Technotes

Download  Acrobat file (K)	Download  AppleWorks file (44K)

The Notification Manager: Problems & Fixes

Technote 1026 **FEBRUARY 1996**

This Technote describes two serious problems in the Notification Manager (NM), one having to do with activate events and the other with update events. These problems can cause windows in your application to be drawn redundantly or not at all. This Technote provides a workaround for the active event problem and some sample code, with explanations, for fixing the update event problem.

If you're an application or app framework developer and want to ensure the windows in your application(s) are always updated and activated properly, you should read this Note.

This Technote augments the information presented in three chapters of *Inside Macintosh* :"Event Manager" (Chapter 2) and "Window Manager" (Chapter 4) of *Macintosh Toolbox Essentials* and "Notification Manager" (Chapter 5) of *Processes*

Contents

Defining Notification Manager Problems

You can use the Notification Manager to present the user with a modal dialog (alert) that opens in front of the windows of all applications. This dialog actually appears in the window list of the frontmost application during its call to WaitNextEvent.

Important:

> Your application should not depend on the details of the way Notification Manager presents its dialog, including but not limited to the fact that NM uses Dialog Manager. The information is presented here only in order to provide a full understanding of the problem at hand.

The first thing to know is that Notification Manager calls the Dialog Manager to manage the dialog. More specifically, NM calls ModalDialog with a dialog filter which in turn calls the standard dialog filter as obtained by GetStdFilterProc.

The fact that Dialog Manager, and in particular ModalDialog and the standard dialog filter, provide imperfect event handling means that some window-oriented events are "swallowed" (i.e., never provided to your app) while the dialog is present.

Swallowed Deactivate Events (and Redundant Activate Events)

Your application does not receive a deactivate event for the (soon-to-be-former) front window when the Notification Manager's dialog appears. This is because ModalDialog has its own event loop (partially implemented by the standard dialog filter) and has no way of passing the deactivate event off to your app's event loop.

When the user dismisses NM's dialog, your app receives a redundant activate event for the (recently-reinstated) front window. Your app **can** make sure it doesn't logically activate a window (i.e., enable text fields, etc.) redundantly by simply checking to see if the window is already active before logically activating it.

Swallowed Update Events

In order to understand how update events get swallowed, you need to understand how they would have been generated in the first place.

Each window has a region, expressed in global coordinates, which describes the portion of the window that needs redrawing. This is called the window's update region. During WaitNextEvent, the Event Manager, Window Manager, and Process Manager collaborate in walking the current window list. If the update region of a window is found to be non-empty, they generate an update event for that window.

When your application receives the update event, it calls BeginUpdate, (re)draws the image for the window, and calls EndUpdate. The BeginUpdate/EndUpdate pair of calls empties the update region for the window so that subsequent searches for windows which need updating do not find that window.

These update regions are the key to understanding how the Notification Manager swallows update events.

The standard dialog filter, to which NM's dialog filter passes control, wants to ensure that background apps get processing time by "solving" the problem described in Macintosh Technical Note TB37, "Pending Update Perils." The standard filter simply calls BeginUpdate and EndUpdate every time it's given an update event, regardless of the window for which the event is bound.

This results in all windows in the current window list having their update regions emptied almost immediately. As a consequence, no update events are generated for those windows, even though they need to be (re)drawn. When the user dismisses NM's dialog, only the windows covered by NM's dialog get update events, and then only for the region covered by NM's dialog.

There is nothing straightforward your app can do to prevent swallowed update events. While your app innocently waits for a call to WaitNextEvent to return, NM suddenly puts up its dialog and seizes control of the event loop until the user dismisses the dialog. Your app can't even predict when this will happen, much less prevent it or easily work around it.

Using the Sample Code Library "UpdateRegionSaver" to Work Around the Problem

Although Apple may fix both of these problems in a future system software release, you might consider using some of the sample code presented here as a workaround. The sample code will continue to work even if there is a fix to the system software.

Most users of UpdateRegionSaver will only need to make three very simple calls, shown in this sample: SaveUpdateRegions, RestoreUpdateRegions, and DeleteUpdateRegions.

```
#ifndef __EVENTS__
#       include
#endif

#include "UpdateRegionSaver.h"

pascal Boolean WaitNextEventWithNMSafeUpdates
        (EventMask eventMask, EventRecord *theEvent,
                UInt32 sleep, RgnHandle mouseRgn)
{
        UpdateRegionSaver *root = SaveUpdateRegions ( );
```

```
        EventRecord event;
        Boolean result = WaitNextEvent
                (eventMask,theEvent,sleep,mouseRgn);
        RestoreUpdateRegions (root);
        DeleteSavedUpdateRegions (root);
        root = nil;
        return result;
}
```

For a full listing of each of the three calls, refer to Appendix A at the end of this Technote.

UpdateRegionSaver Reference

This section describes the data structures and routines specific to the UpdateRegionSaver library.

Data Structures

The UpdateRegionSaver library manipulates a data structure of type UpdateRegionSaver.

```
typedef struct UpdateRegionSaver
{

        RgnHandle                       fRgnH;
        WindowRef                       fRef;
        struct UpdateRegionSaver        *fNext;
}
UpdateRegionSaver;
```

Field Descriptions

fRgnH
: A region copied from the update region of a window. It is in global coordinates (as is the update region itself). Before being restored to the window, it will be copied and converted to the local coordinates of the window.

fRef
: The address of the window from which the copy of the update region came. Before restoring the region, UpdateRegionSaver verifies that there is still a window at this address in the window list. (This is not a perfect test, but it is the best test that can be made without patching a trap or three.)

fNext
: The address of the next structure in the linked list. The last node in the list has a 0 here.

UpdateRegionSaver Routines

To get a list of update regions associated with the windows in the current window list, call SaveUpdateRegions. To restore the regions to their owning windows, call RestoreUpdateRegions. To destroy the list of regions, call DeleteSavedUpdateRegions.

SaveUpdateRegions

To save the update regions associated with the windows in your application's window list, call SaveUpdateRegions.

SaveUpdateRegions returns a pointer to the root of simple singly-linked list which contains a node for each window. In each node your app will find a window pointer, a copy of the window's update region, and a pointer to the next node.

RestoreUpdateRegions

To restore a list of saved update regions to the windows from which they came, call RestoreUpdateRegions.

RestoreUpdateRegions copies the regions in the list and converts the copies to the local coordinates of

each window before calling InvalRgn to merge the region into any update region which may already be there.

DeleteSavedUpdateRegions

To delete a list of saved update regions, call DeleteSavedUpdateRegions.

Summary

The Notification Manager (NM) may present a Dialog Manager dialog whenever your application calls WaitNextEvent. This impedes the flow of window-related events (such as update or activate) to your application's event loop. Make sure your app doesn't logically activate a window which is already active. Use the UpdateRegionSaver library (or something like it) to ensure your app will always get the update events it requires.

Further Reference

- Macintosh Technical Note TB 37, "Pending Update Perils," has a discussion of the problem Notification Manager "solves," as mentioned previously.
- Chapter 6, "Dialog Manager," *Inside Macintosh:Macintosh Toolbox Essentials* has a description of the operation of ModalDialog and the standard dialog filter.

Appendix A

UpdateRegionSaver.h

```
#pragma once

#ifndef __WINDOWS__
#       include <Windows.h>
#endif

typedef struct UpdateRegionSaver
{
        // We don't care about alignment since this is an internal
        // runtime-only structure.

        RgnHandle                       fRgnH;
        WindowRef                       fRef;
        struct UpdateRegionSaver        *fNext;
}
UpdateRegionSaver;

// I can't figure why this next #ifdef would be necessary for
// pascal funcs,but CW7 for PPC says it is.

#ifdef __cplusplus
extern "C" {
#endif

pascal void RestoreUpdateRegions (UpdateRegionSaver *);
pascal void DeleteSavedUpdateRegions (UpdateRegionSaver *);
pascal UpdateRegionSaver * SaveUpdateRegions (void);

#ifdef __cplusplus
}
#endif
```

UpdateRegionSaver.c

```
#ifndef __LOWMEM__
#       include <LowMem.h>
#endif

#include "UpdateRegionSaver.h"
```

```
static pascal Boolean IsWindowStillAround (WindowRef ref)
{
        WindowRef scan = LMGetWindowList ( );

        while (scan)
        {
                if (scan == ref) break;
                scan = GetNextWindow (scan);
        }

        return !!scan;
}

pascal void RestoreUpdateRegions (UpdateRegionSaver *ursp)
{
        while (ursp)
        {
                if (!EmptyRgn (ursp->fRgnH) && IsWindowStillAround (ursp->fRef))
                {
                        RgnHandle localUpdateRgn = NewRgn ( );
                        if (localUpdateRgn)
                        {
                                Point zero;
                                GrafPtr keep = qd.thePort;
                                SetPort (ursp->fRef);
                                zero.h = qd.thePort->portRect.left;
                                zero.v = qd.thePort->portRect.top;
                                GlobalToLocal (&zero);
                                CopyRgn (ursp->fRgnH,localUpdateRgn);
                                OffsetRgn (localUpdateRgn,zero.h,zero.v);
                                InvalRgn (localUpdateRgn);
                                SetPort (keep);
                                DisposeRgn (localUpdateRgn);
                        }
                }

                ursp = ursp->fNext;
        }
}

pascal void DeleteSavedUpdateRegions (UpdateRegionSaver *ursp)
{
        while (ursp)
        {
                UpdateRegionSaver *next = ursp->fNext;
                DisposeRgn (ursp->fRgnH);
                DisposePtr ((Ptr) ursp);
                ursp = next;
        }
}

pascal UpdateRegionSaver * SaveUpdateRegions (void)
{
        //
        //  This function saves as many update regions as it can.
        //  If for some reason memory is so low that some regions
        //  cannot be saved, this function makes a best effort.
        //  (Its best effort is rather stupid, but it does try.)
        //

        UpdateRegionSaver *root = nil;
        WindowRef scan = LMGetWindowList ( );

        while (scan)
        {
                UpdateRegionSaver *newUpdateRegionSaver =
                        (UpdateRegionSaver *) NewPtr (sizeof (UpdateRegionSaver));
                if (!MemError ( ))
```

```
                {
                        RgnHandle rgnH = NewRgn ( );

                        if (!rgnH)
                        {
                                DisposePtr ((Ptr) newUpdateRegionSaver);
                                newUpdateRegionSaver = nil;
                        }
                        else
                        {
                                GetWindowUpdateRgn (scan,rgnH);

                                newUpdateRegionSaver->fRgnH     = rgnH;
                                newUpdateRegionSaver->fRef      = scan;
                                newUpdateRegionSaver->fNext     = root;

                                root = newUpdateRegionSaver;
                        }
                }

                scan = GetNextWindow (scan);
        }

        return root;
}
```
