Technote 1147

Pending Update Perils

By C.K. Haun
Revised by Mark Cookson
Apple Worldwide Developer Technical Support

CONTENTS

This Technote discusses potential problems when pending update events for windows behind modal dialogs are not serviced.

Introduction

Modal dialog boxes have always caused some problems with windows behind dialog windows. Since `ModalDialog` has its own event loop which does not by default cooperate with your application event loop you have always had the potential for not knowing that updates have occured for the other windows in your application when you are in a `ModalDialog` loop.

If you've ever written a filter procedure for a modal dialog, you've probably seen this for yourself. Your filter will get a continual stream of update events. These events are not for the dialog, but are for the window behind the dialog, which has not been updated since the modal dialog came up. The event has not come through your normal event loop, and you have probably not serviced the update since you are only concerned about events for your dialog. This causes the update event to keep getting re-sent. The only way for the update to be stopped is for the update region of the affected window to be cleared by the `Begin/EndUpdate` calls in your drawing routine (see Handling Update Events in Inside Macintosh:Mactintosh Toolbox Essentials).

This situation is exacerbated by screen savers and Balloon Help. If a screen saver becomes active while a modal dialog is up, or if your user has Balloon help on and part of a window behind the dialog is obscured by a balloon, then an update event will be generated for the window.

The Update and `ModalDialog`

If there is an update event pending for your application, no other applications, drivers, control panels, or anything else, will get time.

Updates pending for other applications do not generally cause a problem (unless they too are suffering from pending updates). They will be handled normally by the application in the background. Updates *must* be serviced or other processes will not get time.

This is a potential Bad Thing**.** Many pieces of code need time to keep living, to maintain network connections, or just to look good.

A simple example is the Chooser. Open the Chooser, then launch an application that you know has a modal dialog. Position the Chooser so you can see it, and you'll notice that it refreshes its lists even while it's in the background.

Now make sure there is a document window open in the frontmost application. Turn on Balloon Help from the Help menu.

Open a modal dialog in the application (the About box in most applications will work). Now move the cursor over the window behind the modal dialog. A balloon will appear saying something like "This window is not active because a dialog box is up....", and a piece of the window will be blasted by the balloon. Now look at the Chooser. It has stopped running. The window that got zapped by the balloon now has an update pending for it, that update is going through the `ModalDialog` trap, and not through the program's event loop, so it is not being serviced. Time stops for all other applications.

Note:
This **only** happens if the update is for the same application as the dialog box. If you blast a window in another application (like the Finder) then that update will be processed normally.

Yuck, that's nasty!

You have two choices in your application to prevent this from happening. The first is to have no other open windows in your application when you open a modal dialog. Obviously, this isn't always a realistic solution.

The second, saner, solution is to provide yourself a mechanism to refresh all your windows from within your modal dialog.

A filter procedure (described in Writing an Event Filter Function for Alert and Modal Dialog Boxes in Inside Macintosh:Macintosh Toolbox Essentials) is the proper tool to use to fix this problem. You'll need to add a simple filter procedure to every dialog or alert you bring up in your application. And, in most cases, it can be the same filter for every dialog, so it's not a great deal of extra code.

However, you're going to have to do a little preparation to do this. Your filter proc needs to have a way to call the drawing procedure for any of your windows. There are many ways to do this, dictated by the specific needs of your application and your own programming style. You may want to create a window control object that contains a pointer to your drawing routine, you may want to include the same check and dispatch you have in your main event loop, or use another method which you are comfortable with.

The simplest, bare bones method, would be to include a flag for your drawing procedure in your window record `refCon`, and have your drawing routine vector based on the value in the `refCon`, as shown here:

```
                        /* Window drawing proc, defined somewhere else*/
Boolean MyDrawProc (WindowPtr windowToDraw) {
    Boolean returnVal = true;

                        /* switch off the value you've stored in*/
                        /*your window earlier*/
    switch (GetWRefCon(windowToDraw)) {
        case kMyClipboard: /*draw my clipboard*/
            DrawMyClip (windowToDraw);
            break;
        case kMyDocument:  /*document content*/
            DrawMyDoc (windowToDraw);
            break;
        default:           /*do nothing for anything else, to prevent drawing window*/
            returnVal = false;
                        /*that isn't mine*/
            break;
    }

                        /* this return value is used to tell the Dialog
                        /* Manager if you've handled the update or not when
                        /* this is called from your filter.  In normal uses
                        /* (i.e., in response to an updateEvent in your main
                        /* event loop) the boolean is unnecessary, but it
                        /*doesn't do any harm*/

    return (returnVal);
}
```

Install the flag when you create a window:

```
    myWindowPtr = GetNewWindow (kMyWindowID, nil, (WindowPtr)-1);
    SetWRefCon (myWindowPtr, (long)myDrawingProcFlag);
```

In your filter, the update handling would look something like this:

```
    /* if the update is for the dialog box, ignore it since the regular ModalDialog
       will redraw it as necessary*/
    if(theEventIn->what == updateEvt && theEventIn->message != myDialogPtr ) {
        /* go to my drawing routine, window will be redrawn if I own it*/
        return (MyDrawProc ((WindowPtr)theEventIn->message));
    }
```

In MPW Pascal:

```
{ The function's result is used to tell the Dialog Manager if you've handled the)
( update or not when this is called from your filter. In normal uses (i.e., in)
( response to an updateEvent in your main event loop) the boolean is unnecessary,)
( but it doesn't do any harm. The window drawing procedure is defined somewhere else.}

FUNCTION MyDrawProc(windowToDraw WindowPtr): BOOLEAN;

BEGIN
  CASE GetWRefCon(windowPtr) OF

    kMyClipboard:
      BEGIN
        DrawMyClipboard(windowToDraw);
        MyDrawProc := TRUE;
      END;

    kMyDocument:
      BEGIN
        DrawMyDocument(windowToDraw);
        MyDrawProc := TRUE;
      END;

    OTHERWISE
      MyDrawProc := FALSE;
   END; {CASE}
END;
```

Install the flag when you create a window:

```
    myWindowPtr := GetNewWindow(kMyWindowID, NIL, WindowPtr(-1));
    SetWRefCon(myWindowPtr, myDrawingProcFlag);
```

In your filter, the update handling would look something like this:

```
FUNCTION MyFilter(currentDialog: DialogPtr; VAR theEventIn: EventRecord;
     VAR theItem: INTEGER): BOOLEAN;

{ if the update is for the dialog box, ignore it since the regular ModalDialog
{ function will redraw it as necessary }

BEGIN

  IF (theEventIn.what = updateEvt AND theEventIn.message <> currentDialog)
    BEGIN
      MyFilter := MyDrawProc(currentDialog);
    END;
END;
```

If you do some, you have to do a little more...

The only downside to adding your own filter procedure to a dialog is that the Dialog Manager then assumes that you are handing more than just updates. Specifically, the Dialog Manager assumes that you are handling the standard "return key aliases to item 1" filtering. So, you need to write keystroke handling in the filter yourself.

The Dialog Manager in System 7 has some new calls you can make to ease the load on your program in this situation. These calls were created and tested too late in System 7's development cycle to be documented in Inside Macintosh, so they are presented in Technote 1148: Dialog Manager Helper Functions. They allow you to call on the services of the System to track standard keystrokes in your dialog.

The System 6 Way

Of course, under pre-System 7 applications, you can't use the new calls, so you have to do it yourself. Here's a sample System 6.0.x filter proc that does roughly the same thing that a System 7 filter will do.

```
/* Pre-system 7 dialog filter*/
#define kMyButtonDelay 8

/* declared as `pascal' since it's called by the toolbox*/
pascal Boolean MyFilter (DialogPtr currentDialog,
     EventRecord *theEventIn, short *theDialogItem) {
    Boolean     returnVal = false;
    long        waitTicks;
    short       itemKind;       /* some temporary variables for GetDItem use*/
    Handle      itemHandle;
    Rect        itemRect;

    if (theEventIn->what == updateEvt && theEventIn->message != myDialogPtr) {
        /* myDialogPtr is defined where you created the dialog
           if the update is for the dialog box, ignore it since
           the regular ModalDialog function will redraw it as necessary*/

        returnVal = MyDrawProc (theEventIn->message);    /* go to my drawing routine*/
    } else {
  /* it wasn't an update, see if it was a keystroke. Check for the return or
     enter key, and alias that as item 1. I also included a check here for the
     escape key aliasing as item 2, you may not want to use that*/

   if ((theEventIn->what == keyDown) || (theEventIn->what == autoKey)) {
       /* it was a key*/

  switch (theEventIn->message & charCodeMask) {
      case kReturnKey:
      case kEnterKey:
          *theDialogItem = ok;    /* change whatever the current item is to
                                     the OK item ok is #defined in Dialogs.h
                                     as now we need to invert the button so
                                     the user gets the right feedback*/
          GetDItem (currentDialog, ok, &itemKind, &itemHandle, &itemRect);
          HiliteControl ((ControlHandle)itemHandle, inButton); /* invert the button*/
          Delay (kMyButtonDelay, &waitTicks);   /* wait about 8 ticks so they can see it*/
          HiliteControl ((ControlHandle)itemHandle, false);  /* and back to normal*/

          returnVal = true;           /* tell the Dialog Manager we handled this event*/
          break;
```

```
      /* This filters the escape key the same as item 2 (the cancel button,usually )*/
      case kEscKey:
          *theDialogItem = cancel;     /* cancel is #defined in Dialogs.h as 2*/
          GetDItem (currentDialog, cancel, &itemKind, &itemHandle, &itemRect);
          HiliteControl ((ControlHandle)itemHandle, inButton);
          Delay (kMyButtonDelay, &waitTicks); /* wait about 8 ticks so they can see it*/
          HiliteControl ((ControlHandle)itemHandle, false);

          returnVal = true;  /* tell the Dialog Manager we handled this event*/
          break;
            }
        }
    }

    return (returnVal);
}
```

MPW Pascal

```
{ Your filter for pre-System 7 will look something like this: }

FUNCTION MyFilter(currentDialog: DialogPtr; VAR theEventIn:
     EventRecord; VAR theItem: INTEGER): BOOLEAN;
CONST
kMyButtonDelay = 8;
VAR
  itemKind      : INTEGER;
  itemHandle    : Handle;
  itemRect      : Rect;
  savePort      : GrafPtr;
  waitTicks     : LONGINT;

BEGIN
  { if the update is for the dialog box, ignore it since the regular ModalDialog
  { function will redraw it as necessary }
  IF (theEventIn.what = updateEvt AND theEventIn.message <> currentDialog)
    MyFilter := MyDrawProc(theEventIn.message)
  ELSE { it wasn't an update, see if it was a keystroke }
    BEGIN
 {Check for the return or enter key, and alias that as item "ok".  }
 {I also included a check here for the escape key aliasing as item "cancel", }
 {you may not want to use that }
   IF ((theEventIn.what = keyDown) OR (theEventIn.what = autoKey))
     BEGIN { it was a key }

       CASE CHR(BAnd(theEventIn.message, charCodeMask)) OF

         kReturnKey, kEnterKey:
           BEGIN
             GetDItem(currentDialog, ok, itemKind, itemHandle, itemRect);
             HiliteControl(ControlHandle(itemHandle), TRUE);
             Delay(kMyButtonDelay , waitTicks);  {wait about 8 ticks so they can see it}
             HiliteControl(ControlHandle(itemHandle), FALSE);  {and back to normal}
             MyFilter := TRUE;           {tell the Dialog Manager we handled this event}
           END;

         kEscKey:
           BEGIN
             theItem := cancel;
             GetDItem(currentDialog, cancel, itemKind, itemHandle, itemRect);
```

```
            HiliteControl(ControlHandle(itemHandle), TRUE);
            Delay(kMyButtonDelay , waitTicks);  {wait about 8 ticks so they can see it}
            HiliteControl(ControlHandle(itemHandle), FALSE);  {and back to normal}
            MyFilter := TRUE;           {tell the Dialog Manager we handled this event}
             END;

        END; {CASE}
    END;
  END;
END;
```

Conclusion

Never-ending updates are not a new problem. It is imperative that you do something about never-ending updates. There isn't much extra work involved; just add a simple filter to all of your dialogs and alerts, and put a flag to your drawing proc in your window structure.

The results will allow the system to continue to run smoothly, and as an added benefit your users will always see your application's windows the way they should be, instead of windows with chunks bitten out of them.

Further References

- Inside Macintosh:Macintosh Toolbox Essentials, Chapter 2 - Event Manager
- Inside Macintosh:Macintosh Toolbox Essentials, Chapter 4 - Window Manager
- Inside Macintosh:Macintosh Toolbox Essentials, Chapter 6 - Dialog Manager

Downloadables

 Acrobat version of this Note (35K).

Change History

- Originally written in October 1991, as Technote TB 37 -- Title by C.K. Haun.
- Accompanying code written and revised by C.K. Haun (1991) and Mark Cookson (1999).
- In January 1999, this Technote was updated to better organize the ideas presented.

Acknowledgments

Thanks to Pete Gontier.

To contact us, please use the Contact Us page.
Updated: 11-January-99

Technotes | Contents
Previous Technote | Next Technote