which allow Pro-Cite to be more or less fully MultiFinder aware, and I'll also describe support for the new Notification Manager for System 6.0, which Apple gave PBS the opportunity to test in early 1988. Refer to the attached MPW Pascal code listings to augment the description below as necessary. In many places, I have left out irrelevant statements to try to make the code clearer and easier to read, while in other cases I've left certain items in so that application developers can identify certain "common" areas of a Macintosh application.

The first task before the MultiFinder programmer is to determine if MultiFinder is running in the first place, since the user can turn MultiFinder off, and with the shortage of 1Meg SIMMs this year, will probably often want to. We don't want to write a MultiFinder-only application just yet, do we? Using the technique described in Technical Note 158, the Setup routine, which is called when Pro-Cite first starts up, uses SysEnvirons to see if the 128K ROM is present, since it is required for MultiFinder, and NGetTrapAddress to see if WaitNextEvent is implemented, placing the result in the BOOLEAN variable "WNEisImplemented" (well-named, I think.) It also sets the initial value of the sleep parameter "YieldTime" to 3 as well as the foreground value for this, "frontYieldTime", and the background value "backYieldTime" to 10. These were determined by trial and error and will be used to change the value of "YieldTime" on suspend/resume events. The user can also change these values through Pro-Cite's Configure option (in an undocumented manner), but not all applications are expected to provide such a feature, I would think. Pro-Cite also initializes the BOOLEAN "suspended" to FALSE, of course. The remainder of the Setup code sample initializes support for the Notification Manager, which I will describe at the end of this section.

The main saving grace of Pro-Cite in providing MultiFinder support is the centralization of all event-getting in a routine called "MyGetNextEvent". Actually, many Macintosh applications probably centralize event retrieval and it certainly makes the job easier. In "MyGetNextEvent", the setting of "WNEisImplemented" is tested: if TRUE, we're running under MultiFinder so WaitNextEvent can safely (and should) be called; if FALSE, then we're not running under MultiFinder so SystemTask and possibly GetNextEvent is called. As mentioned earlier, SystemTask may not be needed, but it was left in place in case a future version of Pro-Cite which does NOT require System 4.1 is implemented. More importantly, a BOOLEAN value "callGetNextEvent" is passed in so that background processing may be supported as described below. The result of either WaitNextEvent or GetNextEvent is placed in the BOOLEAN "haveEvent", some other processing is done, and "MyGetNextEvent" returns.

Refer to the abbreviated sample of Pro-Cite's MainEventLoop to see how MyGetNextEvent is used. The MainEventLoop is much as one would expect: MyGetNextEvent is called to get an event near the top and the code proceeds through a CASE statement to dispatch activity depending on the type of the event. Notice a few interesting features of the MainEventLoop with regard to MultiFinder. If Pro-Cite is suspended (the variable "suspended" is TRUE), then neither the HiLiteMenu or the CursorAdjust routine is called, since these might affect the display of the foreground application. However, CheckScrap (which makes sure that the desk scrap and internal scraps are the same as far as Pro-Cite is concerned) is called each time through the maineventloop. This really doesn't take too long and thus scrap coercion becomes unnecessary on suspend/resume events. Notice also that all suspend/resume events (which are implemented as application-defined event type 4, or "app4Evt") are dispatched to a routine called "DoSuspendResume", even in the case where IsDialogEvent returns true: if you have modeless dialogs, be sure to handle suspend/resume events yourself since the Dialog Manager probably won't do much with them. "DoSuspendResume" is the heart of Pro-Cite's MultiFinder support.

"DoSuspendResume" is called, as one might expect, whenever Pro-Cite receives a suspend event or a resume event. It first checks which window is frontmost via FrontWindow, placing the windowkind of the frontmost window into a global "docType" that is used literally everywhere. It then sets suspended based on whether or not the message field of the event is odd: if it's odd, it's a resume event so we're no longer suspended; if not, it's a suspend event so we're about to be suspended. (The temporary setting of suspended to FALSE is merely to allow SetHourGlass to turn on Pro-Cite's hourglass cursor.) If it's a suspend event, then we fake up a deactivate event for the frontmost window, set our YieldTime to the background value, call our activate routine ("MyActivate"), and if one of "our" windows, we also cause an update event just to make sure things are clean (and immediately process it by calling "DrawWindow".) If it's a resume event we do much the same thing, except we fake up an activate event, we set our YieldTime to its foreground value, and we also do a DrawMenuBar (mostly for Switcher). Notice that when handling the activate/deactivate events, we need to pass them to IsDialogEvent (and maybe DialogSelect) if the window is a

modeless dialog or to SystemEvent if the frontwindow is a desk accessory which the user has opened in our application's heap instead of the DA layer.

Background processing in Pro-Cite is implemented by the "GetAnEvent" procedure which is built on top off "MyGetNextEvent". It is merely a small version of the MainEventLoop that only supports suspend/resume events as well as activate and update events. From a process which Pro-Cite wants to support in the background (such as sorting or formatting records), "GetAnEvent" is merely called regularly with the "callGetNextEvent" FALSE (since if we're not under MultiFinder we just want to call SystemTask) and with the event mask for only app4Evt + keydownEvt, the latter to allow Command-period to interrupt the process. The process will continue during null events, plus any windows obscurred and then exposed by the foreground task will continue to be updated, and finally a resume event will be processed when the user "clicks back" into Pro-Cite. "MiniMainEvent" is a similar sort of thing, called by Pro-Cite to bring windows to the front when necessary and to combat "display anomalies" in certain places. It is also called by setup right after FlushEvents to make sure that initial events from MultiFinder at startup are processed immediately: if this isn't done, one could find any dialog or "splash screen" initially produced by the application coming up BEHIND the layer (usually Finder) from which the application launched!

Finally, a few words about the Notification Manager which is available in the 6.0 release of the Macintosh System, released in the spring of 1988. The Notification Manager is NOT really an implementation of inter-process or inter-application communication: that will probably come in some form in System 7.0. It is, however, a means by which an application running in the background can let the user interacting with the current foreground application know that the background application requires attention by a sound, a flashing small icon alternating with the Apple menu icon, a diamond mark on the application's name in the Apple menu, or an alert, or any combination of these. Pro-Cite provides for either sound or the flashing small icon, and it also provides a means for the user to enable or disable these (an idea to keep in mind, if you're going to provide Notification Manager support.) Notification Manager support is actually trivial to provide, if you've accomplished MultiFinder support.

Referring as needed to the attached listing, notice that a record type "NMRec" has been declared for Notification Manager support, and the accompanying variable is "nmforProCite". It is just a standard Macintosh Operating System queue element. This record is initialized in the Setup procedure. The qType is just set to ORD(nmType) which is 8 (in fact, Pro-Cite uses this variable to indicate if a Pro-Cite Notification Manager task is currently active.) The nmMark value is set to 1 to provide a diamond mark in the Apple menu: Pro-Cite always does this and it isn't user-configurable. The nmStr pointer is set to NIL: it provides for a string to be displayed in an alert and Pro-Cite doesn't support this feature. The notification response procedure pointer, nmResp, is set to @NMResponse. This procedure does absolutely nothing in Pro-Cite, but must be present or the flashing icon and menu bar mark will go away immediately after the notification sound occurs! Developers probably will find more for it to do in future applications.

Pro-Cite sets up a Notification Manager task by calling its routine "MyNotify" whenever it has completed a process in the background, such as formatting or sorting records, or when it has encountered an error condition while processing in the background and wants to bring up an error message alert. Pro-Cite could just use the Notification Manager to bring up the alert immediately in the foreground layer, but I think the way I've done it is much less obtrusive. Pro-Cite merely sets up the task with "MyNotify" (checking the variable "notifylevel" to see if the user wanted sound and/or icon, loading the small icon if necessary and setting the nmSIcon handle to it), calls NMInstall to install the queue element, and then hangs around in the MainEventLoop or with MiniMainEvent, whichever is convenient, until the user switches back in. When the user does come back in, Pro-Cite kills the Notification Manager task when processing the resume event by just calling NMRemove in "DoSuspendResume." And that's all there is to it!

A Brief Good-Bye, and Good Luck!

In this paper I've tried to present what I feel are important points to remember when trying to provide MultiFinder support within an application, as disclosed to me by Apple during testing of various versions of MultiFinder and as discovered by me when trying to accomplish the darn task in the first place. This is by no means an exhaustive list, nor may the techniques described herein be entirely accurate or sufficient for the needs of any given application (there are so many Macintosh applications!) I would refer you to both present and future versions of *Inside Macintosh*, the *MultiFinder Development Package*, plus the ongoing series of Technical Notes, for more information.

I hope that I have given you an idea of what it's like to provide MultiFinder support in an application and that I have convinced you to go ahead and DO IT. I think it's a good idea that all applications strive to support MultiFinder as much as possible so as to provide as rich an environment as we can for the Macintosh user. At least until the next Macintosh System release (next year.) Good luck.