New Technical Notes

Macintosh

Developer Support

Pending Update Perils

Toolbox

Revised by: C.K. Haun <TR> October 1991
Written by: C.K. Haun <TR> August 1991

This Technical Note discusses potential problems when pending update events for windows behind modal dialogs are not serviced. This note also documents some new System 7 Dialog Manager calls.
Changes since August 1991: Added note clarifying how to use the new calls, documented use of StdFilterProc in Interface.o,and corrected code errors.

Introduction

Modal dialog boxes have always caused some problems with windows behind dialog windows. Since the `ModalDialog` call makes an internal event call that bypasses your normal 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. Since the event has not come through your normal event loop you have probably not serviced the update since you are only concerned about events for your dialog, so it keeps getting resent. The only way for the update to stop is for the update region of the affected window to be cleared, by the Begin/EndUpdate calls in your drawing routine (see the discussion of update handling in Inside Mac I, the Window Manager chapter).

This situation is exacerbated by screen savers or Balloon Help in System 7. If a screen saver becomes active while a modal dialog is up, or if your user has Balloon help on and part of a behind window is obscured by a balloon, then an update event will be generated for the behind window, and you normally have no way to clear it.

The Update And Modal Dialog

Under System 7 (and in System 6 under MultiFinder), 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 cause the problem, they will be handled normally by the application in the background. But updates for the frontmost application *must* be serviced or the other applications 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 Clock desk accessory. Open the Clock DA, then launch an application that you know has a modal dialog. Position the clock so you can see it, and you'll notice that it refreshes its time count 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 Clock. 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 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 the Dialog Manager chapter of Inside Mac volume I) 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.

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.

In MPW C

```
/* 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 I create the window */
myWindowPtr = GetNewWindow(kMyWindowID,nil,(WindowPtr)-1);
SetWRefCon(myWindowPtr, (long)myDrawingProcFlag);
```

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

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

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;
```

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

The only down side 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. These calls were created and tested too late in System 7's development cycle to be documented in Inside Macintosh, so they are presented here. They allow you to call on the services of the System to track standard keystrokes in your dialog.

NOTE: You must call the standard filter proc (see GetStdFilterProc below) for these new calls to work properly. Automatic cursor tracking, default button bordering, and keystroke aliasing for OK and Cancel will only be active if you call the standard filter procedure. Also, these calls are System 7 specific. You cannot use them in previous system versions.

To make things even easier, MPW 3.2 and later contain glue code to allow you to call the standard filter procedure without calling GetStdFilterProc and dispatching to the procedure pointer returned. The glue routine is called StdFilterProc and is contained in the Interface.o file in the standard MPW libraries. The description of the call is included below. If you are not using the MPW development environment and do not have access to the MPW libraries from your development environment, you will of course have to get the procedure pointer and call it yourself.

New System 7 Dialog Manager call interfaces

MPW C

```
/* Returns a pointer to the Dialog Manager's standard dialog filter */
pascal OSErr GetStdFilterProc(ModalFilterProcPtr *theProc )
       = { 0x303C, 0x0203, 0xAA68 };
```

```
/* Indicates to the Dialog Manager which item is default.  Will then alias the
/* return & enter keys to this item, and also bold border it for you (yaaaaa!) */

pascal OSErr SetDialogDefaultItem(DialogPtr theDialog, short newItem)
       = { 0x303C, 0x0304, 0xAA68 };

/* Indicates which item should be aliased to escape or Command - . */
pascal OSErr SetDialogCancelItem(DialogPtr theDialog, short newItem)
       = { 0x303C, 0x0305, 0xAA68 };

/* Tells the Dialog Manager that there is an edit line in this dialog, and it should
/* track and change to an I-Beam cursor when over the edit line */
pascal OSErr SetDialogTracksCursor(DialogPtr theDialog, Boolean tracks)
       = { 0x303C, 0x0306, 0xAA68 };


/* This routine is included in the MPW 3.2 Interface.o library, and eliminates the
/* need for you to have to dispatch to the ModalFilterProcPtr returned by GetStdFilterProc
*/
/* StdFilterProc will call GetStdFilterProc and dispatch to it for you */
pascal Boolean StdFilterProc(DialogPtr theDialog,EventRecord *theEvent,short *itemHit);
```

MPW Pascal

```
{ Returns a pointer to the Dialog Manager's standard dialog filter }
FUNCTION GetStdFilterProc(VAR theProc: ProcPtr ): OSErr;
       INLINE $303C, $0203, $AA68;

{ Indicates to the Dialog Manager which item is default.  Will then alias the return & }
{ enter key }
{ to this item, and also bold border it for you (yaaaaa!) }
FUNCTION SetDialogDefaultItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
       INLINE $303C, $0304, $AA68;

{ Indicates which item should be aliased to escape or Command - . }
FUNCTION SetDialogCancelItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
       INLINE $303C, $0305, $AA68;

{ Tells the Dialog Manager that there is an edit line in this dialog, and }
{ it should track and change to an I-Beam cursor when over the edit line }

FUNCTION SetDialogTracksCursor(theDialog: DialogPtr; tracks: Boolean):OSErr;
       INLINE $303C, $0306, $AA68 ;

{ This routine is included in the MPW 3.2 Interface.o library, and eliminates the }
{ need for you to have to dispatch to the ModalFilterProcPtr returned by GetStdFilterProc }
{ StdFilterProc will call GetStdFilterProc and dispatch to it for you }
FUNCTION StdFilterProc(theDialog: DialogPtr; VAR event: EventRecord; VAR itemHit: INTEGER):
BOOLEAN;
```

MPW Assembly

```
selectGetStdFilterProc                  EQU                 3
paramWordsGetStdFilterProc              EQU                 2

selectSetDialogDefaultItem              EQU                 4
paramWordsSetDialogDefaultItem          EQU                 3
```

```
selectSetDialogCancelItem               EQU                 5
paramWordsSetDialogCancelItem           EQU                 3

selectSetDialogTracksCursor                     EQU                 6
paramWordsSetDialogTracksCursor         EQU                 3


_DialogDispatch     OPWORD       $AA68
              MACRO
              DoDialogMgrDispatch &routineName
              DoDispatch _DialogDispatch,select&routineName,paramWords&routineName
              ENDM

; Returns a pointer to the Dialog Manager's standard dialog filter
; FUNCTION GetStdFilterProc(VAR theProc: ProcPtr): OSErr;
;
              MACRO
              _GetStdFilterProc
              DoDialogMgrDispatch GetStdFilterProc
              ENDM

; Indicates to the Dialog Manager which item is default.  Will then alias the return ; key
& enter key to this item, and also bold border it for you (yaaaaa!)
; FUNCTION SetDialogDefaultItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
;
              MACRO
              _SetDialogDefaultItem
              DoDialogMgrDispatch SetDialogDefaultItem
              ENDM

; Indicates which item should be aliased to escape or Command - .
; FUNCTION SetDialogCancelItem(theDialog: DialogPtr; newItem: INTEGER): OSErr;
;
              MACRO
              _SetDialogCancelItem
              DoDialogMgrDispatch SetDialogCancelItem
              ENDM

; Tells the Dialog Manager that there is an edit line in this dialog, and
; it should track and change to an I-Beam cursor when over the edit line
; FUNCTION SetDialogTracksCursor(theDialog: DialogPtr; tracks: Boolean): OSErr;

              MACRO
              _SetDialogTracksCursor
              DoDialogMgrDispatch SetDialogTracksCursor
              ENDM
```

Using these calls requires a little preparation on your part. After you create your dialog, you need to tell the Dialog Manager which items you want as the default and cancel items. The button selected as the cancel item will be toggled by the Escape key or by a Command-. keypress. The button specified as the default will be toggled by the return or enter key, and also will have the standard heavy black border drawn around it! The buttons will also be hilited when the correct key is hit.

The `SetDialogTracksCursor` call tells the Dialog Manager that you have edit lines in your dialog. When you pass a 'true' value to the `SetDialogTracksCursor` call the Dialog Manager will constantly check cursor position in your dialog, and change the cursor to an I-Beam when the cursor is over an edit line.

So the complete System 7 filter, incorporating update handling and new Dialog Manager calls, will look something like this;

MPW C

```
/* Before we go into a ModalDialog loop, do a little preparation */
myDialogPtr = GetNewDialog(kMyDialogID, nil, (WindowPtr)-1);
myErr = SetDialogDefaultItem(myDialogPtr,ok);  /* Tell the Dialog Manager that  /* the OK
button is the default */
myErr = SetDialogCancelItem(myDialogPtr,cancel);     /* Tell the Dialog Manager the
                                    /* cancel button is the cancel
                            /* item*/
myErr = SetDialogTracksCursor(myDialogPtr,true);     /* We have an edit item in our   /*
dialog,so tell the Dialog                    /* Manager to change the cursor to     /*
an I-Beam when it's over the                 /* edit line */
do {
   ModalDialog((ModalFilterProcPtr)myFilter, &hitItem);
   }while(hitItem != ok && hitItem !=cancel);

/* and your filter will look something like this */
pascal Boolean myFilter(DialogPtr currentDialog, EventRecord *theEventIn, short
*theDialogItem)
{OSErr myErr;
ModalFilterProcPtr standardProc;
Boolean returnVal = false;
WindowPtr temp;
if(theEventIn->what == updateEvt &&  theEventIn->message != currentDialog) {
                                      /* 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, pass it on to the */
                                     /* system filter */
       GetPort(&temp);                /* save the current port */
       SetPort(currentDialog);        /* and set to the dialog, this is necessary */
                                     /* to track the edit line */
                                    /* cursor change correctly */
/* NOTE: If you are using  MPW 3.2, there is a glue routine in the Interface.o
/* library that will take care of the  details of getting and dispatching to the
/* standard filter for you.  If you are not using MPW 3.2, you will have to call the
/* standard filter procedure yourself.  Both ways will be shown here, remember to
/* only use one of these for you actual implementation */

#ifdef MPW3.2
       /* using MPW 3.2, use the glue */
       StdFilterProc(currentDialog,theEventIn,theDialogItem); /* MPW 3.2 glue   routine */
#else
       /* not using MPW 3.2, get and call the standard filter myself */
       myErr = GetStdFilterProc(&standardProc);   /* get the standard system dialog
                                      /* filter address */
       /* if it was not an update, we pass control to the standard filter */
      if(!myErr)
       returnVal= ((ModalFilterProcPtr)standardProc)
(currentDialog,theEventIn,theDialogItem);
#endif
       SetPort(temp);}
return(returnVal);
}
```

MPW Pascal

```
{ Before we go into a ModalDialog loop, do a little preparation }
{ This inline dispatches to the standard dialog filter for you }
PROCEDURE CallStdFilterProc(theDialog: DialogPtr; VAR event: EventRecord; VAR itemHit:
INTEGER; standardProc : ProcPtr);
INLINE $205F, $4ED0;
{ This pulls the proc pointer of the stack and jumps to the standard filter, }
{ MOVE.L (SP)+,A0 }
{  JMP (A0)       }

myDialogPtr := GetNewDialog(kMyDialogID, NIL, WindowPtr(-1));
myErr :=  SetDialogDefaultItem(myDialogPtr, ok); { Tell the Dialog Manager the default item
}
myErr := SetDialogCancelItem(myDialogPtr, cancel); { Tell  Dialog Manager the cancel item }
myErr :=  SetDialogTracksCursor(myDialogPtr, TRUE); { We have an edit item in our dialog,}
                                        {  so tell the Dialog Manager to change
                                      (  the cursor  to an I-Beam when it's over
                                ( edit line }
REPEAT
 ModalDialog(@MyFilter, hitItem);
UNTIL ((hitItem = ok) OR (hitItem = cancel));

Your filter for System 7 will look something like this:

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

VAR
 savePort    : GrafPort;

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(currentDialog)
 ELSE
   BEGIN
     GetPort(savePort);            { save the current port }
     SetPort(currentDialog);       { set to the dialog, this is necessary to }
                                   { track the edit line cursor change correctly }
{ NOTE: If you are using  MPW 3.2, there is a glue routine in the Interface.o library }
{ that will take care of the  details of getting and dispatching to the standard
( filter for you.  If you are not using MPW 3.2, you will have to call the standard
( filter procedure yourself.  Both ways will be shown here, remember to only use one
( of these for you actual implementation }


{$IFC MPW3.2 }
       { using MPW 3.2, use the glue }
       StdFilterProccurrentDialog, theEventIn, theItem); { MPW 3.2 glue routine }
{$ELSEC}
       { not using MPW 3.2, get and call the standard filter myself }
     myErr := GetStdFilterProc(gStandardProc); {get the current system standard
                                     (filter and store it in a global, so
                                           ( our assembly glue can use it }
                                           ( if it was not an update, pass
                                           ( control to the assembly glue that
                                     ( will call the standard filter }
```

```
     IF myErr = noErr THEN
          MyFilter := CallStdFilterProc(currentDialog, theEventIn, theItem,gStandardProc);
{$ENDC}
     SetPort(savePort);            { restore the saved port }
   END;
END;
```

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. Of course, you can't call the new Dialog Manager routines under System 6.

MPW C

```
/* Pre-system 7 dialog filter */
pascal Boolean MyFilter(DialogPtr currentDialog, EventRecord *theEventIn, short
*theDialogItem)
{ /* declared as 'pascal' since it's called by the toolbox */
#define kMyButtonDelay 8
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

Neverending updates are not a new problem, MultiFinder just makes it imperative that you do something about it. There isn't much extra work involved, just add a simple filter to all 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 windows the way they should be, instead of windows with chunks bitten out of them.

Also, using the new Dialog Manager calls (even when you're not using a filter) allow you to present a consistent user interface across the whole system, a goal we're all striving for.

Further Reference:

- *Inside Macintosh*, Volume I, Window Manager, Dialog Manager, Event Manager