Carbon Events in PowerPlant

Rick Aurbach
Aurbach & Associates, Inc.

October 5, 2006

Abstract

The standard PowerPlant distribution includes initial, incomplete support for Carbon Events. Here, I discuss issues I encountered developing products that use Carbon Events and the modifications I made to the PowerPlant sources to deal with them. This work also led directly to a re-implementation of PowerPlant's contextual menu processing, which is also included.

Credits

Eric Schlegel and contributors to the Carbon-Dev Mailing List have provided me with invaluable assistance in understanding and resolving some of the visual artifacts this work attempts to solve. I have borrowed heavily from the work that John C. Daub (and his collaborators) did in the original PowerPlant implementation of contextual menu support. And finally, I'd like to thank Liz Aurbach for uncovering many of the visual artifacts that this work addresses.

Contents

Issues with PowerPlant 2.2.5 **2**

Existing Carbon Event Handlers 2

What's Missing 4

Other Goals of This Work 5

PowerPlant Changes **5**

Added Files **10**

Contextual Menu Strategy **11**

Details 12

Incorporating Contextual Menus 17

Subclassing Issues 18

Example 19

Issues with PowerPlant 2.2.5

PowerPlant 2.2.5 includes fledgling support for Carbon Events. The presence of this code is controlled by the PP_Uses_Carbon_Events directive, which has a default value of `false` in `PP_Macros.h`.

If PP_Uses_Carbon_Events is `true`, then

- New windows are created with the `kWindowStandardHandlerAttribute` attribute (implemented in two places in `UCarbonDesktop.cp` and two places in `UWMgr20-Desktop.cp`).
- LWindow includes the data member `mEventHandlers`, which is a pointer to an associated LWindowEventHandlers object.
- LWindow manages its associated LWindowEventHandlers object by
  - initializing the `mEventHandlers` data member to `nil` in its constructors.
  - deleting the LWindowEventHandlers object in its destructor.
  - creating and initializing (i.e., installing) the object in its `FinishCreateSelf` method.

This means that when Carbon Events are enabled in PowerPlant, a set of standard event handlers are required. Before delving into what each of these handlers does, we need to understand why they are needed at all.

In a modern Carbon application, windows contain a hierarchy of embedded controls. In this context, a control is an object which is *known* to (i.e., registered with) the OS and which has a standard API that the OS can use to communicate with the controls. In particular, the hierarchy of controls in a window is known to the OS, so that appropriate Carbon Events can be sent directly to them. In a modern Carbon application, *only controls* are embedded in a window.

In contrast, a traditional PowerPlant application handles all event processing internally. It explicitly fields all events, determines which objects should receive them, and dispatches these events internally. PowerPlant objects are *not necessarily* controls — indeed, a number of important PowerPlant objects are derived from LPane or LView and are not controls at all. As long as PowerPlant is solely responsible for event dispatching, that is just fine.

However, when PP_Uses_Carbon_Events is `true`, we are in a hybrid situation. A window will contain some controls (basically Appearance objects*) and some non-control objects. This causes a problem because both the Carbon Event processing system and PowerPlant will attempt to process and dispatch events, in conflict with one another.

The purpose of the handlers implemented in the LWindowEventHanders object is to provide fixes for these inherent incompatibilities.

Existing Carbon Event Handlers

The implementation of LWindowEventHandlers in PowerPlant 2.2.5 provides handlers at the window level (i.e., there are no control-level handlers) for the following events:

*With important exceptions!

`{kEventClassWindow, kEventWindowDrawContent}`
: This handler calls the window's `Draw` method and returns `noErr`.

  The standard window handler processes this event by calling `DrawControls`. Instead, drawing is done by the PowerPlant drawing system.

`{kEventClassWindow, kEventWindowActivated}`
: This handler calls the window's `Activate` method and returns `noErr`.

  The standard window handler processes this event by sending a `kEventWindowHandleActivate` event to itself (Mac OS 10.3 or later) or calling `ActivateControl` on the window's root control. Instead, activation is handled by the standard PowerPlant methods.

`{kEventClassWindow, kEventWindowDeactivates}`
: This handler calls the window's `Deactivate` method and returns `noErr`.

  The standard window handler processes this event by sending a `kEventWindowHandleDeactivate` event to itself (Mac OS 10.3 or later) or calling `DeactivateControl` on the window's root control. Instead, deactivation is handled by the standard PowerPlant methods.

`{kEventClassWindow, kEventWindowClickContentRgn}`
: This handler converts the event into a traditional `EventRecord`, passes the `EventRecord` to the window's `ClickInContent` method and returns `noErr`.

  The standard window handler processes this event by checking for contextual menu clicks and clicks on controls, and sending `kEventWindowContextualMenuSelect`, `kEventControlClick`, and `kEventWindowHandleContentClick` events as appropriate. Instead, this handler connects mouse clicks to the standard PowerPlant event-dispatching system.

`{kEventClassWindow, kEventWindowGetMinimumSize}`
: This handler calls the window's `GetMinMaxSize` method, sets the `kEventParamDimensions` parameter based on the returned value and returns `noErr`.

  On Mac OS 10.2 and later, the standard window handler processes this event by calling `GetWindowResizeLimits` and returning the size obtained in the `kEventParamDimensions` parameter. On pre-10.2 systems, it does nothing. Instead, this handler uses the size information stored by PowerPlant for all OS versions.

`{kEventClassWindow, kEventWindowGetMaximumSize}`
: This handler calls the window's `GetMinMaxSize` method, sets the `kEventParamDimensions` parameter based on the returned value and returns `noErr`.

  On Mac OS 10.2 and later, the standard window handler processes this event by calling `GetWindowResizeLimits` and return the size obtained in the `kEventParamDimensions` parameter. On pre-10.2 systems, it does nothing. Instead, this handler uses the size information stored by PowerPlant for all OS versions.

`{kEventClassWindow, kEventWindowBoundsChanged}`
: This handler gets the value of the event's attributes (`kEventParamAttributes`), passes them to the window's `AdaptToBoundsChange` method, and returns `noErr`.

  In Mac OS X v10.2 and later, the standard window handler can receive this event under the following conditions:

  - the window uses live resizing (if the `kWindowLiveResizeAttribute` attribute is set).

- the user is the one resizing the window
- an update event for the window exists in the event queue

If these conditions are met, the standard window handler removes the update event from the event queue and sends it to the event dispatcher target. Doing so simplifies redrawing window content during live resizing.

Instead, this handler uses the `AdaptToBoundsChange` method (designed to handle window state changes which don't occur via the usual PowerPlant methods) to make the necessary changes to the window's state and to record the changes in the appropriate PowerPlant internal data members.

`{kEventClassWindow, kEventWindowZoom}`

This handler calls the window's `SendAESetZoom` method and returns `noErr`.

The standard window handler zoom's the window using `ZoomWindowIdeal` then, if successful, sends a `kEventWindowZoomed` event. Instead, this handler calls `SendAESetZoom`, which is the same method called when the user clicks in the zoom box in a traditional PowerPlant application.

`{kEventClassWindow, kEventWindowClose}`

This handler calls the window's `ProcessCommand` method, passing it `cmd_Close` and returns `noErr`.

The standard window handle calls `DisposeWindow`. This handler invokes the standard PowerPlant actions associated with closing a window, including checking whether the document is dirty, etc.

`{kEventClassMouse, kEventMouseMoved}`

This handler converts the event to an `EventRecord`, retrieves the window's current content bounds (by calling `GetWindowBounds`), and testing the current mouse location. If the mouse is in the window's contents region, the window's `AdjustContentMouse` method is called; otherwise, the cursor is set to the arrow.

The behavior of the standard window handler to this message is not documented, but presumably involves sending events useful for adjusting the mouse cursor as the mouse enters and leaves appropriate control parts. This method uses the standard PowerPlant mechanism instead.

What's Missing

The above logic handles most situations, but fails to address a number of special cases.

- The standard PowerPlant text objects are not implemented as controls. Therefore, if you use them (instead of alternatives, such as J.W. Walker's CCarbonEditText* class), there will be a number of problems because not all events will be properly routed to them.
- The click handler bypasses normal contextual menu click detection, making implementation of contextual menu clicks difficult.

*http://www.jwwalker.com/pages/ccarboneditext.html

- Drawing of appearance objects can occur without sending a `kEventWindowDrawContent` event. When this occurs, bad things can happen because the PowerPlant object may not have been properly focused.

  Moreover, appearance objects which derive from LControlView will not redraw subviews which are not Carbon controls.

- If a moveable modal dialog enables menu items, the menus are not always updated when the dialog is shown.

The primary goal of this work is to improve the way PowerPlant functions when Carbon Events are enabled.

Other Goals of This Work

Beyond the obvious goal of providing a more complete and robust implementation of Carbon Event handing within a standard PowerPlant application, this work includes some related work involving contextual menus.

The original implementation of contextual menus in PowerPlant was developed by John C. Daub and was based on the Contextual Menu Manager APIs. This new implementation leverages the contextual menu support built in to Carbon and provides some features which were difficult to implement with John's original design.

- Provides an inheritance mechanism so that visual structure (such as a window's subview and its subviews) can display a single common contextual menu with minimal effort.

- Allow users to implement contextual menus both via attachments and via coding subclasses.

  When creating a custom visual-object subclass for an application, it can be convenient to build contextual menu support for that object directly into the subclass, rather than depending on an attachment. But when adding a contextual menu to a standard visual object, an attachment allows this to happen without requiring subclassing.

- Provide both static and dynamic contextual menus. The standard attachment supports a statically-defined contextual menu. However, in some applications it is desirable to customize the menu itself based on application context or (dare we say it?) mode.

We will begin by reviewing changes to the PowerPlant package, then discussing the implementation strategies for contextual menus.

PowerPlant Changes

PP_Macros.h

Added a new conditional compilation directive (`PP_Uses_ContextMenus`) to enable the contextual menu support. This symbol is automatically set to `false` unless `PP_Uses_Carbon_Events` is `true`.

PP_Messages.h

I added `cmd_Help` (with a value of 28) because such a symbol is generally useful in applications which provide interactive help to their users.

Define the `msg_ContextMenu` message. This message is sent to attachments to cause them to display and process a contextual menu.

PP_Resources.h

Define `str_HelpMenuTitle` as a symbol for the `STR#` string index for the title of the Help menu item in a contextual menu.

PP Copy & Customize.ppob

Adds "Help" as a standard string. This string is used for the title of the Help menu item in a contextual menu.

LWindowEventHandler

In the `.h` file, defined the `InputText` method and the `ShowWindow` method.

In `InstallEventHandlers`, added code to install the `InputText` and `ShowWindow` handlers.

If `PP_Uses_ContextMenus` is true, add code to the `ClickContentRgn` handler to detect and process contextual menu clicks. See the section "Contextual Menu Strategy beginning on page 11 for a more detailed discussion.

Add code to the `ClickContentRgn` handler to execute application-level attachments before calling the window's `ClickInContent` method.

```
    if (LCommander::GetTopCommander()->
            LAttachable::ExecuteAttachments(msg_Event, &clickEvent)) {
        mWindow->ClickInContent(clickEvent);
    }
```

This change is needed to support LInPlaceEditField and related objects that depend on application-level attachments being executed in response to mouse clicks.

Add the `InputText` event handler to connect the processing of standard PowerPlant text objects (such as LEditText) with event processing.

```
OSStatus
LWindowEventHandlers::InputText(
    EventHandlerCallRef /* inCallRef */,
    EventRef            inEventRef )
{
    EventRecord     event;
    event.what = keyDown;
    event.when = ::EventTimeToTicks(::GetEventTime(inEventRef));
    event.where.h = event.where.v = 0;
```

```
    EventRef        rawKey;
    ::GetEventParameter(inEventRef,
        kEventParamTextInputSendKeyboardEvent, typeEventRef,
        nil, sizeof(EventRef), nil, &rawKey);
    UInt32          keyCode = 0;
    UInt32          modifiers = 0;
    char            charCode = 0;
    ::GetEventParameter(rawKey,
        kEventParamKeyMacCharCodes, typeChar, nil, sizeof(char),
        nil, &charCode);
    ::GetEventParameter(rawKey, kEventParamKeyCode, typeUInt32,
        nil, sizeof(UInt32), nil, &keyCode);
    ::GetEventParameter(rawKey, kEventParamKeyModifiers, typeUInt32,
        nil, sizeof(UInt32), nil, &modifiers);

    event.message = ((keyCode & 0x000000FF) << 8) + charCode;
    event.modifiers = modifiers;

        // Check if the keystroke is a Menu Equivalent
    SInt32          menuChoice;
    CommandT        keyCommand  = cmd_Nothing;
    LMenuBar*       theMenuBar  = LMenuBar::GetCurrentMenuBar();

    if (theMenuBar != nil) {
        keyCommand = theMenuBar->FindKeyCommand(event, menuChoice);
    }

    LCommander *    tgt = LCommander::GetTarget();
    if (tgt != nil) {
        if (keyCommand != cmd_Nothing) {
            StUnhiliteMenu  unhiliter;
            LCommander::SetUpdateCommandStatus(true);
            tgt->ProcessCommand(keyCommand, &menuChoice);
        } else {
            tgt->ProcessKeyPress(event);
        }
    }

    return noErr;
}
```

Since LEditText is not a registered control, it does not receive keyboard events directly. Without this handler, standard keyboard events would be sent to it via the WNE handlers, but events such as field-to-field tabbing (which are intercepted before the WNE handler) are not handled properly.

Add the `ShowWindow` event handler.

This handler was added to resolve cases where a new window was being drawn incompletely (i.e., not all of its content appeared when the window was first drawn). Here's my conjecture about what happens:

```
OSStatus
LWindowEventHandlers::ShowWindow(
    EventHandlerCallRef /* inCallRef */,
    EventRef            /* inEventRef */ )
```

```
{
    mWindow->Refresh();

    return eventNotHandledErr;
}
```

Conjecture. *It is common practice in PowerPlant to create new windows as initially invisible, then to show them only after building their substructure, loading content into fields, etc. (to reduce flicker). However, when* `PP_Uses_Carbon_Events` *is true, drawing is triggered by the* `kEventWindowDrawContent` *event, which is sent to the window* before the window is shown. *With this order of operations, a window which is constructed initially invisible will not get the needed refresh when it is shown.*

LAMControlImp

If `PP_Uses_Carbon_Events` is true,

- Defined `FinishCreateSelf` and `DoDrawEvent` methods in the `.h` file.
- In the `FinishCreateSelf` install a Carbon Event Handler for the `kEventControlDraw` method.
- Implement the `DoDrawEvent` method. This Carbon Event handler calls the ControlPane's `FocusDraw` method and returns `eventNotHandledErr`, so drawing will occur in a focused context.

```
OSStatus
LAMControlImp::DoDrawEvent (
    EventHandlerCallRef             /* inCallRef */,
    EventRef                        /* inEventRef */ )
{
    mControlPane->FocusDraw();
    return eventNotHandledErr;
}
```

Popup Menus

In traditional PowerPlant, the menu handles of popup menus (in LPopupButton and LPopupGroupBox are set to null except when the popup menu is being explicitly handled. This doesn't work in a Carbon Event context because the controls can be called asynchronously. A number of changes are needed in LAMPopupButtonImp, LPopupButton, LPopupGroupBox, and LAMPopupGroupBoxImp

If `PP_Uses_Carbon_Events` is defined,

- Hide the `PostSetValue` and `GetMacMenuH` methods in LAMPopupButtonImp and LAMPopupGroupBoxImp. They are not used the Carbon Event context.
- Hide the code in the stream constructor of LAMPopupButtonImp and LAMPopupGroupBoxImp that initializes the menu to a special empty menu.

- Eliminate the use of the `StPopupMenuSetter` object in the `TrackHotSpot` [*in LAMPopupButtonImp*], `AdjustControlBounds` [*in LAMPopupGroupBoxImp*] and `DrawSelf` [*in LAMPopupButtonImp and LAMPopupGroupBoxImp*] methods.
- In LPopupButton, initialize the control's menu handle in `InitPopupButton`.
- Add a `SetMacMenuH` method to LPopupButton. This makes sure that if we change the popup menu associated with the control, this change is propagated down to the control's menu handle.

```
void
LPopupButton::SetMacMenuH(
    MenuHandle          inMenuH,
    bool                inOwnsMenu)
{
    LMenuController::SetMacMenuH(inMenuH, inOwnsMenu);
    mControlImp->SetDataTag(kControlNoPart,
        kControlPopupButtonMenuHandleTag,
        sizeof(MenuHandle*), (Ptr)&mMenuH);
}
```

- Analogous changes are made in LPopupGroupBox.

LControlView

If `PP_Uses_Carbon_Events` is true,

- Add code to the `FinishCreateSelf` method to install a Carbon Event handler for the `kControlEventDraw` event.
- Add the `DoDrawEvent` method to use this event handler.

```
OSStatus
LControlView::DoDrawEvent (
    EventHandlerCallRef                 inCallRef,
    EventRef                            inEventRef )
{
    OSStatus                            status;
    status = ::CallNextEventHandler(inCallRef, inEventRef);

    TArrayIterator<LPane*>              iter(mSubPanes);
    LPane *                             theSub;
    while (iter.Next(theSub)) {
        if (theSub != mControlSubPane) {
            theSub->Draw(nil);
        }
    }
    return status;
}
```

The method calls `CallNextEventHandler` to do the basic control drawing, then iterates over the control's subviews and explicitly `Draw`s them. This makes sure that all subviews (including ones which are non-control PowerPlant objects) are drawn.

LIconPane

Added the (missing) `GetIconID` method.

LPane

if `PP_Uses_ContextMenus` is true, the `ContextClick` and `ContextClickSelf` methods are defined. These methods are discussed in detail in section "Contextual Menu Strategy" beginning on page 11.

UKeyFilters

In analogy with the `IsEscapeKey` and `IsCmdPeriod` commands, define and implement the `IsHelpKey` method. This method recognizes the keypad Help key and ⌘ – ?.

UModalDialogs

When using Carbon Events, a dialog (controlled by UModalDialogs) that enables menu items may not properly update the menu bar initially. To fix this, the dialog handler explicitly updates menus the first time that `DoDialog` is called.

To implement this change, if `PP_Uses_Carbon_Events` is true,

- A new data member, `mFirstTime`, is defined. Its value is initialized to `true`.
- In `DoDialog`,

```
1    if (mFirstTime) {
         UpdateMenus();
         mFirstTime = false;
4    }
```

  if `mFirstTime` is `true`, `UpdateMenus` is called and `mFirstTime` is set to `false`.

Added Files

To support the new implementation of contextual menus, the following files were added to the `_Contextual Menus` subfolder of the `_In Progress` folder.

- `LContextMenuAttachment.h`
- `LContextMenuAttachment.cp`

  Source code for the contextual menu attachment class.
- `LContextMenuAttachment.CTYP`

  Constructor templates for the contextual menu attachment classes (class IDs '`CMat`' and `CMfd`').

Please Note!

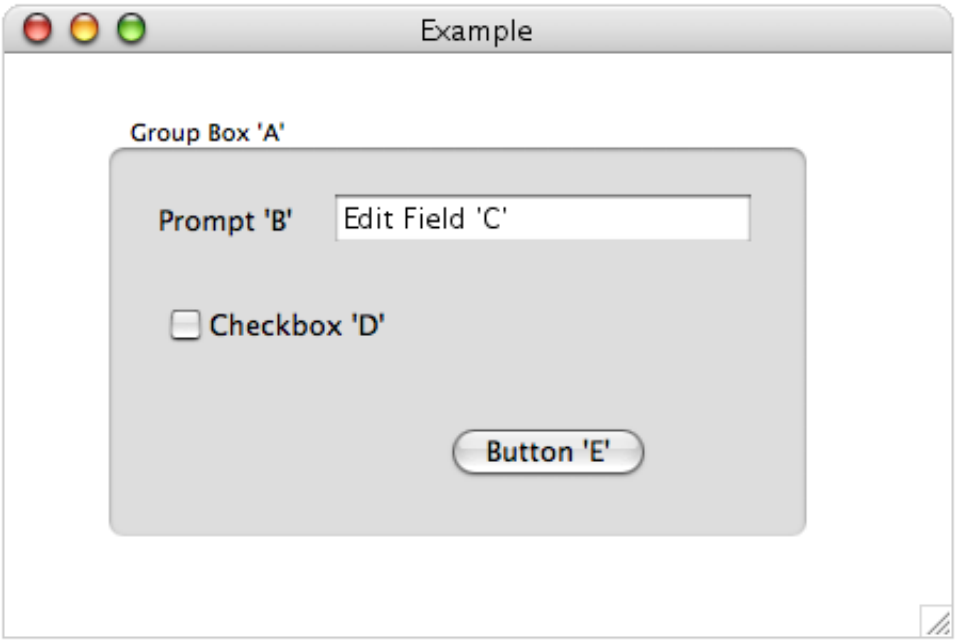
Constructor builds a list of "known" resources from the contents of the Custom Types folder automatically when it launches. However, testing shows that placing this file (in its supplied, data-fork-only format) in Custom Types does not cause its `CTYP` templates to load on Constructor startup. As a result, I recommend that you *convert this file to a traditional resource-fork-based resource file after downloading it.*

- `LContextMenuHelper.h`
- `LContextMenuHelper.cp`

  Source code for the contextual menu helper class.

Contextual Menu Strategy

The starting point for defining a new contextual menu strategy for PowerPlant is the issue of inheritance. Consider this simple example.



There are a variety of ways to handle contextual menus for this window — you might want objects 'C', 'D', and 'E' to have contextual menus; you might want to show the same menu if the user right-clicks over 'B' as over 'C'; or you might want there to be a single contextual menu if the user clicks anywhere inside of 'A', *even if the user clicks over one of 'A's sub-views.* What is "correct" is a matter for the application designer — the framework must permit any of these choices.

The standard PowerPlant implementation (using LCMAttachment) supports contextual menus on individual objects, so it could handle creating a contextual menu when the mouse is over 'C' (for example), but to provide a common contextual menu for the entire

region would require attaching LCMAttachment objects to all five objects ('A' – 'E'). And the current implementation of LCMAttachment only supports a fixed menu, which can be limiting.

This work has taken a different direction. Instead of basing the entire implementation on attachments, I have chosen to provide some of the contextual menu support by adding methods to LPane, thereby making contextual menus a *standard feature* (though often an unused one) of a PowerPlant object.

There are advantages to this approach:

- The object directly under the mouse gets first shot at handling a contextual menu click. It can either handle it itself or pass it up the visual hierarchy. Since all visual objects are ultimately derived from LPane, this works simply and naturally.

  In our example above, this feature allows us to have a single contextual menu for 'A' and all of its subviews without requiring attachments on 'B' – 'E'.

- Attachments are very useful for adding contextual menus to standard objects, but when you create a custom subclass, adding a custom attachment to it is often more work than is needed. This approach lets you choose between implementing a contextual menu by adding an attachment or by overriding a method in a custom subclass.

Details

All changes to existing PowerPlant files are controlled by the `PP_Use_ContextMenus` symbol.

LWindowEventHandlers

We add code to the `ClickContentRgn` method to recognize and intercept contextual menu clicks:

```
OSStatus
LWindowEventHandlers::ClickContentRgn(
    EventHandlerCallRef /* inCallRef */,
    EventRef            inEventRef)
{
#if PP_Uses_ContextMenus
    if (::IsShowContextualMenuEvent(inEventRef)) {
        OSStatus        status = eventNotHandledErr;
        Point           globalPt, portPt;
        ::GetEventParameter(inEventRef, kEventParamMouseLocation,
                typeQDPoint, nil, sizeof(Point), nil, &globalPt);
        portPt = globalPt;
        mWindow->GlobalToPortPoint(portPt);
        LPane *         subPane =
            mWindow->FindDeepSubPaneContaining(portPt.h, portPt.v);
        if (subPane != nil) {
            status = subPane->ContextClick(globalPt);
        } else {
            status = mWindow->ContextClick(globalPt);
        }
```

```
        if (status != eventNotHandledErr) {
            return status;
        }
    }
#endif
        // original code goes here
    return noErr;
}
```

If the event is a contextual menu click, we get the mouse location and see if the mouse is over one of the window's subpanes. If so, we call the pane's `ContentClick` method; else we allow the window to field the click.

If the `ContextClick` method handles the event, return; else allow the standard handler logic to process it.

LPane

So, if a contextual menu click is detected, the appropriate object's `ContextClick` method is called.

```
OSStatus
LPane::ContextClick (
    Point               inGlobalPt)
{
    OSStatus            result = noErr;

    if (ExecuteAttachments(msg_ContextClick, (void*)&inGlobalPt)) {
        if (not ContextClickSelf(inGlobalPt)) {
            LView *     superView = GetSuperView();
            if (superView) {
                result = superView->ContextClick(inGlobalPt);
            } else {
                result = eventNotHandledErr;
            }
        }
    }
    return result;
}
```

The `msg_ContextClick` message was added to `PP_Messages.h`.

This method attempts to handle the contextual menu click by (first) executing an attachment, then (second) executing a virtual LPane method, and (third) by passing the processing on to the pane's superview.

The default LPane `ContextClickSelf` method does nothing.

```
bool
LPane::ContextClickSelf (
    Point               /* inGlobalPt */)
{
    return false;
}
```

LContextMenuHelper

To facilitate the construction of the contextual menu, its display, and the processing of any command selected by the user, a helper class has been created. This class is used by contextual menu attachments and/or `ContextClickSelf` methods to implement the menu and process it.

```
class LContextMenuHelper : public LMenu) {
public:
                        LContextMenuHelper (
                            LCommander *        inCtxCmdr = nil );
                        LContextMenuHelper (
                            ResIDT              inMENUid,
                            LCommander *        inCtxCmdr = nil );
    virtual             ~LContextMenuHelper ( void );

            void        SetContextCommander (
                            LCommander *        inCtxCmdr );
            void        SetContextPane (
                            LPane *             inCtxPane );
            void        SpecifyHelpString (
                            ConstStringPtr      inHelpString );
            void        SpecifyHelpType (
                            UInt32              inHelpType );

            void        AppendMenuCommand (
                            ConstStringPtr      inMenuString,
                            CommandT            inCommand );
            void        AppendMenuCommand (
                            const char *        inMenuString,
                            CommandT            inCommand );
            void        AppendMenuCommand (
                            ResIDT              inStringResID,
                            SInt16              inStringIndex,
                            CommandT            inCommand );
            void        AppendMenuCommandList (
                            ResIDT              inMenuID );
            void        AppendMenuSeparator ( void );

    virtual CommandT    TrackMenu (
                            Point               inGlobalPt,
                            bool                inExecCmd = true );
protected:
    StAEDescriptor  mSelection;
    LCommander *    mCtxCmdr;
    LPane *         mCtxPane;
    LStr255         mHelpStr;
    UInt32          mHelpType;

    virtual void        IsHelpAvailable ( void ) const;
    virtual void        ShowHelp ( void );
    virtual void        GetContext ( void );
    virtual void        PreCMSelect (
                            Point               inGlobalPt );
    virtual void        PostCMSelect (
                            Point               inGlobalPt );
```

```
    virtual void        PrepareMenuItems ( void );
    virtual void        FinalizeMenu ( void );
    virtual void        CheckCommandStatus (
                            CommandT            inCommand,
                            Boolean &           outEnabled,
                            Boolean &           outUsesMark,
                            UInt16 &            outMark,
                            Str255              outName );
};
```

This helper class is typically used by either a contextual menu attachment or by an object's `ContextClickSelf` method to handle the creation, display, user interaction and processing of a contextual menu. Please see Example below.

Attachment Classes

This work provides two attachment classes for use with contextual menus:

LContextMenuAttachment — This attachment is analogous to LCMAttachment. It provides a mechanism to completely specify the content of the contextual menu, as well as display the menu and issue commands to process the selected option.

LContextMenuForwarder — The attachment takes a completely different approach. It does not build or handle the contextual menu at all. Instead, it locates the commander chain associated with the object to which it is attached and sends a `msg_ContextClick` message as a command to that command chain. In an MVC* approach, the attachment forwards the responsibility for the contextual menu from the View to the Controller.

LContextMenuAttachment

The LContextMenuAttachment class defines a "traditional" attachment, i.e. one that is added to a visual layout using its Constructor layout `CTYP`. The `CTYP` includes features that define the menu's items; its `ExecuteSelf` method uses this information to dynamically create the menu, display it, receive user feedback, and send the selected command to the appropriate commander for processing.

```
enum    {
    kCMAItemSeparator       = 0,
    kCMAItemMenuItem        = 1,
    kCMAItemMenuID          = 2
};

typedef struct  {
    SInt16                  mtype;
    UInt32                  mval;
} MENTRY;

```

*Model-View-Controller

```
class   LContextMenuAttachment : public LAttachment {
public:
    enum { class_ID = FOUR_CHAR_CODE('CMat') };

                    LContextMenuAttachment( LStream* inStream );
                    LContextMenuAttachment(
                        MessageT    inMessage = msg_ContextClick,
                        Boolean     inExecuteHost = true,
                        SInt16      inBaseMenuID = 0,
                        UInt32      inHelpType = kCMHelpTypeNoHelp,
                        const LString & inHelpString = Str_Empty,
                        SInt16      inEntryCount = 0,
                        MENTRY *    inEntries = nil );
    virtual         ~LContextMenuAttachment() {}

protected:
    SInt16              mMENUid;
    UInt32              mHelpType;
    LStr255             mHelpString;
    std::vector<MENTRY> mEntryList;

    virtual void    ExecuteSelf(
                        MessageT                inMessage,
                        void*                   ioParam);
    virtual LContextMenuHelper *
                    CreateHelper (
                        SInt16                  inMenuID,
                        LCommander *            inCtxTarget );
    virtual void    PrepareForMenu (
                        LContextMenuHelper* /* inHelper */ );
    virtual bool    FindCommandString(
                        CommandT                inCommand,
                        LString &               outString );
    virtual LCommander* FindCommandTarget();
};
```

The stream constructor collects information from the associated `CTYP` and stores it in data members. The `ExecuteSelf` method responds to `msg_ContextClick` messages by creating an `LContextMenuHelper` object, constructing the menu, then calling the helper's `TrackMenu` method to process it.

LContextMenuForwarder

As was mentioned above, the LContextMenuForwarder attachment takes a different approach to contextual menu handling.

```
class   LContextMenuForwarder   :   public  LAttachment {
public:
    enum { class_ID = FOUR_CHAR_CODE('CMfd') };

                    LContextMenuForwarder( LStream* inStream );
                    LContextMenuForwarder (
                            LCommander *    inCommandTarget = nil,
                            Boolean         inExecuteHost = true );
    virtual         ~LContextMenuForwarder() {}
```

```
protected:
    LCommander *      mTarget;

    virtual void      ExecuteSelf (
                          MessageT            inMessage,
                          void *              ioParam );
    virtual LCommander* FindCommandTarget();
};
```

This attachment responds to the `msg_ContextClick` message by locating the LCommander object most closely associated with the attachment's owner and passing the message to that commander as a command. (If you construct the attachment programmatically, you can specify the command explicitly.)

```
void
LContextMenuForwarder::ExecuteSelf (
    MessageT                    inMessage,
    void *                      ioParam )
{
    bool                        processed = false;

    if (inMessage == msg_ContextClick) {
        LCommander *            cmdr = mTarget;
        if (cmdr == nil) {
            cmdr = FindCommandTarget();
        }
        if (cmdr != nil) {
            SCMForward          param;
            param.globalPt = *(Point*) ioParam;
            param.ownerHost = mOwnerHost;
            processed = cmdr->ProcessCommand(inMessage,
                                             (void*) &param);
        }
    }
    SetExecuteHost(!processed);
}
```

That is, the LCommander is sent a `ProcessCommand` message with `msg_ContextClick` as its command parameter and a custom structure as its `ioParam`. The use of the custom structure provides the commander with both the context click location and information which allows it to determine which of its views sent the message.

Incorporating Contextual Menus

To use the contextual menus mechanism discussed here, you must build a Carbon application (either CFM or MachO). This code is compatible with Mac OS9/CarbonLib.

- The application prefix file should

  ```
  #define PP_Uses_Carbon_Events 1
  #define PP_Uses_ContextMenus 1
  ```

- You will need to include

```
LWindowEventHandlers.cp
LContextMenuHelper.cp
UCMMUtils.cp
```

in the project. If you plan to specify contextual menus using attachments, you should also include `LContextMenuAttachment.cp` in your project.

- If you are using attachments, be sure to register them as needed.
- Attach LContextMenuAttachment or LContextMenuForwarder objects where appropriate in the application's `PPob`s and/or code `ContextClickSelf` methods in custom visual classes.

Subclassing Issues

While the classes described above are designed to be as complete and self-contained as possible, they will usually need to be subclassed.

In OS X, contextual menus normally include a Help item as the first item of the menu. This item is inserted into the menu by `ContextMenuSelect`. It will be enabled unless `kCMHelpItemNoHelp` is specified. (If `kCMHelpItemRemoveHelp` is specified on 10.2 or later, the item will not be added to the menu; pre-10.2, it will be handled like `kCMHelpItemNoHelp`).

In the LContextMenuHelper method `TrackMenu`, user selection of this help item is handled by calling the `ShowHelp` method. But, since PowerPlant does not (and should not) place requirements on how the application provides help to users, the `ShowHelp` method is empty. To provide help in your contextual menus, you will need to override the `ShowHelp` method.

Other methods of LContextMenuHelper that you may wish to override include

`IsHelpAvailable`	This method is called if the specified help type is `kCMHelpItemOtherHelp`. If it returns false, the help type is converted to `kCMHelpItemNoHelp` for this call to `ContextualMenuSelect`. The idea is to provide a mechanism to test the availability of a help item prior to enabling help support.
`GetContext`	This method specifies the selection used by `ContextualMenuSelect` to determine which contextual menu plugins should add items to the menu. The default method calls the context-pane's `GetSelection` method.
`PreCMSelect`	This method is a hook called before `ContextualMenuSelect`. The default method is empty.
`PostCMSelect`	This method is a hook called after `ContextualMenuSelect` but before dispatching for command processing. The default method is empty.
`PrepareMenuItems`	This method is called prior to displaying the menu. The default method calls `ProcessCommandStatus` for each item and updates it accordingly.

`FinalizeMenu` This method is called prior to displaying the menu. The default method makes sure that the construction process did not inadvertently leave behind extra separator lines.

Note

If you override LContextMenuHelper, you will also want to override LContextMenuAttachment. It uses LContextMenuHelper in its `ExecuteSelf` method. To do this, simply override the `CreateHelper` method.

Example

The following example shows how to use LContextMenuHelper. The specific context of this example is as a ContextClickSelf override, but the same approach would be used (for example) in a `msg_ContextClick` command handler (invoked from an LContextMenuForwarder attachment.

```
bool
MyPane::ContextClickSelf (
    Point                inGlobalPt )
{
    LContextMenuHelper   helper(myCmdr);

    helper.SpecifyHelpType(kCMHelpItemAppleGuide);
    helper.AppendMenuCommandList(menu_ID);
    helper.AppendMenuSeparator();
    helper.AppendMenuCommand("\pMy Command", cmdID);

    helper.TrackMenu(inGlobalPt);

    return true;
}
```

line 5 Create an instance of the helper, passing it a pointer to the LCommander object that is responsible for handling commands for this instance of MyPane.

line 7 Specify that the help item is supplied by an Apple Guide.

line 8 Add a set of menu items (based on the context of the specified `MENU` resource) to the contextual menu.

line 9 Add a separator line to the contextual menu.

line 10 Add a single menu item to the contextual menu, with the specified menu item text and command number.

line 12 Display the contextual menu. Allow the user to make a selection. If the user does so, determine what command should be issued and send the appropriate `ProcessCommand` to `myCmdr`.

line 14 Return `true` because the contextual menu has been handled.