PennyWise™ Application Framework

Introduction

PennyWise Application Framework is a development framework which makes implementing a Macintosh application much easier. The framework handles all the details of how to dispatch events and to whom they should be dispatched. This will make implementing a Macintosh application a much less daunting task, particularly to the new programmer.

While I feel that this framework will be a large asset to the new programmer, this is by no means a lightweight beginners framework. This framework model is a high powered application core that I have yet to find a project that was not made easier by using it. More than anything else, it is, simply put, just a good design. This is what I wish I had available when I started Macintosh development.

Additionally, this framework does not do things it's own way, forcing you to learn a custom interface system rather then the Macintoshes. It hides very little from the user, but rather enforces a structure that makes building complex applications much simpler. This framework ***will not*** avoid the need to learn the Macintosh Toolbox; it will make learning, understanding and using it easier.

The three major concerns while developing this framework were ease of use, speed of execution, and flexibility. I believe I've succeeded in all three areas.

PennyWise Application Framework was built using **Think Project Manager** so that is the form that the framework takes. As time goes on and as demand arises it will be ported to other platforms.

NOTE: This document should explain all the intricacies of using the PennyWise Application Framework. However, documentation is far from my specialty. It may be easier to understand if you were to try to build the sample project and then read this document. Yeah, you heard me right, do it first, then read the directions.

Price

A great deal of time and effort went into making this framework a viable development tool. I would like nothing more than to say "I don't need the money," unfortunately that is not the case. Therefore pricing is as follows:

Individual—$25 This license fee grants one individual the right to use PennyWise Application Framework for any and/or all software development projects.

Group—$100/6 This license fee grants the same rights as the individual license fee, but gives a discount for groups working together. The licensing is $100 for up to 6 individuals and $15 for each additional person, up to a maximum of $200.

Site—$200 This license give all people within your organization the right to use PennyWise Application Framework.

Tutorial—?? If you have no intention of using PennyWise Application Framework for development, but found it to be a valuable learning tool. The tutorial fee is whatever you think it is worth. (a postcard would be nice).

NOTE: *Please read the License Agreement for the details.*

License Agreement

When the above mentioned fee has been paid, PennyWise grants the use of PennyWise Application Framework on a single computer for each license bought. Licensees may use the PennyWise Application Framework to create freeware, shareware, or commercial software provided that 1) Credit is given to PennyWise Software in an appropriate place in the software and/or documentation. 2) Only applications are distributed—not source and/or object code.

You may not include any part of PennyWise Application Framework in a product for which you are or have given up your rights to the source code.[Contract Developers] If you which to use PennyWise Application Framework in such a product the individual or company who will acquire your code must buy an additional license.

Payment should be sent to:

PennyWise Software
PO Box 32153
Raleigh, NC 27622

Checks should be made payable to: Peter Kaplan

Overview

*The **PennyWise Application Framework** is broken into 4 parts. It is written this way to allow the greatest flexibility. While they are built to be used together you can decide what parts make sense in your development methodology.*

1 • The PWFramework.LIB.

*The **PWFramework.LIB** is the application core. This library is a very tightly written, flexible event handler. It does all of the behind the scenes work handling and/or dispatching events to the proper places. The inclusion of this library adds about 3K of code to your project. But it is code that would have to be written anyway.*

2 • The PennyWise Application Framework Project Model.

*The **PennyWise Application Framework** Project Model is a Think Project Manager Project Model which uses the **PWFramework.LIB** and implements a working Application around it. This is the best starting point for your projects. This allows you to start new projects with a working skeleton already implemented. This requires PWFramework.LIB to work.*

4 • The Window Templates.

The Window Templates are stationary pad files that implement the different window types. To add window types to your application simply open the appropriate template and make changes to fit your new window type. Step by step directions are given at the top of the template's source file to make implementing new WindowIDs as painless as possible.

3 • The Utility Libraries.

The Utility Libraries are utility routines that would be of use to all Macintosh programmers. All experienced programmers probably already have similar libraries. They can be used either with the PWFramework.LIB or separately, in other projects; there are no interdependencies with PWFramework.LIB. This set of utilities is far from a robust programming library. It will increase with each revision of the PennyWise Application Framework. (If you can think of other routines that you would like to see added let me know and I'll try to include them).

To make the PWFramework.LIB work you must assign each kind of window a separate ID. The IDs start at 1 and work upwards to the maximum number of window types your application handles. (0 is reserved for default behavior). At your application's initialization you register each WindowID's handlers with the PWFramework.LIB. Along with each WindowID's handlers you must select what type of window it will be. (Described Below).

NOTE: WindowIDs are not individual windows, but rather kinds of windows. You can have many windows of the same ID open at the same time.

Additionally, each WindowID is assigned a Window Type. For each Window Type there are proper methods of interaction. All of the proper methods of interaction are handled by the PWFramework.LIB. All that you as the programmer need to do is assign the proper Type to your WindowIDs. The list of supported window Types is maintained by PWFramework.LIB, but that does not mean the "look" of the window is controlled; only the actions. The "look" of a window is controlled by the WDEF. See Inside Macintosh: Macintosh Toolbox Essentials for details.

The currently available window Types are:

kWINDOW_TYPE_APPLICATION
This will behave as a standard Macintosh window, it supports all the dragging, sizing and switching you would expect. It conforms completely with the Guidelines laid out in Macintosh Human Interface Guidelines.

kWINDOW_TYPE_MOVABLE_MODAL
This will behave like kWINDOW_TYPE_APPLICATION in all situations except when the user tries to select another window in your application. Instead of bringing the other window forward the user will just get a beep.

kWINDOW_TYPE_DIALOG
This will behave like kWINDOW_TYPE_MOVABLE_MODAL in all situations except when the user tries to switch out of your application. Instead of switching out of your application the user will just get a beep.

kWINDOW_TYPE_BACKGROUND,
This window is not like the others, it will not come to the foreground when the user click in it. It stays where it was. This is not a standard Macintosh Type of window. Possible uses are: if you want a Logo screen to always be behind your running application. The only reason I implemented this is a customer wanted all other applications to disappear while working in his system (Even the trashcan); so I made a virtual desktop window the size of the screen and just filled it with the desktop pattern.

kWINDOW_TYPE_FLOATING.
This window is a floating palette window, It will reside in a layer above the application windows. NOTE: To implement a floating window you must use the TextServices utility windows. THIS SYSTEM DOES NOT IMPLEMENT ITS OWN FLOATING PALETTES.

NOTE: **kWINDOW_TYPE_FLOATING** windows have not yet been fully implemented.

PWFramework.LIB Functions:

These four functions **must** be called by your application at the proper time. They are the heart of the PWFramework system. Implementing them is simple, two of the four get called right from the main segment of your code. The other two get called at very well defined times. These routines are already properly implemented in the PennyWise Application Framework Project Model.

void PWInitMac(short *masters*, Size growStack, short *noOfWindowTypes*);

This function initializes all of the Macintosh Managers and the PWFramework itself. It allocates all memory for the window handlers and allocates the gMouseMovedRgn global variable.

short
masters
The number of times *MoreMasters* should be called by your Application. This will determine how many Master Pointer Blocks are allocated in your Application's Heap.

Size
growStack
This is how many bytes you would like to increase the stack size by. This is handled in the method outlined in Inside Macintosh: Memory. Unless you have large stack requirements you should leave this 0.

short
noOfWindowTypes
This is the number of different WindowIDs your Application uses. If you do everything correctly, passing in the value kMAX_WINDOW_IDS will work.

void PWMainEventLoop(long *sleepValue*, Boolean *AllowGlobalIdle*);

Call this routine after calling PWInitMac, initializing all the window handlers, and doing any other things at startup your application may need. This routine starts the PWFramework.LIB event processing. You do not ever come out of this call so any code after this call in main will never get called.

long
sleepValue
This is the value that will be passed into WaitNextEvent for the sleep parameter. This is the maximum number of ticks that your application agrees to give up to background applications while no events need processing.

Boolean
AllowGlobalIdle
The PWFramework allows you a high degree of flexibility in event processing by giving you access to the event before the PWFramework handles it. This is done by implementing a GlobalIdle routine. However, because most applications do not need this flexibility it can be turned off by passing false in this parameter. NOTE: You must still have a GlobalIdle routine, but it will never get called.

void PWQuitApplication(void);

Call this routine when the user selects Quit from the menu. It will go about asking all the windows to shut down and if they all agree to do so it calls ExitToShell.

short PWCallClose(EventRecord *theEvent, WindowPtr theWindow);

Call this function when the user selects Close from the menu. This function will get called by the framework automatically when the user clicks in the goAway box.

Developer Provided Functions:

These functions **must** be provided by you, the developer. Even if you are not going to use them, they must be defined or a link error will occur. These routines are already properly implemented in the PennyWise Application Framework Project Model. In the Project Model they do nothing, but they are there.

```
short GlobalIdle(EventRecord *theEvent);
```

This function can be used to get access to the event before the PWFramework handles it. You can do as little or as much preprocessing as you like. If the return value is TRUE the framework will assume that your GlobalIdle function handled the event and it will do nothing. If the return value is FALSE the PWFramework.LIB will handle the event.

NOTE: If you pass ALLOW_GLOBAL_IDLE to PKMainEventLoop this function will get called once at the beginning of each new event, if you pass !ALLOW_GLOBAL_IDLE this routine will never get called. It was done this way because very few applications will need that kind of access so if it will not be used why waste time in the main event loop calling it.

```
void GlobalPreMenu(void);
```

This function will be called after a user presses the mouse button in the menu bar, but before a call is made to MenuSelect. This is a good place to build a “Windows” menu.

```
void GlobalPostMenu(void);
```

This function will be called directly after the MenuSelect call. Here you would undo any changes you made in the GlobalPreMenu call.

NOTE: Toggling menu items depending on which window is front most should not be done here. Each window has a Handler to do just that.

PWFramework Global Variable:

This global variable is the only public access global for the PWFramework.LIB. In fact, the Framework only uses 10 bytes of global space. Additionally, speaking of memory, when InitMac is called the PWFramework.LIB will allocate one handle for use by the framework. The handle size will vary depending on how many window IDs you have implemented. This is the only memory requirements made by PWFramework.LIB

RgnHandle gMouseMovedRgn;

This Rgn is allocated and initialize in the InitMac function. It is used as the mouseRgn in the call to WaitNextEvent. This is the Rgn that moving the mouse ***will not*** cause an event. See WaitNextEvent in *Inside Macintosh:Macintosh Toolbox Essentials* for the details. Your window will want to alter this from the ThisWindowCursorCall.

NOTE: It sounds backwards, but it make sense to do it this way.

Window Handlers:

For each window ID you will have to create handler routines that will get called when a given event belongs to your window. Additionally, you will have to register each handler with the PWFramework.LIB. (This will be explained in the next section). If your window does not have to do anything for a particular event it you can pass in an empty routine. (The same empty routine can be reused for all empty handlers—This will save space).

void ThisWindowCreate(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine creates a new window. It should create the window in the most generic way possible. Unless your window needs no input parameters you will want to write a different "Create" routine that can take parameters of your choosing. There is no routine in the PennyWise framework that currently calls this routine, it is included for completeness only.

Regardless of what routine actually creates the window it must perform certain tasks. It must allocate a handle to its private data structure and install it in the refCon field of the WindowPtr. The first field of this data structure MUST be the PWFramework Window Header. (This is explained in the templates and demos).

(NOTE: What I do is write a Create routine for all my windows using the parameters that are appropriate for that window ID and then just make this routine a wrapper where it calls my custom routine with generic parameters).

The parameters, theEvent and theWindow, are irrelevant and are only supplied for uniformity.

short ThisWindowDispose(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine will close and dispose of the window specified in the parameter *theWindow*. It will return TRUE if the window was closed and FALSE if it remains open.

The common thing for this routine to do is check to see if changes have been made to this window (Is_Dirty Flag); if changes have been made, present the user with a save changes dialog. If they say save it save the data and dispose of the window, if they say cancel exit without saving and return FALSE.

This routine will be called by PWCloseCall.

void ThisWindowZoomIn(EventRecord* *theEvent*, WindowPtr *theWindow*);

The PW Application Framework will do the actual Tacking and Zooming of the window. This routine gets called directly before the framework Zooms the window in. So you can do any pre-processing that you would like. (I have yet to find a need to do anything here, but you never know).

void ThisWindowZoomOut(EventRecord* *theEvent*, WindowPtr *theWindow*);

The PW Application Framework will do the actual Tacking and Zooming of the window. This routine gets called directly before the framework Zooms the window out. So you can do any pre-processing that you would like. (This is useful when you want to ask the print manager about the size of the page, so you can force that to be the size "Zoomed" to.

void ThisWindowResize(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called whenever the user resizes the window (GrowBox or ZoomBox). It must reposition controls, lists etc. according to the new size.

void ThisWindowClick(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine will get called when the user clicks in the content region of your window. You must process the click appropriately.

void ThisWindowUpdate(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine will get called for an Update Event in your window. You must draw the window and controls in your window. NOTE: When it gets called from the PW Application framework it is already between calls to BeginUpdate and EndUpdate; So you need not make those calls. (And you should NOT make them).

void ThisWindowActivate(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine will get called for an Activate event in your window. You must Activate controls and such.

void ThisWindowDeactivate(EventRecord**theEvent*, WindowPtr *theWindow*);

This routine will get called for an Deactivate event in your window. You must Deactivate controls and such.

void ThisWindowDrag(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called when the user drags this window to a different position on screen. It gets called directly after the drag takes place. (I have yet to find a reason to do something here.) The dragging will be based on screenBits.bounds.

void ThisWindowIdle(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine will get called on NULL events. Only the front most window will receive this call (Even if the application is in the background). You can perform periodic tasks here. But, don't use this to adjust the mouse cursor, use the Cursor routine via gMouseMovedRgn.

void ThisWindowCursor(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called when there is a mouse moved event. In this routine you must determine what cursor to display based on the mouse's position and manipulate the gMouseMovedRgn global variable to reflect the move. On exit the mouse must be showing the correct cursor and the gMouseMovedRgn should be set to the Rgn where the mouse movements should NOT cause mouse moved events. (NOTE: This was a bit confusing to me at first, but it makes perfect sense to do it this way; If you need clarification look in Inside Macintosh: Processes)

void ThisWindowKeyDown(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called when the user presses a key (one that is not a Command Key combination). You must do what is appropriate for your window.

void ThisWindowPreMenu(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called directly before a call to MenuSelect. So if your window wants to enable, disable or change names of menu items here is where you should do it. (Ex. Make 'Close' read 'Close "untitled"' or turn on Cut, Copy, Paste as appropriate). REMINDER: theWindow is the front most window when this is called so you have access to the window and its data to make the changes appropriately.

short ThisWindowDoMenu(EventRecord**theEvent*, WindowPtr *theWindow*,
short *theMenu*, short *theItem,short theWindowID*);

This routine gets called when the user selects a valid item from the menu (or a command key). When this routine is called you must either handle the event entirely and return TRUE or not handle it at all and return FALSE. (If you return false it will be given to the defaultMenuDispatch (GlobalMenuDispatch) after which it is assumed handled no matter what the return value is.

void ThisWindowPostMenu(EventRecord* *theEvent*, WindowPtr *theWindow*);

This routine gets called directly after the call to MenuSelect. So you can clean up the changes you made in ThisWindowPreMenu.

void ThisWindowBackground(EventRecord**theEvent*, WindowPtr *theWindow*);

This routine will only get called if theWindow IS NOT the front most window. It is intended for background processing. If you need a procedure to get called while both in the background and in the foreground make it an Idle and Background procedure.

NOTE: if you do not wish to do any background processing you can set this routine to NULL in the WindowList.

THIS IS THE ONLY ROUTINE YOU CAN SET TO NULL. Others just supply an empty routine if nothing is to be done.

void ThisWindowGrowRect(EventRecord* *theEvent*, WindowPtr *theWindow*,
Rect* *theRect*);

This gets called before the Framework calls the GrowWindow procedure. What this routine must pass back in theRect is theRect->top = min height of window, theRect->left = min width of window, theRect->bottom = max height or window, and theRect->right = max width of window. If you wish to handle the GrowWindow call yourself pass back a Rect of 0,0,0,0.

void ThisWindowGetScrap(EventRecord**theEvent*, WindowPtr *theWindow*);

This routine will be called on a resume event when the convertClipboard bit is set. What you must do is convert the system scrap to your private scrap.

void ThisWindowPutScrap(EventRecord**theEvent*, WindowPtr *theWindow*);

This routine will be called on a suspend event . What you must do is convert your private scrap to the system scrap.

INSTALL FUNCTIONS:

This is the way you register your WindowIDs with PWFramework.LIB. These install functions must get called before you create or access any windows. For each WindowID you must call these routines with the WindowID and the proper parameter.

This sets the window Type for this WindowID.

```
void  PWInstallWindowType     (short id, short theType);
```

All the remaining functions install the proper handlers. They have the format:

```
void  PWInstallCreate         (short      id, ProcPtr proc);
void  PWInstallDispose        (short      id, ProcPtr proc);
void  PWInstallZoomIn         (short      id, ProcPtr proc);
void  PWInstallZoomOut        (short      id, ProcPtr proc);
void  PWInstallResize         (short      id, ProcPtr proc);
void  PWInstallClick          (short      id, ProcPtr proc);
void  PWInstallUpdate         (short      id, ProcPtr proc);
void  PWInstallActivate       (short      id, ProcPtr proc);
void  PWInstallDeactivate     (short      id, ProcPtr proc);
void  PWInstallIdle           (short      id, ProcPtr proc);
void  PWInstallCursor         (short      id, ProcPtr proc);
void  PWInstallKeyDown        (short      id, ProcPtr proc);
void  PWInstallDrag           (short      id, ProcPtr proc);
void  PWInstallPreMenu        (short      id, ProcPtr proc);
void  PWInstallMenu           (short      id, ProcPtr proc);
void  PWInstallPostMenu       (short      id, ProcPtr proc);
void  PWInstallGrowRect       (short      id, ProcPtr proc);
void  PWInstallBackground     (short      id, ProcPtr proc);
void  PWInstallScrap2Appl     (short      id, ProcPtr proc);
void  PWInstallAppl2Scrap     (short      id, ProcPtr proc);
```

MENUS

A great deal of time and effort went into developing methods for handling menus, but all of the methods tried only made working with menus more obfuscated. So what the framework does is dispatches menu selections to your routines to handle.

Of course, menu dispatching is all automatic, but it is up to you to install the menus at startup and up to your routines to handle the menu selections. Each window has its own handlers for doing just that. (ThisWindowDoMenu—As described in the previous sections).

PennyWise Application Framework Project Model

The PennyWise Application Framework Project Model is a skeleton project that implements the PWFramework.LIB. This is the best starting point for your PWFramework.LIB projects. It is a complete application that will compile and run without any modification. (Although it does nothing). From this skeleton you can build your application. All that you need to do is install your window handlers and it will work. Then, as the need arises, you can modify the skeleton into your full blown Macintosh Application.

Once properly installed all that you need to do in order to use the Project Model is

1) Start a New Project in Think Project Manager.
2) When the **New Project** dialog appears select *PennyWise Application Framework* from the list and press **Create.**
3) Name and Save your new project wherever you keep your projects.

That's it! Now the project will compile to a running framework without any modification.

Window Templates

In order to make implementing your application as painless as possible several templates have been written for the different window types defined by PennyWise Application Framework. These templates will be placed in the folder with a new project started from the PennyWise Application Framework Project Model. They are stationary pad files so they will open in untitled windows; this way you can name your files as you need without destroying the originals.

These templates have all the handler routines already written and have general instructions on how to incorporate them into your project. You can start from these templates and then modify the routines to fit your needs as you add then to your project.

UTILITY LIBRARIES

ALSO INCLUDED WITH THE PENNYWISE APPLICATION FRAMEWORK IS A SET OF UTILITY LIBRARIES. THE SET OF LIBRARIES WILL INCREASE WITH EACH REVISION OF THE PENNYWISE APPLICATION FRAMEWORK. THE HEADERS ARE IN THE *PENNYWISE INCLUDES* FOLDER IN THE SAME FOLDER AS THINK PROJECT MANAGER. THE LIBRARIES ARE IN THE *PENNYWISE LIBRARIES* FOLDER ALSO IN THE SAME FOLDER AS THINK PROJECT MANAGER.

THE CURRENT LIBRARIES ARE DIALOGUTILS, WINDOWUTILS, MENUUTILS, PRINTUTILS, AND PREFSMGR.

DialogUtils

DialogUtils implements several common routines that are performed within the context of dialog boxes. Several of the drawing routines rely upon WindowUtils. So including this file means including that one as well.

---

pascal OSErr SetDialogDefaultItem (DialogPtr theDialog, short newItem);
pascal OSErr SetDialogCancelItem (DialogPtr theDialog, short newItem);

These routines are implemented by Apple as part of the Macintosh Dialog Manager. The headers supplied with my copy of THINK C did not have them defined. So I put them here. If your headers already have them defined remove them from this file. To see how they work look to Inside Macintosh: Toolbox Essentials.

---

short GetDItemType(WindowPtr theDialog, short theItem);
Handle GetDItemHandle(WindowPtr theDialog, short theItem);
Rect GetDItemRect(WindowPtr theDialog, short theItem);

Given a DialogPtr and item number these routines will return the dialog item type, Handle, or Rect respectively.

Boolean IsButton(WindowPtr theDialog, short theItem);
Boolean IsRadio(WindowPtr theDialog, short theItem);

BOOLEAN ISCHECKBOX(WINDOWPTR THEDIALOG, SHORT THEITEM);

BOOLEAN ISOTHERCONTROL(WINDOWPTR THEDIALOG, SHORT THEITEM);

GIVEN A DIALOGPTR AND ITEM NUMBER THESE ROUTINES WILL RETURN TRUE IF THE ITEM IS A BUTTON, RADIO BUTTON , CHECKBOX, OR CUSTOM CONTROL RESPECTIVELY.

VOID FLASHBUTTON(WINDOWPTR THEDIALOG, SHORT THEITEM);

THIS ROUTINE WILL HIGHLIGHT THE BUTTON FOR AN INSTANT AND THEN RETURN IT TO ITS PREVIOUS STATE. THIS IS USEFUL FOR WHEN YOU WANT A KEYSTROKE TO SIMULATE PRESSING A BUTTON.

VOID HILITEDITEM(WINDOWPTR THEDIALOG, SHORT THEITEM, SHORT VALUE);

THIS ROUTINE SETS THE HIGHLIGHT STATE OF A DIALOG ITEM.

VOID FRAMEITEM(WINDOWPTR THEDIALOG, SHORT THEITEM, LONG COLOR);

THIS ROUTINE DRAWS A FRAME AROUND THE RECTANGLE OF A GIVEN DIALOG ITEM. THE FRAME WILL BE DRAWN IN THE COLOR. (OLD QUICKDRAW COLORS)

VOID DRAWBUTTONOUTLINEITEM(WINDOWPTR THEDIALOG, SHORT THEITEM, LONG COLOR);

THIS ROUTINE DRAWS A BOLD OUTLINE AROUND THE BUTTON AT THE GIVEN DIALOG ITEM. THE FRAME WILL BE DRAWN IN THE COLOR. (OLD QUICKDRAW COLORS)

VOID ERASEBUTTONOUTLINEITEM(WINDOWPTR THEDIALOG, SHORT THEITEM);

THIS ROUTINE ERASES A BOLD OUTLINE AROUND THE BUTTON AT THE GIVEN DIALOG ITEM.

VOID DRAWFOCUSITEM(WINDOWPTRTHEDIALOG, SHORT THEITEM, LONG COLOR);

THIS ROUTINE DRAWS A FOCUS OUTLINE AROUND THE GIVEN DIALOG ITEM. THE FRAME WILL BE DRAWN IN THE COLOR. (OLD QUICKDRAW COLORS)

VOID ERASEFOCUSITEM(WINDOWPTR THEDIALOG, SHORT THEITEM);

THIS ROUTINE ERASES A FOCUS OUTLINE AROUND THE GIVEN DIALOG ITEM.

VOID GETDTEXT(DIALOGPTR THEDIALOG, SHORT THEITEM, UNSIGNED CHAR* THESTRING);

VOID SETDTEXT(DIALOGPTR THEDIALOG, SHORT THEITEM, UNSIGNED CHAR * THESTRING);

These routines will get or set the text of a dialog item. The item must be either a static text or edit

text item.

```
void SetDValue(DialogPtr theDialog, short   theItem, short theValue);
short GetDValue(DialogPtr theDialog, short  theItem);
```

These routines will get or set the value of a dialog item. The item must be a control.

```
void GetPopUpItemString( WindowPtr theDialog, short theItem, unsigned char *
                         theString);
```

This routine will get a string from a PopupMenu control. The string will be the currently selected item in the popup menu.

PrefsMgr

PrefsMgr implements a uniform way for your application to access a preferences file held in the Preferences folder (or sub folder). This manager requires System 7.0 or better to run. It uses 2 bytes of global data space, but has no other memory requirements. The PrefsMgr routines **will** move memory if the equivalent Resource Manager routine moves memory.

```
short OpenPrefsFile(char *dirName, char *fileName, NewRoutine newProc,
                    OSType theCreator);
```

This routine will open a preferences file named *fileName*. If you would like the preference file to be in a sub directory supply the directory name in *dirName* (If not pass NULL). The file will be of type 'pref' and have a creator code of *theCreator*. Upon exit the PrefsMgr will have been initialized and be ready to use. If the call was unable to open the preferences file it will return NULL. All other values mean it was successful.

The preferences manager remains active until you call ClosePrefsFile. However, calls to the resource manager will not be sent to the preferences file; but rather your application. Only calls to the PrefsMgr routines will get and save resources to the preferences file.

If the prefs file does not exist this file will create one and then call the routine you supply in NewRoutine. NewRoutine must be of the format void NewRoutine(short). When your routine gets called your preferences file will be the Current Res File and the short that gets passed in will be the oldCurrentResFile (Your Applications Res File). It is then up to your routine to set up any needed resources in the prefs file.

```
void  ClosePrefsFile(void);
```

This routine will close the currently opened Prefs file. After this call is made you can not use the prefs manager routines again (unless you open another prefs file).

```
long  GetOrMakeDir( short theVolume , long parID, char *theName);
```

This routine is provided because it is useful outside the realm of preferences. It will return the dirID of the directory at theVolume with parID of the name theName. If one does not exist it will be created.

The remainder of the routines are directly analogous to the Resource Manager routines, but they will only look in preferences file. The routines are as follows:

```
Handle      GetPrefsResource(ResType theType, short   theID);
Handle      GetPrefsNamedResource(ResType theType, char * theName);
Handle      GetPrefsIndResource (ResType theType,short  theIndex);
void  GetPrefsIndType (ResType *theType,short  theIndex);
short CountPrefsTypes (void);
short CountPrefsResources (ResType theType);
void  RmvePrefsResource ( Handle theHandle);
short UniquePrefsID (ResType theType);
void  AddPrefsResource ( Handle theHandle, ResType theType,short
                  theID, char * theName);
```

MenuUtils

These routines are for customizing menu items at run time. They can be used to make **Close** become **Close this Document** and then change it back to **Close** when *thisDocument* is no longer open. (See PennyView for an example).

```
void AddNameToMenu( MenuHandle theMenuHandle, short theItem, unsigned
        char* theNameString, Boolean elipse, Boolean quotes);
```

This routine will add the string pointed to by theNameString to the end of the menu item at theItem in menu theMenuHandle. The parameter elipse should be true if the item has an elipse at the end. The parameter quotes should be true if you want theNameString in quotes.

```
void SetMenuItemToIndString(short  theID, short theIndex, MenuHandle
        theMenu);
```

This routine will get a string from STR# resource number theID at index theIndex and make it the menu item theIndex in menu theMenu.

PrintUtils

The PrintUtils hides a great deal of the detail of the printing manager from you. It is good for printing if you don't have very elaborate needs. It uses four bytes of global space and allocates only the required data structures for printing.

```
void DoPageSetup(void);
```

Call this routine when someone selects PageSetup from the File Menu. It will do the right thing.

```
Boolean DoPrintDialog(void);
```

Call this routine to get a Print Job Dialog box. It will do the right thing. If the user selected cancel it will return FALSE, if the user selected OK it will return TRUE.

```
void    GetPageSize(Rect *theRect);
```

This routine will return the Rect of the page that is currently selected in the Page Setup.

```
void    GetPageSizeWhileOpen(Rect *theRect);
```

This routine does the same thing as the previous one, but you call this one if you are currently printing. (Between StartPrinting and StopPrinting calls).

```
void StartPrinting(void);
```

This call will initialize printing, and set the current port to the printing port. Whatever Quickdraw calls you now make are going to a printed page.

```
void StopPrinting(void);
```

This call will close the print page and end printing. Then it will send the print job off to be printed.

```
void NextPage(void);
```

This routine will close the current print page and open a new one. Call this when you fill up a page and want to go on to the next.

WindowUtils

The window utilities have several routines that are of general interest for windows. The WindowUtils.LIB relies on PrefsMgr.LIB so inclusion of these windows routines means including PrefsMgr as well.

---

void DrawButtonOutlineRect(WindowPtr theWindow,Rect*theRect,long color);

This routine will draw a default button outline around the given Rect. It will draw the outline in the color. (Old Quickdraw colors)

---

void EraseButtonOutlineRect(WindowPtr theWindow, Rect *theRect);

This routine will Erase a default button outline around the given Rect.

---

void DrawFocusRect(WindowPtr theWindow, Rect *theRect, long color);

This routine will draw a Focus Rect around the given Rect. It will draw the focus in the color. (Old Quickdraw colors)

---

void EraseFocusRect(WindowPtr theWindow, Rect *theRect);

This routine will erase a Focus Rect around the given Rect.

---

void SetWindowPosition(WindowPtr theWindow, short theID);

This routine will set the window position of the window with theID. (The ID is usually a ResID). It will move the window to that position if that position is not offscreen. This routine relies on the PrefsMgr, so it must be available and initialized with a prefs file.

---

void ChangeWindowPosition(WindowPtr theWindow, short theID);

This routine will tell the prefs file that the user has moved with window with theID. (The ID is usually a ResID). It will update the prefs file accordingly. This routine relies on the PrefsMgr, so it must be available and initialized with a prefs file.

---

void CascadePosition(WindowPtr theWindow);

This routine will move theWindow slightly to the right and down from the current FrontWindow. It will

only do this if the windows would otherwise be in the same place.

Final Word

I hope that you find PennyWise Application Framework as useful as I do. I think it is a good learning tool and a good development tool.

> What is in the future for PennyWise Application Framework? This is entirely up to market demand. So, if you can think of something that you want implemented tell me. I think the next thing I'd like implemented would be the Drag and Drop Manager. Additionally, I'll be adding new Utility routines.

If you have any questions, comments, concerns, praise or insults feel free to contact me:

AOL: PennyWs SW
eWorld:MacRTP

PennyWise Software
PO Box 32153
Raleigh, NC 27622