



### #180: MultiFinder Miscellanea

Revised by: Dave Radcliffe & Keith Rollin  
Written by: Jim Friedlander

August 1989  
November 1987

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

**Changes since June 1988:** Updated and generalized sample code to reflect new MPW 3.0 calls in both C and Pascal for saving and restoring A5 for interrupt code that accesses application globals. Removed text that can be found in *Programmer's Guide to MultiFinder*, and added a note about `_PostEvent`.

### Switching

For conceptual clarity, it is best to think of MultiFinder 6.0 and earlier as using three types of switching: major, minor, and update. All switching occurs at a well defined times, namely, when a call is made to either `_WaitNextEvent`, `_GetNextEvent`, or `_EventAvail`.

**Major switching** is a complete context switch, that is, an application's windows are moved from the background to the foreground or vice versa. A5 worlds are switched, and the application's low-memory world is switched. If the application accepts Suspend and 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 occur. To determine whether major switching will occur, MultiFinder checks (among other things) if the window definition procedure of that window is `dBoxProc`. If it is, then MultiFinder won't allow a switch via the user clicking on another application. A window definition procedure of `dBoxProc` is 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 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 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 and 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 set. This switch is very similar to minor switching, except that update events are sent to the application whose window need updating.

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

## Suspend and Resume Events

If your application does not accept Suspend and Resume events (as set in the 'SIZE' -1 resource), then if a mouse-click event occurs in a window that isn't yours, MultiFinder will send your application a mouse-down event with code `inMenuBar` (with `menuItem` 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 desk accessory is opening, 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 mouse-down events in the menu bar before finally giving 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 and 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 mouse-down event in the menu bar 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 switches 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 `_GetNextEvent` or `_WaitNextEvent` time). MultiFinder also switches low-memory globals for you. To get the value of the application's A5, use the routines from Technical Note #208, Setting and Restoring A5.

If an application uses routines that execute at interrupt time and accesses globals, then it needs to be concerned about A5. MultiFinder affects four basic types of interrupt routines:

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

### 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 condition, 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 its value around with the VBL, 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 the new MPW 3.0 calls mentioned in Technical Note #208. 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 the application's A5 in a location where

we can find it from our VBL task. Nor does it depend on the VBL task information being allocated globally. This gives you more flexibility setting up your VBL.

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.

## MPW Pascal 3.0

```
UNIT VBLS;

{$R-}

INTERFACE

USES
    Dialogs, Events, OSEvents, Retrace, Packages, Types, Traps;

CONST
    Interval = 6;
    rInfoDialog = 140;
    rStatTextItem = 1;

TYPE
    { Define a record to keep track of what we need. Put theVBLTask into the
      record first because its address will be passed to our VBL task in A0. }
    VBLRec = RECORD
        theVBLTask:   VBLTask;           { the actual VBLTask }
        VBLA5:        LongInt;           { saved CurrentA5 where we can find it }
    END;
    VBLRecPtr = ^VBLRec;

VAR
    gCounter:        LongInt;            { Global counter incremented by VBL }

PROCEDURE InstallVBL;

IMPLEMENTATION

{ GetVBLRec returns the address of the VBLRec associated with our VBL task.
  This works because on entry into the VBL task, A0 points to the theVBLTask
  field in the VBLRec record, which is the first field in the record and that
  is the address we return. Note that this method works whether the VBLRec
  is allocated globally, in the heap (as long as the record is locked in
  memory) or if it is allocated on the stack as is the case in this example.
  In the latter case this is OK as long as the procedure which installed the
  task does not exit while the task is running. This trick allows us to get
  to the saved A5, but it could also be used to get to anything we wanted to
  store in the record. }
FUNCTION GetVBLRec: VBLRecPtr;
    INLINE $2E88;                        { MOVE.L A0,(A7) }

PROCEDURE DoVBL (VRP: VBLRecPtr);
{ DoVBL is called only by StartVBL }
BEGIN
    gCounter := gCounter + 1;            { Show we can set a global }
    VRP^.theVBLTask.vblCount := Interval; { Set ourselves to run again }
END;

PROCEDURE StartVBL;
{ This is the actual VBL task code. It uses GetVBLRec to get our VBL record
  and properly set up A5. Having done that, it calls DoVBL to increment a
  global counter and sets itself to run again. Because of the vagaries of
  MPW C 3.0 optimization, it calls a separate routine to actually access
  global variables. See Tech Note #208 for the reasons for this, as well
```

```

as for a description of SetA5. }

VAR
    curA5:          LongInt;
    recPtr:         VBLRecPtr;

BEGIN
    recPtr := GetVBLRec;                { First get our record }
    curA5:= SetA5(recPtr^.VBLA5);       { Get our application's A5 }

                                     { Now we can access globals }
    DoVBL (recPtr);                    { Call another routine to do actual work }

    curA5:= SetA5(curA5);              { restore original A5, ignoring
result }
END;

PROCEDURE InstallVBL;
{ InstallVBL creates a dialog just to demonstrate that the global variable
is being updated by the VBL Task. Before installing the VBL, we store
our A5 in the actual VBL Task record, using SetCurrentA5 described in
Tech Note #208. We'll run the VBL, showing the counter being incremented,
until the mouse button is clicked. Then we remove the VBL Task, close the
dialog, and remove the mouse down events to prevent the application from
being inadvertently switched by MultiFinder. }

VAR
    theVBLRec:      VBLRec;
    infoDPtr:       DialogPtr;
    infoDStorage:   DialogRecord;
    numStr:          Str255;
    theErr:          OSErr;
    theItemHandle:   Handle;
    theItemType:     INTEGER;
    theRect:         Rect;

BEGIN
    gCounter:= 0;                      { initialize the global variable }
    infoDPtr:= GetNewDialog(rInfoDialog, @infoDStorage, Pointer(-1));
    DrawDialog(infoDPtr);
    GetDItem(infoDPtr, rStatTextItem, theItemType, theItemHandle, theRect);

    theVBLRec.VBLA5:= SetCurrentA5;    { get our A5 }
    WITH theVBLRec.theVBLTask DO
        BEGIN
            vblAddr:= @StartVBL;        { pointer to VBL code }
            vblCount:= Interval;         { frequency of VBL in System ticks }
            qType:= ORD(vType);          { qElement is a VBL type }
            vblPhase:= 0;                { no phases }
        END;

    theErr:= VInstall(@theVBLRec.theVBLTask); { install this VBL task }
    IF theErr = noErr THEN                { we'll show the global value in }
        BEGIN                            { the dialog until a mouse click }
            REPEAT
                NumToString(gCounter, numStr);
                SetIText(theItemHandle, numStr);
            UNTIL Button;
            theErr:= VRemove(@theVBLRec.theVBLTask); { remove the VBL task }
        END;

    CloseDialog(infoDPtr);               { get rid of the info dialog }
    FlushEvents(mDownMask, 0);           { remove all mouse down events }

```

END;

END.

## MPW C 3.0

```
#include <Events.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Retrace.h>
#include <Traps.h>

#define INTERVAL          6
#define rInfoDialog      140
#define rStatTextItem    1

/*
 * These are globals which will be referenced from our VBL Task
 */
long gCounter;                /* Counter incremented each time our VBL gets called */

/*
 * Define a struct to keep track of what we need. Put theVBLTask into the
 * struct first because its address will be passed to our VBL task in A0
 */
struct VBLRec {
    VBLTask          theVBLTask; /* the VBL task itself */
    long             VBLA5;      /* saved CurrentA5 where we can find it */
};
typedef struct VBLRec VBLRec, *VBLRecPtr;

/*
 * GetVBLRec returns the address of the VBLRec associated with our VBL task.
 * This works because on entry into the VBL task, A0 points to the theVBLTask
 * field in the VBLRec record, which is the first field in the record and that
 * is the address we return. Note that this method works whether the VBLRec
 * is allocated globally, in the heap (as long as the record is locked in
 * memory) or if it is allocated on the stack as is the case in this example.
 * In the latter case this is OK as long as the procedure which installed the
 * task does not exit while the task is running. This trick allows us to get
 * to the saved A5, but it could also be used to get to anything we wanted to
 * store in the record.
 */
VBLRecPtr GetVBLRec ()
    = 0x2008;                /* MOVE.L    A0,D0 */

/*
 * DoVBL is called only by StartVBL ()
 */
void DoVBL (VRP)
VBLRecPtr  VRP;
{
    gCounter++;              /* Show we can set a global */
    VRP->theVBLTask.vblCount = INTERVAL; /* Set ourselves to run again */
}

/*
 * This is the actual VBL task code. It uses GetVBLRec to get our VBL record
 * and properly set up A5. Having done that, it calls DoVBL to increment a
 * global counter and sets itself to run again. Because of the vagaries of

```

```

* MPW C 3.0 optimization, it calls a separate routine to actually access
* global variables. See Tech Note #208 - "Setting and Restoring A5" for
* the reasons for this, as well as for a description of SetA5.
*/
void StartVBL ()
{
    long          curA5;
    VBLRecPtr     recPtr;

    recPtr = GetVBLRec ();                /* First get our record */

    curA5 = SetA5 (recPtr->VBLA5);        /* Get the saved A5 */
                                          /* Now we can access globals */
    DoVBL (recPtr);                      /* Call another routine to do actual work */

    (void) SetA5 (curA5);                /* Restore old A5 */
}

/*
* InstallVBL creates a dialog just to demonstrate that the global variable
* is being updated by the VBL Task. Before installing the VBL, we store
* our A5 in the actual VBL Task record, using SetCurrentA5 described in
* Tech Note #208. We'll run the VBL, showing the counter being incremented,
* until the mouse button is clicked. Then we remove the VBL Task, close the
* dialog, and remove the mouse down events to prevent the application from
* being inadvertently switched by MultiFinder.
*/
void InstallCVBL ()
{
    VBLRec         theVBLRec;
    DialogPtr       infoDPtr;
    DialogRecord     infoDStorage;
    Str255          numStr;
    OSErr           theErr;
    Handle           theItemHandle;
    short           theItemType;
    Rect            theRect;

    gCounter = 0;                        /* Initialize our global counter */
    infoDPtr = GetNewDialog (rInfoDialog, (Ptr) &infoDStorage, (WindowPtr) -1);
    DrawDialog (infoDPtr);
    GetDItem (infoDPtr, rStatTextItem, &theItemType, &theItemHandle,
              &theRect);

    /*
    * Store the current value of A5 in the MyA5 field. For more
    * information on SetCurrentA5, see Tech Note #208- "Setting and
    * Restoring A5".
    */
    theVBLRec.VBLA5 = SetCurrentA5 ();
    /* Set the address of our routine */
    theVBLRec.theVBLTask.vblAddr = (VBLProcPtr) StartVBL;
    theVBLRec.theVBLTask.vblCount = INTERVAL;    /* Frequency of task, in ticks */
    theVBLRec.theVBLTask.qType = vType;          /* qElement is a VBL task */
    theVBLRec.theVBLTask.vblPhase = 0;

    /* Now install the VBL task */
    theErr = VInstall ((QElemPtr)&theVBLRec.theVBLTask);

    if (!theErr) {
        do {
            NumToString (gCounter, numStr);

```

```

        SetIText (theItemHandle, numStr);
    } while (!Button ());
    theErr = VRemove ((QElemPtr)&theVBLRec.theVBLTask); /* Remove it when done */
}

/* Finish up */
CloseDialog (infoDPtr);
FlushEvents (mDownMask, 0);
/* Get rid of our dialog */
/* Flush all mouse down events
*/
}

```

## Completion Routines

Currently, MultiFinder will not do a major, minor, or update switch if an asynchronous File Manager call is pending. This may not be true in the future. We recommend that you use the above technique to save A5 for asynchronous File Manager calls. MultiFinder does allow a switch if an asynchronous Device Manager or Sound 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 access a global). Sounds pretty hopeless, huh?

Well, actually this one is quite easy, you just need to put the value of A5 that “belongs” to your partition in a place where you can find it from your completion routine. It is guaranteed that A0 will point to your parameter block when your completion routine is called, so you can use the same technique shown with VBL tasks to put the value of A5 at a known offset from the beginning of the parameter block, and then reference it from 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 in MPW Pascal and C can be found in the previous section about VBL tasks (it was easier to provide a clear, concise example for VBL tasks than for asynchronous Device Manager completion routines).

## Time Manager Tasks

The Time Manager was rewritten for System 6.0.3. The new version will put a pointer to the TMTTask record in A1. This is not true in System 6.0.2 or earlier. The technique shown in the example VBL for accessing an application's globals is possible using System 6.0.3 and the Time Manager. Prior to System 6.0.3, the task must also store the application's A5 into its code. This method is not a very good idea and runs the risk of incompatibility (self-modifying code).

## 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 store the application's A5 into its code (since there is no parameter block for interrupt service routines). This method is not a very good idea and runs the risk of compatibility (self-modifying code).

**Note:** WDEFs should also maintain a copy of A5 in the same fashion as Time Manager tasks (prior to System Software 6.0.3) 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.

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

## The Scrap and MultiFinder

MultiFinder 6.0 and earlier keeps separate scrap variables for each partition. MultiFinder only checks to see whether or not to increment the other partitions' `scrapCount` variables in response to a user-initiated Cut or Copy. To do this, it watches for a call to `_SysEdit` (`SystemEdit`) or a menu event to determine if an official Cut or Copy command has been issued.

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

## `_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 when MultiFinder is not active, if a DA unmounted a volume “out from under” an application).

System 4.2 changes the behavior of `_UnmountVol` (whether or not MultiFinder is active) so that it returns 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, a call to `_UnmountVol` asks it to close the Desktop file so you won't get an error if the only file open is the Desktop file. However, there is a bug with this new behavior. In System 6.0.3, and earlier, `_UnmountVol` does not close the Desktop file for MFS-formatted volumes. Only the Finder can unmount a MFS volume (when the user drags the disk icon to the trash).

## Displaying 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 should call `_EventAvail` several times (or `_WaitNextEvent` or `_GetNextEvent`) 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 `_GetNextEvent`, `_WaitNextEvent`, or `_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. It is important to consider the Apple Menu owned by the system. You can have the standard about item, but other than this, you should avoid using the Apple menu. Don't make any assumptions about the contents of this menu. Even reading from its data may be a compatibility risk since its structure may change.



## Interprocess Communication

MultiFinder 6.0, and earlier, does not have full-fledged interprocess communication facilities. There is no standard way to communicate between applications in MultiFinder 6.0. There are, however, a couple of ways to communicate between applications. For recommendations on attempting interprocess communication with MultiFinder 6.0, contact MacDTS at the address listed in Technical Note #0.

**Note:** It is in your best interest to wait until Apple implements Interapplication Communication (IAC) in System 7.0.

## \_PostEvent

Even though you can have many applications running at once, each with a fairly independent world, the Event Manager maintains only one event queue. Because of this single queue, and because there is no facility implemented to keep track of which events belong to which layer, all events in the queue are passed to the frontmost application. This situation can cause problems for applications that take advantage of application-defined events. If the application is in the background and posts one of these events, then it is the foreground application that receives it.

This does not apply to events which are not really stored in the event queue. The list of these events include, but is not limited to, activate and update events, which are generated by the Window Manager as needed, and are correctly routed to the right application.

## Miscellaneous Miscellanea

The sound driver glue that shipped with MPW 1.0 and 2.0 is **not** MultiFinder compatible and should not be used. This also includes much of the glue supplied with older development systems. Instead, applications should be using the Sound Manager.

All code needs to be aware of the shared environment; this includes screen savers. Screen savers should make sure that background processing continues. A simple scenario for a screen saver 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 screen saver to go away has occurred.

## Further Reference:

- 
- *Inside Macintosh*, Volume V, Compatibility Guidelines
  - *Programmer's Guide to MultiFinder* (APDA)
  - Technical Note #126, Sub(Launching) from a High-Level Language
  - Technical Note #129, `_SysEnviron`: System 6.0 and Beyond
  - Technical Note #158, Frequently Asked MultiFinder Questions
  - Technical Note #205, MultiFinder Revisited: The 6.0 System Release
  - Technical Note #208, Setting and Restoring A5