Bitwise Operator

By Matt Slot

For this column, we'll talk about several changes to the sample code (found at http://www.ambrosiasw.com/~fprefect/bitwise/issue5/bitwise.sit.hqx). The most striking change was the reorganization of the code into multiple source files. Next, there are several places where the window handling code was moved to create several helper functions. Finally, so that the changes aren't entirely administrative, the application now supports several types of application dialogs.

Because there is no longer a single source file, I felt free to distribute the code among several files and to include a resource file. This makes it difficult to provide a single Web link to the entire source code, so I've archived the code -- you'll need Stuffit Expander to uncompress it.

Project Management: Multiple Source Files

As a software program grows and matures, it's important to make sure that the source code changes with it. While it would be nice to keep the entire source for an application in a single file, it becomes quite impractical after a point. This said, there are a few things you should do that will end up saving both compile and debug time in the end. If you already have written projects with multiple source files, you can skip this section.In the first pass at the source, there were "function prototypes" (using the "extern" keyword) declared at the top of the source file. These declare the calling conventions for the indicated name, which simplifies the compiler's job (and reduces confusion) when the code invokes that function. On the MacOS, and in general, you should get in the habit of prototyping your functions.

Because we want functions in one source file to call functions in another source file (and because we don't want to maintain the same prototype in many places), we move that declaration to a header file. Whenever a source file needs to invoke a function from another source file, it uses the #include preprocessor directive to load the associated header file and get all of the declarations.

In addition to function prototypes, we can also place preprocessor macros, struct declarations, and even declarations for our globals in the header file. Notice that each header is bracked with a #ifndef and #endif pair, so that the contents are actually only processed once, no matter how many times the header is included.

After examining the sample source files, it should become apparent that each file contains related functions. This makes it easier to find specific functions, as well as reducing the number of file-dependencies. To add the source and resource files to your project, either drag them from the project window or choose "Add Files..." from the Project menu.

This is also the first column that includes a resource file to link with. The menus, error dialog, and the new dialogs are all loaded from resources, that were created using ResEdit. Resources are an easy way to change the layout or language of interface elements without forcing a recompile -- which can be handy for localizing software.

User Interface: More Window Handling

In preparation for doing floating windows in the next article, we need to migrate the window handling code from various places within the program into specific functions. We call these functions "bottlenecks" because all of the window handling code is forced through a narrow set of functions, and then dispatched to be handled elsewhere.Looking in "windows.h", there are handlers for almost every window-specific event: click, select, update, activate, drag, grow, zoom, and disposing. Most of these consist of essentially the same code as the previous version, except they now reside in functions the same file. Also, several functions have been adjusted to recognize and dispatch events to dialogs.

User Interface: Dialogs

The most substantial change to the sample application is support for multiple kinds of dialogs. There are 3 important kinds of dialogs on the MacOS: modeless, modal (including alerts), and movable modal. This article concludes with a discussion of the differences between them, and an implementation of the first 2 kinds. Movable modal dialogs will be addressed in the next column, when we dive fully into window layers with
floating windows.A dialog is a special kind of window that is used to interact with the user and retrieve information about a desired action or setting. The Mac Toolbox helps by providing routines that simplify the creation of "controls", text fields, and other interface elements in dialogs. In addition, the Dialog Manager can create dialogs from templates stored in the resource file, which means that you can use ResEdit to create or modify the layout of items in your dialogs easily.

A common class of dialogs are "modeless" -- which means that to the user, they behave pretty much like a normal window. These are typically used for Search and Replace dialogs, which can remain on screen for longperiods of time. To that end, the DoSampleModeless() creates the dialog and then returns. In the main event loop, a call to the Toolbox function IsDialogEvent() now redirects events to a handler HandleModelessEvent()
as necessary.

We let the Dialog Manager process the event using DialogSelect(), and then handle the indicated item as appropriate. Note that each type of item has its own nuances, both for initializing and responding to
clicks. However, as demonstrated below, essentially the same switch statement is used for the modal dialog as well. Finally, modeless dialogs are not dismissed by clicking an OK or Cancel button, so the sample dialog includes a close box and also responds to the Close Window menu command.

By far the most popular kind of dialog is "modal". These stop the normal execution of a program until certain information can be exchanged with the user. Typically, an application restricts the user's freedome (and places itself into a "mode") because it cannot continue processing a request until it gets additional feedback (such as where to save a file, or how many copies to print). When the information is complete and the user dismisses the modal dialog, the application event loop resumes as normal.

The DoSampleModal() function creates, interacts, and then disposes of the dialog before returning. Event handling differs from modeless dialogs because it doesn't poll DialogSelect(), but calls ModalDialog() repeatedly while handling clicked items. When the user clicks a button that should dismiss the dialog ("OK" or "Cancel" in this case), a flag is set and the loop terminates. Again, note that the switch statement is essentially the same as a modeless dialog.

A special variant of the modal dialog is the alert. An alert is posted to notify the user of important information or report an error code. It typically has one or more buttons for the user to indicate what to do (such as "Save", "Cancel", "Don't Save", etc). Alerts rarely need additional controls like checkboxes or radio buttons.

The functions DoSampleAlert() and HandleError() both load and display an alert on the screen. Note that the Toolbox makes it easy, by providing a single function, Alert(), that manages everything -- in contrast with a
modal dialog.

The last kind of dialog is the "movable modal", which are replacing modal dialogs as the way to request important information. The primary difference between modals and movable modals is that the latter can be moved and that the user can switch to another application while a movable modal is onscreen. This column doesn't illustrate one of these, but stay tuned because the next one will.

Matt Slot, Bitwise Operator