

TransSkel

**A Transportable Skeleton for
Macintosh Application Development**

Reference Manual

**Release 3.11
26 February 1994**

Introduction to TransSkel Release 3

This document describes TransSkel 3, a programmer's aid for Macintosh application development under THINK C or THINK Pascal. The name "TransSkel" means "transportable skeleton," reflecting two of its objectives:

- TransSkel is an application skeleton. It is not in itself a finished, working application. Rather, it provides a framework for application development that you flesh out by adding muscle and connective tissue.
- TransSkel is transportable from application to application. It provides a limited set of services in a general manner so that it can be plugged into applications of many different kinds without modification.

TransSkel has a third, pedagogical, objective. It is widely known that acquisition of Macintosh programming skills is a process roughly akin to trial by ordeal. Although the many available volumes on the topic contain a good deal of useful information, access to all this information may still leave one with a dearth of practical knowledge of how to write a Macintosh application that actually *does* something. Coupled with the fact that even trivial applications involve a rather large amount of code, the necessity of understanding *much* of the Macintosh Toolbox before you can use *any* of it leaves the beginning programmer with a daunting task.

TransSkel addresses this problem on two fronts. First, TransSkel handles many tasks for you automatically so you don't have to learn

how to do them right away, if at all. This reduces the programmer's initial burden. Second, TransSkel is provided in source form. It has long been my observation that, while basic documentation is a necessity, it is just as important to have someone else's source code to look at. If you can find some other program that does something similar to what you want to do, you can often learn a great deal about how to write your own program. Thus you can study TransSkel itself to learn basic event-handling techniques, and the standard distribution includes a number of demonstration programs that you can examine to learn something about the interaction between event-handling and the code that handles application-specific tasks.

The claims that can be made for TransSkel are rather modest. It is neither the most general nor the most capable Macintosh application development tool. Nevertheless, it has enjoyed widespread distribution and use, perhaps due principally to three fairly attractive characteristics:

- It's free.
- It's in the public domain. You can use it for any purpose with no restrictions whatsoever. No royalties need be paid. No "About" box attributions are required.
- As already mentioned, TransSkel comes in source form. You can check for yourself how or why it does something the way it does — and if you don't like it, you can change it.

Major Differences Between Releases

TransSkel release 1 provided the basic window and menu registration and event routing services which remain as its core today. This initial release was criticized (justly) for its defective port-setting policy. Release 2 addressed those criticisms. David Berry and Owen Hartnett contributed other changes, so that release 2 also offered support for zoom boxes (which didn't exist when release 1 was written), modeless dialogs, better memory handling and improved menu support. Translations of TransSkel into other languages began to appear at this time, with Owen's port to THINK Pascal (then Lightspeed Pascal) being the most widely distributed.

Two glaring deficiencies in release 2 were failure to provide hierarchical menu or multitasking (MultiFinder) support. A number of people contributed changes to address these shortcomings, to varying degrees of completeness. The most thorough treatment was contributed by Bob Schumaker, who also made many other changes and additions, a number of which appear in release 3.^{1†}

TransSkel release 3 provides, among other things, multitasking support, multiple-monitor awareness, fully prototyped source code, additional improvements to menu handling (including hierarchical menu support), and Apple Event support. TransSkel now also departs from the "everything in one source file" orientation, with its services having been split into a set of core services oriented around tasks that are almost universally needed, and a set of auxiliary services that are often useful but not needed so pervasively.

TransSkel 3 also includes a set of purely convenience routines. These perform functions that may have no direct relationship to event processing, but which are needed commonly enough that it's often useful to have them available in the form of library routines. This has two immediate benefits. First, the application developer doesn't have to write the code to perform these functions and include it over and over in multiple applications. Second, as with all libraries, if improvements are made or bugs eliminated from library functions, any application using them benefits simply by being recompiled. It is unnecessary to track down source in multiple applications and fix each of them.

I classify a routine as core, auxiliary, or convenience as follows:

- Core Routine — Necessary for TransSkel initialization, shutdown, or event dispatching, or required by another core routine.
- Auxiliary Routine — Related to, but not essential for event processing, but likely to be used by many applications in the course of event processing. The TransSkel Apple menu handler is an example of this.
- Convenience Routine — Not justifiably a core or auxiliary routine, but performs a task likely to be common to multiple applications.

Beginning with release 3, a series of TransSkel Programmer's Notes has been instituted in order to provide detailed discussion of particular aspects of TransSkel programming. Think of them as vignettes that don't belong in the manual.

TransSkel and System Compatibility

^{1†} In addition to the TransSkel I maintain, Bob has followed a different development track and maintains his own version, currently numbered at release 2.9. The numbering shouldn't be taken to mean that his version is more primitive than mine. Quite the opposite. He has, for instance, added support for tool (floating) windows.

An effort has been made to ensure that TransSkel code will run on as many Macintoshes and versions of the system software as possible. This includes 64K ROM machines and software not supporting system calls such as `SysEnviron()`, `Gestalt()`, `GetMBarHeight()`, or `GetGrayRgn()`. The intent is that if applications developed with TransSkel run only on certain machines, that will be due to a choice you have made, not to an incompatibility imposed on you by TransSkel.

TransSkel has not been tested on all possible configurations, of course, so if you discover compatibility bugs, please report them—especially if you have a fix.

The following aspects of the code are in special need of being exercised on a wide variety of machines to make sure they work:

- 64K ROM support. Compatibility with 64K ROMs is written into the code, but has not been well tested.
- Window dragging, sizing and zooming on multiple-monitor systems. TransSkel is known to respond properly on single monitors of varying size. If you have a multiple-monitor system, please test the following and let me know the results:
 - You should be able to drag windows to within 4 pixels of the edges of the desktop.
 - You should be able to grow windows to the full size of the desktop.
 - Windows should zoom onto the monitor containing most of the window, not simply onto the monitor containing the menu bar. There should be a 3 pixel gap between all edges of zoomed windows and the desktop boundary (don't include window drop shadows when checking this).
- No demonstration program should crash under any circumstances.

Although TransSkel should be compatible with a wide range of systems, little effort was expended to ensure compatibility of this release of TransSkel with any previous release. The myriad changes between releases 2 and 3 involve a number of incompatibilities. Applications written for release 2 and earlier need modification in order to compile or run properly under release 3. If you have applications that need to be converted from earlier TransSkel releases, read the “Release Notes and Compatibility Issues” document, which is devoted to discussion of the differences between releases 2 and 3, and summarizes the issues involved in porting code to the latter.

A particularly significant “incompatibility release” occurred at release 3.11, when TransSkel was converted to use Pascal-compatible bindings for the public (interface) functions, so that it could be used from within Pascal applications without translating TransSkel itself into Pascal. This results in some incompatibilities for existing TransSkel

applications written in C. See TransSkel Programmer's Note 11 for details about modifying code written for pre-3.11 applications.

If you find TransSkel useful, please feel free to send a note describing what you use it for, or perhaps a copy of applications you build with it. Other comments or suggestions are welcome; criticisms and bug fixes are especially valuable.

Paul DuBois
dubois@primate.wisc.edu

Overview

Skeleton programs are a common vehicle by which to demonstrate techniques involved in programming particular machines, operating systems, software packages, etc. In the Macintosh world, skeleton programs often take the form of simple applications that support a window and a few menus, and handle a number of standard operations such as window dragging and menu item selection. These provide straightforward sample applications and serve as valuable learning tools by illustrating a number of typical Macintosh programming techniques. They help one get up to speed by providing live source code to study. The disadvantage of skeleton applications is that they are not directly reusable. To use one for a different application, it has to be modified and recompiled.

The philosophy underlying TransSkel is different. It's an application skeleton, distinct in both form and purpose from a skeleton application. It's not a complete application, but a standard framework on which to build complete applications. The goal is to provide a limited set of services in a general manner. By concentrating on operations that virtually all applications require and that remain relatively stereotypical across applications, the code implementing them can be abstracted out and decoupled from any particular application. This frees the programmer from having to be much concerned about the services provided, and induces a discipline through which they are accessed.

So what is it that TransSkel provides? Its capabilities have expanded with time, but the central focus remains quite simple: to function as an event-routing engine. TransSkel offers a mechanism for registering window and menu objects, and an event loop which automatically routes events to those objects. There are a number of routines which perform additional tasks, but the basic premise is that you're more interested in responding to events rather than in the mechanics of gathering and dispatching them. This leaves you free to concentrate on writing application-specific code.

This simple approach, together with an emphasis on modularity, results in code which is reusable from application to application. TransSkel is reasonably general, and thus transportable without modification across a wide variety of applications. For instance, it doesn't limit you to some fixed number of windows or menus — it doesn't care how many you have, or whether you create or dispose of them all at once or dynamically.

This document is organized into the following sections:

- Introduction to TransSkel
- Overview of TransSkel
- Distribution Notes — Availability, Layout, Installation, Demonstrations
- Programming interface specification
 - Initialization and Termination

- Querying the Execution Environment
- Event Processing
- Menus
- Windows
- Window Properties
- Multitasking
- Apple Events

- Convenience routines
- Future Changes
- Additional documentation
- Authorship

The version of TransSkel described in this document can be used with Symantec C++/THINK C 6.0 or THINK Pascal 4.0, which are trademarks of:

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
USA

Several other pieces of documentation are referred to here and in the source code. The key to these references is:

HIN	Human Interface Notes
IM	Inside Macintosh
PGMF	Programmer's Guide to MultiFinder
TN	Macintosh Technical Notes
TPN	TransSkel Programmer's Notes

References to the old series and new series of Inside Macintosh can be distinguished by the fact that volumes in the old series are numbered (e.g., IM V) whereas volumes in the new series are named (e.g., IM: Files).

Technical Notes come in two series as well. The original series was numbered (e.g., TN 79 for Technical Note 79). The new series is numbered as well, but the number is also accompanied by a section designator (e.g., TN TE 26, for Technical Note 26 in the TextEdit section). Most references to Technical Notes in TransSkel documentation are to Notes in the new series. There are, however, some Notes in the old series which were not carried forward into the new series.

Distribution Notes

This section discusses how to obtain TransSkel, how the distribution is organized and installed, and describes the demonstration applications.

How to get TransSkel

TransSkel is in the public domain and may be freely redistributed. The software is, however, provided “as is,” with no warranty express or implied.

TransSkel is available for anonymous FTP access on Internet host *ftp.imate.wisc.edu* in the directory */pub/mac/TransSkel*. You can also obtain TransSkel using *gopher* by connecting to *gopher.imate.wisc.edu* and looking under “Primate Center Software Archives”. If you don’t have FTP or gopher access, send an electronic mail request to *software@imate.wisc.edu*.

Distribution Layout

The standard TransSkel distribution includes:

Documentation

- Release Notes

- TransSkel — A Transportable Macintosh Application Skeleton (Reference)

- TransSkel — A Transportable Macintosh Application Skeleton (Tutorial)

- TransSkel Programmer’s Notes

TransSkel library C source code

- TransSkel core routines

- TransSkel auxiliary routines

- TransSkel convenience routines

TransSkel interface files

- C header file

- Pascal interface file

- Precompiled binary library document

Demonstration applications (C and Pascal)

- MiniSkel: minimal application

- Hello: “hello, world” application

- Skel: simulation of traditional Skel

- MultiSkel: multiple-window application — fixed number of windows and menus

- ManyWind: multiple-window application — variable number of menus and windows

DialogSkel: application using modeless dialogs

Filter: demonstrates use of dialog filter for alerts and dialogs

Button: demonstrates button-outlining convenience routines

The TransSkel distribution is organized into the following folders:

Interface — C and Pascal interface files, precompiled library document

- Core — source for core routines
- Auxiliary — source for auxiliary routines
- Convenience — source for convenience routines
- Documents — Documentation (manual and Programmer's Notes)
- Demos — C and Pascal demonstration applications

Installing TransSkel

TransSkel can be used either from THINK C or THINK Pascal.

Interface Installation for C

In order for the TransSkel header files and library to be easily accessible to your projects, you should install them inside the same folder as that in which the THINK Project Manager is located. I do this by creating folders named “Local #includes” and “Local Libraries”. This makes it clear that their contents are not part of Symantec's distribution.

Make a copy of *TransSkel.h* from the TransSkel “Interface” folder and put it in “Local #includes”. To build the library, open *TransSkel.π* in the TransSkel folder and update it. You can install the library two ways, as a copy of updated project document, or as a library. To install it as a copy of the updated project, just make a copy and put it in “Local Libraries” and rename the copy to *TransSkel*. To install the project as a library, copy *TransSkel* from the “Interface” folder into the “Local Libraries” folder. (Or select Project/Build Library after updating *TransSkel.π*, and install the library as *TransSkel* in the “Local Libraries” folder.) You can then add *TransSkel* to your application project documents.

The advantage of installing the project as a library is library documents are smaller than project documents. This reduces the disk space required. The advantage of installing a copy of the project document is that if no functions are referenced for a given file in the project, the object code for that file isn't linked into your application, resulting in smaller applications. If you install the project as a library, all the code is linked in, whether you use it all or not.

TransSkel can be compiled with or without support for modeless dialogs. This is controlled by #define-ing or #undef-ing the symbol `supportDialogs` in *TransSkel.c*. TransSkel may be compiled with dialog support turned off for all of the demonstrations except DialogSkel. You may wish to compile two libraries, one with dialog support turned on, the other with it turned off.

Interface Installation for Pascal

The Pascal interface consists of two files in the “Interface” folder, the interface file *TransSkel.intf* and the library document *TransSkel*. The latter is included in the distribution in precompiled form as a library document, because you cannot use THINK C project documents from Pascal projects. Copy *TransSkel.intf* and *TransSkel* and put them in a location that’s convenient for your projects to be able to access them. (If you wish to recompile the library, you may do so using the project *TransSkel.π* located at the top level of the TransSkel distribution.)

The precompiled TransSkel library document is compiled with modeless dialog support enabled.

The TransSkel Demonstration Applications

The Demos folder contains a C Demos and Pascal Demos folder. These two folders are similar, each containing one folder per demonstration application, in the appropriate language.

The demonstrations are examples of the kinds of applications for which TransSkel can be used. The programs differ in emphasis and complexity. To build any C demonstration application, open its project document and update it. Then select Project/Run. To build any Pascal demonstration application, open its project document and build it. (You probably need to tell THINK Pascal where to find the interface files the first time you build a project.) Then select Run/Go.

Any of the demonstration applications may be terminated by selecting Quit from the File menu or by typing command-Q.

MiniSkel

This demonstration puts up an Apple menu with desk accessories in it, and a File menu with a Quit item. Desk accessories may be run as usual. The source illustrates in minimal fashion the general way in which a host uses TransSkel.

Hello

This demonstration puts up a single window that says “hello, world.” If the application is suspended, the message changes to “goodbye, world.” Demonstrates simple window handling.

Skel

This demonstration mimics the traditional Skel program: one sizable, draggable, non-closable dark gray window, Apple and File menus, two dialog boxes, support for desk accessories.

MultiSkel

This demonstration is a simple multiple-window application. There are two text windows (one editable, the other not), and two graphics windows (one manipulable, the other not). MultiSkel shows how to use TransSkel to support some of the basic Macintosh programming techniques: graphics, scroll bar use, region manipulation, shift-click and double-click detection, text editing and use of the clipboard.

All four windows may be resized, dragged, closed and opened. If any window is resized, its display adjusts itself to the window’s new shape. Use the close box in a window to close it, or select Close from the File menu. To make hidden windows visible, select Open from the File menu.

Help Window

This window describes MultiSkel via a scrolling text display.

Edit Window

This window allows text editing. The Edit menu is functional for this window, except Undo. The clipboard is supported. Scrolling is not.

Zoom Window

This window presents a series of randomly selected rectangles, each smoothly interpolated into the next. The display pauses if the mouse button is held down in the window.

Region Window

If the mouse is clicked and dragged in this window, a gray selection rectangle is drawn. Rectangles drawn this way are combined into a region, the outline of which is drawn as a

“marching ants” marquee. Rectangles drawn with the shift key down are subtracted from the region. Clear the window by double-clicking.

ManyWind

This application demonstrates dynamic object handler creation and disposal. Initially there are two menus and no windows. The Apple menu operates as usual. The File menu allows new windows to be created with the New item. Up to 20 windows may exist at once; after that, the New item is dimmed until some window is destroyed. Whenever any windows are present, a Windows menu and a Color menu are also present. The Windows menu contains one item for each visible window. Selecting an item brings the corresponding window to the front. The Color menu may be used to change the background color of the frontmost window. Clicking the close box of a window destroys it and removes it from the Windows menu. If all the windows are closed, the Windows and Color menus are destroyed as well until another window is created.

DialogSkel

This application supports two modeless dialogs, each of which is used to influence the appearance of the other, and a standard document window.

The dialogs disappear when the application is suspended, and reappear when it's resumed.

Filter

This application demonstrates the use of `SkelDialogFilter()` to show how event loss during modal dialog processing can be avoided with dialog filters. See also TPN 2.

Button

This application demonstrates how to use the button-outlining convenience routines in various situations. It accompanies the discussion in TPN 10.

HierMenu

This application demonstrates how to use `TransSkel` to set up hierarchical menus, using both `NewMenu()` and `GetMenu()`.

The TransSkel Programming Interface — General Information

TransSkel installation directions are given in the “Distribution Notes” section. I assume here that you’ve already followed those instructions.

Using TransSkel from C

Source Files

Host applications interact with TransSkel entirely through function calls; there are no global variables. C source files that use TransSkel functions should include the following line:

```
#include "TransSkel.h"
```

TransSkel.h contains the constants, types, and functions needed for referring to TransSkel routines and the services they provide.

Project Document

Include the *TransSkel* library in your project document. It can be either a project document or a library document.

THINK Project Manager

You should turn on “prototypes required” under Options/THINK C so that you'll be sure to catch all function name changes and argument list changes to TransSkel routines.

Also, turn off “Separate STRs” under Project/Set Project Type or your application will crash on machines running early versions of the system software (e.g., 4.1).

The 'SIZE' Resource

Several aspects of the way your application behaves in a multitasking environment are controlled by the application’s 'SIZE' resource. You can set these directly from THINK C under the dialog presented by Project/Set Project Type.

Using TransSkel from Pascal

The Pascal interface consists of the interface file *TransSkel.intf* and the library document *TransSkel*. Both are located in the Interface folder. You should include both files in your project document, and source files that use TransSkel routines, types, or constants must list *TransSkel* in the `uses` statement:

```
uses  
    TransSkel;
```

The Rest of this Manual

The manual sections following this one describe the programming interface provided by TransSkel. The interface is described using C notation, so if you want to use TransSkel from Pascal, consult the file *TransSkel.intf*, which provides the appropriate translation for the C syntax used here. (The calling sequences for callback routines are given in comments in the `type` section.)

Each of the following sections discusses a particular aspect of the interface:

Initialization and Termination

This section describe the functions used to initialize TransSkel when your application begins and how to shut it down when your application is ready to exit.

Querying the Execution Environment

This section describes routines that provide access to information about the system or the application's state.

Event Processing

This section discusses the routines that provide your application's event loop.

Menus

This section discusses how to write menu handlers and how to arrange for TransSkel to call them automatically when menu selections are made.

Windows

This section discusses how to write window handlers and how to arrange for TransSkel to call them automatically when window events occur.

Window Properties

This section discusses window properties — a mechanism allowing you to associate arbitrary pieces of information with your application's windows.

Multitasking

This section discusses how to use TransSkel's multitasking support.

Apple Events

This section discusses how to use TransSkel's high-level event support.

Convenience

This section discusses convenience routines provided by TransSkel.

Initialization and Termination

TransSkel must be initialized to work properly. You do this by calling `SkelInit()` near the beginning of your application. In addition, if you wish to have TransSkel clean up menus, windows, etc., when your application is ready to exit, you call `SkelCleanup()`. Thus a typical application looks like this:

```
int
main (void)
{
    SkelInit (void);
    /* set up menus, windows, etc. */
    /* run event loop */
    SkelCleanup ();
}
```

An argument of `nil` to `SkelInit()` means “use the default initialization parameters”. If you want to use values different than the defaults, declare a `SkelInitParams` structure, call `SkelGetInitParams()` to fill it with the default values, change those parameters you wish to be different, and pass the structure to `SkelInit()`. For example, if you want `SkelInit()` to call `MoreMasters()` 840 times instead of the default 6 times, do this:

```
SkelInitParams initParams;

SkelGetInitParams (&initParams);
initParams.skelMoreMasters = 840;
SkelInit (&initParams);
```

The `SkelInitParams` structure looks like this:

```
typedef struct SkelInitParams SkelInitParams, *SkelInitParamsPtr;

struct SkelInitParams
{
    short                skelMoreMasters;
    GrowZoneProcPtr     skelGzProc;
    ResumeProcPtr       skelResumeProc;
    Size                skelStackAdjust;
};
```

- `skelMoreMasters` is the number of times to call `MoreMasters()`. The default is 6 times.
- `skelGzProc` is either a pointer to a user-written `GrowZone()` routine or `nil` if no `GrowZone()` function should be installed. The default is `nil`.
- `skelResumeProc` is a pointer to a resume function passed to `InitDialogs()`, or `nil` if no function should be installed. The default is `nil`.
- `skelStackAdjust` is the amount by which to adjust the application stack size. Positive values increase the stack size, negative values decrease it. The default is no adjustment.

The reason you call `SkelGetInitParams()` is that the `SkelInitParams` structure may be extended in the future. If you try to initialize such a structure statically, your program may be incompatible with future versions of TransSkel. By calling `SkelGetInitParams()` to get the defaults and modifying the parameters you want to change, your application will be compatible with new versions of TransSkel simply by recompiling. This is discussed further in TPN 5.

Interface Specification

```
pascal void
SkelInit (SkelInitParamsPtr initParams);
```

The host application should call `SkelInit()` to initialize the various Macintosh managers and some internal variables. `initParams` is a pointer to a `SkelInitParams` structure containing the initialization values you want to use, or `nil` if you want to use the defaults.

`SkelInit()` should be called before any other TransSkel routines, with the exception of `SkelGetInitParams()`.

```
pascal void
SkelGetInitParams (SkelInitParamsPtr initParams);
```

This function fills in the initialization parameters structure you pass it with the default initialization values. Use this function if you plan on modifying some of the defaults when you call `SkelInit()`.

```
pascal void
SkelCleanup (void);
```

`SkelCleanup()` destroys all registered objects (windows, dialogs and menus). It's generally called before the application exits, after the main event loop has terminated.

Before removing window handlers, `SkelCleanup()` hides all windows and closes desk accessories. Desk accessories are closed because on some early systems (e.g., System 4.1), if you leave a DA open when the application exits, the system crashes the next time you open that DA. Windows are hidden from back to front. This is more esthetic (less obtrusive), and it's also quicker because the system doesn't have to do as much painting of underlying windows.

Querying the Execution Environment

Applications often need to know something about the capabilities of the system on which they're executing (e.g., are Apple Events available?), or about their own state (e.g., am I running in the foreground?). This section describes routines that provide access to this kind of information.

Interface Description

```
pascal long  
SkelQuery (short selector);
```

`SkelQuery()` may be thought of as a `TransSkel-specific Gestalt()` call. It provides a mechanism for the host application to query `TransSkel` about certain aspects of the execution environment. The argument is a query selector indicating the information desired. Interpretation of the return value depends on the selector, as follows:

Selector	Return Value
<code>skelQVersion</code>	<code>TransSkel</code> version number. The major version number is contained in byte 1 (bits 8-15) and the minor version number is contained in byte 0 (bits 0-7). For example, 0x00000304 means 3.04.
<code>skelQSysVersion</code>	System software version. The major version number is contained in byte 1 (bits 8-15) and the minor and subminor version numbers are contained in bits 4-7 and 0-3 of byte 0. For example, 0x00000701 means 7.0.1.
<code>skelQHasWNE</code>	Non-zero if <code>WaitNextEvent()</code> is implemented, zero otherwise.
<code>skelQMBarHeight</code>	Menu bar height, in pixels. This works on any Macintosh, in contrast to the global <code>MBarHeight</code> or the function <code>GetGrayRgn()</code> , thus no machine-specific test is needed.
<code>skelQHas64KROM</code>	Non-zero if the system has the 64K ROM. This can be useful to the application if it uses certain features which are known not to exist on in the 64K ROM.
<code>skelQHasColorQD</code> <code>skelQQDVersion</code>	Non-zero if the system has Color QuickDraw, zero otherwise. QuickDraw version number. The major version number is contained in byte 1 (bits 8-15) and the minor version number is contained in byte 0 (bits 0-7). For example, 0x00000230 means 2.30 (2.3).
<code>skelQInForeground</code>	Non-zero if the application is in the foreground, zero otherwise.
<code>skelQHasGestalt</code>	Non-zero if the system has the <code>Gestalt()</code> call, zero otherwise.
<code>skelQHasAppleEvents</code> <code>skelQGrayRgn</code>	Non-zero if Apple Events are available, zero otherwise. Handle to copy of desktop region, zero if handle could not be allocated. Return value should be cast to <code>RgnHandle</code> type.

Region must be disposed of by caller. Works even on systems where `GetGrayRgn()` is not implemented.

The return value is meaningless if the selector is not one of those defined above.

Note

`skelQHasWNE` was known as `skelQWNEImplemented` in some early release 3 versions.

The set of selectors provided may seem somewhat arbitrary. It happens to contain selectors for those types of information I've found useful for implementing TransSkel itself.

```
pascal Boolean  
SkelTrapAvailable (short trap);
```

This function returns `true` if the given trap is available, `false` otherwise. You can use it to determine whether particular Toolbox routines are available, for example.

Event Processing

TransSkel provides a simple means of initiating and terminating a basic event loop: you call `SkelEventLoop()` to process events and `SkelStopEventLoop()` to stop processing events.

Typically you'll call `SkelEventLoop()` from your `main()` function and `SkelStopEventLoop()` from within a menu or window handler function that's called by TransSkel. Here's a simple `main()` function:

```
int
main (void)
{
    SkelInit (nil);
    /* set up menus, windows, etc. here */
    SkelEventLoop ();
    SkelCleanup ();
}
```

If you have a File menu with a Quit item in it, and you've registered a handler function for the menu with TransSkel, you can terminate the event loop when Quit is selected like this:

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

        /* handler other items here */

        case quitApp:
            SkelStopEventLoop();
            break;

    }
}
```

Normally, TransSkel applications call `SkelEventLoop()` to initiate the main event loop that drives the application. You can use `SkelDoEvents()` and `SkelDoUpdates()` to run a "mini-event loop" outside of the main event loop. Each routine runs as long as any events of the desired type(s) are pending. While such events are pending, they are retrieved from the event queue and dispatched.

`SkelDoEvents()` takes an event mask parameter specifying which events you're interested in. `SkelDoUpdates()` processes update events only.

These routines are intended to make it easier to dispose of any pending events of a given type before resuming the main event loop. For instance, you might put up a standard file dialog to solicit an input file. When the user dismisses the dialog by clicking Open, your application might read in the selected file and present its contents in a document window. However, when the dialog disappears, parts of any windows over which the dialog came up will be erased. Unless you take steps to ensure otherwise, update events for these windows will be pending while you read in the

file. To avoid leaving the user staring at a damaged display, you can call `SkelDoUpdates()` to repair window contents after the dialog is dismissed and before reading the file.

TransSkel also provides functions allowing you to process a particular event record, to control what kind of events are requested by TransSkel, to install a hook allowing you to inspect events before TransSkel processes them, and to set up an “idle” function that’s called when no events are pending (i.e., when a null event occurs).

Interface Description

Event Routing Routines

```
pascal void  
SkelEventLoop (void);
```

The host calls this function to run an event loop, typically after calling menu and window registration functions. `SkelEventLoop()` solicits events from the system and passes them to `SkelRouteEvent()` until `SkelStopEventLoop()` is called. When there are no events pending, `SkelEventLoop()` also polls window handler idle functions as appropriate.

```
pascal void  
SkelStopEventLoop (void);
```

`SkelStopEventLoop()` terminates the current event loop, causing the last call to `SkelEventLoop()` to return. Generally, host applications call this function when they are ready to exit, for example in the code that processes the Quit item from the File menu. The application may then call `SkelCleanup()` and exit.

```
pascal void  
SkelRouteEvent (EventRecord *event);
```

This function processes a single event, either by handling it itself, or by routing it to a handler function, such as a window, menu, or OS event handler.

If the host wishes to inspect events before TransSkel processes them, a hook function should be installed with `SkelSetEventHook()`.

`SkelRouteEvent()` performs the following actions itself, without routing them to host-installed handlers:

- Window dragging, if the window can be dragged.
- Window sizing, if the window has a grow box or zoom box. The host program detects when the window has been resized according to the convention described

in the discussion of the `SkelWindow()` function for the `fUpdate` parameter.

- If the mouse is clicked in the content region of an inactive window, that window is brought to the front. Normally, window-activating clicks are not passed to mouse-click handler functions. Exception: If the `getFrontClicks` flag is set in the application's 'SIZE' resource, mouse clicks in the content area of a suspended application's window *are* passed to the application .

- Command-key equivalents for menu items are processed.
- Disk-insert events for uninitialized disks are handled by allowing the user to eject or initialize the disk. You don't end up with a dead floppy drive due to such an event having been thrown away.
- System item selections from the Apple menu (e.g., desk accessories) are processed, if `SkelApple()` has been called.

Window sizing and zooming behavior can be modified by the application.

```
pascal void
SkelDoEvents (short mask);
```

Runs the TransSkel event loop to process any events of the types specified in the mask parameter. Returns when no events of the requested types are available.

Example: to process any outstanding activates and updates, do this:

```
        SkelDoEvents (activMask + updateMask);

pascal void
SkelDoUpdates (void);
```

This function runs the TransSkel event loop to process any outstanding update events.

Event Inspection Hook Routines

```
pascal void
SkelSetEventHook (SkelEventHookProcPtr p);
```

`SkelSetEventHook()` installs a function to be called each time an event (including null events) is routed by `SkelRouteEvent()`. A pointer to the event record is passed to the function, which can inspect the event and take whatever action it deems necessary. The function should be declared as follows:

```
pascal Boolean
MyEventHook (EventRecord *event)
{
}
```

If your hook returns `true`, `SkelRouteEvent()` assumes the handler processed the event and ignores it, otherwise TransSkel handles the event as it would otherwise. To turn event inspection off, pass a value of `nil` to `SkelSetRouteEvent()`. There is no event inspection hook initially.

Note

Not every event that occurs passes through the hook. For example, many desk accessory events are processed by the system, mouse up events following clicks in the menu bar and window drag regions are eaten by the Toolbox, etc. Notwithstanding this caveat, the

events seen by the hook are generally the only ones the host really cares about, anyway.

```
pascal SkelEventHookProcPtr
```

```
SkelGetEventHook (void);
```

`SkelGetEventHook()` returns the current event-inspecting hook, or `nil` if there isn't one.

Idle Time Routines

```
pascal void  
SkelSetIdle (SkelIdleProcPtr p);
```

`SkelSetIdle()` installs an idle function to be run when no events are available. To turn the idle function off, pass a value of `nil`. There is no idle function initially.

If you supply an idle function, it should be declared as follows:

```
pascal void  
MyEventHook (void)  
{  
}  
  
pascal SkelIdleProcPtr  
SkelGetIdle (void);
```

`SkelGetIdle()` returns the current idle function, or `nil` if there isn't one.

Event Mask Routines

```
pascal void  
SkelSetEventMask (short mask);
```

`SkelSetEventMask()` is used to specify the types of events requested for processing by `SkelEventLoop()`. The mask is constructed by or-ing together the event masks used to specify individual event types. The default is the same as the system event mask default, i.e., `everyEvent - keyUpMask`.

`SkelSetEventMask()` does not modify the system event mask; it simply specifies which events `TransSkel` will ask the system for.

IM II-70 and TN 202 both warn against changing the system event mask, in particular, against restricting the set of events which can be posted. `SkelSetEventMask()` does not change the system event mask, it only changes the set of events requested by `TransSkel` on behalf of your application.

Normally `SkelSetEventMask()` is called by an application to restrict the set of events requested, for instance, to process only pending update events. It is legitimate to expand the set of events which can be posted, to allow key-up events (which are by default ignored). However, if this is done, the application must also change the system

event mask and should be careful to restore that mask before terminating. See TransSkel Programmer's Note 3 for more details.

```
pascal short  
SkelGetEventMask (void);
```

`SkelGetEventMask()` returns the current TransSkel event mask.

```
pascal void
SkelSetDlogMask (short mask);
```

`SkelSetDlogMask()` is used to specify the types of events that should be allowed to pass through to modeless dialogs. The mask is constructed by or-ing together the event masks used to specify individual event types. The actual mask used within `TransSkel` will always have bit 0 turned on, even if not turned on in the mask parameter. This forces null events to be passed to modeless dialogs; otherwise the caret in `TextEdit` items would not blink.

`SkelSetDlogMask()` does not modify the system event mask; it simply specifies which events, out of those specified by the regular `TransSkel` event mask, that will be passed to dialogs if a dialog is in front. The default is: activate, update, key, and mouse down events.

Use of a secondary selector mask for dialogs, in addition to the regular `TransSkel` mask, makes it easy to keep non-dialog events from being gobbled up incorrectly. The mechanism recommended in IM I (pp. 278-279) for handling dialog events uses `IsDialogEvent()` and `DialogSelect()`. If the event “is an activate or update event for a dialog window, a mouse down event in the content region of an active dialog window, or *any other type of event when a dialog window is active*,” then `IsDialogEvent()` returns `true` (emphasis added). Note especially the last part of that quote. One of the implications of it is that it is necessary to prevent some events from going into the dialog handling code or they will be lost, e.g., disk inserts, `MultiFinder` events.

```
pascal short
SkelGetDlogMask (void);
```

`SkelGetDlogMask()` returns the current `TransSkel` dialog event mask.

Dialog Event Filter Routines

```
pascal ModalFilterProcPtr
SkelDlogFilter (ModalFilterProcPtr filter, Boolean
doReturn);
```

This routine provides a standard dialog/alert event filter function. The first argument is the dialog-specific filter function you would normally pass to `ModalDialog()` or to an alert call. The standard function takes care of routing updates and activates that aren't for the dialog or alert through the `TransSkel` event loop so they get passed to

your application normally. It also routes OS events. In addition, the standard filter can map Return or Enter key events onto clicks in the default button , and Escape or Command-period key events onto clicks in the cancel button. If the standard filter doesn't handle the event, it passes it to function you passed as the `filter` argument, if that argument isn't `nil`. If non-`nil`, filter should point to a function defined like this:

```
pascal Boolean
FilterFunction (DialogPtr dlog, EventRecord *evt, Integer *item)
{
}
```

The `doReturn` argument is true if you want the standard filter to treat the item specified in the dialog record as the default button and map Return and Enter onto it. If you want to specify a different item as the default, or to specify a cancel button, use `SkelDlogDefaultItem()` and `SkelDlogCancelItem()`.

`SkelDlogFilter()` maintains a stack internally and may be called in nested fashion if you are processing multiple dialogs or alerts — although you should try to avoid such situations as they can be confusing to the user. You must call `SkelRmveDlogFilter()` to pop the stack after each call to `SkelDlogFilter()`.

Example:

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

See TransSkel Programmer's Notes 2 and 8 for further information about dialog event filters.

```
pascal ModalFilterYDProcPtr
SkelDlogFilterYD (ModalFilterYDProcPtr filter, Boolean doReturn);
```

This routine is like `SkelDlogFilter()`, but it's used for dialog event filter functions like those used by the System 7 Standard File dialogs `CustomGetFile()` and `CustomPutFile()`, i.e., filter functions that are defined like this:

```
pascal Boolean
FilterFunctionYD (DialogPtr dlog, EventRecord *evt, Integer *item, void *data)
{
}
```

Currently, you should always pass false for the `doReturn` argument, since the Standard File dialogs handle key mapping for themselves.

You must call `SkelRmveDlogFilter()` to remove the filter installed by `SkelDlogFilterYD()`.

See TransSkel Programmer's Note 8 for further information.

```
pascal void
SkelRmveDlogFilter (void);
```

Removes the dialog filter installed by `SkelDlogFilter()` or `SkelDlogFilterYD()`. You must call this function once for each call to those two functions.

See TransSkel Programmer's Notes 2 and 8 for further information.

```
pascal void
SkelDlogDefaultItem (short item);
```

Designates the given item as the item that should be considered the default button by the standard dialog event filter function. The filter function maps Return and Enter onto clicks in this button. If `item` is zero, Return/Enter mapping is turned off. If `item` is -1, the filter maps Return and Enter onto the item specified as the default in the dialog record.

See TransSkel Programmer's Note 8 for further information.

```
pascal void
SkelDlogCancelItem (short item);
```

Designates the given item as the item that should be considered the cancel button by the standard dialog event filter function. The filter function maps Escape and Command-period onto clicks in this button. If `item` is zero, Escape/Command-period mapping is turned off.

See TransSkel Programmer's Note 8 for further information.

Miscellaneous Routines

```
pascal EventRecord *
SkelGetCurrentEvent (void);
```

This function returns a pointer to the last event processed by `SkelRouteEvent()`. Note that this will be `nil` if no events have been processed yet.

```
pascal short
SkelGetModifiers (void);
```

This function returns the modifiers word from the last event processed by `SkelRouteEvent()`.

```
pascal Boolean
SkelCmdPeriod (EventRecord *evt);
```

Returns `true` the given event contains a key-down event representing command-period (i.e., “cancel the current operation”), `false` otherwise. The test is done in an internationally-compatible way.

This function is used by the standard dialog event filter function, but it can also be useful in other contexts, such as during print operations.

Related Routines

`SkelClose()` and `SkelActivate()` under “Window Management”.

`SkelSetWaitTimes()` and `SkelGetWaitTimes()` under “Multitasking Support”.

Menu Management

pascal Boolean

```
SkelMenu (MenuHandle menu,  
          SkelMenuSelectProcPtr doSelect,  
          SkelMenuClobberProcPtr doClobber,  
          Boolean subMenu,  
          Boolean drawBar);
```

`SkelMenu()` registers a menu with `TransSkel`. This causes a handler to be created for the menu so that `TransSkel` knows how to route selections to it. The host should already have created menu via `GetMenu()` or `NewMenu()`. `SkelMenu()` allocates memory for the handler and returns `true` if it succeeded. If `SkelMenu()` returns `false`, it failed.

`SkelMenu()` installs menus at the end of the menu bar; the host should not call `InsertMenu()`.

menu

A handle to the menu to be registered.

doSelect

A pointer to the function to call when items are selected from the menu. The function should be defined to take one parameter, the number of the item that was selected:

```
pascal void  
MySelect (short item)  
{  
}
```

If `doSelect` is `nil`, the menu is installed but selecting items from it has no effect.

doClobber

A pointer to the function to call to take care of disposing of the menu. The function should be defined to take one parameter, a handle to the menu to dispose of:

```
pascal void  
MyClobber (MenuHandle m)  
{  
}
```

`doClobber` can be `nil` for any menu that persists throughout program execution. Handlers for temporary menus should include a disposal function.

Note

The host should never call the `doClobber` function itself. Let `SkelCleanup()` or `SkelRmveMenu()` do it.

If the host calls `SkelMenu()` to install a handler for a menu for which a handler already exists, the previous handler is removed (without disposing of the menu itself). This is functionally equivalent to modification of existing handlers.

`subMenu`

This parameter should be `true` if the menu is a submenu in a hierarchical menu, `false` otherwise.

`drawBar`

This parameter should be `true` if the menu bar should be drawn after the menu is installed. Typically the host installs all menus but the last with a `drawBar` value of `false`, to avoid menu bar flicker due to unnecessary redrawing. Alternatively, you can pass `false` as you install all your menus, then call `DrawMenuBar()` afterward.

```
pascal void
SkelRmveMenu (MenuHandle menu);
```

`SkelRmveMenu()` removes the menu handler for the given menu. It removes the menu from the menu bar (which is redrawn) and calls the handler's disposal function, if one was specified when the handler was created.

`SkelRmveMenu()` deletes the menu from the menu bar; the host should not call `DeleteMenu()`. `SkelRmveMenu()` redraws the menu bar; the host does not need to.

It is permissible for menu handler selection functions to call `SkelRmveMenu()`. Handlers can remove themselves this way. Menu handlers can also be removed by window handler functions, for instance, if a menu is only present when a particular window is active.

```
pascal void
SkelApple (StringPtr items, SkelMenuSelectProcPtr doSelect);
```

The Apple menu and the way it is processed are generally highly stereotypical: an application-specific item that says something like "About MyApp...", then a gray line, then system entries for the items in the Apple Menu Items folder under System 7 (or desk accessories otherwise). Selecting the application item results in a display giving some information about the program. Selecting a system item causes it to be opened.

If your application has an Apple menu that fits this description, call `SkelApple()` to install one. The `items` argument supplies the title of the application item as a Pascal string (e.g., "\pAbout MyApp...") and `doSelect` is the function to execute when that item is selected.

`items` can be `nil` or the empty string, or, if non-empty, can contain multiple items separated by semicolons. For instance, if you want to specify an "About" item and a "Help" item, `items` might look like this:

```
"About MyApp...;Help"
```

`doSelect` is declared the same way as the selection function passed to `SkelMenu()`. Normally the item number passed to the Apple menu selection function will be 1, but not necessarily, since you can specify multiple item titles in `items`.

If `items` is `nil` or the empty string, only system items appear in the menu. If `items` is non-empty but `doSelect` is `nil`, the “About” item appears in the Apple menu but selecting it has no effect.

`SkelApple()` creates its own menu, so you do not need to supply one. It uses a menu ID of `skelAppleMenuID` (128), so the host should not use that ID for any of its other menus. If the Apple menu is to be handled in a non-standard way, use `SkelMenu()` to install your own Apple menu.

`SkelApple()` calls `SkelMenu()` to install the Apple menu but returns no value. It is assumed that `SkelApple()` will be called so early in the application that it is virtually certain to succeed, and that if it doesn’t, there is no hope for the application anyway.

`SkelApple()` does not draw the menu bar; if your only menu is the Apple menu, then you’ll need to call `DrawMenuBar()` yourself to make the menu show up.

```
pascal void
SkelSetMenuHook (SkelMenuHookProcPtr p);
```

This routine registers a function to be called when a menu selection is about to be made (i.e., when there is a mouse-down event in the menu bar or a command-key equivalent was typed). This gives your application a chance to adjust menu items to disable or enable menu items according to your application’s state. For example, if you have a Close item in your File menu, you might want to disable it when no windows are visible.

The menu hook function takes no arguments and returns no value:

```
pascal void
MyMenuHook (void)
{
}
```

Pass `nil` to turn off menu selection notification.

```
SkelMenuHookProcPtr
SkelGetMenuHook (void);
```

Returns the current menu notification function, or `nil` if there isn’t one.

Window Management

pascal Boolean

```
SkelWindow (WindowPtr wind,  
            SkelWindMouseProcPtr doMouse,  
            SkelWindKeyProcPtr doKey,  
            SkelWindUpdateProcPtr doUpdate,  
            SkelWindActivateProcPtr doActivate,  
            SkelWindCloseProcPtr doClose,  
            SkelWindClobberProcPtr doClobber,  
            SkelWindIdleProcPtr doIdle,  
            Boolean idleFrontOnly);
```

`SkelWindow()` registers a window with TransSkel. This causes a handler to be created for the window so that TransSkel knows how to route events to it. The window's port is also made the current port. `SkelWindow()` allocates memory for the handler and returns `true` if it succeeded. If `SkelWindow()` returns `false`, it failed, and the current port remains unchanged. The application should already have created the window via `NewWindow()` or `GetNewWindow()`. The rest of the parameters are values that are installed into TransSkel's handler structure for the window. Most of the parameters are addresses of functions that are called to handle events in the window (`doMouse`, `doKey`, `doUpdate`, `doActivate`, `doClose`) or program execution conditions (`doClobber`, `doIdle`). If the handler doesn't need a particular function, pass `nil` for the corresponding parameter. For example, if the handler doesn't process key clicks, pass `nil` as the value of `doKey`.

If a handler function is associated with the handlers for multiple windows, the function can call `GetPort()` to determine which window it was called for.

If the application calls `SkelWindow()` to install a handler for a window for which a handler already exists, the previous handler is removed (without disposing of the window itself). This is functionally equivalent to modification of existing handlers.

The window's initial property list is empty, unless `SkelWindow()` is used to register an already-registered window. In that case, any existing properties attached to the window are transferred to the new handler.

Note

Since `SkelWindow()` sets the port to the window being registered, problems can ensue if you don't show the window when you're done initializing it. That is, you can end up with the port no longer set to the front window. In this case, you should save the port before and restore it after calling `SkelWindow()`.

`wind`

A pointer to the window to be registered.

`doMouse`

A pointer to the function to execute when `wind` is the active window and the mouse is clicked in its content region. The function should be declared to take three parameters:

```
pascal void
MyMouse (Point where, long when, short modifiers)
{
}
```

The location of the mouse click is passed in `where`, in coordinates local to the window. The time of the click is passed in `when`, and the modifier flags word of the mouse click event record is passed in `modifiers`. The time of the click can be used for double-click detection, while the modifiers can be used to detect shift-click, option-click, etc.

`doKey`

A pointer to the function to execute to handle key clicks when `wind` is the active window. The function should be declared to take three parameters:

```
pascal void
MyKey (short c, short code, short modifiers)
{
}
```

The character is passed in `c`, the raw character code in `code`, and the modifier flags word of the key event record is passed in `modifiers`. Note that the character is a `short`; this is consistent with the way the THINK C headers declare `char` arguments in Toolbox function prototypes, so you may pass `c` directly to the Toolbox.

Key clicks are routed to the `doKey` function of the active window and are thrown away if no windows are visible. If this is a problem, install an event-inspecting hook with `SkelSetEventHook()`.

`doUpdate`

A pointer to the function that draws the content region of the window. `TransSkel` brackets calls to this function by calls to `BeginUpdate()` and `EndUpdate()`, so that drawing shows on the screen only in the region that actually needs updating (see IM:Toolbox Essentials 4-448-450). The update function should be declared to take one parameter:

```
pascal void
MyUpdate (Boolean resized)
{
}
```

The `resized` parameter indicates whether or not the window has been resized. The convention for such notification is this: if the mouse is clicked in the grow box or zoom box of the active window, `TransSkel` resizes it automatically and invalidates the entire window port to cause an update event to be generated. When that event occurs, `TransSkel` passes a value of `true` to the update function. The application may take whatever action is appropriate to respond to a resizing. Since the entire port is invalidated, drawing shows up in the whole window.

`doActivate`

A pointer to the function to execute when the window is activated or deactivated. The function should be declared to take one parameter:

```
pascal void
MyActivate (Boolean active)
{
}
```

The parameter has a value of `true` if the window is coming active, `false` if it is going inactive. The application program should draw the grow box if the window has one and take whatever other steps are appropriate, such as enabling or disabling of controls, and highlighting text selections properly.

You can cause a window's `doActivate` function to be invoked from anywhere in your application by passing the window pointer to `SkelActivate()`. This is useful in suspend/resume functions, for example.

`doClose`

A pointer to the function to execute when the mouse is clicked in the window's close box. If the window has a close box but the handler has a `doClose` function of `nil`, the window is simply hidden (not disposed of). In that case, the application should probably provide a method for reopening the window (such as a menu selection).

If a window should be disposed of when it's closed, the `doClose` function can call `SkelRmveWind()`. This will cause the window to be disposed of (via the `doClobber` function) and its handler removed.

The `doClose` function should take no parameters:

```
pascal void
MyWindClose (void)
{
}
```

You can cause a window's `doClose` function to be invoked from anywhere in your application by passing the window pointer to `SkelClose()`. This is useful in menu handlers, e.g., for the Close item of a File menu.

`doClobber`

A pointer to the function to execute to dispose of the window and any associated data structures that may have been created at the same time as the window (e.g., controls, TextEdit records). This is the function in which `CloseWindow()` or `DisposeWindow()` should be called. The function should take no parameters:

```
pascal void
MyWindClobber (void)
{
}
```

Your application should never call the `doClobber` function itself. Let `SkelRmveWind()` or `SkelCleanup()` do it.

`doIdle`

A pointer to the function to execute when there are no events pending. Essentially, this is the “no-event” handler. For example, `doIdle` functions for TextEdit window handlers can call `TEIdle()` to blink the caret. Window handlers that change the cursor shape depending on the current mouse location can do so here.

`doIdle` differs from the general application idle function that’s specified with `SkelSetIdle()` in that the window idle function is attached to a particular window.

The `doIdle` function should take no parameters:

```
pascal void
MyWindIdle (void)
{
}

idleFrontOnly
```

This argument determines whether or not the window idle function is called only when the window is active (frontmost). If `doIdle` is `nil`, the value of `idleFrontOnly` is irrelevant, of course.

`doIdle` functions that should execute only when the window is visible must themselves check whether that is so. However, window idle functions are never called when the application is suspended.

```
pascal void
SkelRmveWind (WindowPtr wind);
```

`SkelRmveWind()` removes the window handler for `wind`. It calls the handler’s `doClobber` function, if one was specified when the handler was created.

It is dangerous for `doIdle` functions to call `SkelRmveWind()` to remove handlers for windows other than the one for which they are called. Otherwise, it is permissible for any window handler function except `doClobber` to call `SkelRmveWind()`. Handlers can remove themselves this way. (`doClobber` is called by `SkelRmveWind()`; it does not make sense for the latter to be called by the former.)

Window handlers can also be removed by menu handler functions.

```
pascal Boolean
SkelDialog (DialogPtr dlog,
            SkelWindEventProcPtr doEvent,
            SkelWindCloseProcPtr doClose,
            SkelWindClobberProcPtr doClobber);
```

`SkelDialog()` registers a dialog with `TransSkel`. This causes a handler to be created for the dialog so that `TransSkel` knows how to route events to it. The dialog port is also made the current port. `SkelDialog()` allocates memory for the handler and returns `true` if it succeeded. If `SkelDialog()` returns `false`, it failed, and the current port remains unchanged. The application should already have created the dialog via `NewDialog()` or `GetNewDialog()`. The other parameters are addresses of functions that are called to handle

events in the dialog (`doEvent`, `doClose`) or program execution conditions (`doClobber`). If the handler doesn't need a particular function, pass `nil` for the corresponding parameter.

`SkelDialog()` allocates memory for the handler, attaches a `skelWPropModeless` property to it, and returns `true` if it succeeded. If `SkelDialog()` returns `false`, it failed, and the current port remains unchanged.

If the application calls `SkelDialog()` to install a handler for a dialog for which a handler already exists, the previous handler is removed (without disposing of the dialog itself). This is functionally equivalent to modification of existing handlers.

`dlog`

A pointer to the dialog to be registered.

`doEvent`

A pointer to the function to execute to handle events in the dialog. This will be called after the usual `IsDialogEvent()/DialogSelect()` sequence is performed. The function should be declared to take two parameters.

```
pascal void
MyEvent (short item, EventRecord *event)
```

A pointer to the event to be handled is passed in `event`, and `item` is the item to which the event applies.

`IsDialogEvent()` considers any event at all to be a dialog event when a modeless dialog is active. Events such as disk-inserts, however, are not usually related to the dialog. Because of this, `TransSkel` only processes update, activate, key, mouse and null events for dialogs. The set of events that should be passed to dialogs may be changed with `SkelDlogMask()`.

`doClose`

A pointer to the function to execute when the mouse is clicked in the dialog's close box. If the dialog has a close box but the handler has a `doClose` function of `nil`, the dialog is simply hidden (not disposed of). In that case, the application should probably provide a method for reopening the dialog, e.g., a menu item.

`doClose` is useful for writing handlers for temporary dialogs that are thrown away when the dialog is closed. The `doClose` function can call `SkelRmveDlog()` to cause the dialog to be disposed of and its handler to be removed.

The `doClose` function is defined the same way as for `SkelWindow()`.

`doClobber`

A pointer to the function to execute to dispose of `dlog` and any associated data structures that may have been created at the same time as the dialog. The function is defined the same way as for `SkelWindow()`.

Note:

The application should never call the `doClobber` function itself. Let `SkelRmveDlog()` or `SkelCleanup()` do it.

```
pascal void
SkelRmveDlog (DialogPtr dlog);
```

`SkelRmveDlog()` removes the dialog handler for `dlog`. It calls the handler's `doClobber` function, if one was specified when the handler was created.

`SkelRmveDlog()` can be used to dynamically alter the set of available dialogs.

```
pascal void
SkelClose (WindowPtr wind);
```

If `wind` is `nil`, `SkelClose()` does nothing. If the window is a Desk Accessory window, `SkelClose()` closes it properly. Otherwise `SkelClose()` calls the window's close procedure. This is useful for situations when you may not know a window's close procedure. For example, you can call it to process a Close item in your File menu:

```
SkelClose (FrontWindow ());
```

```
pascal void
SkelActivate (WindowPtr wind, Boolean active);
```

`SkelActivate()` calls the given window's activate procedure. `active` indicates whether the window should be activated or deactivated. This is useful in suspend/resume handlers when you may not know a window's activate procedure:

```
SkelActivate (wind, inForeground);
```

```
pascal void
SkelSetGrowBounds (WindowPtr wind,
                   short hLo, short vLo,
                   short hHi, short vHi);
```

`SkelSetGrowBounds()` tells the handler for `wind` what the lower and upper limits on window sizing should be. The lower limits horizontally and vertically are given by `hLo` and `vLo`. The upper limits are given by `hHi` and `vHi`.

Grow limits for a window are initialized by `SkelWindow()` when it creates the window's handler. To reset these general defaults (as opposed to the defaults for a particular window), pass `nil` as the value of `wind`. The initial defaults are 80 pixels in either direction for the lower limits. The default upper limits are the dimensions of the desktop region bounding rectangle, and are therefore machine dependent.

```
pascal void
SkelSetZoom (WindowPtr wind, SkelWindZoomProcPtr doZoom);
```

`SkelSetZoom()` installs a function to be used to zoom the window when its zoom box is clicked by the user. If `wind` is `nil`, the function becomes the default window zooming function that is used for all windows subsequently made known to `TransSkel` with `SkelWindow()`. If `wind` is non-`nil`, the function is used only for the given window. If

`doZoom` is non-`nil`, it is used as the function to be called for zooming. If `doZoom` is `nil`, the initial default function (the “default default”) is used.

By default, `TransSkel` will zoom a window to the full size of the screen (or the screen containing most of the window, on multiple-screen systems), inset by 3 pixels on all sides and the height of the menu bar on the top if the window is on the screen containing the menu bar. Applications may wish to modify this behavior, particularly for large-screen monitors. For instance, text editors need only zoom windows to the width of a page, which may be less than the full width of the screen.

```
pascal SkelWindZoomProcPtr
SkelGetZoom (WindowPtr wind);
```

This function returns the current function used to zoom the given window, or `nil` if the window is zoomed using `TransSkel`’s default zoom function.

```
pascal void
SkelGetWindContentRect (WindowPtr wind, Rect *rp);
```

Returns the window’s content rectangle in the second argument, in global coordinates.

```
pascal void
SkelGetWindStructureRect (WindowPtr wind, Rect *rp);
```

Returns the window’s structure rectangle in the second argument, in global coordinates.

```
pascal short
SkelGetWindTitleHeight (WindowPtr wind);
```

This function returns the height of the window’s title bar by calculating the difference between the tops of the structure and content regions. It works whether or not the window is visible (the structure region is invalid for invisible regions) and in a script-system independent way.

Note

`SkelGetWindTitleHeight()` is intended to be used for rectangular window with the title bar on the top. It will not work for windows that have a title bar on the side, and it may not work for strangely-shaped windows.

```
pascal Boolean
SkelGetWindowDevice (WindowPtr wind, GDHandle *gd, Rect *devRect);
```

`SkelGetWindowDevice()` determines which active screen device contains the largest area of the content region of the window and returns the device and the usable area on that device. The return value is `true` if the window is actually overlaps some device, `false` otherwise. You can pass `nil` for the device or device rectangle parameters if you’re not interested in the device.

If the return value is `false` (the window overlaps no devices), non-`nil` arguments are filled in with the main device, the main device rectangle, and `true`, respectively

On return, `gd` will be `nil` if Color QuickDraw is not supported, since no graphics devices will be present.

`SkelGetWindowDevice()` doesn't check whether the visible flag is set in the window record, because it can be useful for determining where to position not-yet-shown windows — if the return value is false, you know the window will not be visible on any device even after a `ShowWindow()`, and that it should be positioned somewhere within the returned rectangle.

Applications that save document positions should use `SkelGetWindowDevice()` when a document is opened to determine whether the window will be invisible if positioned where it was last saved, and move it somewhere within the returned rectangle if the function result is false.

If you use this routine for positioning windows before you show them, remember to take into account the height of the title bar. This can be determined by calling `SkelGetWindTitleHeight()`.

`SkelGetWindowDevice()` calls `SkelGetRectDevice()`, which see.

```
pascal Boolean  
SkelWindowRegistered (WindowPtr wind);
```

Returns true if the window is registered with TransSkel, false otherwise.

The Port-Setting Model

TransSkel uses three principles in deciding when to change the current port. These are described below.

Follow the active window. The main port-setting principle is that TransSkel sets the port to a window when the window becomes active (frontmost). Thus the current port follows window activation events. This is congruent with the way users perceive themselves to be directing their attention to a given window by clicking in it to bring it to the front, if it isn't already frontmost.

This often means the programmer need do no `SetPort()` calls at all, since many events are directed toward the active window. It also means that many events naturally go to the correct port without scattering a lot of `SetPort()` calls throughout the TransSkel code itself. For instance, key clicks are directed by TransSkel to the frontmost window, thus to the current port. Mouse clicks in the close box, the grow region or the zoom box of a window can only occur when that window is frontmost, so the port will already be set correctly. Mouse clicks in the content region of a window are passed to the window's click handler only if the window is active.

Make local port changes for non-active windows. A second port-setting principle is that port changes not resulting from a window coming active are local changes only. That is, the port is set temporarily to handle some event, and then restored after the event has been processed. For instance, update events can occur for any window. Thus, TransSkel saves the port, sets the port to the window needing updating, calls the update handler, and restores the port. Clobber and idle procedures may also be executed for any window and the port is saved, set and restored similarly.

One implication of this port-setting model is that your window handler procedures can expect to find the port set to the window for which they're being called. This means your

application may be able to share handler procedures among windows that have the similar characteristics. By doing a `GetPort()`, those procedures can determine which window is to be affected. For example, if multiple windows share a clobber function, that function can determine the proper window to dispose of like this:

```

pascal void
MyClobber (void)
{
    WindowPtr    w;

    GetPort (&w);
    /* dispose of window here */
}

```

Some anomalous situations are handled specially and deserve comment.

Zooming and closing can both be initiated under application control, thus for windows which aren't necessarily active. Thus the port is set before performing the operation and restored afterward.

A local port change is made for deactivate events. This is necessary for the case that the window coming active is a modal dialog. Modal dialogs are not handled by TransSkel, so setting the port to the deactivated window and leaving it there would leave the port set to the wrong window during dialog processing.

Set the port when a window is registered. The current port is also set by `SkelWindow()` when a window is registered. I concede that setting the port to any newly-created window does not strictly follow from either the user's perception of port-setting, or from the local-changes-only convention. However, in my experience, typically when a window is created and its handler installed, other initialization is performed on the window (setting font size or face, installing controls, etc.) and it is convenient to find the port already set. In any case, should the application wish to override this behavior, the port can be saved before and restored after the call to `SkelWindow()`. For instance, the model will fail under the following circumstances:

- Window A is created and activated (port becomes A)
- Window B is created but is not visible, i.e., it will be used later (port becomes B)
- User draws into A, the active window. The port, however, is set to B.

Under conditions such as these, the creation of B and its initialization should be bracketed by a pair of calls to `GetPort()` and `SetPort()`. I felt that these circumstances are sufficiently infrequent that the burden on the developer is less when the port is set by `SkelWindow()` than when it is not.

Treatment of modeless dialogs is similar; when a dialog becomes active, the port is set to it.

Window Properties

Windows may have property lists. A window property consists of an short property type, a long property data value, and a handle to the next property in the list (nil if none).

```
typedef struct SkelWindProperty SkelWindProperty, **SkelWindPropHandle;
struct SkelWindProperty
{
    short          skelWPropType;
    long           skelWPropData;
    SkelWindPropHandle skelWPropNext;
};
```

You can store whatever you like in the `skelWPropData` field, for instance, a handle to an auxiliary data structure, or a pointer to a function.

The current property types are:

```
skelWPropAll           /* pseudo-property */
skelWPropModeless      /* modeless dialog */
skelWPropModal         /* modal dialog */
skelWPropTool          /* tool window */
skelWPropMovableModal  /* movable modal dialog */
skelWPropHelp          /* help window */
skelWPropText          /* text window */
skelWPropDisplayWind   /* TransDisplay window */
skelWPropEditWind      /* TransEdit window */
skelWPropApplBase      /* first general-use property number */
```

Property types less than 256 (a constant available as `skelWPropApplBase`) are reserved for use by TransSkel and those window packages which are now or in the future become part of the TransSkel “canon” — e.g., TransDisplay, TransEdit. `skelWPropAll` is a pseudo-property used as indicated under the descriptions of the `SkelGetWindProp()` and `SkelRmveWindProp()` functions.

Properties from `skelWPropApplBase` and up are available for use on an application-defined basis.

When a window is registered with TransSkel using `SkelWindow()`, it has no properties by default. `SkelDlogWindow()` assigns the `skelWPropModeless` property.

You can assign properties with `SkelAddWindProp()` and remove them with `SkelRmveWindProp()`. Properties may be retrieved with `SkelGetWindProp()`. By attaching distinct properties to different types of windows, an application can use this function to easily determine whether a window is a given type or not.

TransSkel allows you to call `SkelWindow()` for a window that has already been passed to `SkelWindow()`. In such a case, the current window handler is destroyed, and replaced with a new one. However, if the window has any existing properties, they are not destroyed, but are transferred to the new handler.

Why properties?

TransSkel manages windows by associating each one with a structure that describes various things about the window: pointer to window structure, etc.

Window-handling attributes that apply to all or most windows are valid candidates for inclusion in the handler structure. However, for attributes that distinctly apply only to a given type of window, it is unwarranted to put the machinery for them directly in the structure.

If applications assigned no more than one property to a given window, the property mechanism could be implemented as a static slot in the TransSkel window handler structure. However, since multiple properties might be useful in unforeseen contexts, and to forestall the need for changes to the property mechanism due to unanticipated requirements, the implementation uses a list of dynamically-allocated structures which can be added and deleted.

This allows an arbitrary number of properties and eliminates the need for each application to invent its own method of identifying windows. The window property structure carries window type identification information, and points to additional data that can be used by the machinery that manipulates that window type.

Properties allow different types of windows to be easily identified without building information necessary to identify and manipulate each type directly into the handler structure. Furthermore, the mechanism to process information associated with the property remains in the application, where it belongs; there is no need to build idiosyncratic window-handling mechanisms into TransSkel.

Interface Specification

```
pascal SkelWindPropHandle  
SkelGetWindProp (WindowPtr w, short propType);
```

Returns a handle to the property structure for the given window and property type. Returns `nil` if the window is unregistered or does not have the given property.

Passing the pseudo-property `skelWPropAll` as the property type causes a handle to the first property in the property list to be returned, which can be useful when you want to scan through the list yourself. This will be `nil` if the window has no properties.

Here's how to walk the property list for a window w:

```
SkelWindPropHandle  wh;

wh = SkelGetWindProp (w, skelWPropAll);
while (wh != (SkelWindPropHandle) nil) /* walk list until end reached */
{
    /* ... look at (**wh).skelWPropType... */
    wh = (**wh).skelWPropNext;
}

pascal Boolean
SkelAddWindProp (WindowPtr w, short propType, long propData);
```

`SkelAddWindProp()` adds a property to a window and returns `true` if it succeeded. It returns `false` if the window is not registered with `TransSkel`, if memory could not be allocated for the new property structure, if the window already has a property of the given type, or if the pseudo-type `skelWPropAll` is passed as the property type.

Since `SkelAddWindProp()` fails if the window already has a property of the given type, you should remove any instance of a given property before creating a new one.

```
pascal void
SkelRmveWindProp (WindowPtr w, short propType);
```

Removes the property of the given type for the given window. Note that since `TransSkel` has no way of interpreting what you store in the `skelWPropData` field, if you use it to store a pointer or handle to other data structures, you are responsible for disposing of those structures yourself. Use `SkelGetWindProp()` to get a handle to the property structure before you remove the property. For example:

```
wh = SkelGetWindProp(w, propType);
MyDisposePropData ((*wh).skelWPropData);
SkelRmveWindProp (w, propType);
```

Passing `skelWPropAll` as the property type removes all properties. Again, if you need to dispose of auxiliary structures, do so for all structures before removing any of them. However, if you try walking the property list while simultaneously removing properties from the list, you'll encounter problems:

Incorrect:

```
wh = SkelGetWindProp (w, skelWPropAll);
while (wh != (SkelWindPropHandle) nil)
{
    MyDisposePropData ((*wh).skelWPropData);
    SkelRmveWindProp (w, ((*wh).skelWPropType));
    wh = ((*wh).skelWPropNext);
}
```

Correct:

```
wh = SkelGetWindProp (w, skelWPropAll);
while (wh != (SkelWindPropHandle) nil)
{
    MyDisposePropData ((*wh).skelWPropData);
    wh = ((*wh).skelWPropNext);
}
SkelRmveWindProp (w, skelWPropAll);
```

A window's property list is disposed of automatically by `SkelRmveWind()` after calling the window's clobber function. You may therefore wish to dispose of any auxiliary data structures pointed to by the `skelWPropData` field in the clobber function.

Multitasking Support

System software versions that allow multiple applications to be open at once manage those applications by moving them between foreground and background. The system uses OS events to communicate to applications that they're being suspended or resumed and that the contents of the Clipboard (the system scrap) need to be converted to or from the application's private scrap.

TransSkel provides support for multitasking by allowing your application to be notified when suspend/resume events occur, and when the clipboard needs to be converted between the private and system scrap. There are also routines for setting and getting the foreground and background event wait times used by `WaitNextEvent()`, and there is a `SkelQuery()` selector your application can use to find out whether it is currently in the foreground or background.

There is presently no support for receiving mouse-moved OS events. This may be provided in the future. There is also no support for child-died information delivered through OS events, which is unlikely to change. This was used primarily only for debuggers, and the bits that you test in the event message seem now not even to be documented. Under System 7, you can receive child-died events using Apple Events.

To receive OS events:

- You must set the proper flags in your application's 'SIZE' resource to inform the system that your application wants to know about OS events.
- You must register handler functions for OS events with TransSkel, so it knows how to route those events into your application code.

'SIZE' resource -1 controls whether or not the system tells your application about OS events. For TransSkel, the relevant flags are `acceptSuspendResumeEvents`, `doesActivateOnFGSwitch`, and `getFrontClicks`. These flags correspond to the Suspend & Resume Events, MultiFinder-Aware, and Get FrontClicks items that you can set with the SIZE Flags popup menu from Set Project Type... under the Project menu in THINK C.

- `acceptSuspendResumeEvents`

If set, this flag specifies that your application wishes to receive OS events (that is, `osEvt` events, formerly known as `app4Evt` events). The system still suspends and resumes your application if the flag isn't set, it simply doesn't tell you when that happens.

- `doesActivateOnFGSwitch`

If set, this flag specifies that your application will handle deactivating and activating of the front window itself when the application is suspended and resumed. If this flag isn't set, the system fools your application into doing the right thing by creating a false window. That's easier for your application but it takes longer. The result for the user is more sluggish response.

If the `doesActivateOnFGSwitch` flag is set, you also set the `acceptSuspendResumeEvents` flag, or your application won't know when to do the activates..

- `getFrontClicks`

Normally, a click in the content area of a window belonging to a suspended application brings the application to the front but is otherwise thrown away. Setting the `getFrontClicks` flag specifies that your application wishes to receive such content-area mouse clicks.

To receive suspend and resume events, you should set the appropriate flag(s) in the 'SIZE' resource and install a handler for `TransSkel` to call when your application is suspended or resumed. The function should look like this:

```
pascal void
DoSuspendResume (Boolean inForeground)
{
    /* respond to event here */
}
```

`inForeground` is `false` if your application is being suspended, `true` if it's being resumed. Register your suspend/resume handler with `TransSkel` like this:

```
SkelSetSuspendResume (DoSuspendResume);
```

Pass `nil` to disable suspend/resume notification.

`SkelGetSuspendResume()` returns the current suspend/resume handler. The return value is `nil` if there isn't one.

If your application supports a private scrap, you can arrange to be notified when the application needs to convert its contents to or from the Clipboard. This allows interapplication communication using the Clipboard.

To manage Clipboard conversion, write a handler function:

```
pascal void
DoClipCvt (Boolean inForeground)
{
    if (inForeground)
        /* convert system scrap to private scrap */
    else
        /* convert private scrap to system scrap */
}
```

`inForeground` is `false` if your application is being suspended, which means you should unload the application's private scrap to the system scrap. Otherwise you should convert the contents of the system scrap to your application's private scrap.

Register your Clipboard conversion handler with `TransSkel` like this:

```
SkelSetClipCvt (DoClipCvt);
```

Pass `nil` to disable Clipboard notification.

`SkelGetClipCvt()` returns the current Clipboard conversion handler. The return value is `nil` if there isn't one.

Note

When an application is brought to the foreground, it only needs to convert the system scrap if that was changed while the application was suspended. Thus, it's not necessarily true that your Clipboard conversion handler will be called on every resume.

If your application needs to know whether it's running in the foreground or background, it can do that by maintaining a global variable that you set in your suspend/resume handler. However, you can get the same information by asking `TransSkel`:

```
inForeground = SkelQuery (skelQInForeground);
```

Interface Specification

```
typedef pascal void (*SkelSuspendResumeProcPtr) (Boolean
inForeground);
```

typedef for the kind of function pointer you pass to or receive from `SkelSetSuspendResume()` and `SkelGetSuspendResume()`, respectively. The function argument indicates whether or not the application is becoming the foreground application (being resumed). The function returns no value.

```
typedef pascal void (*SkelClipCvtProcPtr) (Boolean
inForeground);
```

typedef for the kind of function pointer you pass to or receive from `SkelSetClipCvt()` and `SkelGetClipCvt()`, respectively. The function argument indicates whether or not the application is becoming the foreground application (being resumed). This indicates the direction in which the conversion should take place. The function returns no value.

```
pascal void
SkelSetSuspendResume (SkelSuspendResumeProcPtr p);
```

Register a function to be called when a suspend/resume event occurs. Pass `nil` to disable notification.

```
pascal SkelSuspendResumeProcPtr
SkelGetSuspendResume (void);
```

Returns the current suspend/resume handler, or `nil` if there isn't one.

```
pascal void
SkelSetClipCvt (SkelClipCvtProcPtr p);
```

Register a function to be called when the Clipboard needs converting. Pass `nil` to

disable notification.

```
pascal SkelClipCvtProcPtr  
SkelGetClipCvt (void);
```

Returns the current Clipboard conversion handler, or `nil` if there isn't one.


```
pascal void  
SkelSetWaitTimes (long fgTime, long bgTime);
```

```
pascal void  
SkelGetWaitTimes (long *pFgTime, long *pBgTime);
```

These functions are used to set or get the foreground and background event wait times used by `WaitNextEvent()`. The values are specified in ticks (60ths/second). The defaults are 6 and 300 ticks (.1 second, 5 seconds).

Either argument to `SkelGetWaitTimes()` may be `nil` if you're not interested in the corresponding time.

Related Routines

`SkelQuery()`, re: the `skelQInForeground` selector.

Apple Event Support

TransSkel provides some support for Apple Events (high-level events), although the interface is pretty rudimentary and may change in the future. Presently, to receive high-level events:

- You must set the proper flags in your application's 'SIZE' resource to inform the system that your application wants to know about high-level events.
- You must register handler functions for high-level events with TransSkel, so it knows how to send those events to your application. You will probably also need to register handler functions with the system.

'SIZE' resource -1 controls whether or not the system tells your application about high-level events. For TransSkel, the relevant flags are `isHighLevelEventAware` and `localAndRemoteHLEvents`. These flags correspond to the HighLevelEvent-Aware and Accept Remote HighLevelEvents items that you can set with the SIZE Flags popup menu from Set Project Type... under the Project menu in THINK C.

- `isHighLevelEventAware`

This flag must be set to specify that your application can receive high-level events (that is, `kHighLevelEvent` events). Actually, it means your application can send high-level events, too, but TransSkel has nothing to do with sending them.

- `localAndRemoteHLEvents`

If set, this flag specifies that your application can send and receive high-level events across a network.

Assuming the 'SIZE' resource is set properly, your application must do several things to receive and process Apple Events.

You need to write handler functions for the events in which you're interested. For instance, to handle the required core Apple Events (Open Application, Quit Application, Open Documents, and Print Documents), you can write handlers for them like this:

```
pascal OSErr
MyAEOpenApp (AppleEvent *theAEvt, AppleEvent *replyEvt, long refCon)
{
    /* handle Open Application event here */
}
pascal OSErr
MyAEQuitApp (AppleEvent *theAEvt, AppleEvent *replyEvt, long refCon)
{
    /* handle Quit Application event here */
}
```

```
}  
pascal OSErr  
MyAEOpenDoc (AppleEvent *theAEvt, AppleEvent *replyEvt, long refCon)  
{  
    /* handle Open Documents event here */  
}
```

```

}
pascal OSErr
MyAEPrintDoc (AppleEvent *theAEvt, AppleEvent *replyEvt, long refCon)
{
    /* handle Print Documents event here */
}

```

Note that these handlers must be pascal functions or your application will crash.

Your handlers must be registered using `AEInstallEventHandler()`. However, first you should test whether Apple Events are available:

```

if (SkelQuery (skelQHasAppleEvents))
    /* Apple Events are available */
else
    /* Apple Events are not available */

```

If Apple Events are available, you tell the system about your handlers:

```

if (AEInstallEventHandler (kCoreEventClass, kAEOpenApplication,
                          MyAEOpenApp, 0L, false) != noErr)
    /* handle error */
if (AEInstallEventHandler (kCoreEventClass, kAEQuitApplication,
                          MyAEQuitApp, 0L, false) != noErr)
    /* handle error */
if (AEInstallEventHandler (kCoreEventClass, kAEOpenDocuments,
                          MyAEOpenDoc, 0L, false) != noErr)
    /* handle error */
if (AEInstallEventHandler (kCoreEventClass, kAEPrintDocuments,
                          MyAEPrintDoc, 0L, false) != noErr)
    /* handle error */

```

Now you need to write a function that TransSkel can call to notify your application about high-level events. The function should be defined like this:

```

pascal void
MyAEProc (EventRecord *theEvent)
{
    /* event-handling stuff here */
}

```

And you register it with TransSkel like this:

```

SkelSetAEHandler (MyAEProc);

```

What you do inside `MyAEProc()` depends on how you intend to process Apple Events. If you respond only to Apple Events for which you've registered handlers by calling `AEInstallEventHandler()`, `MyAEProc()` doesn't need to do anything other than pass the events along to `AEProcessAppleEvent()`, which will convert them to Apple Events and route them to whichever handler is appropriate for the event type:

```

pascal void
MyAEProc (EventRecord *theEvent)
{
    AEProcessAppleEvent (theEvent);
}

```

If your application responds to other high-level events, then `MyAEPProc()` may need to do additional processing.

Interface Specification

```
typedef pascal void (*SkelAEHandlerProcPtr) (EventRecord
*);
```

This is a typedef for the kind of function you write as a TransSkel Apple Event handler. It takes a pointer to an `EventRecord` and returns no value.

```
pascal void
SkelSetAEHandler (SkelAEHandlerProcPtr p);
```

`SkelSetAEHandler()` registers a function for TransSkel to call when a high-level event occurs. Pass `nil` to turn off high-level event notification.

```
pascal SkelAEHandlerProcPtr
SkelGetAEHandler (void);
```

`SkelGetAEHandler()` returns the current high-level event handler. It returns `nil` if there isn't one.

Convenience Routines

This section describes the TransSkel convenience routines.

Note

`SkelGetRectDevice()` is actually a core routine.

Control Manipulation

pascal Boolean

```
SkelHiliteControl (ControlHandle ctrl, short hilite);
```

This routine sets the control hilite value. It differs from the `HiliteControl()` Toolbox routine in that it does nothing if the hilite value is already set to the value passed. `SkelHiliteControl()` returns true if the hilite value was set, false if not. The return value can be used to avoid unnecessary drawing. For instance, if you draw a heavy outline around a button, you should redraw it appropriately (black or gray) when the button hilite value changes, but you don't need to if the value's already set properly.

pascal void

```
SkelDrawButtonOutline (ControlHandle ctrl);
```

Draw a heavy outline around the control (which should be a push-button). If the button is inactive, the outline is drawn in gray. True gray is used if it's available and the window uses a color GrafPort, otherwise dithered gray is used.

pascal void

```
SkelEraseButtonOutline (ControlHandle ctrl);
```

Assuming there's a heavy outline around the control (which should be a push-button) this function erases it by drawing the outline in white.

pascal void

```
SkelFlashButton (ControlHandle ctrl);
```

Simulates a click in the control (which should be a push-button) by inverting it momentarily.

pascal short

```
SkelToggleCtlValue (ControlHandle ctrl);
```

Toggle the value of the control and return the resulting value. This is most useful for

checkbox items.

Dialog Item Management

This section describes TransSkel routines for dealing with dialog items. These routines assume that you call them appropriately. For example, don't call `SkelGetDialogCtl()` to get a control handle

for an item that's not actually associated with a control; don't call `SkelGetDlogStr()` to get the text of a dialog item unless it's a static text or edit text item.

```
pascal ControlHandle  
SkelGetDlogCtl (DialogPtr d, short item);
```

Return a handle to the control associated with the dialog item.

```
pascal Boolean  
SkelSetDlogCtlHilite (DialogPtr d, short item, short  
hilite);
```

Set the hiliting value of the control associated with the dialog item, and return true if the value was changed, false if the hiliting value was already set to the given value.

```
pascal short  
SkelGetDlogCtlHilite (DialogPtr d, short item);
```

Return the hiliting value of the control associated with the dialog item.

```
pascal void  
SkelSetDlogCtlValue (DialogPtr d, short item, short value);
```

Set the value of the control associated with the dialog item.

```
pascal short  
SkelGetDlogCtlValue (DialogPtr d, short item);
```

Return the value of the control associated with the dialog item.

```
pascal short  
SkelToggleDlogCtlValue (DialogPtr d, short item);
```

Toggle the value of the control associated with the dialog item and return the resulting value. This is most useful for checkbox items.

```
pascal void  
SkelSetDlogCtlRefCon (DialogPtr d, short item, long value);
```

Set the reference constant of the control associated with the dialog item.


```
pascal long  
SkelGetDlogCtlRefCon (DialogPtr d, short item);
```

Return the reference constant of the control associated with the dialog item.

```
pascal void  
SkelSetDlogStr (DialogPtr d, short item, StringPtr str);
```

Set the text of the dialog item using the string pointed to by str.

```
pascal void  
SkelGetDlogStr (DialogPtr d, short item, StringPtr str);
```

Get the text of the dialog item into the string pointed to by `str`.

```
pascal void
SkelSetDlogRect (DialogPtr d, short item, Rect *r);
```

Set the bounding rectangle of the dialog item using the rectangle pointed to by `r`.

```
pascal void
SkelGetDlogRect (DialogPtr d, short item, Rect *r);
```

Get the bounding rectangle of the dialog item into the rectangle pointed to by `r`.

```
pascal void
SkelSetDlogProc (DialogPtr d, short item,
SkelDlogItemProcPtr p);
```

Set the procedure associated with the dialog item.

```
pascal SkelDlogItemProcPtr
SkelGetDlogProc (DialogPtr d, short item);
```

Get the procedure associated with the dialog item.

```
pascal void
SkelSetDlogType (DialogPtr d, short item, short type);
```

Set the dialog item's type.

```
pascal short
SkelGetDlogType (DialogPtr d, short item);
```

Get the dialog item's type.

```
pascal void
SkelSetDlogRadioButtonSet (DialogPtr dlog,
                           short first, short last, short
choice);
```

For dialog items `first` through `last`, set the value of item `choice` to 1 and the others to 0. The items should be consecutive radio buttons.

```
pascal void
```

```
SkelSetDlogButtonOutliner (DialogPtr d, short item);
```

`item` must be a user item. This routine associates a drawing procedure with the item that draws a heavy outline around the dialog's default button. It does this by moving and sizing the item's bounding rectangle to surround the default button's bounding rectangle. To force the outline to be redrawn (e.g., if you change the button's highlighting state), you can use `SkelGetDlogRect()` to get item's bounding rectangle and pass it to `InvalRect()`.

Since this function moves and sizes the user item bounding rectangle so it's appropriate for the button, the initial size and location of the rectangle doesn't matter. However, the user item should be disabled. Also, because the user item needs to be set up, you should create the dialog invisible, call `SkelSetDlogButtonInstaller()` and then show the dialog with `ShowWindow()`.

If you create the dialog with `GetNewDialog()` and it is possible for the default button to become inactive, create a 'dctb' resource for the dialog. This will allow the outline to be drawn in true gray on systems and monitors that support color or grayscale. Otherwise dithered gray will always be used.

You will also get dithered gray if you create a dialog with `NewDialog()`. To get true gray when possible, test whether `Color QuickDraw` is available, and use `NewCDialog()` instead if it is. You can do this as follows:

```
if (SkelQuery (skelQHasColorQD))
    dlog = NewCDialog ( ... arguments ... );
else
    dlog = NewDialog ( ... arguments ... );
```

Alert Presentation

```
pascal short
SkelAlert (short alrtResNum, ModalFilterProcPtr filter,
           short positionType);
```

Present the alert stored in the 'ALRT' resource `alrtResNum`. `positionType` specifies how to position the alert on the screen. The legal values for this parameter are the same as discussed under `SkelGetReferenceRect()`. `filter` is the alert event filter. It's analagous to the filter function passed to the Toolbox routine `Alert()`.

If the positioning type is `skelPositionOnParentWindow`, the alert can end up positioned partly off the desktop if the parent window has been moved partly off the desktop. If this happens, the alert is positioned on the parent device instead of the parent window.

```
pascal void
SkelSetAlertPosRatios (Fixed hRatio, Fixed vRatio);
```

Set the screen positioning ratios that `SkelAlert()` uses to position alerts. The default ratios are:

horizontal	<code>FixRatio (1, 2)</code>	center horizontally
vertical	<code>FixRatio (1, 5)</code>	1/5 of border above alert, 4/5 below

```
pascal void
SkelGetAlertPosRatios (Fixed *hRatio, Fixed *vRatio);
```

Get the current positioning ratios used by `SkelAlert()`. If you're not interested in a particular ratio, pass `nil` for the corresponding argument.

Miscellaneous Convenience Routines

pascal Boolean

```
SkelGetRectDevice (Rect *rp, GDHandle *gd, Rect *devRect,  
                  Boolean *isMainDevice);
```

`SkelGetRectDevice()` determines which active screen device contains the largest part of the given rectangle and returns the device, the device rectangle, and a flag indicating whether or

not the device is the main device. These values are stuffed into the arguments, which are passed as variable addresses. If you're not interested in a particular value, pass `nil` for the corresponding argument.

The rectangle must be specified in global coordinates. The return value is `true` if the rectangle overlaps some device, `false` otherwise.

If the return value is `false` (the rectangle overlaps no devices), the device, device rectangle parameters are filled in with the values for the main device.

On return, `gd` will be `nil` if Color QuickDraw is not supported, since no graphics devices will be present.

```
pascal void
SkelGetMainDeviceRect (Rect *r);
```

Returns in `r` the usable area on the main device (the screen containing the menu bar).

```
pascal void
SkelPositionRect (Rect *refRect, Rect *r,
                  Fixed hRatio, Fixed vRatio);
```

Position the rectangle `r` relative to the reference rectangle `refRect`. `hRatio` and `vRatio` indicate the ratios to use for dividing the border between the inner and outer rectangles.

Examples:

Center a rectangle inside the reference rectangle:

```
SkelPositionRect (&ref, &r, FixRatio (1, 2), FixRatio (1, 2));
```

Of vertical border between a rectangle and the reference rectangle, leave 1/3 above the rectangle and 2/3 below:

```
SkelPositionRect (&ref, &r, FixRatio (1, 2), FixRatio (1, 3));
```

```
pascal void
SkelPositionWindow (WindowPtr w, short positionType,
                   Fixed hRatio, Fixed vRatio);
```

Position a window according to the given positioning type, using `hRatio` and `vRatio` to divide the border between the window and the reference frame. The frame used is determined by `positionType`, which should be one of the following:

<code>skelPositionNone</code>	Don't position window; leave as is
<code>skelPositionOnMainDevice</code>	Position on main device
<code>skelPositionOnParentWindow</code>	Position over the frontmost visible window
<code>skelPositionOnParentDevice</code>	position on device on which the frontmost visible

window lies.

For those cases in which a window is positioned against a device, the positioning does not include the menu bar area if the device is the main device. Those cases which use the frontmost visible window use the main device if no window is visible.

It is best that the window be invisible before calling `SkelPositionWindow()` and made visible after, otherwise the window will appear to jump on the screen.

Note

If a window is positioned against a parent window that has been moved partly off the desktop, the positioned window might end up partly off the desktop, too. This may be undesirable. If so, you can force the window to be positioned against the parent device instead if it goes off the desktop by writing your positioning code like this:

```
SkelPositionWindow (wind, positionType, hRatio, vRatio);
SkelGetWindStructureRect (wind, &r);
if (positionType == skelPositionOnParentWindow && !SkelTestRectVisible (&r))
    SkelPositionWindow (wind, skelPositionOnParentDevice, hRatio, vRatio);
```

```
pascal void
SkelGetReferenceRect (Rect *r, short positionType);
```

Return in `r` the rectangle to be used as a reference against which to position another rectangle. `positionType` should be one of the following:

<code>skelPositionOnMainDevice</code>	Reference is main device
<code>skelPositionOnParentWindow</code>	Reference is frontmost visible window
<code>skelPositionOnParentDevice</code>	Reference is device on which the frontmost visible window lies

For those cases in which a device rectangle is the reference, the device rectangle does not include the menu bar area if the device is the main device. Those cases which use the frontmost visible window use the main device if no window is visible.

If `positionType` is `SkelPositionNone`, the result is undefined.

```
pascal Boolean
SkelTestRectVisible (Rect *r);
```

Return `true` if the given rectangle is entirely visible, i.e., entirely contained within the desktop region, `false` otherwise.

```
pascal void
SkelPause (long ticks);
```

Pause process execution for the given number of ticks (60ths of a second). If the application is running under a system that supports multitasking, time is given to other processes during the pause.

Using TransSkel from Pascal

This section discusses how to use TransSkel from within Pascal.

The Pascal interface consists of the interface file *TransSkel.intf* and the library document *TransSkel*. These are both located in the Pascal folder.

You should include both files in your project document, and source files that use TransSkel routines, types, or constants must list *TransSkel* in the `uses` statement:

```
uses  
    TransSkel;
```

The calling sequence for TransSkel routines are as given by the procedure/function declarations in *TransSkel.intf*. The calling sequences for callback routines are given in comments in the `type` section. These provide you with the appropriate translation for the various callbacks described elsewhere using C syntax in the rest of this manual.

Future Changes

This section describes some of the changes that might happen in future TransSkel releases.

- Modeless dialog support may disappear.
- I might add support for mouse-moved events.
- The Apple Event interface may change if I start to actually understand Apple Events.

Authorship

The principal authors of TransSkel are listed below, with an indication of the thrust of their contributions. You may judge which author is most likely to be able to field your questions from this information, or you may simply address them to the first author.

Paul DuBois

Original author of TransSkel.

U.S. Mail

Paul DuBois
Wisconsin Regional Primate Research Center
1220 Capitol Court
Madison, WI 53715-1299 USA

Internet

dubois@primate.wisc.edu

Owen Hartnett

Responsible for most of the improvements between releases 1.02 and 2, and for the port from THINK C to THINK Pascal.

U.S. Mail

Owen Hartnett
OHM Software Company
163 Richard Drive
Tiverton, RI 02878 USA

Internet:

omh@cs.brown.edu (Brown University Computer Science)

UUCP

uunet!brunix!omh

Bob Schumaker

Responsible for most of the improvements between releases 2 and 3.

U.S. Mail

Bob Schumaker
The AMIX Corporation
1881 Landings Drive
Mountain View, CA 94043-0848

Internet

bob@markets.amix.com

UUCP

{sun, uunet, netcom}!markets!bob
CIS
72227,2103
AOL
BSchumaker

Other contributors, of code or, what is just as useful, criticism, include David Berry, Nick Rothwell, William Gilbert, Danny Zerkel, Duane Williams, Denis Cohen, Lionel Cons...

Related Efforts

The philosophy underlying TransSkel is that it should serve as a module that can be plugged into a large number of applications without change. Two other packages sharing the same philosophy are available. Both aim at the goal of “compile it and plug it in.” TransDisplay provides an arbitrary number of generic display windows. These can be used for displaying debugging output or on-line documentation, for example. The module automatically handles scrolling, autoflushing, reformatting when the window is resized, etc. TransEdit is similar, providing generic text edit windows. Keyboard input, text selection with the mouse, scrolling, file I/O, and so forth are handled with little or no host intervention required.

Both are public domain, free and come in source form, with documentation and example programs.

With few exceptions, code in pre-3 versions of TransSkel performed operations that applications wanted to avoid doing at all costs, so the interface offered only functions of no general interest to applications except for interacting with TransSkel itself.

Macintosh programming is now sufficiently complex that some of the operations TransSkel needs to perform may duplicate functions needed in the main application code, so those functions have been packaged as additional interface routines.

`SkelGetWindowDevice()` is an example of this, because finding the device that contains most of a window has application other than in window zooming (which is how TransSkel uses it).

Developing with TransSkel

Hierarchical menu support doesn't work with 64K ROM machines, and there's nothing you can do about it. Should your application require them and provide no workaround, it should test for that ROM via `SkelQuery()` and refuse to run if it's present.

No test needs to be made for the presence of multiple monitors, for the functions that TransSkel itself provides (window dragging, sizing and zooming).

Miscellany and Random

Suspend/Resume proc might be used to know when to hide subsidiary windows (e.g., tool windows, modeless dialogs, clipboard). Perhaps this should be handled by TransSkel itself.

If you modify the event mask, you are responsible for resetting it before your application exits. PGMF 2–11 claims that “in the long run, low memory will disappear,” so TransSkel avoids anything that requires modifying or even accessing low memory variables directly. The rationale is that there is no point in doing anything that looks like it will cause an application to break under future versions of system software.

Introduction to TransSkel Release 3	1
Major Differences Between Releases	1
TransSkel and System Compatibility	2
Overview	4
Distribution Notes	6
How to get TransSkel	6
Distribution Layout	6
Installing TransSkel	7
Interface Installation for C	7
Interface Installation for Pascal	7
The TransSkel Demonstration Applications	7
The TransSkel Programming Interface — General Information	10
Using TransSkel from C	10
Source Files	10
Project Document	10
THINK Project Manager	10
The 'SIZE' Resource	10
Using TransSkel from Pascal	10
The Rest of this Manual	11
Initialization and Termination	12
Querying the Execution Environment	14
Event Processing	16
Menu Management	23
Window Management	26
The Port-Setting Model	33
Window Properties	35
Multitasking Support	38
Apple Event Support	42
Convenience Routines	45
Control Manipulation	45
Dialog Item Management	45
Alert Presentation	48
Miscellaneous Convenience Routines	48
Using TransSkel from Pascal	51
Future Changes	52
Authorship	53
Miscellaneous	55