TransSkel

A Transportable Skeleton for
Macintosh Application Development

Tutorial

Release 3.17
1 June 1994

Table of Contents

Introduction

This document is a tutorial for writing TransSkel applications in C. You should consult the TransSkel Reference Manual for details about particular routines. There is also a series of TransSkel Programmer's Notes (TPN's) that provide an adjunct to the Reference Manual.

A Pascal interface is available, but this tutorial doesn't discuss how to use TransSkel from Pascal. See the Reference Manual for details.

Getting Started

The general logic of TransSkel applications is:

- Call `SkelInit()` to tell TransSkel to initialize itself.
- If you use menus or windows, create them and register them with TransSkel. This is usually done with `SkelApple()`, `SkelMenu()`, and `SkelWindow()`.
- If you take advantage of multitasking, register handler functions with TransSkel. This is usually done with `SkelSetSuspend()`, `SkelSetClipCvt()`, and `SkelSetWaitTimes()`.
- If you respond to Apple Events, register a handler function with TransSkel using `SkelSetAEHandler()`.
- Call `SkelEventLoop()`, TransSkel's main event-processing loop. It routes events to the proper object handlers automatically.
- Call `SkelStopEventLoop()` to terminate the event loop. This will often be in the code that handles the Quit item of a File menu handler.
- When event processing terminates, call `SkelCleanup()` to tell TransSkel to dispose of menus, windows, and their handlers.

This section develops in a simple fashion a small application SimpleApp that illustrates some basic ways to use TransSkel. We'll do this by starting with a minimal application and then add extra functionality to it. The sections following this one provide more detail on particular aspects of TransSkel application writing.

A Minimal Application

The first step is to write an application that does nothing but start up and shut down. Thus the first version of the source file *SimpleApp.c* looks like this:

```
# include    "TransSkel.h"

int
main (void)
{
       SkelInit (nil);     /* initialize */
       SkelCleanup ();     /* clean up */
}
```

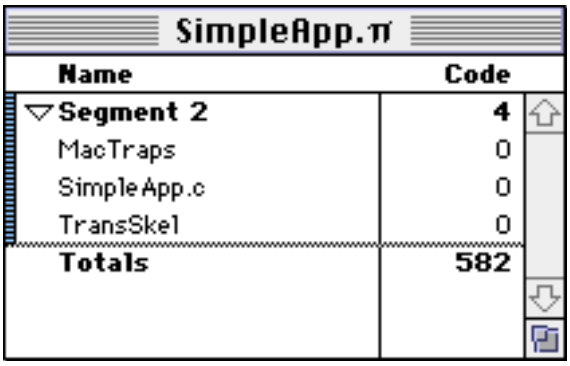
The two functions invoked by this application should be called by all TransSkel applications.

`SkelInit()` initializes TransSkel and is normally the first TransSkel call your application makes. It sets up your application's heap zone, initializes the Toolbox managers, and determines various characteristics of the machine on which your application is running. You can pass `nil` to `SkelInit()` to use the default initialization parameters. (The argument is actually a pointer to an initialization parameters structure; if you want know how to specify initialization parameters explicitly, consult the Reference Manual and TPN 5.)

`SkelCleanup()` removes any internal data structures that TransSkel maintains.

The application also includes the header file *TransSkel.h*, which declares the constants, types, and functions that make up the TransSkel interface. It's necessary to include this file in any source file you write that uses the TransSkel interface in some way. We won't show it in any further examples, but remember to use it.

To build SimpleApp, we need a project document. It should include the source file *SimpleApp.c*, the MacTraps library, and TransSkel:



You can build and run SimpleApp to verify that it launches and exits properly.

Using Menus

The minimal application isn't very interesting, so let's write something slightly less trivial. Here's `main()` again, revised to support an Apple menu and a File menu with a Quit item. The intent is for the application to run until you select Quit from the File menu.

```
int
main (void)
{
MenuHandle    m;

        SkelInit (nil);             /* initialize */
        SkelApple (nil, nil);       /* handle Apple menu */
        /* create File menu, install handler */
        m = NewMenu (skelAppleMenuID + 1, "\pFile");
        AppendMenu (m, "\pQuit/Q");
        (void) SkelMenu (m,         /* menu handle */
                    DoFileMenu,     /* item selection function */
                    nil,            /* menu disposal function */
                    false,          /* not a submenu */
                    true);          /* draw menu bar */
        SkelEventLoop ();           /* loop 'til Quit selected */
        SkelCleanup ();             /* clean up */
}
```

Here are the steps taken by `main()`:

- Call `SkelInit()` to initialize TransSkel.
- Call `SkelApple()` to set up the Apple menu.
- Create the File menu, put the Quit item in it, and register it with TransSkel by calling `SkelMenu()`.
- Call `SkelEventLoop()` to run the main event loop. This will return after Quit is selected from the File menu.
- Call `SkelCleanup()` to shut down TransSkel.

`SkelApple()` allows your application to create and manage an Apple menu easily.[1] The two arguments are a string indicating the title of the application-specific item that begins the Apple menu (usually something like "About MyApp…") and the name of the function that's called when you select the application item. In the source shown above, no application item is given, and thus no function is needed to handle it, so both arguments are `nil`.

For all menus other than the Apple menu, TransSkel requires you to create the menu yourself. You can do this either by calling `NewMenu()` or by storing the menu in a resource file and calling `GetMenu()`. The code above calls `NewMenu()` and uses a menu ID of `skelAppleMenuID+1`. TransSkel uses `skelAppleMenuID` (128) as the ID for the Apple menu, so numbers higher than that are safe for your own menus.

After you create your menu, call `SkelMenu()` to register it with TransSkel. `SkelMenu()` also puts the menu in the menu bar for you. The arguments to `SkelMenu()` specify the menu handle, the function to call when items are selected from the menu, the function to call to dispose of the menu, whether or not the menu is a submenu, and whether or not to draw the menu bar after inserting the menu into it. We've specified `DoFileMenu()` to be called to handle item selections, no disposal function, that the menu is not a submenu, and that the menu bar should be drawn after inserting the File menu. `SkelMenu()` returns `true` or `false` depending on whether or not it was able to create the internal data structures needed for keeping track of the menu. (We're assuming it succeeds by ignoring the return value.)

Now we need to write `DoFileMenu()`. Menu selection handler functions should be written to return no value and take a single argument that indicates the number of the item that was selected. Our File menu has only one item, so we can ignore the item number in the body of the function and write `DoFileMenu()` like this:

```
static pascal void
DoFileMenu (short item)
{
	SkelStopEventLoop ();	/* tell SkelEventLoop() to quit */
}
```

The application's main event loop is initiated by calling `SkelEventLoop()`. For our current application, the only things the event loop processes are selections from the two menus. TransSkel handles items selected from the Apple menu. `DoFileMenu()` is called from the event loop when

[1] You should always have an Apple menu since System 7 appears to crash when you click in the menu bar if you don't have one.

Quit is chosen from the File menu; it calls `SkelStopEventLoop()` to terminate the event loop. When `DoFileMenu()` returns, TransSkel stops running the loop and passes control back to `main()`. The application then calls `SkelCleanup()` to clean up and exits.

Applications often have menu handlers that do more than what we've done above. Typically an application-specific item will be placed in the Apple menu, and selection functions for other menus usually are written to handle more than a single item. We'll see how to do these things in a later section.

Note that `DoFileMenu()` is declared as a `pascal` function. All other functions that are passed to TransSkel routines should be declared as `pascal` functions, too, because TransSkel expects to call any functions you pass to it using Pascal calling conventions. (The reason for this is so that TransSkel can be used from within Pascal applications.)

Using Windows

Now let's add some code to manage a window. In addition to the existing method of terminating the application by selecting Quit from the File menu, we'll arrange to allow clicking the window's close box to do the same thing.

Here's what the `main()` function looks like:

```
int
main (void)
{
MenuHandle    m;
WindowPtr     w;
Rect          r;

        SkelInit (nil);             /* initialize */
        SkelApple (nil, nil);       /* handle Apple menu */
        /* create File menu, install handler */
        m = NewMenu (skelAppleMenuID + 1, "\pFile");
        AppendMenu (m, "\pQuit/Q");
        (void) SkelMenu (m,         /* menu handle */
                    DoFileMenu,     /* item selection function */
                    nil,            /* menu disposal function */
                    false,          /* not a submenu */
                    true);          /* draw menu bar */

        SetRect (&r, 40, 40, 200, 120);
        w = NewWindow (nil, &r, "\pA Window", true,
                            documentProc+8, (WindowPtr) -1, true, 0L);
        (void) SkelWindow (w,
                    nil,            /* mouse click handler */
                    nil,            /* key click handler */
                    nil,            /* update event handler */
                    nil,            /* activate event handler */
                    Close,          /* close box click handler */
                    Clobber,        /* disposal function */
                    nil,            /* idle-time handler */
                    true);          /* idle only when frontmost */
        SkelEventLoop ();
        SkelCleanup ();
}
```

The new code in `main()` creates a window and calls `SkelWindow()` to register it with TransSkel. `SkelWindow()` returns `true` or `false` depending on whether or not it was able to

create the internal data structures needed for keeping track of the window. (We're assuming it succeeds by ignoring the return value.) `SkelWindow()` takes several arguments. The first is a pointer to the window that's to be registered. The rest are for handling various kinds of events and operations on the window, as indicated by the comments in the code.

For now, we'll pass `nil` for most of the window handler functions. We'll show the `Close()` and `Clobber()` functions to begin with and then fill in the rest in a later section.

To make the application quit when the close box is clicked, we can write `Close()` like this:

```
static pascal void
Close (void)
{
	SkelStopEventLoop ();
}
```

To dispose of the window when it's no longer needed, `Clobber()` looks like this:

```
static pascal void
Clobber (void)
{
WindowPtr	w;

	GetPort (&w);
	DisposeWindow (w);
}
```

This disposal function calls `GetPort()` to find out the current port. That's because TransSkel sets the port to the window to be disposed of before calling a window's disposal function. This allows you to use the same disposal function for multiple windows but still be able to tell which one to dispose of.

Once `Close()` and `Clobber()` are written, you can build and run the application. Note that you can move the window by clicking in the title bar, zoom the window by clicking the zoom box, and resize the window by clicking the size area in its lower right corner, even though you've written no code to perform these manipulations. That's because TransSkel does them for you.

What's Next

More complex applications are an elaboration of the simple themes illustrated by the application we've written so far. Applications create window and menu objects, and tell TransSkel which functions should be called when certain things happen to those objects. The collection of functions attached to an object constitutes an object handler. TransSkel manages the routing of events to the proper handler functions automatically. This means that although event processing is central to Macintosh programming, the application often need not even know such things as event records exist.

The following sections of this tutorial roughly parallel the sections of the Reference Manual. They discuss various aspects of application development in terms of the services TransSkel provides to help you.

Menus

This section describes how you use TransSkel to manage menus. The Apple menu is a special case that we'll consider shortly. For all other menus your application uses, you create the menu and call `SkelMenu()` to register it with TransSkel. This tells TransSkel what to do when the user selects an item from the menu, and how to dispose of the menu when you're done with it.

If you need to get rid of the menu before your application exits (e.g., if it's a temporary menu), call `SkelRmveMenu()`. This function takes the menu out of the menu bar and calls your disposal function. For permanent menus this is unnecessary because when you call `SkelCleanup()` at application termination time, it calls `SkelRmveMenu()` automatically for any registered menus that remain.

Here's an example that creates a File menu with Open, Close, and Quit items and registers it with TransSkel:

```
MenuHandle fileMenu;
Boolean result;

fileMenu = NewMenu ("\pFile");
AppendMenu (fileMenu, "\pOpen/O;Close/W;Quit/Q");
result = SkelMenu (fileMenu, FileMenuSelect, FileMenuDispose, false, true);
```

The arguments to `SkelMenu()` are:

- The menu handle
- The item selection and menu disposal functions
- Whether or not the menu is a submenu in a hierarchical menu
- Whether or not to draw the menu bar

`SkelMenu()` returns `true` if it was able to create the menu handler structure properly, `false` otherwise. In the case of failure, your application should take appropriate action.

When the user selects an item from a menu (either with the mouse or by typing the command-key equivalent), TransSkel calls the menu's selection function and passes it the item number.

If you specify the menu selection function as `nil`, the user can make selections from the menu, but nothing happens. If you write a selection function, it should be declared to take a single argument and return no value. The argument is the number of the item selected. For the example menu above, the selection function looks like this:

```
pascal void
FileMenuSelect (short item)
{
        switch (item)
        {
```

```
        case 1:
               /* process Open */
               break;
        case 2:
               /* process Close */
               break;
        case 3:
               /* process Quit */
               break;
        }
}
```

To remove the menu, pass the menu handle to `SkelRmveMenu()`, which removes the menu from the menu bar and calls your disposal function. Thus, the following call:

```
SkelRmveMenu (fileMenu);
```

removes the File menu from the menu bar and calls the disposal function `FileMenuDispose()`. This should be a function that returns no value and takes a menu handle as a parameter. If the menu is created using `NewMenu()`, as we did above, the following disposal function is suitable:

```
pascal void
FileMenuDispose (MenuHandle m)
{
        DisposeMenu (m);
}
```

For menus created using `GetMenu()`, do this instead:

```
pascal void
FileMenuDispose (MenuHandle m)
{
        ReleaseResource ((Handle) m);
}
```

Note that you never call your disposal function yourself; let `SkelRmveMenu()` do it.

For temporary menus, or menus that you want to dispose of explicitly, you should be sure to write a disposal function. However, it's okay to pass `nil` for the disposal function for menus that are permanent and exist until the application exits.

Handling the Apple Menu

The Apple menu is a special case of menu handling because it's highly stereotyped across applications: zero or more application-specific items, then a gray line, then a set of system items. Under System 7, the system items correspond to the items in the Apple Menu Items folder; otherwise there's one item for each desk accessory installed in the System file..

To tell TransSkel to create and manage an Apple menu, call `SkelApple()`:

```
SkelApple (appItemTitle, DoAppleMenu);
```

When this function is called, TransSkel sets up an Apple menu and adds the application and system items to it. TransSkel manages system item selections automatically. The arguments to `SkelApple()` indicate how to handle application items.

The first argument is a Pascal string to be used as the text of the application item or items. If you have no application items, pass `nil` or an empty string. If there are multiple items, separate them using semicolons:

```
SkelApple (nil, nil);                                   /* no application items */
SkelApple ("\pAbout MyApp…", DoAppleMenu);                  /* one application item */
SkelApple ("\pAbout MyApp…;Help…;Credits…", DoAppleMenu);   /* several items */
```

The second argument is a pointer to the function you want called when an application item is selected. If there are no application items, pass `nil` for the selection function. If there are application items, the function can still be `nil`, but then nothing will happen when the user selects an application item. So normally, if there are application items, you'll write a function for processing them. The form of the selection function is like that of any other menu selection function, i.e., a function that returns no value and takes the item number as a parameter.

If there is only one item, we can ignore the item number in the body of the function. Suppose we want `DoAppleMenu()` to display an alert that presents application information. We can write it like this:

```
static pascal void
DoAppleMenu (short item)
{
       (void) Alert (aboutAlertResNum, nil);
}
```

When there are multiple application items, the selection function must switch on the item number:

```
static pascal void
DoAppleMenu(short item)
{
       switch (item)
       {
       case 1:
              (void) Alert (aboutAlertResNum, nil);
              break;
       case 2:
              /* process Help */
              break;
       case 3:
              /* process Credits */
              break;
       }
}
```

Some problems with simple `Alert()` calls like the ones just shown are that the window underlying an alert isn't properly deactivated unless you take steps to ensure otherwise, and the alert isn't necessarily positioned very well on the screen. These issues are discussed in the section "Modal Dialogs and Alerts."

Document Windows

For any document window your application uses, you should:

- Create the window, e.g., with `NewWindow()`, `NewCWindow()`, `GetWindow()`, or `GetCWindow()`.

- Write handler functions to process mouse clicks, key clicks, update and activate events, clicks in the close box, and a disposal function to get rid of the window when you're done with it. You can also write an "idle time" function to be called periodically when there are no other events for the window.

- Call `SkelWindow()` to register the window and associate it with the handler functions.

- Call `SkelRmveWind()` to destroy the window when you're done with it. Alternatively, you can call `SkelCleanup()` when your application is ready to exit. `SkelCleanup()` calls `SkelRmveWind()` for all registered windows that haven't been destroyed.

The arguments to `SkelWindow()` are:

- A pointer to the window that's to be registered

- The functions to be called to handle mouse clicks in the content area, key clicks, update and activate events, and clicks in the close box

- The window disposal function

- The function to call during idle time and a flag indicating whether the idle-time handler should run only when the window is the application's frontmost window; if there's no idle-time function, the value of the flag is irrelevant.

In the "Getting Started" section, we wrote a simple application that put up a window, but didn't do much with it. We specified only the close box and disposal functions for the window handler:

```
(void) SkelWindow (w,
                   nil,          /* mouse click handler */
                   nil,          /* key click handler */
                   nil,          /* update event handler */
                   nil,          /* activate event handler */
                   Close,        /* close box click handler */
                   Clobber,      /* disposal function */
                   nil,          /* idle-time handler */
                   true);        /* idle only when frontmost */
```

Let’s flesh out window handling a little to show the basic form of all the other functions. We’ll do the following:

- Remember the last character typed and display it in the window

- Display the character at the location of the last mouse click
- Display the size box

All drawing will occur in the window update procedure. We'll cause redrawing to occur when a key or mouse click occurs by invalidating the window port.

First, we'll need a couple of global variables to remember the last character and mouse click location. The initial character value is "no character", but we'll give the location a default that should cause the first key typed to be visible:

```
static char lastChar = '\0';
static Point lastLocation = { 20, 20 };
```

Next, change the call to `SkelWindow()` so it looks like this:

```
(void) SkelWindow (w,
                   Mouse,       /* mouse click handler */
                   Key,         /* key click handler */
                   Update,      /* update event handler */
                   Activate,    /* activate event handler */
                   Close,       /* close box click handler */
                   Clobber,     /* disposal function */
                   Idle,        /* idle-time handler */
                   true);
```

Now we have to write the functions `Mouse()`, `Key()`, `Update()`, `Activate()`, and `Idle()`. Most of these functions will need to access the window pointer. One way of allowing this is to keep track of the window using a global variable. That's okay in a single-window situation, but if you're managing multiple windows that share handler functions, it doesn't work. A more general technique, illustrated below, makes use of the fact that TransSkel's port-setting model is such that when any of the window-handler functions are called, the current port is set to the window that the handler function should operate on. This means we can call `GetPort()` to get the current port.

Mouse handlers are written to receive the current point location, click time, and the event modifiers word. Our handler will remember the click location and invalidate the window port to cause an update:

```
static pascal void
Mouse (Point where, long when, short modifiers)
{
WindowPtr    w;

       lastLocation = where;
       GetPort (&w);
       InvalRect (&w->portRect);
}
```

Key handlers receive the character typed, the character code, and the event modifiers word. Our handler will remember the character and invalidate the window port to cause an update:

```
static pascal void
Key (short c, short code, short modifiers)
{
WindowPtr    w;

       lastChar = c;
```

```
        GetPort (&w);
        InvalRect (&w->portRect);
}
```

Update handlers receive a boolean parameter indicating whether the window was resized or not. This is because updates often need to be handled differently depending on whether or not the window size has been changed. Our handler ignores the parameter. It erases the window since the character or its position may have changed, draws the current character in the current location, and draws the size box.

```
static pascal void
Update (Boolean resized)
{
WindowPtr    w;

        GetPort (&w);
        EraseRect (&w->portRect);
        if (lastChar != '\0')
        {
                MoveTo (lastLocation.h, lastLocation.v);
                DrawChar (lastChar);
        }
        DrawGrowIcon (w);
}
```

Activate handlers receive a boolean parameter which is true if the window is coming active, false if the window is going inactive. Our handler does nothing but draw the size box. This needs to be done because the box is drawn differently to reflect the activation state of the window:

```
static pascal void
Activate (Boolean active)
{
WindowPtr    w;

        GetPort (&w);
        DrawGrowIcon (w);
}
```

Idle-time handlers receive no parameters. Ours does nothing:

```
static pascal void
Idle (void)
{
}
```

Now the application can be compiled and run.

You'll notice that `DrawGrowIcon()` actually draws more than the size box. It also draws horizontal and vertical lines at the bottom and right of the window. This is actually more than we want drawn. The problem can be avoided by drawing the grow box after setting the clip region down. We do this with a function `DrawGrowBox()` and calling that instead of `DrawGrowIcon()` from `Update()` and `Activate()`:

```
static void
DrawGrowBox (WindowPtr w)
{
RgnHandle    oldClip;
Rect         r;
```

```
        r = w->portRect;
        r.left = r.right - 15;
        r.top = r.bottom - 15;
        oldClip = NewRgn ();
        GetClip (oldClip);
        ClipRect (&r);
        DrawGrowIcon (w);
        SetClip (oldClip);
        DisposeRgn (oldClip);
}
```

Modeless and Movable Modal Dialogs

For any modeless dialog or movable modal dialog your application uses, you should:

- Create the dialog window.
- Write handler functions to filter events in the window, process item selections, and clicks in the close box, and a disposal function to get rid of the window when you're done with it.
- Call `SkelDialog()` to register the window and associate it with the handler functions.
- Call `SkelRmveDlog()` to destroy the window when you're done with it. Alternatively, you can let `SkelCleanup()` call `SkelRmveDlog()` for any remaining dialogs when your application is ready to exit.

The arguments to `SkelDialog()` are:

- The dialog to be registered
- The function to call to filter events in the dialog
- The function to call when an item in the dialog is selected
- The function to call when the close box is clicked
- The dialog disposal function

The event filter function gets first crack as events for the dialog, in case you need to intercept certain events and take special action. The function is defined just like filter functions that are passed to `ModalDialog()`, although `ModalDialog()` isn't used to process modeless or movable modal dialogs:

```
static pascal Boolean
MyFilter (DialogPtr dlog, EventRecord *evt, short *item)
{
}
```

TransSkel Programmer's Note 12 describes filter functions in some detail.

The item selection function receives parameters indicating the dialog and the number of the item in the dialog that was selected:

```
static pascal void
DlogEvent (DialogPtr dlog, short item)
{
}
```

The close box and dialog disposal functions are the same as those for document windows. They return no value and take no parameters. The dialog that the function should operate on can be obtained using `GetPort()`. For instance, you might write a dialog disposal function like this:

```
static pascal void
DlogClobber (void)
{
DialogPtr    dlog;

       GetPort (&dlog);
       DisposeDialog (dlog);
}
```

Earlier, we wrote a simple function to display an alert when the user selects the application item from the Apple item:

```
static pascal void
DoAppleMenu (short item)
{
	(void) Alert (aboutAlertResNum, nil);
}
```

Some problems with a simple `Alert()` call like this are that the window underlying an alert isn't properly deactivated unless you take steps to ensure otherwise, and the alert isn't necessarily positioned very well on the screen.

To make sure that deactivates are handled properly, you can pass TransSkel's standard filter function to the `Alert()` routine:

```
static pascal void
DoAppleMenu (short item)
{
	(void) Alert (aboutAlertResNum, SkelDlogFilter (nil, true));
	SkelRmveDlogFilter ();
}
```

`SkelDlogFilter()` returns a pointer to a filter function that is passed to `Alert()` to be called during alert processing. The filter handles activates for underlying windows, and maps the Return and Enter keys to the alert's default button. `SkelRmveDlogFilter()` removes the filter. (TransSkel maintains filters invocations on a stack in case you have nested dialogs and/or alerts, so each call to `SkelDlogFilter()` *must* be matched by a call to `SkelRmveDlogFilter()`.)

If you want to display the alert in a standard position, call `SkelAlert()` instead of `Alert()`:

```
static pascal void
DoAppleMenu (short item)
{
	(void) SkelAlert (aboutAlertResNum, SkelDlogFilter (nil, true),
	                                        skelPositionOnParentDevice);
	SkelRmveDlogFilter ();
}
```

`SkelAlert()` is like `Alert()` except that it takes one additional argument, a selector indicating how to position the alert. The value `skelPositionOnParentDevice` tells `SkelAlert()` to display the alert on the device containing your application's frontmost window, or the main device if your application has no windows. The alert is centered horizontally and positioned so that 1/5 of the vertical border between the alert and the screen boundary lies above the alert and 4/5 below. This conforms to Apple's current human interface guidelines.

You can change the positioning ratios if you like with `SkelSetAlertPosRatios()`. For instance, to set the ratios so that 1/2 of the horizontal border and 1/3 of the vertical border appear to the left and top of the alert, do this:

```
SkelSetAlertPosRatios (FixRatio (1, 2), FixRatio (1, 3));
```

More on Shutting Down

If you expect that you'll need to terminate your application from various locations in the application code as a result of encountering abnormal circumstances or fatal errors, you may want to set up a function that calls `SkelCleanup()` and exits:

```
void
Terminate (void)
{
       SkelCleanup ();
       ExitToShell ();
}
```

Here, `ExitToShell()` is called to force the application to exit. `SkelCleanup()` simply returns to its caller when its done, so without `ExitToShell()`, `Terminate()` would return to the point at which the fatal error occurred!

Button Outlining

It doesn't matter where the user item is positioned because the item will be positioned and sized properly to surround the default button. However, you should observe the following constraints:

- The item must be disabled. Otherwise `ModalDialog()` returns the item number when you click in the item.

- The item should appear later in the item list than any enabled items, and it *must* be later in the item list than any button with which it can be associated. This is because the Dialog Manager apparently determines which item is clicked in by running through the item list until it finds an item with a bounding rectangle containing the click. If the item is disabled, the click is ignored, otherwise the item number is returned. Therefore, if the user item appears in the item list before the button with which it is associated, clicks in the button are interpreted as clicks in the user item and ignored.