TransSkel Programmer’s Notes

2: Orphan Events During Dialog Processing

Who to blame: Paul DuBois, dubois@primate.wisc.edu
Note creation date: 01/28/91
Note revision: 1.05
Last revision date: 01/15/94
TransSkel release: 3.00

This Note discusses some event processing difficulties that occur when a modal dialog comes up over the frontmost window. These difficulties are not specific to the TransSkel environment, although the solution proposed to alleviate them is.

Familiarity with the TransSkel routines `SkelDoEvents()`, `SkelRouteEvent()` and `SkelWindowRegistered()` is assumed. These routines are new in release 3.00.

Since this note was originally written, Apple has released Macintosh Technical Note TB 37, which deals with some of the same issues discussed here. Revisions were made here to reflect some of additional information presented there.

Beginning with TransSkel 3.02 and release 1.03 of this Note, the argument list for `SkelDlogFilter()` includes a `Boolean` allowing the caller to indicate whether processing of special keys should be done or not.

Beginning with TransSkel 3.06 and release 1.04 of this Note, some statements about (in)capabilities of TransSkel and the THINK C compiler that are no longer true are flagged and annotated in an Addenda section.

Some limitations of `SkelDlogFilter()` have been identified and eliminated in TransSkel 3.06. This is discussed in TransSkel Programmer’s Note 8: Dialog Events Revisited.

The Problem

When a dialog (or alert, which is a special type of modal dialog) comes up in front of a window, the window manager takes care of deactivating that window’s frame and generates a deactivate event for it. When the application receives the deactivate, it knows that the window content region should be changed to reflect its new state. This may include actions such as hiding scroll bars, dimming buttons and check or radio boxes, dimming the outline around a default button, or deactivating highlighted text and list selections.

The problem is that the application might never actually see the deactivate event. The default alert and dialog filter functions respond to activates for dialog windows, but have no way to respond to such for other non-modal windows, since they have no way to know where to route them so they can be processed. Consequently, the deactivate is simply discarded, leaving the window visually inconsistent. The window frame properly reflects the deactivated state, but the window contents look as though the window is still active.

Loss of deactivates is actually less serious than other problems that occur in the context of dialog processing. At least the window contents remain drawn, albeit in an incorrect state. A more difficult problem is that non-delivery of update events can also occur. This frequently happens in situations involving two or more alerts or dialogs.

In one common scenario, by selecting the first item under the Apple menu, the user can see an alert that has a button to dismiss it and a button for further help. Suppose that when the user clicks the Help button, the alert is dismissed and a help dialog is presented. If the dialog window is a different size or shape than the alert, part of the underlying window erased by the alert dismissal may still be visible (and not redrawn) when the dialog comes up. That window will have a non-empty update region.

Using the default dialog filter, that part of the window will remain erased until the dialog is dismissed, even though an update event is generated for the window. Why? Because the update will not be processed — the filter has no way of knowing how to route the event. In fact, an update event will be generated *repeatedly*, because the update region never gets cleared with `BeginUpdate()` and `EndUpdate()`. Thus, as long as the dialog is present, a continual stream of update events is presented to (and ignored by) the filter function. The dialog itself appears to function normally because user actions — mouse and key clicks — have higher priority than updates, but update processing for the window beneath it does not actually occur until the dialog is dismissed.

This same problem occurs with Apple's movable modal dialog window type, introduced with System 7. While it is useful to be able to move a modal dialog around, doing so damages underlying windows and results in generation of update events. If they are left unprocessed, it does no good to move the dialog, since presumably the reason you move it is to see what's under it!

Balloon Help, another feature of System 7, can damage arbitrary portions of the desktop or other windows. A balloon may come up anywhere; when it disappears, update events are generated.

Deactivate and update events which are discarded as described above are *orphan* events — no window handler is found to adopt them as its own.

In addition to deactivates and updates, MultiFinder events (`osEvt` events, formerly known as `app4Evt` events) are also ignored by the default filters and become orphaned, too.

The occurrence of orphan events is not disastrous for the frontmost application. It continues to function and not crash, and the right thing happens when dialogs or alerts go away. Nevertheless, it doesn't look right for window contents to appear as though the window is active when it's not, or for hunks of the content region to be blank when they shouldn't be.

More serious is the potential impact on background applications. Apple says (TN TB 37) that under System 7, failure to service update events for an application's own windows during modal dialog processing can cause other applications to stop dead because they will no longer get any time. (I suppose this is because updates are a ToolBox event and not an OS event, and `WaitNextEvent()` keys its switching activities on OS events, or something like that.) Worse yet, you cannot prevent update events of this nature from occurring under System 7, given the existence of Balloon Help.

Solutions to Problem

A consistent mechanism to allow orphan events to be processed properly is desirable. Some of the TransSkel auxiliary routines are designed to provide such a mechanism, in a manner that (i) is general purpose; (ii) is compatible with the TransSkel core; (iii) requires minimal effort on the part of the application programmer.

Several solutions to the problem of handling orphan events are discussed below, illustrating some of the process through which the auxiliary routines were designed. These approaches are all framed in terms of implementation within the TransSkel processing environment, although the basic problem is a general Macintosh phenomenon.

Approach 1: Anticipate orphan events

If one knows that a dialog is going to come up over the frontmost window, the deactivate event for that window can be anticipated. The effect of its arrival can be simulated, by calling the window's activate handler function to perform deactivate processing before actually showing the dialog. Then it doesn't matter if the real event is discarded by the dialog filter function.

The problem with this in the TransSkel environment is that you must know the window's activate function. This knowledge cannot, in the general case, be guaranteed since it's not necessarily true that your application will know all the handlers for the windows it uses. It's perfectly possible to write window-creating modules that install handlers but export no information about them to the main program.[1]

To get around this, it seems like you should just be able to use `PostEvent()` to put a deactivate event for the frontmost window into the event queue, let TransSkel route it to the correct handler, then bring up the dialog. However, deactivates are not among the events that can be posted, because they never actually go into the event queue (IM I-245, IM II-67, 68). It's necessary to let the Window Manager itself generate the deactivate event. If the dialog resource specifies that the dialog is initially invisible, this can be done as follows:

```
dlog = GetNewDialog (…);
        …set up dialog…
SelectWindow (dlog);
SkelDoEvents (activateEvt);
ShowWindow (dlog);
```

Here the dialog is created, set up and selected, then TransSkel is told to dispatch pending activate (and deactivate) events. Then the dialog is made visible. This relies on the fact that selecting an invisible window deactivates the currently frontmost window, which conveniently allows the ensuing deactivate event to be processed. When the dialog is made visible, its activate event is generated (presumably to be processed by a call to `ModalDialog()` following the code above).

This method doesn't work for alerts, though — there's no corresponding `GetNewAlert()` call.

Another disadvantage of the event anticipation approach is that update events cannot, in general, be anticipated because you don't know when they might occur and for which underlying windows, particularly if Balloon Help is turned on. Pending updates will not be processed until the dialogs go away. MultiFinder events cannot be anticipated, either.

Approach 2. Bypass `ModalDialog()`

[1]This is no longer a limitation in TransSkel. See Addendum 1.

Modal dialogs are processed by calling `ModalDialog()`. One could register modal dialogs with TransSkel and bypass `ModalDialog()` entirely, allowing all events to pass through the usual TransSkel event dispatcher, and unifying treatment of modal dialogs with treatment of modeless dialogs and non-dialog windows. This is appealing, especially in light of Apple's apparent recommendation that movable modal dialogs be handled in the application's main event loop (MEL).

Unfortunately, the method doesn't work for alerts, since the ToolBox alert calls use `ModalDialog()` internally and no way is provided to bypass it. It also doesn't work for certain non-alert dialogs. For instance, some of the dialogs provided by the Standard File package don't allow `ModalDialog()` to be bypassed.

Approach 3: Always provide a filter function

Orphan events occur because modal dialogs and alerts processed with `ModalDialog()` are, in effect, running a different event loop than your application's own main event loop. Consequently, events for the application's non-dialog windows are going through the wrong loop — one not designed with those windows in mind. Since one cannot, in general, get rid of `ModalDialog()`, what's needed is a way to integrate the two event loops, or a way to break into the `ModalDialog()` event loop and "steal" certain events so they can be routed into the application's MEL.

Integrating the two loops would be pretty difficult, but it is possible to break into the `ModalDialog()` loop, because alert and dialog processing calls allow a filter function to be specified. This filter gets a shot at events retrieved by `ModalDialog()` — what should it do?

Since one problem is lost deactivates and updates, the filter needs to recognize those events, determine whether the window to which the event applies has a TransSkel handler (i.e., has been registered), and if so, pass the event to `SkelRouteEvent()`, which is part of the MEL. To tell whether the window has a handler, it's not sufficient to check that it's not a dialog window, since modeless dialogs may be registered. The function `SkelWindowRegistered()` may be used for this purpose. The other problem is loss of MultiFinder events, so the filter must recognize and reroute those, too.

This works: you don't have to know anything about window handler functions, and you don't even have to bother setting the port to the right window, since TransSkel makes sure it's set properly while activate and update handlers are executing. If the application has made itself MultiFinder-aware, `SkelRouteEvent()` also makes sure that

MultiFinder events get sent to the right place.

The disadvantages of this approach are the necessity of always using a filter function, and that you have to include this kind of code in all of your filter functions. (It’s actually worse: by not using the default filter, you become responsible to do your own Enter and Return key processing for the default button.)

Approach 4: Use piggybacked filter functions

The previous approach seems to do what’s needed, but the cost in code duplication may be high. It would be better to have a single filter function which could be reused for every alert or dialog, but which would allow you to use whatever filter function you would otherwise use — in other words, a *standard filter* onto which a alert- or dialog-specific filter can be piggybacked. The standard filter would handle updates and activates, and the Return and Enter keys if necessary. All other events would pass through to the piggyback filter. (Return/Enter processing is necessary, because by using a non-`nil` standard filter, the application becomes responsible for handling those keys.)

This seems to be a reasonable approach. Orphan events are handled and the code to do so is packaged into a single place. The question remains as to implementation of this mechanism.

Implementation

Dialogs and alerts are typically invoked by calls like these:

```
ModalDialog (filter, &item);

Alert (resourceNum, filter);
```

where `filter` is the filter function. Below, several ways of implementing piggybacked filters are discussed. The requirements to be met are:

- Must work with any existing call that takes filter arguments
- Should work without change if Apple invents new calls that take filter arguments
- Must be reentrant since multiple alerts/modal dialogs may be shown simultaneously
- Should work both for alerts and for modal dialogs.

The function to be created is `SkelDlogFilter()`, which looks like this:

```
typedef pascal Boolean (*SkelDlogFilterProcPtr)
                                  (DialogPtr, EventRecord *, Integer *);

SkelDlogFilterProcPtr
SkelDlogFilter (SkelDlogFilterProcPtr filter, Boolean doReturn);
```

`filter` is the filter that would normally be passed to `ModalDialog()` or an alert call. `doReturn` is `true` if Return/Enter key processing is to be done. It’s used so that the calling program can take advantage of the key processing facilities of `SkelDlogFilter()` even if filter is not `nil`, to avoid duplicating functionality.

Note: `SkelDlogFilterProcPtr` is logically the same type as `ModalFilterProcPtr`. It’s declared explicitly in *TransSkel.h* because the THINK C compiler considers `ModalFilterProcPtr` the same as `void *` by default, which is not helpful for type checking (and in some cases results in spurious compiler messages for correctly written code).[2]

Method 1. Interpolate the standard filter into alert and dialog calls

From the application programmer’s point of view, it would be easy if calls to functions that took filters could be written something like this:

```
ModalDialog (SkelDlogFilter (filter, true), &item);

Alert (resourceNum, SkelDlogFilter (filter, true));
```

`SkelDlogFilter()` returns a pointer to the standard filter and stashes a copy of its argument somewhere for use by the standard filter. Unfortunately, this method does not provide reentrancy

[2]This problem no longer exists. See Addendum 2.

— you can't have more than one alert or dialog active simultaneously. If a second is presented, its filter is installed — but not uninstalled when control returns to the first dialog.

Method 2. Use replacement functions

An alternative would be to provide replacement functions for all routines that take filter function arguments, e.g.,

```
        SkelModalDialog (filter, &item);

        SkelAlert (resourceNum, filter);
```

In terms of textual modification of existing sources, this is a nice solution : you just prepend "Skel" to the names of the alert and dialog calls.

The replacement functions could take care of the reentrancy problem by installing and uninstalling their filters before and after calling the functions they replace. This could be done by maintaining a common stack internally, for instance.

The problem is that a replacement function is needed for every ToolBox and OS call that takes a filter argument. This approach does not automatically extend to any new traps that Apple might invent in the future, either. New replacements would have to be invented each time that happened.

Method 3. Require explicit filter function installation and removal

Two functions `SkelDlogFilter() and SkelRmveDlogFilter()` are defined. They can be used for any function that takes a filter, and require the programmer to explicitly install and remove the filters that would otherwise be passed.

```
        ModalDialog (SkelDlogFilter (filter, true), &item);
        SkelRmveDlogFilter ();

        Alert (resourceNum, SkelDlogFilter (filter, true));
        SkelRmveDlogFilter ();
```

This requires more modification of existing programs than one might like. On the plus side, it satisfies all the requirements:

- It works for all current routines that take filter arguments
- It should work for any new traps Apple might invent that take filter arguments
- It can be made reentrant
- It works both for alerts and dialogs

This approach is also flexible enough that the minimum-modification implementation (method 2) can be easily layered on top of it if you wish. Recall that method 2 would replace `ModalDialog()`, `Alert()`, etc., with `SkelModalDialog()`, `SkelAlert()`, etc. The latter are easy to write in terms of implementation 3. For instance, `SkelAlert()` becomes:

```
        SkelAlert (Integer resourceNum, SkelDlogFilterProcPtr filter)
        {
        Integer       result;

                result = Alert (resourceNum, SkelDlogFilter (filter, true));
                SkelRmveDlogFilter ();
        return (result);
    }
```

When the default dialog and alert filters are used (i.e., when you supply a `nil` filter function to `ModalDialog()` or alert calls), Return and Enter key processing is performed for you. Using `SkelDlogFilter()`, the filter passed to `ModalDialog()` and alert calls is never `nil`. For consistency, the standard filter emulates this behavior. It performs Return and Enter processing by returning the value of the `aDefItem` field of the dialog record. For dialogs, this will always be 1 (IM I-409, 415). For alerts, the default button may vary depending on the current alert stage (IM I-419).

In order to accommodate `SkelDlogFilter()`, it was necessary to modify TransSkel's activate event router slightly. The TransSkel port-setting model is based on on activates and sets the port to any window which comes active. This was also done for deactivates, to make sure the port was correct while the window's activate handler executed. However, for deactivates, no attention was paid to saving and restoring the port, on the assumption that an activate for some other window would soon follow and the port would then be set to that window.

Unfortunately, modal dialogs fall outside of this model. Typically, when presenting a dialog, the port setting is done explicitly by the application, not by TransSkel: one creates the dialog, saves the current port, sets the port to the dialog, processes it, and restores the port. Formerly (before the invention of `SkelDlogFilter()`) the deactivate for the underlying window would not have gone through TransSkel — because it would have been thrown away! Using `SkelDlogFilter()`, that's no longer true. It was therefore not correct for TransSkel to change the port for a deactivate and leave it changed. To fix this, TransSkel now saves and restores the port when handling deactivates (the port is still set and *not* restored on activates, as before).

Unsightly Blemishes

- A fixed-size stack of depth 10 is used. This limits reentrancy to 10 simultaneous alerts and dialogs. This limitation could be avoided by dynamic allocation of linked list elements, but that seems like overkill. I figure a depth of 10 is deep enough, since too many dialogs will confuse the user, anyway.

- The Standard File package presents modal dialogs allowing the user to specify files. To use the filter mechanism provided by `SkelDlogFilter()`, you must use `SFPGetFile()` and `SFPPutFile()` rather than `SFGetFile()` and `SFPutFile()`. (The filter arguments passed to the latter routines are *file* filters, not *event* filters.) For instance, a call to `SFGetFile()` would be converted as follows:

  From:

  ```
          SFGetFile (pt, "\p", nil, 1, &type, nil, &sfReply);
  ```

  To:

  ```
          SFPGetFile (pt, "\p", nil, 1, &type, nil, &sfReply,
                                  getDlgID, SkelDlogFilter (nil, true));
          SkelRmveDlogFilter();
  ```

  The conversion from `SFPutFile()` to `SFPPutFile()` is similar.[3]

[3]System 7 introduces Standard File calls that use a different dialog filter. See Addendum 3.

- There are circumstances over which one simply has no control. For instance, if the user inserts an uninitialized disk, the disk formatting routines present dialogs, but make no provision for specifying dialog event filters. The same is true for the page setup and print job dialogs presented by `PrStlDialog()` and `PrJobDialog()`, although you can use the information in Tech Note PR 09 to perform equivalent tasks in a manner that allows you to get at the dialog event filter.

Using `SkelDlogFilter()` with Movable Modal Dialogs

From what I've read, Apple recommends that `ModalDialog()` not be used to process movable modal dialogs. Some of the rationale for this stems from exactly the sort of problems that `SkelDlogFilter()` is designed to solve, e.g., if the dialog is movable, moving it generates update events for underlying windows that need to be serviced. The other part of the rationale was that MultiFinder patched `ModalDialog()` in some unfriendly way to know when not to switch. MultiFinder has since been changed to detect when not to switch in a different way, so it may well be perfectly suitable to use `ModalDialog()` with `SkelDlogFilter()` to handle movable modal dialogs. This is all conjecture, though, until someone actually tries it and reports the results.

Summary

This Note discusses some difficulties that occur during modal dialog event processing, and describes a dialog filter mechanism provided by TransSkel to deal with those difficulties.

Use of the TransSkel dialog filter is not mandatory; it's not part of the TransSkel core. It's up to you to decide whether or not to use it. You might not use it, for instance, if your windows take a significant amount of time to deactivate or update. An important principle to keep in mind is to be consistent within your application. By this principle you would use the filter ubiquitously, or not at all.

However, the existence of the problems described in Tech Note TB 37 (background applications stopping) lends a certain degree of support to the practice of using `SkelDlogFilter()` for *every* alert and `ModalDialog()` call you make.

References

Inside Macintosh, Volume I. Dialog Manager, Standard File Package, ToolBox Event Manager, and Window Manager chapters
Inside Macintosh, Volume II. Operating System Event Manager chapter
Inside Macintosh, Volume VI. Help Manager chapter
Macintosh Technical Note PR 09: How To Add Items to the Print Dialogs
Macintosh Technical Note TB 35: MultiFinder Miscellanea

Macintosh Technical Note TB 37: Pending Update Perils
movable-modal-wdef-zen.txt (available via FTP on *ftp.apple.com*, in the directory *~ftp/dts/mac/hi*)
TransSkel Programmer’s Note 8: Dialog Events Revisited

Addenda

1. Beginning with TransSkel 3.05, you can call `SkelActivate()` to deactivate your front window without knowing its activate handler explicitly. However, that’s an extra step you must do before showing your dialog, and it remains true that other events such as updates cannot be predicted and must be handled as they occur.

2. The THINK C 5 default precompiled MacHeaders to treat function pointers as `void *`, losing prototype information. Thus, although `SkelDlogFilterProcPtr` and `ModalFilterProcPtr` were logically equivalent, for type-checking purposes they were not. The THINK C 6 default precompiled MacHeaders retains full prototype information. `SkelDlogFilterProcPtr` is thus fully equivalent to `ModalFilterProcPtr` and has become vestigial. The return value of `SkelDlogFilter()` and the type of its first argument are now declared as `ModalFilterProcPtr`.

3. The Standard File calls `CustomGetFile()` and `CustomPutfile()` introduced in System 7 use dialog filters with a slightly different argument structure, so `SkelDlogFilter()` doesn't work with them. This is discussed in TPN 8.