Macintosh Technical Notes

#180: MultiFinder Miscellanea

See also: Technical Note #126—Sublaunching
Technical Note #158—
Frequently Asked MultiFinder Questions

Written by: Jim Friedlander November 2, 1987
Updated: March 1, 1988

This Technical Note discusses MultiFinder issues of which programmers should be aware.

Desk Accessories and MultiFinder

There is a persistent rumor that DAs no longer function under MultiFinder. While we are certainly encouraging people who write DAs to write small applications instead (it's a lot easier to write a small application than a DA), MultiFinder 1.0 does indeed support the standard desk accessory model. The world of the DA has changed somewhat, though, under MultiFinder.

The major change is that DAs are now loaded into the system heap, instead of the application heap. This fact means that **certain** DAs will **not** work under MultiFinder 1.0, though we've gone to great lengths to make sure that most DAs will work correctly. DAs that are "self-sufficient" will work just fine.

Self-sufficient DAs

"Self-sufficient DAs" are DAs that don't rely on a specific application being present in order to function, that is, they don't rely on being in a specific application's heap in order to do what they do. A self-sufficient DA also doesn't care about global context at `accEvent` or `accRun` time (not guaranteed under MultiFinder) since it is only drawing to its own window and dealing with its own storage. Under MultiFinder, a DA has no way of knowing who called it.

Non self-sufficient DAs

So, you're probably wondering, what in the heck is a DA that **isn't** self-sufficient. A good example of a DA that won't work very well under MultiFinder is a spell checker. Spell check DAs generally rely on posting events to get word processors to save to the scrap so the DA can then get the text from the scrap and scan it for speling erors. Unfortunately, at `accEvent` or `accRun` time, you don't know which MultiFinder partition called you, so you can't post an event (you don't know where it will go) so you can't get the text, so you get hopelessly confused and give up.

Error checking

DAs still need to do robust error checking to see if they have enough memory to load, even though MultiFinder loads DAs into the system heap, and will automatically grow the system heap if it can. A DA can't know if MultiFinder has loaded it, or (even if MultiFinder has loaded it) if there is room to grow the system heap. A DA can check to see if there is room for it to load by actually trying to allocate all the memory it needs and then backing out gracefully if it can't get it.

Switching

For conceptual clarity, it is best to think of MultiFinder 1.0 as using three types of switching: major, minor and update. All switching occurs at a very well defined time, namely when either `WaitNextEvent`, `GetNextEvent` or `EventAvail` is called.

Major switching is a complete context switch, that is, an application's windows are moved from the background to the foreground or rice-a-roni. `A5` worlds are switched and the application's low-memory world is switched. If the application accepts Suspend/Resume events, it is so notified at major switch time.

Major switching will not occur when a modal dialog is the frontmost window of the front layer, though minor and update switching will. To determine this, MultiFinder looks to see (among other things) if the window definition procedure of that window is `dBoxProc`. If it is, then it won't allow a switch via the user clicking on another application. `dBoxProcs` are specifically reserved for modal dialogs—when most users see a `dBoxProc`, they are expecting a modal situation. If you are using a `dBoxProc` for a non-modal window, we'd strongly recommend that you change it to some other window type, or risk the wrath of the User-Interface Thought Police (UITP).

Minor switching occurs when an application needs to be switched out in order to give time to background processes. In a minor switch, `A5` worlds are switched, as are low-memory worlds, but the application's layer of windows is **not** switched, and the application won't be notified of the switch via Suspend/Resume events.

Update switching occurs when MultiFinder detects that one or more of the windows of an application that is not frontmost needs updating. This happens whether or not the application has the `canBackground` bit in the SIZE -1 resource. This switch is very similar to minor switching, except that update events are sent to the application whose window(s) need updating.

Both minor and update switches should be transparent to the frontmost application.

Suspend/Resume events

If your application does not accept Suspend/Resume events (as set in the SIZE –1 resource), then if a mouseclick occurs in a window that isn't yours, MultiFinder will send your application a mouse-down event with code `inMenuBar` (with `menuID` equal to the ID of the Apple menu and `menuItem` set to "About MultiFinder ..."). The reason that MultiFinder does this is to force your application to think that a DA is coming up, so that it will convert any private scrap that it might be keeping. MultiFinder is expecting your application to call `MenuSelect`—if you don't, it will currently issue a few more `mouseDowns` in the menu bar and then finally give up. This isn't really a problem, but a lot of developers have run into it, especially in quick and dirty applications.

If you are switching menu bars with `SetMenuBar` (and switching the Apple Menu) during the execution of your application, then you should definitely make sure that your application accepts Suspend/Resume events. MultiFinder records the ID of the original Apple menu that you use and won't keep track of any changes that you make to the Apple menu. So, in the above situation, MultiFinder will give you a `mouseDown` in the `menuBar` with the `menuItem` set to the item number of "About MultiFinder..." that was in the original Apple menu, which could be quite a confusing situation. If you set the `MultiFinder friendly` bit in the SIZE resource, MultiFinder will never give you these mouse down events.

Referencing global data (A5 and MultiFinder)

MultiFinder maintains a separate `A5` world for each application. MultiFinder will switch `A5` worlds as appropriate so most applications don't have to worry about `A5` at all (except to make sure that it points to a valid QuickDraw global record at GNE/WNE time). MultiFinder also switches low-memory globals for you, so, if you need to get at `CurrentA5`, you should be OK.

If an application uses routines that execute at interrupt time, then it does need to be concerned about `A5`. There are four basic types of interrupt routines that are affected by MultiFinder:

- VBL tasks
- Completion routines
- Time manager tasks
- Interrupt service routines

from VBL tasks

If an application installs a VBL task into its application heap, MultiFinder will currently "unhook" that VBL routine when it switches that application out (using either a major or a minor switch). It will "rehook" it when the application is switched back in. A VBL task that is installed in the system heap will always receive time, that is, it will never be "unhooked." Given this, it is technically not necessary for a VBL task that is in the application's heap to worry about its `A5` context, since it will only be running when that application's partition is switched in. However, we would still like to encourage you to set up `A5` by carrying the value for `CurrentA5` around with you, since we may change the way this works in future versions of MultiFinder (and even without MultiFinder the VBL could trigger at a time when `A5` is not correct).

The following short MPW examples show how to do this using `INLINEs`. Please note that this technique does **not** involve writing into your code segment (we'll get to that later), we just put our value of `CurrentA5` in a position where we can find it from our `VBLtask`. These examples relys on the fact that we know that `A0` points to our `VBLTask`. Since we store our `CurrentA5` into the 4 bytes before our `VBLTask`, we know that we can get our `CurrentA5` from `-4(A0)`.

This example also serves to demonstrate how one might write a completion routine for an asynchronous Device Manager call. It is not intended to be a complete program, nor to demonstrate optimal techniques for displaying information. In MPW Pascal:

```
PROGRAM InlineVBL;

        USES
           {$PUSH}                        {save current compiler options}
           {$LOAD PasDump.dump}           {load symbol table dump}
           Memtypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint,WLW,JimLib;
           {$LOAD}                        {turn off LOAD}
           {$POP}                       {restore compiler options}
           {$D+}                        {debug symbols}

        CONST
           Interval         = 6;      {how often we want our VBL called, in ticks}
           CurrentA5        = $904;      {low-memory global}

        TYPE
           MyVBLType        = RECORD
                               CurA5: Longint; {put CurA5 where we can find it}
                               MyVBL: VBLTask; {the actual VBLTask}
                              END;    {MyVBLType}

        VAR
           Err              : Integer;
           MyVBLRec         : MyVBLType;
           Counter          : Integer;
           MyEvent          : EventRecord;

        PROCEDURE _DataInit;
           EXTERNAL;

        PROCEDURE PushA5;
           INLINE $2F0D;              {MOVE.L A5,-(SP)} {Push A5 onto the Stack}

        PROCEDURE PopA5;
           INLINE $2A5F;              {MOVE.L (SP)+,A5} {Pop the stack into A5}

        PROCEDURE GetMyA5;
           INLINE $2A68,$FFFC;           {MOVE.L -4(A0),A5} {Get the value of A5 we've
                                 stored before the parameter block and put
                                    it in A5. Since we know that when
                                    VBL task is called, A0 will
                                    point to our parameter block, we
                                    also know that the value of CurrentA5 that
                                    we stored will be at -4(A0)}

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

        PROCEDURE DoVBL;              {our whizzy VBL task}

           BEGIN                      {DoVBL}

           {First, we'll make sure that we have our A5, that we stored before our
        parameter block}
              PushA5;                 {Push the value of A5 onto the stack}
              GetMyA5;                {Get our A5 from right before the parameter block}
```

```
            {now we can access our globals: }
            MyVBLRec.MyVBL.vblCount := Interval; {we wish to run again}
            Counter := Counter + 1;{to show we can set a global}

            {since we're leaving, put back the A5 that was there before we changed it}
            PopA5;                  {put back original A5}
         END;                       {DoVBL}

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

      BEGIN                         {main PROGRAM}
         MaxApplZone;               {grow the heap to ApplLimit}
         UnloadSeg(@_DataInit);     {unload data init code before any allocations}
         InitMac;                   {initialize Macintosh managers}
         InitWW(NIL);               {initialize WritelnWindow with default window}

         Counter := 0;              {initialize this}
         WITH MyVBLRec,MyVBL DO BEGIN
            CurA5 := LongPtr(CurrentA5)^;{Get the current value of CurrenA5}
            vblAddr := @DoVBL;      {point to our task}
            vblCount := Interval;     {set up the interval at which we'll be called}
            qType := ORD(vType);      {this to is necessary}
            vblPhase := 0;
         END;                       {With}

         Err := VInstall(@MyVBLRec.MyVBL); {Install our VBLTask}
         writeln('VInstall err = ',Err);

         REPEAT
            writeln(Counter);         {write out counter}
         UNTIL GetNextEvent(mDownMask,MyEvent); {this allows a switch}

         Err := VRemove(@MyVBLRec.MyVBL);{we're done, remove the task}
         writeln('VRemove err = ',Err);

         beep;                      {show 'em we're done}
      END.
```

In MPW C (with assembly):
First, the assembly routines:

```
      CASE ON      ; for C

;-------------------------------------------------------------------------
PushA5  PROC   EXPORT ;pushes A5 onto the stack -- BE CAREFUL NOT TO DISTURB A0 here,
                  ; since GetMyA5 relies on it        MOVE.L (SP)+,A1     ; get return
address off the stack
         MOVE.L A5,-(SP)  ; push A5
         JMP    (A1)      ; return to caller
         ENDP

;-----------------------------------------------------------------------------------
PopA5   PROC   EXPORT
         MOVE.L (SP)+,A1  ; get return address off the stack
         MOVE.L (SP)+,A5  ; pop into A5
         JMP    (A1)      ; return to caller
         ENDP
```

```
;----------------------------------------------------------------------------------------
GetMyA5 PROC   EXPORT
          MOVE.L (SP)+,A1  ; get return address off the stack
          MOVE.L -4(A0),A5 ; get our saved value of A5 and put it in A5
          JMP    (A1)      ; return to caller
          ENDP
          END
```

Now the MPW C programette:

```
#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 <retrace.h>
#include <packages.h>

extern void PushA5();       /*  MOVE.L A5,-(SP) */ /*Push A5 onto the Stack*/
extern void PopA5();    /*  MOVE.L (SP)+,A5 */ /*Pop the stack into A5*/
extern void GetMyA5();  /* MOVE.L -4(A0),A5 */
                        /*Get the value of A5 we've
       stored before the parameter block and put
                        it in A5. Since we know that when
                        VBL task is called, A0 will
                        point to our parameter block, we
                        also know that the value of CurrentA5 that
                        we stored will be at -4(A0)
                        */

void DoVBL();

typedef struct MyVBLType {
       long      CurA5; /* put CurA5 where we can find it */
       VBLTask      MyVBL; /* the actual VBLTask */
} MyVBLType;

MyVBLType  MyVBLRec; /* a variable of the above type */
short      Counter;  /* this needs to be global so the VBL task can get to it */

main()
{
#define    Interval  6      /*how often we want our VBL called, in ticks*/
#define    CurrentA5 0x904  /*low-memory global*/

       WindowPtr        MyWindow;
       Rect          myWRect,rectToErase;
       OSErr         err;
       EventRecord   MyEvent;
       char          myStr[40];       /* this should be enough room */

       InitGraf(&qd.thePort);
       InitFonts();
       FlushEvents(everyEvent, 0);
       InitWindows();
       InitMenus();
       TEInit();
```

```
        SetRect(&myWRect,50,260,150,340);
        MyWindow = NewWindow(nil,&myWRect,"\pVBL",true,0,(WindowPtr)-1,false,0);
        SetPort(MyWindow);

        Counter = 0;  /*initialize this*/

        MyVBLRec.CurA5 = *(long *)(CurrentA5);/* Get the current value of CurrentA5 */
        MyVBLRec.MyVBL.vblAddr = DoVBL;  /* point to our task */
        MyVBLRec.MyVBL.vblCount = Interval;
           /* set up the interval at which we'll be called */
        MyVBLRec.MyVBL.qType = vType; /* this too is necessary */
        MyVBLRec.MyVBL.vblPhase = 0;

        err = VInstall(&MyVBLRec.MyVBL);/* Install our VBLTask */

        PenMode(patXor);              /* so we can see the drawing flicker */
        SetRect(&rectToErase,60,20,100,50);
        MoveTo(10,76);
        DrawString("\pClick to quit");

        while (!GetNextEvent(mDownMask,&MyEvent))     /* this allows a switch */
        {
           MoveTo(20,20);             /* draw a box */
           LineTo(20,50);LineTo(50,50);LineTo(50,20);LineTo(20,20);LineTo(50,50);
           MoveTo(20,50);LineTo(50,20);MoveTo(60,43);

           EraseRect(&rectToErase);     /* erase the last number */
           NumToString(Counter,myStr);
           DrawString(myStr);         /* draw the current value of Counter */
        }

        err = VRemove(&MyVBLRec.MyVBL);  /* we're done, remove the task */
        if (err != noErr) debugger();

        /* wait around until the user clicks before exiting */
        while (!Button());
        while (Button());
} /*main*/

void DoVBL()  /* our whizzy VBL task */
{       /* DoVBL */
        /* First, we'll make sure that we have our A5, that we stored before our
        parameter block */
        PushA5();        /* Push the value of A5 onto the stack */
        GetMyA5();       /* Get our A5 from right before the parameter block */

        /* now we can access our globals: */
        MyVBLRec.MyVBL.vblCount = Interval;/* we wish to run again */
        Counter += 1;/* to show we can set a global */

        /* since we're leaving, put back the A5 that was there before we changed it */
        PopA5();  /* put back original A5 */
}       /*DoVBL*/
```

from Completion routines

Currently, MultiFinder will not do a major, minor, or update switch if an asynchronous **File Manager** call is pending, so an application doesn't really need to worry about whether or not its `A5` or low-memory globals are correct, but we still recommend that you use the above technique to save `A5` for asynchronous File Manager calls. MultiFinder will, however, switch if an asynchronous **Device Manager** call is pending. When the call completes, the completion routine has no way of knowing whose partition is active, that is, it doesn't know if `A5` is valid (it needs `A5` if it wants to set a global) or even if the value of `CurrentA5` is valid. Sounds pretty hopeless, huh?

Well, actually this one is quite easy, you just need to put the value of `CurrentA5` that "belongs" to your partition in a place where you can find it from your completion routine. Since it is guaranteed that `A0` will be pointing to your parameter block when your completion routine is called, you can put the value of `CurrentA5` at a known offset from the beginning of your parameter block and then reference it off of `A0`. Completion routines are normally written in assembly language, though you can also write them in a high-level language. A simple example of how to do this from MPW Pascal can be found in the previous section about VBL tasks (it was easier to provide a clear, concise example for VBLs than for asynchronous Device Manager completion routines).

from Time Manager Tasks

You might think that you could use the same technique for Time Manager tasks as for VBL tasks, but, unfortunately you can't. Unlike VBL tasks (and completion routines), a Time Manager task is **not** called with `A0` pointing to the task block (`A0` points to the task's routine instead). So, if you need to get at your application's globals from your Time Manager task, you'll have to actually write the value of `CurrentA5` into your code segment at a time when you know that `CurrentA5` is valid and then use that value to set up `A5` when your Time Manager task is called. (I know, I know: Technical Note #2 says not to do this, but there's no alternative in this case).

from Interrupt service routines

If your application needs to get to its application globals and it replaces the standard 68xxx interrupt vectors (levels 1-7) with pointers to its own routines, it must also write `CurrentA5` into its code (since there is no parameter block for interrupt service routines).

Note: WDEFs should also carry around a copy of `A5` in the same fashion as Time Manager tasks and set up `A5` when called; WDEFs should also be non-purgeable.

Launching and MultiFinder

Technical Note #126 discusses the sublaunching feature of Systems 4.1 and newer. If you are running MultiFinder and you use the technique demonstrated in that technical note, your application will be able to launch the desired application and remain open. **Note:** MultiFinder does not support `Chain`; your application should never call this trap.

The application that you launch will become the foreground application. Unlike non-MultiFinder systems, when the user quits the application that you have sublaunched, control will not necessarily return to your application, but rather to the next frontmost layer.

Unlike non-MultiFinder systems, if you set both high bits of LaunchFlags, your application will continue to execute after calling Launch, so be prepared! Calling Launch with both high bits of LaunchFlags set can be thought of as a **request** to sublaunch. The actual launch (and hence **suspend** of your application) won't happen in the Launch trap, but at a later time (after calls to GNE/WNE/EventAvail).

Launch under MultiFinder currently will return an error if there isn't enough memory to launch the desired application, if the desired application can't be located or if the desired application is already open. In the latter case, that application will **not** be made active—if you sublaunched, control will return to your application, if you didn't sublaunch, your application will be terminated and the next frontmost layer will become active. If you didn't sublaunch and an error occurred, MultiFinder will do a SysBeep, since your application will be terminated. If you sublaunched, MultiFinder will not beep and it is up to your application to report the error to the user.

Launch now returns an error in register D0 if you are sublaunching. You can check for D0<0 after the sublaunch to see if the Launch failed. If D0>=0 then the application will be launched.

Note: The warnings in Technical Note #126 about using Sublaunching still apply, but, if you still wish to use Sublaunching, we strongly recommend that you set both high bits of LaunchFlags.

The Scrap and MultiFinder

MultiFinder 1.0 keeps separate scrap variables for each partition. MultiFinder only checks to see whether or not to increment the other partitions' scrapCounts in response to a user-initiated Cut or Copy. To do this, it watches the SystemEdit call to determine whether an official Cut or Copy has been issued.

When an application calls PutScrap or ZeroScrap in response to a Cut or Copy menu selection, the other partitions' scrapCounts will be incremented (the other partitions will know that something new has been put in the scrap).

System Resources and MultiFinder

MultiFinder is a shared environment. Resources that were formerly loaded into the application heap can now be loaded into the system heap for use by all applications. Basically, if a resource came from the System file, then it will be loaded into the system heap, even if the resSysHeap bit isn't set.

Since other applications may need to use system resources, applications should not call ReleaseResource or DetachResource on system resources, such as cursors and fonts, nor should they change resource attributes or modify the resource data directly. Applications should also not make assumptions about where a resource has been loaded (system heap or application heap).

UnmountVol and MultiFinder

`UnmountVol` was changed in System 4.2 so that it would work better in a shared environment. In systems 4.1 and prior, `UnmountVol` would successfully unmount a volume even if files were open on that volume. Under MultiFinder, that would be disastrous, since one application could unmount a volume that another application was using (this exact problem could occur under UniFinder if a DA unmounted a volume “out from under” an application).

System 4.2 changes the behavior of `UnmountVol` (whether or not MultiFinder is running) so that it will return a `-47` (`FBsyErr`) error if any files are open on the volume you wish to unmount. Since the Finder always has a DeskTop file open for each volume, it is asked to close the DeskTop file by `UnmountVol`, so you won’t get an error back if the only file open on a volume is the DeskTop file.

Putting up a splash screen

Some applications like to put up a “splash screen” to give the user something to look at while the application is loading. If your application does this **and** has the `canBackground` bit set in the size resource, then it must call `GetNextEvent` several times (or `WaitNextEvent` or `EventAvail`) before putting up the splash screen, or the splash screen will come up behind the frontmost layer. If the `canBackground` bit is set, MultiFinder will not move your layer to the front until you call `GNE/WNE/EventAvail`.

The Apple Menu and MultiFinder

Applications should avoid doing anything untoward with the Apple menu. For example, if your application puts an icon next to the “About MyApplication...” item, MultiFinder may unceremoniously write over it.

Interprocess Communication

MultiFinder 1.0 doesn’t have full-fledged interprocess communication facilities (that is planned for MultiFinder 2.0). There is no standard way to communicate between applications in MultiFinder 1.0. One of MultiFinder 1.0’s design goals was to not change the programming model (that is, so that as few applications as possible would have to be altered to run under MultiFinder); this made it impossible to add meaningful IPC.

There are, though, a couple of ways to communicate between applications. You can communicate through a file or through the clipboard (not guaranteed to be successful, since an intervening application might do a `ZeroScrap`) or you could use a bulletin-board type driver, where one application sends a message to the driver, which records the message and another application can query the driver to see if any messages have arrived. You can also do IPC with AppleTalk. If you’re running AppleTalk drivers version 48 or later you have the capability to send packets to your own node. This feature is enabled by making a `SetSelfSend` call. With AppleTalk self sending enabled, two applications on the same machine can send packets to each other. If you want to ensure that the socket you’re sending data to is on your local machine, (as opposed to another machine on the network), you can check the node portion of the socket’s address you look up with NBP. **Note:** it is most definitely to your advantage to wait until we implement real IPC.

Running as a Background Application

For applications that run in the background (i.e. applications that have the `canBackground` bit set in the SIZE resource), the programming model changes somewhat. Since background applications aren't guaranteed any time, an application that is running in the background should **not** count on being able to communicate with the user at any time. For example, a background application should not put up a modal dialog to inform the user that something that requires immediate attention has occurred (in fact, the modal dialog will **not** come up in the foreground, but rather behind other layers and the user might not be able to see the dialog at all). If an application does require urgent attention of the user, then it probably should not run in the background, or, at the least, should not perform any operation that may require urgent attention while it is in the background until notification services are provided in system software.

Background applications also should not do anything that might affect the foreground application such as changing the cursor or altering the menu bar.

Background applications do not receive user events of any kind. This includes application-defined events. If an application posts an application-defined event from the background, it will be sent to the foreground application (which will probably confuse that application). Rather than post an application-defined event, an application that is running in the background could set a global that is checked at Main Event Loop time to indicate a change in program status.

Miscellaneous Miscellanea

The sound glue that shipped with MPW 1.0 and 2.0 is **not** MultiFinder compatible and should not be used. Instead, applications should make direct calls to the sound driver.

All code needs to be aware of the shared environment; this includes ScreenSavers. ScreenSavers should make sure that background processing continues. A simple scenario for a ScreenSaver that's an INIT might be: patch `PostEvent` at INIT time, put up a full-screen black window spider, call `WaitNextEvent`, and watch `PostEvent` to see if an event that should cause the ScreenSaver to go away has occurred.