Electric Image Animation System

EIAS version 9.0
User Interface API

Table of Contents

Abstract

Reference

Electric Image Animation System

Introduction

The Electric Image User Interface API (hereafter known simply as “the API”) contains over 200 functions that allow you to create full featured dialog boxes in a cross-platform environment.

The API was designed to allow plugin developers to manipulate dialog boxes created using Interface Builder. However, there are limitations. The API does not, for example, provide support for creating non-modal windows, nor does it allow modification of the dialog box by creating new controls or destroying existing ones.

Conventions

Names

All function and type names begin with the prefix “EI_.” All constants are defined as enumeration constants and begin with the prefix “cEI_.”

Arguments to functions always begin with “i,” “o,” or “io.” The prefix “i” is for input-only arguments, “o” is for output-only arguments, and the prefix “io” is for arguments which are both input and output arguments. Usually, input-only arguments are const.

All macro names begin with “EI_” and all letters are upper case.

Pointers

Many functions take arguments that are of pointer type. Unless otherwise noted in a function description, passing NULL as a function argument will cause the function to return without taking any action.

Indices

Many functions deal with indices. For example, you can obtain a list box item by index. In all cases, the indices in the API are zero-based. That is, the first item indexed uses the index value 0. Unless otherwise noted, passing invalid indices to a function (such as a negative index) will cause the function to return without taking any action.

Strings

Throughout the API are functions which take arguments of string type. The strings used by the API are always C strings: an array of character bytes terminated by a zero byte. For operations that return a string (such as the function which retrieves the text from an edit text control), there are two functions provided: one which returns the size of a character buffer big enough to hold the string (note that the size returned includes the terminating zero byte), and another which copies the characters of the string into a buffer provided by the caller. It is not necessary to call the length function in order to call the copy function, but doing so will allow you to allocate a buffer large enough to hold all of the characters in the string.

Boolean Values

Many functions accept arguments of boolean type, and return boolean results. In all cases, the boolean argument is passed as type “int.” The interpretation of the int as a boolean is in accordance with the normal C convention:

a non-zero int is considered true, while a zero int is false. Beyond this convention, you cannot count on any particular value of the int. In particular, in the relationship between the function EI_StopDialog and the return result of EI_ExecuteDialog, the exact integer value of the argument iResult of EI_StopDialog will not be returned from EI_ExecuteDialog. The value of iResult will be converted to either 1 or 0 before it is returned from EI_ExecuteDialog.

API Versioning

The API contains one function, EI_GetAPIVersion, which returns the version of the API that is exported by the host program. As the API evolves, this will allows your code to determine what features are available through the API.

Opaque Types

Many of the types that the API deals with are opaque. For example, in the API there is no definition of the data contained in an EI_Dialog. These types are all defined similarly in the file EI_Types.h. For example, here is how EI_Dialog is defined:

```
typedef struct {
        int     mData;
} EI_Dialog;
```

The only method available to you to manipulate these types is through the functions of the API. There is no way to directly manipulate the internal data of an opaque type.

The types of the API are defined this way so that

- each type is unique and provides some type safety (for example, you cannot accidentally pass an EI_Dialog pointer to a function expecting an EI_Control)

- your code is prevented from depending on implementation details of the underlying implementation

All opaque types in the API are passed by pointer.

Universal Data types for 32/64-bit Platforms

The following types, defined in file PlatformTypes.h, have same size and range of values in both 32/64- bit modes, for Macintosh and Windows platforms:

SInt32 - 4 bytes signed integer on all platforms
UInt32 - 4 bytes unsigned integer on all platforms

Avoid using long and unsigned long data types. These cause problems because these types are 8 bytes long on Macintosh 64-bit but 4 bytes long on Windows and Macintosh 32-bit.

The following types, defined in file PlatformTypes.h, have mutable size depending on whether the 32 or 64 bit mode is in use:

slong - 4 bytes signed integer in 32 bit, but 8 bytes signed integer in 64-bit mode.

ulong - 4 bytes unsigned integer in 32-bit, but 8 bytes unsigned integer in 64-bit mode.

There are very few cases when you need to use these types.

File I/O

The API provides support for binary endian-correct I/O to files. Endian-correct means that writing a file on one platform and then reading the file on another platform will work, even if the platforms use different byte ordering. By convention, all binary I/O is performed using big-endian data. All of the details of byte-swapping are handled by the API I/O functions.

You also have the option of using block-level I/O calls to read and write arbitrary blocks of raw data. However, you should use these functions only with data that does not use a specific byte ordering, such as text strings, or data which you know will not be moved to another platform.

You should avoid code that reads and writes data by writing entire structs at a time. For example

```
typedef struct {
        int                     mData1;
        unsigned char           mData2;
        float                   mData3;
} MyStructure;

MyStructure             data;
SInt32                  count;

count = sizeof(MyStructure);
EI_WriteStream(myStream, &data, &count);
```

This kind of I/O is problematic for at least two reasons:

1) It will not byte-swap the members of the structure.

2) It also writes out any padding bytes the compiler inserts into the structure MyStructure. This forces anyone who wants to read the data, on any platform, to be aware of these padding bytes and account for them somehow.

Instead, write out each member of the structure individually, using the byte-swapping I/O functions.
The file I/O system revolves around three types of objects: directory references, file references and streams.
A directory reference is a representation of a path name to a directory. The path name is stored as a sequential list of strings. The directory does not actually have to exist in the user's file system for the directory reference to be valid.

There are three predefined directory references you can access: the home directory, the preferences directory and the temporary directory. The home directory is the directory in which your plugin resides. The preferences directory is a directory you can use to store preference information, if necessary. The temporary directory is a directory you should use for temporary files.

A file reference is essentially a directory reference plus a file name which names a file in the directory. In a manner analogous to directory references, the file referred to by a file reference does not need to exist in the user's file system for the file reference to be valid.

File references also carry a file type and extension with them. A file type is a 4 byte value, expressed as an unsigned integer value, used on the Macintosh to encode the type of the file, while the extension is a short string used to encode the type on Microsoft Windows.

NOTE: Changing the file type or extension of a file reference does not affect the corresponding file in the user's file system.

NOTE: If you include an extension in the file reference's name (e.g., "SomeFile.txt"), then the file reference's extension string will be ignored. If an extension appears in the file reference's name, it override's the extension of the file reference.

Note that file references and directory references do not make use of directory separator characters; however, you can convert file references and directory references into strings, which are pathnames which include directory separator characters between directory names.

A stream is an object that provides I/O services for an open file. Given a file reference, you can open a stream that you can use to perform I/O to the corresponding file in the user's file system.

Many file-related functions return error codes. In all cases, a zero result indicates successful completion of the function, while non-zero indicates failure. Note that this is the reverse of the boolean convention of zero meaning false and non-zero meaning true.

Resources

The API makes use of resources. A resource is a piece of data from a file that is addressed by an unsigned long type and an integer ID. Dialog box definitions and images are stored in resources. The API provides functions for reading images and dialog box definitions from resource files, as well as reading raw resource data of any type.

There is no support for writing resource files in the API. Currently, resource files can be created and edited only on the Macintosh, using a program such as ResEdit or Resorcerer.

Resources in the API will be familiar to Macintosh programmers, with one difference: resources are always returned as a pointer and a size rather than the ubiquitous "Handle" type.

To read any resource, you must first open the resource file that contains it. You can open as many resource files as you like, but you must close each file when you are done with it.

User Interface Controls

The user interface objects inside dialog boxes are called controls. The controls supported by the API are

- push buttons
- check boxes
- radio button groups
- divider lines
- group boxes
- tab controls
- color sliders
- scrollers

- static text
- edit text fields
- color buttons
- scrolling lists
- popup menus
- pictures
- user controls
- OpenGL controls

When users interact with the controls in a dialog box, your code will need to receive some kind of notification of the interaction. This notification comes from a callback function called a hit function. Hit functions are called when the following events occur:

- the user clicks a push button, radio button or check box
- the user drags the thumb of a color slider or scroller (in this case, the hit function is called every time the thumb moves while the user drags it)
- the user types a character into an edit text field
- the user clicks a color button and chooses a new color in the color picker that comes up
- the user clicks on an item in a scrolling list
- the user chooses a new item in a popup menu
- the user clicks on a user control
- the user clicks on an OpenGL control

Controls have attributes. There is a core set of attributes that all controls have, while some controls contain additional attributes. For example, push buttons have title strings, but scrollers don't. All controls have the following attributes:

- ID number
- a boundary rectangle
- an immediate draw flag
- a visibility flag
- an enabled flag
- a hit function

- an “extra data” pointer

The extra data pointer allows you to define and attach your own attributes to controls.

NOTE: Dialog boxes have an extra data pointer attribute as well.

Other attributes, with the controls that have them, are listed here:

- a title (push button, check box, group box, popup menu, static text, edit text)
- a value (push buttons that are in “toggle” mode, check box, radio button control, scroller, color slider, popup menu)
- a maximum value (scroller, color slider)
- an icon ID (push button, picture)
- title/icon layout (push button)
- an edit string (edit text)
- an edit filter type (edit text)
- an edit filter function (edit text)
- a selection range (edit text)
- a color (color button)
- a list of items (list box)
- a set of list flags (list box)
- a list double-click function (list box)
- a list reorder function (list box)
- a user control draw function (user control)
- a user control click function (user control)
- a user control mouse moved function (user control)
- an OpenGL draw function (OpenGL control)
- an OpenGL click function (OpenGL control)
- an OpenGL mouse moved function (OpenGL control)

When making a call to get or set the value of an attribute of a control which does not actually have the attribute, the function will do nothing. For example, attempting to get the value of a push button will return 0.

A description of each control’s interpretation of its optional attributes follows.

Push Buttons

Push buttons have a title and icon, which are drawn inside the button. If the push button is in toggle mode (which is set with Interface Builder), then the push buttons also have a value. Push buttons also have two layout attributes describing how the title and icon are positioned out with respect to each other.

Check Boxes

Check boxes have a title and a value. The title string is displayed next to the check box's button, and the value of a check box is boolean.

Radio Buttons

Radio button groups have a value. Each radio button control contains a set of 1 or more individual radio buttons. The value of the radio button control is simply the index of the chosen button. To turn off all radio buttons in a group, use a value of -1.

Pictures

Pictures have an icon ID.

Group Boxes

Group boxes have a title, which is drawn at the upper left corner of the box.

Color Sliders

Color sliders have a value and a maximum value. The value of the slider is always in the range [0, slider maximum]. When the thumb is all the way to the left, the value is 0, and when the thumb is all the way to the right the value is equal to the maximum. It is not possible to set the color slider's value less than zero or greater than the maximum.

Scrollers

Scrollers have a value and a maximum value. The value of the slider is always in the range [0, slider maximum]. When the thumb is all the way to the left (for a horizontal scroller) or all the way at the top (for a vertical scroller), the value is 0, and when the thumb is all the way to the right (for a horizontal scroller) or all the way to the bottom (for a vertical scroller) the value is equal to the maximum. It is not possible to set the scroller's value less than zero or greater than the maximum.

Static Text Controls

Static text controls have a title, which is displayed as the text of the control.

Edit Text Controls

Edit text controls have a title, an edit string, an edit filter type, an edit filter function and a selection range. The title is drawn to the left of the edit box, and the edit string is drawn within the edit box, where the user can edit it. Whenever the user enters text into the edit box, the edit text control filters the text according to its edit filter type. The edit filter type is an enumeration that names several predefined filter types, such as unsigned integer or signed floating point. However, if the enumeration indicates a filter function is used, the edit filter function (if it is not NULL) is called to see if the text typed by the user should be allowed. The selection range is a pair of integers (start, end), and represents the indices of the text currently highlighted by the user in the edit box. Note that if there is no text selected (that is, if there is an insertion point blinking), the start and end values will be equal to each other.

Color Buttons

Color buttons have a color attribute, which is used to fill the interior of the color button.

List Boxes

A list box has a list of items, a set of list flags, a double-click function and a reorder function.
The list items are simply strings. Each item can be drawn using stylistic variations such as bold or italics.
Currently, only two flag values are defined. One flag value enables or disables whether the list allows multiple selections or constrains the user to selecting single items at a time, and the other flag determines whether or not the user is allowed to drag items in the list to reorder them.
The double-click function is called whenever the user double-clicks an item. Note that the first click of a double-click sequence will cause the hit function (if any) to be called first.
The reorder function will be called if the user drags an item to a new position in the list.
List box controls can be the keyboard focus control of their dialog box. When a list box is the keyboard focus, it lets the user move the list selection up and down using the arrow keys of the keyboard.

Graph Controls

The Graph Control provides a way for EIAS plug-ins and shaders to create and manipulate editable graphs.

The service is provided by 2 functions:
EI_GraphDataRequest
EI_GraphControlRequest

They are defined in the header files, EI_GraphData.h and EI_GraphControl.h. Rather than list all of the details shown in these header files, this document will consider the principal aspects of the Graph Control.

Creating the Graph Control Interface

In Interface Builder, create a EI control of “Group” type. The standard size is 233x132 for the graph. In addition:

For shaders: specify “Graph” type in the shader’s PTKC resource for the item.

For Plug-ins: use EI_GraphControlRequest with the EI_GRAPH_CONTROL_REQUEST_CREATE command.

What is GraphData? Why is GraphData and Graph Control needed?

The Graph Ctonrol is just a visual UI element associated with a concrete dialog. At render time, there is no actual Graph Control(s). Therefore to use the results of an edited graph (or to load data into a graph) you need to pass the GraphData pointer to it (or extract it from the control).

The GraphData is an opaque data type. The client's side (plug-in or shader) is not aware about the internal graph data format and works through the EI_GraphData Request API interface. For example: to get a graph's value y(x), a client will pass the EI_GRAPH_CONTROL_REQUEST_GET_VALUE command value to the EI_GraphDataRequest call.

Loading and saving graph data edited by the user

For Shaders: The host provides this function automatically.

For Plug-ins:

Steps to save graph settings:

Get the GraphData pointer via EI_GRAPH_CONTROL_REQUEST_GET_GRAPHDATA
Get the GraphData size via EI_GRAPH_DATA_REQUEST_GET_DATASIZE
Save the GraphData into your data via EI_GRAPH_DATA_REQUEST_EXPORT

Steps to load graph settings:

Get the GraphData pointer via EI_GRAPH_CONTROL_REQUEST_GET_GRAPHDATA
Load the GraphData from your data via EI_GRAPH_DATA_REQUEST_IMPORT
Redraw the visible graph control with the EI_InvalControl call

How can I obtain the GraphData pointer during rendering when there is no Graph Control?

For Shaders: The host automatically passes the pointer to the Graph Data shader's interface. You should NOT convert this field between little/big Endian. For shaders, you should not destroy the passed GraphData and there is no need to create new ones.

For Plug-ins:

Create the new GraphData via EI_GRAPH_CONTROL_REQUEST_CREATE
Load the GraphData from your data via EI_GRAPH_DATA_REQUEST_IMPORT
Destroy the GraphData via EI_GRAPH_DATA_REQUEST_DESTROY after you are done with it.

Gradient Controls

The Gradient Control provides a way for the user to create a color gradient for use as a material property.

The service is provided by 2 functions:

EI_GrandientDataRequest
EI_GradientControlRequest

They are defined in the header files, EI_GradientData.h and EI_GradientControl.h, Functionally the Gradient Control is just like the Graph Control described above. Just use "GrandientData" instead of "GraphData".

Popup Menus

Popup menus have a title and a value. The title is drawn to the left of the popup button, and the value is the index of the selected item.

User Controls

A user control has a draw function, a click function, and a mouse moved function. The draw function is called when the user control needs to be drawn, the click function is called when the user presses the mouse button while the cursor is in the user control, and the mouse moved function is called whenever the mouse is moved within the user control.

NOTE: The click function is different from the hit function. The click function is meant to track the user's mouse activity when the control is clicked, perhaps changing its appearance as appropriate, while the hit function is a general notification mechanism that is used to alert your code that the user acted on a control.

NOTE: The hit function is not called for a user control unless either the user control has no click function OR the click function returns a non-zero result.

OpenGL Controls

An OpenGL control provides an OpenGL context you can use to draw 3D scenes using the OpenGL API. An OpenGL control has a draw function, a click function, and a mouse moved function, similar to the user control. The draw function is called when the OpenGL control needs to be drawn, the click function is called when the user presses the mouse button while the cursor is in the OpenGL control, and the mouse moved function is called whenever the mouse is moved within the OpenGL control.

NOTE: The click function is different from the hit function. The click function is called immediately when the user clicks the OpenGL control; you use it to track the user's mouse activity, perhaps changing its appearance as appropriate. The hit function is a general notification mechanism that is used to alert your code that the user acted on a control.

NOTE: The hit function is not called for an OpenGL control unless either the OpenGL control has no click function OR the click function returns a non-zero result.

Keyboard Focus

At any given time, there may be at most one keyboard focus control. The keyboard focus control will be the control that responds to keyboard events. Edit text controls and list box controls can be keyboard focus controls.

NOTE: The dialog's key filter function can intercept key events before they get to the keyboard focus. You can use the key filter function to implement custom accelerator keys.

Drawing

Two dimensional drawing takes place in the API through the use of a drawing context object. A drawing context object contains all of the information needed by the API to perform the drawing, including the destination of the drawing (either a dialog box window or an off-screen buffer), a clipping rectangle, a foreground and background color, a pen location, a pen mode and font information.

The pen location is used for drawing lines and text. When you use EI_MoveTo, you move the pen to a new location. EI_MoveTo does not perform any drawing. EI_LineTo, however, moves the pen to a new location while drawing a line from the previous pen location to the new location. Similarly, EI_DrawText and EI_DrawTextToFit draw text beginning at the current pen location, moving the pen to a new location after the last character drawn. When drawing text, the text is positioned so that the pen's location is aligned near the bottom left of the first character drawn.

Note that it is not possible to change the actual font used for drawing text, although you can change the size and style attributes (such as bold or underline) of the font.

Reference

Constants

The constants described in this section are defined in the header file EI_Types.h.

API Version Numbers

```
enum {
        cEI_InitialVersion = 1,

        cEI_CurrentVersion = cEI_InitialVersion
};
```

The constants defined by this enumeration define the supported versions of the API. The constant cEI_InitialVersion is the value that will be returned by the first shipping version of the API. In the future, as the API evolves, additional version number constants will be defined to represent newer versions of the API.

The constant cEI_CurrentVersion represents the version number of the API which corresponds to the definitions in the API header files. The version number of the API which is compiled into the host application is returned by the function EI_GetAPIVersion.

As the API evolves, subsequent version numbers, and the value of cEI_CurrentVersion, will always increase (though not necessarily by any fixed amount).

Font Style Flags

```
enum {
        cEI_TextBold = 1 << 0,
        cEI_TextItalic = 1 << 1,
        cEI_TextUnderline = 1 << 2,
        cEI_TextOutline = 1 << 3,
        cEI_TextShadow = 1 << 4
};
```

The constants defined by this enumeration represent the font style variations usable by a drawing context object to draw text. The constants may be bit-wise or'ed together to produce different style combinations.

Image Buffer Pixel Constants

```
enum {
        cEI_AlphaShift,
        cEI_RedShift,
        cEI_GreenShift,
        cEI_BlueShift,

        cEI_AlphaOffset,
        cEI_RedOffset,
        cEI_GreenOffset,
        cEI_BlueOffset
};
```

The constants defined by this enumeration can be used for low-level access to the individual components of the pixels contained in an image buffer. Each of these constants is defined differently for different operating systems.

NOTE: You should prefer to use the pixel access macros defined in EI_ImageBuffer.h whenever possible. Use the pixel access constants only when the macros don't provide the functionality you need.

The "shift" constants (cEI_AlphaShift, cEI_RedShift, cEI_GreenShift, and cEI_BlueShift) define the number of bit positions to left-shift an 8 bit integer value into the proper position in a 32 bit pixel. For example:

```
EI_32BitPixel           pixel;
unsigned char           red, green, blue;

// kind of a lime green color:
red = 0x7F;
green = 0xFF;
blue = 0x3F;
pixel = ((EI_32BitPixel) red << cEI_RedShift) |
        ((EI_32BitPixel) green << cEI_GreenShift) |
        ((EI_32BitPixel) blue << cEI_BlueShift);
```

Likewise, the shift constants can be used to extract color components from a 32 bit pixel using the right-shift operator:

```
red = (unsigned char) ((pixel >> cEI_RedShift) & 0xFF);
```

The "offset" constants allow access to pixel components using pointer offsets. Given a pointer to a 32 bit pixel, the offset constants give the offset in bytes from the pixel pointer to the address of the corresponding color component within that pixel. This works because the pixel is actually stored as four bytes. The offsets allow you to address each bytes individually. For example, to get the red component from a pixel pointer, you cast the pixel pointer to a pointer to an unsigned char and then add the cEI_RedOffset to it:

```
EI_32BitPixel           *pixelPtr;
unsigned char           red;

red = *((unsigned char*) pixelPtr + cEI_RedOffset);
```

List Box Flags

```
enum {
        cEI_ListMultipleSelect = 1 << 0,
        cEI_ListDragReorder = 1 << 1,
        cEI_ListHasItemIcons = 1 << 2,
        cEI_ListHasVisibilityColumn = 1 << 3,
        cEI_ListHasLockedColumn = 1 << 4,
        cEI_ListCanInlineEdit = 1 << 5
};
```

The constants defined by this enumeration define the flag bits returned by EI_GetListFlags. If the bit represented by the constant cEI_ListMultipleSelect is set, the list allows the user to make multiple item selections. If it is clear, the user may select only one item at a time. If the bit represented by the constant cEI_DragReorder is set, then the list allows the user to drag items up and down in the list to reorder the list. If it is clear, the user may not drag items up and down in the list.

You may set or clear either or both of these flags using the function EI_SetListFlags.

Modifier Key Flags

```
enum {
        cEI_ShiftKey = 1 << 0,
        cEI_ControlKey = 1 << 1,
        cEI_ModPrimary = 1 << 2,
        cEI_ModSecondary = 1 << 3,
        cEI_ModTernary = 1 << 4
};
```

When a user control's click function is called, the state of the modifier keys is passed as an integer argument containing bits for each modifier key. You can test the modifier keys argument by performing a bit-wise "and" operation using the constants defined in this enumeration.

If the bit represented by the constant cEI_ShiftKey is set, the shift key is being pressed. If the bit represented by the constant cEI_ControlKey is set, the control key is being pressed.

The constants cEI_ModPrimary, cEI_ModSecondary and cEI_ModTernary are meant to abstract the modifier keys. cEI_ModPrimary represents the "most important" modifier key, cEI_ModSecondary the next most important and so on.

Currently, on the Macintosh cEI_ModPrimary maps to the command key, cEI_ModSecondary maps to the option key and cEI_ModTernary maps to the control key. On the PC, cEI_ModPrimary maps to the control key, cEI_ModSecondary maps to the Alt key and cEI_ModTernary is not mapped.

NOTE: f the user clicks the right mouse button on a Windows PC, the cEI_ControlKey (as well as cEI_ModPrimary) bit will be set even if the control key is not pressed.

Push Button Layout Constants

```
enum {
        cEI_Left,
        cEI_Top,
        cEI_Right,
        cEI_Bottom,
        cEI_Center
};
```

The constants defined by this enumeration can be passed to the function EI_SetPushButtonLayout, and are used to position the text and icon of a push button.

Mouse Moved Constants

```
enum {
        cEI_MouseEnter = 1 << 0,
        cEI_MouseWithin = 1 << 1,
        cEI_MouseExit = 1 << 2
};
```

The constants defined by this enumeration are passed to your mouse moved function to tell you how the cursor has moved. If the value is cEI_MouseEnter, the cursor has just entered your control. If the value is cEI_MouseWithin, the cursor has moved within the control. If the value is cEI_MouseExit, the cursor has just left the control.

Types

This section describes the types available to the clients of the API, beginning with the non-opaque types and followed by the opaque types. The types described in this section are defined in the header file EI_Types.h.

EI_Rect

```
typedef struct {
        int             left;
        int             top;
        int             right;
        int             bottom;
} EI_Rect;
```

EI_Rect represents an axis-aligned rectangle. Many drawing functions take EI_Rects as arguments.

EI_Color

```
typedef struct {
        unsigned char           red;
        unsigned char           green;
        unsigned char           blue;
} EI_Color;
```

EI_Color represents a single RGB color. The values of each member may be in the range [0, 255], where zero represents the complete absence of the corresponding color component, and 255 represents the maximum amount of that color component.

EI_32BitPixel

```
typedef unsigned SInt32 EI_32BitPixel;
```

This type represents the pixels contained in an image buffer. You can use the pixel access macros described in the section “Image Buffers” for accessing the color components of a pixel, or you can use the image buffer pixel constants if you need lower level component access.

EI_PenMode

```
typedef enum {
        cEI_PenColor,
        cEI_PenXOR
} EI_PenMode;
```

EI_PenMode represents the two styles of drawing that can be done using a drawing context object. When the pen mode is cEI_PenColor, the drawing uses the foreground color directly for drawing. When the pen mode is cEI_PenXOR, however, the drawing causes the pixels in the destination to be inverted.

EI_ControlType

```
typedef enum {
        cEI_NoControlType,
        cEI_PushButton,
        cEI_CheckBox,
        cEI_RadioGroup,
        cEI_DividerLine,
        cEI_GroupBox,
        cEI_TabGroup,
        cEI_ColorSlider,
        cEI_Scroller,
        cEI_StaticText,
        cEI_EditText,
        cEI_ColorButton,
        cEI_ListBox,
        cEI_PopupMenu,
        cEI_Picture,
        cEI_UserControl,
        cEI_OpenGLControl
} EI_ControlType;
```

The constants defined by this enumeration define the various control types.

EI_EditFilterType

```
typedef enum {
        cEI_NoFilter,
        cEI_UnsignedInt,
        cEI_SignedInt,
        cEI_UnsignedFloat,
        cEI_SignedFloat,
        cEI_Custom
} EI_EditFilterType;
```

The constants defined by this enumeration are used when setting the type of edit filter used with an edit text control.

EI_StreamAccess

```
typedef enum {
        cEI_ReadOnly,
        cEI_WriteOnly,
        cEI_ReadWrite
} EI_StreamAccess;
```

The constants defined by this enumeration are used when creating an EI_Stream for I/O.

EI_ResourceFile

This is an opaque type.

EI_ResourceFile represents an open resource file. You can get resources, including dialog box definitions and disk images, from a resource file.

EI_DrawContext

This is an opaque type.

EI_DrawContext represents a drawing context object. Using an EI_DrawContext you can draw lines, text, rectangles, ovals and image buffers.

EI_ImageBuffer

This is an opaque type.

EI_ImageBuffer represents an off-screen graphics buffer. You can draw to the off-screen buffer by creating a drawing context object for it, and you can copy pixels to or from an off-screen buffer to a context. You can also manipulate the pixels of an image buffer directly

EI_DiskImage

This is an opaque type.

EI_DiskImage represents an image that is read from a resource file. The types of images which can be read into an EI_DiskImage are

- ‘PICT’
- ‘argb’
- ‘cicn’

EI_Control

This is an opaque type.

EI_Control represents a single control inside a dialog box.

EI_Dialog

This is an opaque type.

EI_Dialog represents a dialog box that is read from a resource file and displayed on screen.

EI_DirectoryRef

This is an opaque type.

EI_DirectoryRef represents a cross-platform reference to a directory on a user’s file system.

EI_FileRef

This is an opaque type.

EI_FileRef represents a cross-platform reference to a file on a user’s file system. A file reference contains a directory reference, a file name, and a file type and extension.

EI_Stream

This is an opaque type.

EI_Stream is used for performing binary, endian-correct file I/O.

EI_Timer

This is an opaque type.

EI_Timer is used for writing timer functions, functions that get called periodically when the system is idle.

EI_Cursor

This is an opaque type.

EI_Cursor is used to represent the image displayed on the cursor.

API Version

The function described in this section is declared in the header file EI_API.h.

EI_GetAPIVersion

```
int EI_GetAPIVersion(void);
```

return value	an integer value

EI_GetAPIVersion returns the version number of the API that is currently executing. This value may differ from the value of the constant cEI_CurrentVersion if the plugin is running in a host which contains a different version of the API than the version being used to compile the plugin.

Dialog Boxes

The functions described in this section allow you to create modal dialog boxes. The functions described in this section are declared in the header file EI_Dialog.h.

EI_MakeDialog

```
EI_Dialog *EI_MakeDialog(
                    EI_ResourceFile *iResourceFile,
                    int              iDialogID);
```

iResourceFile	a pointer to a resource file object
iDialogID	an integer which must be the ID of a dialog definition resource in the resource file indicated by iResourceFile
return value	a pointer to a dialog box object

EI_MakeDialog constructs a modal dialog box by reading the dialog definition resource iDialogID from iResourceFile. If the dialog box cannot be constructed, EI_MakeDialog returns NULL. The dialog box is initially invisible, and will remain invisible until you call EI_ExecuteDialog.

EI_DestroyDialog

```
void EI_DestroyDialog(EI_Dialog *iDialog);
```

iDialog	a pointer to a dialog box object

EI_DestroyDialog destroys a dialog box previously created with EI_MakeDialog.

EI_SetDialogTitle

```
void EI_SetDialogTitle(
        EI_Dialog  *ioDialog,
        const char *iDialogTitle);
```

ioDialog	a pointer to a dialog box object
iDialogTitle	a pointer to a C string

EI_SetDialogTitle changes the title of the modal dialog box’s window to the string given by iDialogTitle.

EI_MoveDialog

```
void EI_MoveDialog(
        EI_Dialog *ioDialog,
        int        iNewLeft,
        int        iNewTop);
```

ioDialog	a pointer to a dialog box object
iNewLeft	a horizontal coordinate on the screen
iNewTop	a vertical coordinate on the screen

EI_MoveDialog moves the dialog box’s position on screen so that its upper left corner is moved to the point (iNewLeft, iNewTop).

EI_GetDialogPosition

```
void EI_GetDialogPosition(
        const EI_Dialog *iDialog,
        int             *oLeft,
        int             *oTop);
```

iDialog	a pointer to a dialog box object
oLeft	a pointer to an int
oTop	a pointer to an int

EI_GetDialogPosition returns in *oLeft and *oTop the current location of the upper left corner of *iDialog on the screen.

EI_SetDialogSize

```
void EI_SetDialogSize(
        EI_Dialog *iDialog,
        int        iWidth,
        int        iHeight);
```

iDialog	a pointer to a dialog box object
iWidth	the new width
iHeight	the new height

EI_SetDialogSize resizes the dialog box to the size given by iWidth and iHeight.

EI_GetDialogSize

```
void EI_GetDialogSize(
        const EI_Dialog *iDialog,
        int             *oWidth,
        int             *oHeight);
```

iDialog	a pointer to a dialog box object
oWidth	a pointer to an int
oHeight	a pointer to an int

EI_GetDialogSize returns in *oWidth and *oHeight the size of the dialog box.

EI_GetMonitorBounds

```
void EI_GetMonitorBounds(EI_Rect *oMonitorBounds);
```

oMonitorBounds	a pointer to an EI_Rect

EI_GetMonitorBounds returns in oMonitorBounds the boundary rectangle of the main monitor. This boundary rectangle is somewhat platform-dependent in nature. On the Macintosh, it will be the boundary rectangle of the main screen, minus the menu bar. On Windows, it will be the boundary rectangle of the root window of the running application.

EI_SetDialogExtraData - *Not 64-bit compatible*

```
void EI_SetDialogExtraData(
        EI_Dialog *ioDialog,
        void       *iExtraData);
```

ioDialog	a pointer to a dialog box object
iExtraData	a pointer

EI_SetDialogExtraData sets the extra data pointer of a dialog box object. The value passed in iExtraData is not dereferenced or used in any way by the API, so you are free to use this argument in any way you like. You can pass a pointer to a structure containing extra data for your dialog box. During hit functions or other callbacks, you can extract the extra data pointer using the EI_GetDialogExtraData function.

NOTE: If the extra data pointer points to dynamically allocated memory, the API will not free that memory for you when the dialog box is destroyed; you are responsible for freeing any memory you allocate when you are through using it.

EI_GetDialogExtraData - **Not 64-bit compatible**

```
void *EI_GetDialogExtraData(const EI_Dialog  *iDialog);
```

iDialog	a pointer to a dialog box object
return value	a pointer

EI_GetDialogExtraData returns the extra data pointer currently assigned to *iDialog. If you have not yet assigned an extra data pointer to *iDialog, EI_GetDialogExtraData returns NULL.

EI_ExecuteDialog

```
int EI_ExecuteDialog(EI_Dialog *ioDialog);
```

ioDialog	a pointer to a dialog box object
return value	a boolean value

EI_ExecuteDialog shows the given dialog and allows the user to interact with it. Any callbacks you have installed into the dialog or its controls will be executed at the appropriate times. EI_ExecuteDialog will not return until the user dismisses the dialog. If the user dismisses the dialog by clicking the OK button, EI_ExecuteDialog returns true (non-zero); if the user dismisses a dialog by clicking the Cancel button, EI_ExecuteDialog returns false (zero). Also, if one of your callbacks calls EI_StopDialog, the boolean result value passed to EI_StopDialog will be returned by EI_ExecuteDialog.

NOTE: If the user clicks the OK button and there is a validation function in your dialog box, the validation function must return a true result for the dialog box to actually be dismissed.

EI_ShowDialogModeless

```
int EI_ShowDialogModeless(EI_Dialog *iDialog);
```

ioDialog	a pointer to a dialog box object

EI_EShowDialogModeless shows the given dialog and allows the user to interact with it. Any callbacks you have installed into the dialog or its controls will be executed at the appropriate times. Unlike EI_ExecuteDialog, EI_ShowDialogModeless returns control immediately. The developer is responsible for closing the modeless dialog. It is necessary to call EI_HideDialogModeless before calling EI_DestroyDialog for all modeless dialogs.

EI_HideDialogModeless

```
int EI_HideDialogModeless(EI_Dialog *iDialog);
```

ioDialog	a pointer to a dialog box object

EI_EHideDialogModeless hides a modeless dialog previously shown with EI_ShowDialogModeless. It is necessary to call EIHideDialogModeless before calling EI_DestroyDialog for all modeless dialogs.

EI_StopDialog

```
void EI_StopDialog(
        EI_Dialog *ioDialog,
        int        iResult);
```

ioDialog	a pointer to a dialog box object
iResult	a boolean value

EI_StopDialog may be called from within a dialog or control callback function to shut down the dialog box. The boolean value contained in iResult will be returned from the enclosing call to EI_ExecuteDialog.

NOTE: You cannot depend on the exact integer value in iResult being returned by EI_ExecuteDialog. The return value of EI_ExecuteDialog, and the iResult argument of EI_StopDialog, are booleans. The API only guarantees that the result will be zero or non-zero.

NOTE: If you pass a true value in iResult and there is a validation function in your dialog box, the validation function must return a true result for the dialog box to actually shut down.

EI_SetDialogValidationFunction

```
void EI_SetDialogValidateFunction(
        EI_Dialog                  *ioDialog,
        EI_DialogValidationFunction iFunction);
```

ioDialog	a pointer to a dialog box object
iFunction	a pointer to a dialog box validation function

EI_SetDialogValidationFunction sets the dialog box's validation function. Whenever the user presses the OK button in your dialog box, it may not be appropriate for the dialog box to be dismissed. For example, if the value entered into an edit text field is out of range, the user must be told to correct the entry. The validation function allows you to check that the data entered is valid.

The validation function is called whenever the user presses the OK button, or you call EI_StopDialog and you pass a true value in iResult. In either case, if the validation function returns false, the dialog is not dismissed. If the user clicks the cancel button, or EI_StopDialog is called with a false result, the validation function is not called.

EI_SetDialogHitFunction - Not 64-bit compatible

```
void EI_SetDialogHitFunction(
       EI_Dialog           *ioDialog,
       EI_DialogHitFunction iFunction);
```

ioDialog	a pointer to a dialog box object
iFunction	a pointer to a dialog hit callback function

EI_SetDialogHitFunction sets *ioDialog's current dialog hit callback function to iFunction. To clear the dialog hit function, pass NULL in iFunction.

The dialog's hit function is called if the user clicks in "empty space" in the dialog; that is, if the user clicks the mouse button when the cursor is not within a control.

EI_SetDialogKeyFilterFunction - Not 64-bit compatible

```
void EI_SetDialogKeyFilterFunction(
       EI_Dialog         *ioDialog,
       EI_DialogKeyFilter iFunction);
```

iDialog	a pointer to a dialog box object
iFunction	a pointer to a dialog key down filter function

EI_SetDialogKeyFilterFunction sets *ioDialog's current dialog key down filter function to iFunction.
The dialog's key down filter function is called whenever there is a key down event for the dialog, and before the key down event is given to the key board focus. The filter function returns a boolean result; if the filter function returns true, the key down event is not given to the key focus.

Callbacks

EI_DialogValidationFunction

```
int MyDialogValidationFunction(EI_Dialog *ioDialog);
```

iDialog	a pointer to a dialog box object
return value	a boolean value

The dialog validation function is called when the user attempts to dismiss the dialog box by clicking the OK button. Your validation function can examine the data entered into the dialog box and decide if it is valid or not. For example, the value entered into a text field might need to be in a certain range in order for the dialog to be filled out correctly. If the validation function decides the dialog is not correct, it should return false, which will cause the dialog box to remain on screen. If the dialog is correctly filled out, the validation function should return true, which will cause the dialog box to shut down and return true from EI_ExecuteDialog.

Note that if your function is going to return false, it should probably tell the user what went wrong by presenting another dialog box with a message in it. Also, if you want to further direct the user's attention to the problem, you can call EI_SetKeyControl to set the keyboard focus control to the field that was entered incorrectly.

EI_DialogHitFunction

```
void MyDialogHitFunction(EI_Dialog* iDialog);
```

iDialog	a pointer to a dialog box object

When the mouse is clicked in the dialog *iDialog, but not in any control, the dialog's hit function is called.

EI_DialogKeyFilter

```
int MyDialogKeyFilter(
	EI_Dialog *iDialog,
	int       iKey,
	int       iModifiers);
```

iDialog	a pointer to a dialog box object
iKey	a character value
iModifiers	an integer value
return value	a boolean value

If a dialog box has a key down event filter, then whenever a key is pressed, the filter is called before giving the key to the key focus control. The return value tells the host application whether or not the filter function handled the key down completely, and thus whether or not to give the key down to the key focus control. A return value of true tells the host application that the filter function handled the key down event and that the key focus control should not get it. A return value of false tells the host application that the filter function did not handle the event, and that the key focus control should handle it.

You can use the key down event filter to implement custom accelerator keys.

Controls

The functions described in this section allow you to manipulate controls in dialog boxes. The functions described in this section are declared in the file EI_Control.h.

General Control Functions

EI_GetControlID

```
int EI_GetControlID(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a control ID number

EI_GetControlID returns the control's ID number.

EI_SetControlID

```
void EI_SetControlID(
        EI_Control *ioControl,
        int        iNewID);
```

ioControl	a pointer to a control object
iNewID	a new control ID number

EI_SetControlID sets the control’s ID number to iNewID.

EI_CountControls

```
int EI_CountControls(const EI_Dialog *iDialog);
```

iDialog	a pointer to a dialog box object
return value	the number of controls in the dialog box

EI_CountControls returns the number of controls contained in *iDialog.

EI_FindControlByIndex

```
EI_Control *EI_FindControlByIndex(
        const EI_Dialog *iDialog,
        int              iControlNumber);
```

iDialog	a pointer to a dialog box object
iControlNumber	a control index
return value	a pointer to a control object

EI_FindControlByIndex returns a pointer to the iControlNumber’th control contained in *iDialog. If iControlNumber is not a valid index, EI_FindControlByIndex returns NULL.

Using EI_CountControls and EI_FindControlByIndex, you can iterate over all of the controls contained in a given dialog box.

EI_FindControlByID

```
EI_Control *EI_FindControlByID(
        const EI_Dialog *iDialog,
        int              iControlID);
```

iDialog	a pointer to a control object
iControlID	a control ID
return value	a pointer to a control object

EI_FindControlByID will return a pointer to the control contained in *iDialog with ID equal to iControlID. If no control has this ID, EI_FindControlByID returns NULL.

EI_InvalControl

```
void EI_InvalControl(const EI_Control *iControl);
```

iControl	a pointer to a control object

EI_InvalControl will force the control to be redrawn. Normally, you should not need to call this function for most controls, since changing control attributes will cause the correct redrawing to occur automatically. But you may need to call this if you modify data associated with a user control and drawn by the user control's draw procedure.

EI_GetControlDialog

```
EI_Dialog *EI_GetControlDialog(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a pointer to a dialog box object

EI_GetControlDialog returns a pointer to the dialog box object which contains the control.

EI_GetControlType

```
EI_ControlType EI_GetControlType(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a control type

EI_GetControlType returns the type of the control.

EI_GetControlBounds

```
void EI_GetControlBounds(
        const EI_Control *iControl,
        EI_Rect          *oBounds);
```

iControl	a pointer to a control object
oBounds	a pointer to an EI_Rect

EI_GetControlBounds sets *oBounds to the boundary rectangle of *iControl.

EI_SetControlBounds

```
void EI_SetControlBounds(
        EI_Control    *ioControl,
        const EI_Rect *iBounds);
```

ioControl	a pointer to a control object
iBounds	a pointer to an EI_Rect

EI_SetControlBounds sets *ioControl’s boundary rectangle to *iBounds. This function can be used to change the position or size of a control.

EI_SetControlDrawImmediate

```
void EI_SetControlDrawImmediate(
       EI_Control *ioControl,
       int        iDrawImmediateFlag);
```

ioControl	a pointer to a control object
iDrawImmediateFlag	a boolean value

EI_SetControlDrawImmediate sets the control’s immediate draw flag to the value of iDrawImmediateFlag. If a control’s immediate draw flag is true, then when changing attributes of the control (such as its value or its title), it is redrawn immediately; otherwise, the control is invalidated, as if you had called EI_InvalControl.

Usually, it is best to leave a control’s flag set to false. However, sometimes you can’t wait until the next window update event is handled. For example, if the user clicks on a slider and drags the thumb and you want to update a static text with the value of the slider, you would install a callback into the slider which gets called while the user drags the thumb. Your callback obtains the value of the slider, converts it to a string, and then calls EI_SetControlTitle to set the text of the static text control. You must therefore set the control’s immediate draw flag to true so that the static text control is drawn when you call EI_SetControlTitle; otherwise it won’t be redrawn until the next window update event is processed, after the user releases the mouse button.

EI_GetControlDrawImmediate

```
int EI_GetControlDrawImmediate(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a boolean value

EI_GetControlDrawImmediate returns the value of the controls immediate draw flag.

EI_SetControlVisible

```
void EI_SetControlVisible(
       EI_Control *ioControl,
       int        iVisible);
```

ioControl	a pointer to a control object
iVisible	a boolean value

EI_SetControlVisible can show or hide a control. If iVisible is true, the control is made visible. If iVisible is false, the control is made invisible. In either case the dialog box containing the control is updated appropriately.

EI_GetControlVisible

```
int EI_GetControlVisible(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a boolean value

EI_GetControlVisible returns true if the *iControl is visible and false if *iControl is invisible.

EI_SetControlEnable

```
void EI_SetControlEnable(
        EI_Control *ioControl,
        int         iEnabled);
```

ioControl	a pointer to a control object
iEnabled	a boolean value

EI_SetControlEnable sets the control’s enable flag. The enable flag determines whether or not the control responds to events, and also influences the appearance of the control. Most controls, when disabled, appear “grayed out.” If iEnabled is true, the control will be enabled, and if iEnabled is false, the control will be disabled. In either case the dialog box containing the control is updated appropriately.

EI_GetControlEnable

```
int EI_GetControlEnable(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a boolean value

EI_GetControlEnable returns the value of the control’s enable flag.

EI_SetControlTitle

```
void EI_SetControlTitle(
        EI_Control *ioControl,
        const char *iTitleString);
```

ioControl	a pointer to a control object
iTitleString	a pointer to a C-string

EI_SetControlTitle sets the control’s title to the given string.

EI_GetControlTitleSize

```
int EI_GetControlTitleSize(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	the number of characters in the control’s title

EI_GetControlTitleSize returns the number of characters in the control’s title, including the terminating zero byte.

EI_GetControlTitle

```
void EI_GetControlTitle(
        const EI_Control *iControl,
        char             *oText,
        int              *ioNumChars);
```

iControl	a pointer to a control object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetControlTitle copies the control’s title string to the buffer pointed at by oText. If the title is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetControlTitle will copy as many characters as will fit, including a terminating zero. EI_GetControlTitle will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_GetControlValue

```
int EI_GetControlValue(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	an integer value

EI_GetControlValue will return the value of the given control.

EI_SetControlValue

```
void EI_SetControlValue(
        EI_Control *ioControl,
        int         iNewValue);
```

ioControl	a pointer to a control object
iNewValue	an integer value

EI_SetControlValue will change the control’s value to iNewValue. If iNewValue is greater than the control’s maximum, it will be truncated to the maximum value.

EI_GetControlMax

```
int EI_GetControlMax(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	an integer value

EI_GetControlMax returns the current maximum value of the control.

EI_SetControlMax

```
void EI_SetControlMax(
        EI_Control *ioControl,
        int         iNewMax);
```

ioControl	a pointer to a control object
iNewMax	an integer value

EI_SetControlMax sets the control’s maximum value to iNewMax.

EI_SetControlExtraData - **Not 64-bit compatible**

```
void EI_SetControlExtraData(
        EI_Control *ioControl,
        void        *iExtraData);
```

ioControl	a pointer to a control object
iExtraData	a pointer

EI_SetControlExtraData sets the extra data pointer of a control object. The value passed in iExtraData is not dereferenced or used in any way by the API, so you are free to use this argument in any way you like. You can pass a pointer to a structure containing extra data for your control. During hit functions or other callbacks, you can extract the extra data pointer using the EI_GetControlExtraData function.

NOTE: If the extra data pointer points to dynamically allocated memory, the API will not free that memory for you when the control is destroyed; you are responsible for freeing any memory you allocate when you are through using it.

EI_GetControlExtraData - **Not 64-bit compatible**

```
void *EI_GetControlExtraData(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	a pointer

EI_GetControlExtraData returns the extra data pointer currently assigned to *iControl. If you have not yet assigned an extra data pointer to *iControl, EI_GetControlExtraData returns NULL.

Electric Image Animation System

EI_SetControlHitFunction - Not 64-bit compatible

```
void EI_SetControlHitFunction(
       EI_Control          *ioControl,
       EI_ControlHitFunction iFunction);
```

ioControl	a pointer to a control object
iFunction	a pointer to a control hit function

EI_SetControlHitFunction sets the hit function of a control to iFunction.

Control Images

EI_GetControlImageID

```
int EI_GetControlImageID(const EI_Control *iControl);
```

iControl	a pointer to a control object
return value	an image ID

EI_GetControlImageID returns the resource ID of the image used by *iControl. This function applies to push buttons and pictures.

EI_SetControlImageID

```
void EI_SetControlImageID(
       EI_Control *ioControl,
       int         iNewImageID);
```

ioControl	a pointer to a control object
iNewImageID	an image ID

EI_SetControlImageID changes the image used by *ioControl to the image indicated by iNewImageID. The value of iNewImageID must the ID of an existing ‘argb’, ‘cicn’ or ‘PICT’ resource.

NOTE: Use IDs in the range 12000 to 18000 for your images, or you may collide with images defined by Electric Image.

Push Button Layout

EI_GetPushButtonLayout

```
void EI_GetPushButtonLayout(
       const EI_Control *iControl,
       int              *oLabelPosition,
       int              *oIconPosition);
```

iControl	a pointer to a push button control
oLabelPosition	a pointer to an int
oIconPosition	a pointer to an int

EI_GetPushButtonLayout returns, in *oLabelPosition and *oIconPosition, the layout values used by *iControl. Push buttons display a label that can contain an icon, a title string, or both. The value returned in *oLabelPosition controls the position of the label, and can be any one of cEI_Left, cEI_Top, cEI_Right, cEI_Bottom, or cEI_Center. For cEI_Left, the label is positioned against the left side of the button, for the cEI_Top, the label is positioned against the top side of the button, and so on. For cEI_Center (the default), the label is placed in the center of the button.

The value returned in *oIconPosition controls the position of the icon with respect to the title string. This value is meaningful only if the label contains an icon and a title string, and can be any one of cEI_Left, cEI_Top, cEI_Right, or cEI_Bottom. For cEI_Left, the icon will be positioned to the left of the title string, for cEI_Top, the icon will be positioned above the title string, and so on. The value cEI_Center has no meaning for *oIconPosition.

EI_SetPushButtonLayout

```
void EI_SetPushButtonLayout(
       EI_Control *ioControl,
       int        iLabelPosition,
       int        iIconPosition);
```

ioControl	a pointer to a push button control
iLabelPosition	a layout value
iIconPosition	a layout value

EI_SetPushButtonLayout lets you change the label's layout values. The meaning of the layout values is described above in the description of the function EI_GetPushButtonLayout.

Keyboard Focus Controls

EI_SetKeyControl

```
void EI_SetKeyControl(EI_Control *iControl);
```

iControl	a pointer to a keyboard focus control object

EI_SetKeyControl makes *iControl become the current keyboard focus for its dialog box. If *iControl is not a valid keyboard focus control (either an edit text control or a list box control), EI_SetKeyControl has no effect.

EI_GetKeyControl

```
EI_Control *EI_GetKeyControl(const EI_Dialog *iDialog);
```

iDialog	a pointer to a dialog box object
return value	a pointer to a keyboard focus control object

EI_GetKeyControl returns the control object which is the current keyboard focus in the given dialog box. If there is no keyboard focus for the dialog, EI_GetKeyControl returns NULL.

Edit Text Controls

EI_SetEditTextString

```
void EI_SetEditTextString(
        EI_Control *ioControl,
        const char *iEditString);
```

ioControl	a pointer to an edit text control object
iEditString	a pointer to a C string

EI_SetEditTextString changes the text in the edit box of an edit text control to the string passed in iEditString.

EI_GetEditTextStringSize

```
int EI_GetEditTextStringSize(const EI_Control *iControl);
```

iControl	a pointer to an edit text control object
return value	the number of characters in the control's edit box

EI_GetEditTextStringSize returns the number of characters in an edit text control's edit box, including the terminating zero byte.

EI_GetEditTextString

```
void EI_GetEditTextString(
        const EI_Control *iControl,
        char             *oText,
        int              *ioNumChars);
```

iControl	a pointer to an edit text control object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetEditTextString copies the text from an edit control's edit box to the buffer pointed at by oText. If the text is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetEditTextString will copy as many characters as will fit, including a terminating zero. EI_GetEditTextString will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_SetEditTextSelection

```
void EI_SetEditTextSelection(
        EI_Control *ioControl,
        int        iSelectionStart,
        int        iSelectionEnd);
```

ioControl	a pointer to an edit text control object
iSelectionStart	an integer value
iSelectionEnd	an integer value

EI_SetEditTextSelection changes the selection range of an edit text control to the range given by iSelectionStart and iSelectionEnd. Both values are indexes into the string, and begin at zero. If iSelectionStart or iSelectionEnd indicates a position after the end of the text, the last text position will be used in its place.

EI_GetEditTextSelection

```
void EI_GetEditTextSelection(
        const EI_Control *ioControl,
        int              *oSelectionStart,
        int              *oSelectionEnd);
```

ioControl	a pointer to an edit text control object
oSelectionStart	a pointer to an integer value
oSelectionEnd	a pointer to an integer value

EI_GetEditTextSelection sets *oSelectionStart and *oSelectionEnd to the edit text control's current selection range.

EI_SetEditTextFilter - Not 64-bit compatible

```
void EI_SetEditTextFilter(
        EI_Control           *ioControl,
        EI_EditFilterType     iFilterType,
        EI_EditFilterFunction iFilterFunction);
```

ioControl	a pointer to an edit text control object
iFilterType	a filter type constant
iFilterFunction	a pointer to an edit text filter function

EI_SetEditTextFilter changes the filter used with an edit text control. The value passed in iFilterType determines how the edit text will filter text typed by an end user.

The meanings of the EI_EditFilterType constants are:

cEI_NoFilter	The edit text control performs no filtering; all text typed by the user is accepted into the control.
cEI_UnsignedInt	The edit text control only allows an unsigned integer value to be typed into the control.
cEI_SignedInt	The edit text control only allows an integer value to be typed into the control, with an optional sign.
cEI_UnsignedFloat	The edit text control only allows an unsigned floating point number to be typed into the control.
cEI_SignedFloat	The edit text control only allows a floating point number to be typed into the control, with an optional sign.

cEI_Custom	The edit text calls the filter function pointed to by iFilterFunction to provide filtering. See the discussion of EI_EditFilterFunction callback functions.

NOTE: If iFilterType is cEI_Custom and the iFilterFunction pointer is NULL, the edit text control performs no filtering; all text typed by the user is accepted into the control.

Color Button Controls

EI_GetColorButtonColor

```
void EI_GetColorButtonColor(
        const EI_Control *iControl,
        int              *oRed,
        int              *oGreen,
        int              *oBlue,
        int              *oAlpha);
```

iControl	a pointer to a color button control object
oRed	a pointer to an integer value
oGreen	a pointer to an integer value
oBlue	a pointer to an integer value
oAlpha	a pointer to an integer value

EI_GetColorButtonColor sets *oRed, *oGreen, *oBlue and *oAlpha to the current color value in the color button control. The values will all be in the range 0 to 255.

Any or all of the integer pointers can be NULL. If a pointer is NULL, it is ignored. This way you can ask for only the components you are interested in. For example, if you only want the RGB part but don't care about the alpha, you can pass NULL as oAlpha.

EI_SetColorButtonColor

```
void EI_SetColorButtonColor(
        EI_Control *ioControl,
        int         iRed,
        int         iGreen,
        int         iBlue,
        int         iAlpha);
```

ioControl	a pointer to a color button control object
iRed	an integer value
iGreen	an integer value
iBlue	an integer value
iAlpha	an integer value

EI_SetColorButtonColor changes the color of a color button to the color given by iRed, iGreen, iBlue and iAlpha. The values are expected to be in the range 0 to 255. If a value is greater than 255, it will be truncated to 255. If any value is less than zero, that argument will be ignored and will leave the corresponding component in the color button unchanged. For example, if you want to change the color to a particular RGB value but don't want to disturb the alpha, you can pass -1 as iAlpha.

Tab Controls

The functions in this section allow you to change the current tab displayed in a tab control.

NOTE: Group controls can also have multiple "branches." Group branches are similar to tab panes in a tab control, in that only one group branch, and the controls within it, are visible at a time. The tab control functions described in this section also work with groups containing multiple branches. Group branches can be edited in Interface Builder.

EI_GetNumTabPanels

```
int EI_GetNumTabPanels(const EI_Control *iControl);
```

iControl	a pointer to a tab control
return value	the number of tab panels in the tab control

EI_GetNumTabPanels returns the number of tab panels in a tab control.

EI_GetCurrentTabPanel

```
int EI_GetCurrentTabPanel(const EI_Control *iControl);
```

iControl	a pointer to a tab control
return value	the index of the tab panel currently displayed in the tab control

EI_GetCurrentTabPanel returns the index of the tab control's current tab panel. The index is zero based.

EI_SetCurrentTabPanel

```
void EI_SetCurrentTabPanel(
        EI_Control *ioControl,
        int         iPanelNumber);
```

ioControl	a pointer to a tab control
iPanelNumber	a tab panel index

EI_SetCurrentTabPanel switches the tab control's current tab panel to the tab panel indicated by iPanelNumber.

List Box Controls

EI_CountListItems

```
int EI_CountListItems(const EI_Control *iControl);
```

iControl	a pointer to a list box control object
return value	an integer value

EI_CountListItems returns the number of items in a list box control.

EI_InsertListItem

```
void EI_InsertListItem(
        EI_Control *ioControl,
        int        iBeforeWhichItem,
        const char *iItemString);
```

ioControl	a pointer to a list box control object
iBeforeWhichItem	an item index
iItemString	a pointer to a C string

EI_InsertListItem adds a string to a list box. The new item is inserted before the item whose index is passed in iBeforeWhichItem, and the string is passed in iItemString.

EI_InsertListItemEx

```
void EI_InsertListItemEx(
EI_Control          *ioControl,
int                  iBeforeWhichItem,
const char          *iItemString,
int                  iItemID,
int                  iItemFlag,
int                  iItemStyle,
Const EI_ImageBuffer *iItemIcon);
```

ioControl	a pointer to a list box control object
iBeforeWhichItem	an item index
iItemString	item's text (C string)
iItemID	ID associated with the item (see EIGetListItemID)
iItemFlag	item's flags (See EI_SetListItemFlags)
iItemStyle	Item's text style
iItemIcon	item's icon image buffer (can be null)

EI_InsertListItemEx inserts a new item into a list.

EI_GetListItemID

```
void EI_GetListItemID(
        EI_Control *ioControl,
        int        iWhichItem);
```

ioControl	a pointer to a list box control object
iWhichItem	an item index

EI_GetListItemID reutnrs an ID of an item previously created with InsertListItemEx. It is an easy way to operate with items regardless of the their order.

EI_RemoveListItem

```
void EI_RemoveListItem(
        EI_Control *ioControl,
        int         iWhichItem);
```

ioControl	a pointer to a list box control object
iWhichItem	an item index

EI_RemoveListItem removes an item from a list box. The index of the item to remove is passed in iWhichItem.

EI_SetListString

```
void EI_SetListString(
        EI_Control *ioControl,
        int         iWhichItem,
        const char *iItemString);
```

ioControl	a pointer to a list box control object
iWhichItem	an item index
iItemString	a pointer to a C string

EI_SetListString changes the text of an existing list item to the string passed in iItemString. The index of the item to change is passed in iWhichItem.

EI_GetListStringSize

```
int EI_GetListStringSize(
        const EI_Control *iControl,
        int               iWhichItem);
```

iControl	a pointer to a list box control object
iWhichItem	an item index
return value	an integer value

EI_GetListStringSize returns the number of characters in an item of a list box control, including the terminating zero byte. The index of the item is passed in iWhichItem.

EI_GetListString

```
void EI_GetListString(
        const EI_Control *iControl,
        int               iWhichItem,
        char             *oText,
        int              *ioNumChars);
```

iControl	a pointer to a list box control object
iWhichItem	an item index
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetListString copies the string from the list box item whose index is passed in iWhichItem to the buffer pointed at by oText. If the text is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetListString will copy as many characters as will fit, including a terminating zero. EI_GetListString will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_SetListItemStyle

```
void EI_SetListItemStyle(
        EI_Control *ioControl,
        int         iWhichItem,
        int         iStyleFlags);
```

ioControl	a pointer to a list box control object
iWhichItem	an item index
iStyleFlags	an integer value

EI_SetListItemStyle sets the font style flags of the list box item whose index is passed in iWhichItem to iStyleFlags. The style flags can be any bit-wise combination of cEI_TextBold, cEI_TextItalic, cEI_TextUnderline, cEI_TextOutline, and cEI_TextShadow. To remove all stylistic variation from the list box item, pass 0 as iStyleFlags.

EI_GetListItemStyle

```
int EI_GetListItemStyle(
        const EI_Control *iControl,
        int               iWhichItem);
```

iControl	a pointer to a list box control object
iWhichItem	an item index
return value	an integer value

EI_GetListItemStyle returns the font style flags of the list box item whose index is passed in iWhichItem.

EI_GetListSelectedItem

```
int EI_GetListSelectedItem(const EI_Control *iControl);
```

iControl	a pointer to a list box control object
return value	an item index

EI_GetListSelectedItem returns the index of the currently selected item. If more than one item in the list is selected, the index returned is the index of the first item in the list that is selected. For example, if items 5 and 10 are selected, EI_GetListSelectedItem will return 5. If no item is selected, EI_GetListSelectedItem returns -1.

EI_SelectListItem

```
void EI_SelectListItem(
        EI_Control *ioControl,
        int         iWhichItem,
        int         iSelect);
```

ioControl	a pointer to a list box control object
iWhichItem	an item index
iSelect	a boolean value

EI_SelectListItem either selects or deselects an item in the list box control. The index of the item is passed in iWhichItem. If iSelect is true, the item is selected. If iSelect is false, the item is deselected.

EI_IsListItemSelected

```
int EI_IsListItemSelected(
        const EI_Control *iControl,
        int               iWhichItem);
```

iControl	a pointer to a list box control object
iWhichItem	an item index
return value	a boolean value

EI_IsListItemSelected returns true if the item whose index is iWhichItem is selected and false if it is not selected.

EI_SetListFlags

```
void EI_SetListFlags(
        EI_Control *ioControl,
        int         iListFlags);
```

ioControl	a pointer to a list box control object
iListFlags	an integer value

EI_SetListFlags sets the list box control's flags to the flags passed in iListFlags.

EI_GetListFlags

```
int EI_GetListFlags(const EI_Control *iControl);
```

iControl	a pointer to a list box control object
return value	an integer value

EI_GetListFlags returns the current value of a list box control's flags.

EI_SetListItemFlags

```
void EI_SetListItemFlags(
        EI_Control *ioControl,
        int        iWhichItem,
        int        iFlags
        int        iMask);
```

ioControl	a pointer to a list box control object
iWhichItem	index of item to set
iFlags	set of following flags (EI_Types.h)

cEI_ItemSelected
cEI_ItemEnabled
cEI_ItemLocked

iMask	set of flags listed above. If a bit of iMsak is set, then the list items attribute will be modified according to the iFlags bit. Otherwise the attribute remains unchanged.

EI_SetListItemFlags item's states: enabled, locked and selected.

EI_GetListItemFlags

```
int EI_GetListItemFlags(
        const EI_Control *iControl,
        int               iWhichItem);
```

iControl	a pointer to a list box control object
iWhichItem	index of item to set
return value	set of the following flags (EI_Types.h)

cEI_ItemSelected
cEI_ItemEnabled
cEI_ItemLocked

EI_GetListItemFlags returns the current state of the list item: enabled, locked and selected.

EI_SetListItemIcon

```
int EI_setListItemIcon(
        const EI_Control     *ioControl,
        int                   iWhichItem
        const EI_ImageBuffer *iImage);
```

ioControl	list control
iWhichItem	index of item to set
iImage	icon's image buffer

EI_setListItemIcon sets the list item's icon that appears rightost in the list and is indvidual for each item.

EI_SetListIcon

```
void EI_SetListIcon(
        EI_Control           *ioControl,
        int                   iFlag,
        Const EI_ImageBuffer *iIcon);
```

ioControl	a pointer to a list box control object
iFlag	a flag from EI_Types.h

cEI_ItemEnabled = 1 << 1
cEI_ItemLocked = 1 << 2

iIcon	icon’s image buffer

EI_SetListIcon sets the lists’s “enable” or “locked” icons that are common for all items.

EI_SetListDoubleClickFunction - Not 64-bit compatible

```
void EI_SetListDoubleClickFunction(
        EI_Control                *ioControl,
        EI_ListDoubleClickFunction iFunction);
```

ioControl	a pointer to a list box control object
iFunction	a pointer to a list double-click function

EI_SetListDoubleClickFunction sets the double-click function of *ioControl to iFunction. Whenever the end-user double-clicks an item in the list, the double-click function will be called. To clear the double-click function, pass NULL in iFunction.

EI_SetListReorderFunction - Not 64-bit compatible

```
void EI_SetListReorderFunction(
        EI_Control             *ioControl,
        EI_ListReorderFunction  iFunction);
```

ioControl	a pointer to a list box control object
iFunction	a pointer to a list reorder function

EI_SetListReorderFunction sets the reorder function of *ioControl to iFunction. Whenever the end-user drags an item to a new position in the list, the reorder function will be called. To clear the reorder function, pass NULL in iFunction.

EI_GetListHitInfo

```
void EI_GetListHitInfo(
        EI_Control *iControl,
        int         *oWhichItem,
        Int         *oWichPart,
        int         *oWhereX,
        int         *oWhereY);
```

iControl	a pointer to a list box control object
oWhichItem	hit item (-1 otherwise)
oWhichPart	the item’s part, one of the following (EI_Types.h)

cEI_VisibilityHit	hit in “enabled” list’s column
cEI_LockingHit	hit in “locked” list’s column
cEI_SElectionHit	item’s hit
cEI_IconClickHit	item’s icon hit

oWhereX, oWhereY	the actual mouse position (in dialog coordinates)

EI_GetListHitInfo returns advanced information about what happened when a list item was picked. EI_GetListHitInfo should be called from the ControlHit function when it receives control.

EI_ScrollList2Item

```
void EI_ScrollList2Item(
        EI_Control *ioControl,
        int         iWhichItem);
```

ioControl	a pointer to a list box control object
iWhichItem	item index to scroll to

EI_ScrollList2Item tells the list to adjust the scroll bar to make the specified item in the list visible.

Popup Control

EI_PopupMenuTrack

```
int EI_PopupMenuTrack(
        const EI_Dialog *iiDialog,
        int              imenuID,
        int              iItemSelect,
        int              iItemCheck,
        int             *iWhereX,
        int             *iWhereY);
```

iDialog	dialog window the popup service is attached from
iWhichItem	ID of ‘FMNU resource
iItemSelect	index of initially selected item (0 = none)
iItemCheck	index of initially checked item (0 = none)
iWhereX, iWhereY	mouse position (in dialog coordinates) where the popup appears

return value	high 2 bytes = menu ID low 2 bytes = index of selected item (0 = none)

EI_PopupMenuTrack proveds a floating popup service for plug-ins and shaders.

NOTE: EI_PopupMenuTrack should be called when the mouse button is pressed but not yet released. The typical usage is when the “icon” control is picked.

User Controls

EI_SetUserControlDraw - *Not 64-bit compatible*

```
void EI_SetUserControlDraw(
       EI_Control          *ioControl,
       EI_UserDrawFunction iFunction);
```

ioControl	a pointer to a control object
iFunction	a pointer to a user control draw function

EI_SetUserControlDraw sets the draw function of a user control to the function pointed at by iFunction.

EI_SetUserControlClick - *Not 64-bit compatible*

```
void EI_SetUserControlClick(
       EI_Control           *ioControl,
       EI_UserClickFunction iFunction);
```

ioControl	a pointer to a control object
iFunction	a pointer to a user control click function

EI_SetUserControlClick sets the click function of a user control to the function pointed at by iFunction.

EI_SetUserControlMouseMoved - *Not 64-bit compatible*

```
void EI_SetUserControlMouseMoved(
       EI_Control                *ioControl,
       EI_UserMouseMovedFunction iFunction);
```

ioControl	a pointer to a control object
iFunction	a pointer to a user control mouse moved function

EI_SetUserControlMouseMoved sets the mouse moved function of a user control to the function pointed at by iFunction.

EI_TrackMouseDown

```
int EI_TrackMouseDown(
        const EI_Control* iControl,
        int*              oMouseX,
        int*              oMouseY);
```

iControl	a pointer to a control object
oMouseX	a pointer to an integer value
oMouseY	a pointer to an integer value
return value	a boolean value

EI_TrackMouseDown can be called from within a user control click function. Pass the user control in iControl. EI_TrackMouseDown will return when the mouse moves and set *oMouseX and *oMouseY to the new location of the mouse. The return value will be true if the user is still pressing the mouse button and false if the user has released the mouse button.

NOTE: EI_TrackMouseDown will return false as soon as the mouse button is released, whether the mouse moves or not.

OpenGL Controls

EI_SetGLDrawFunction - **Not 64-bit compatible**

```
void EI_SetGLDrawFunction(
        EI_Control        *ioControl,
        EI_GLDrawFunction iFunction);
```

ioControl	a pointer to an OpenGL control
iFunction	a pointer to an OpenGL control draw function

EI_SetGLDrawFunction sets the draw function for the OpenGL control *ioControl to iFunction.

EI_SetGLClickFunction - **Not 64-bit compatible**

```
void EI_SetGLClickFunction(
        EI_Control        *ioControl,
        EI_GLClickFunction iFunction);
```

ioControl	a pointer to an OpenGL control
iFunction	a pointer to an OpenGL control click function

EI_SetGLClickFunction sets the click function for the OpenGL control *ioControl to iFunction.

EI_SetGLControlMouseMoved - Not 64-bit compatible

```
void EI_SetGLControlMouseMoved(
        EI_Control              *ioControl,
        EI_GLMouseMovedFunction iFunction);
```

ioControl	a pointer to an OpenGL control
iFunction	a pointer to an OpenGL mouse moved function

EI_SetGLControlMouseMoved sets the mouse moved function of an OpenGL control to the function pointed at by iFunction.

EI_BeginGLDrawing

```
int EI_BeginGLDrawing(const EI_Control *iControl);
```

iControl	a pointer to an OpenGL control
return value	a boolean value

EI_BeginGLDrawing sets up the OpenGL state so that drawing can occur within the OpenGL control *iControl. If the OpenGL state cannot be properly set up for *iControl, EI_BeginGLDrawing returns 0; otherwise it returns non-zero.

You use EI_BeginGLDrawing and EI_EndGLDrawing within your OpenGL control callbacks to bracket the OpenGL calls that you use to draw a 3D scene.

See the description of the EI_GLDrawFunction for an example of using EI_BeginGLDrawing.

EI_EndGLDrawing

```
void EI_EndGLDrawing(const EI_Control *iControl);
```

iControl	a pointer to an OpenGL control

EI_EndGLDrawing finishes up the OpenGL drawing that was previously begun by calling EI_BeginGLDrawing. EI_EndGLDrawing also flushes the drawing to the screen, so you should not call glFinish or glFlush.

EI_WindowToGL

```
void EI_WindowToGL(
        const EI_Dialog *iDialog,
        int             *ioX,
        int             *ioY);
```

iDialog	a pointer to a dialog box object
ioX	a pointer to an integer value
ioY	a pointer to an integer value

EI_WindowToGL transforms the two dimensional point given by (*ioX, *ioY) from the API's window coordinates to OpenGL's window coordinates. The API's window coordinates have their origin at the upper left corner of the window, with X and Y increasing to the right and down, respectively. OpenGL's window coordinates, on the other hand, have the origin at the lower left corner of the window, with X and Y increasing to the right and up, respectively.

You can use this function to transform a mouse location (as returned by the function EI_TrackMouseDown) to OpenGL coordinates when handling mouse clicks in an OpenGL control.

EI_GLToWindow

```
void EI_GLToWindow(
        const EI_Dialog *iDialog,
        int             *ioX,
        int             *ioY);
```

iDialog	a pointer to a dialog box object
ioX	a pointer to an integer value
ioY	a pointer to an integer value

EI_GLToWindow is the inverse of EI_WindowToGL. Given a point (*ioX, *ioY) given in OpenGL window coordinates, EI_GLToWindow transforms the point to the equivalent point in API window coordinates.

Callbacks

EI_ControlHitFunction

```
void MyHitFunction(
        EI_Control     *iControl,
        EI_DrawContext *iContext);
```

iControl	a pointer to a control
iContext	a pointer to a drawing context

A control's hit function is a notification that the user has interacted with the control in some way. Your can take whatever actions are appropriate for the given control. For example, if you need to update another control when the user interacts with this control, you can do so from within the hit function. A pointer to the control that was interacted with is passed in iControl, and iContext contains a pointer to a drawing context you can use to perform drawing if necessary.

NOTE: You should only need to drawing context in unusual circumstances, such as when dealing with user controls with special drawing needs. Most of the time, the iContext argument can be ignored.

EI_EditFilterFunction

```
int MyEditFilterFunction(
        EI_Control *iControl,
        const char *iNewText);
```

iControl	a pointer to an edit text control
iNewText	a pointer to a C string
return result	a boolean value

When an edit text control is set to use an edit filter function, the edit filter function will be called each time the user enters text into an edit text control (either by typing individual characters or by pasting a string of characters). The edit text control constructs a new string based on the edit string and the user's action, and then passes the new string in as iNewText. The edit filter function must examine the string passed in iNewText and determine if it is acceptable or not. If so, then the edit filter function should return true. If the edit filter returns false, the string will be rejected and the edit string will remain unchanged.

EI_ListDoubleClickFunction

```
void MyListDoubleClickFunction(
        EI_Control     *iControl,
        int             iDoubleClickedItem,
        EI_DrawContext *iContext);
```

iControl	a pointer to a list box control
iDoubleClickedItem	the index of the item that was double-clicked
iContext	a pointer to a drawing context

When a user double-clicks an item in a list box control, the double-click function is called to allow you to respond.

EI_ListReorderFunction

```
void MyListReorderFunction(
        EI_Control     *iControl,
        EI_DrawContext *iContext);
```

iControl	a pointer to a list box control
iContext	a pointer to a drawing context

When the user drags an item to a new position within a list box control, the reorder function is called to allow you to respond. Because the user may be dragging multiple items, there is no “iWhichItem” argument; you must inspect the list to see how it has changed.

EI_UserDrawFunction

```
void MyUserDrawFunction(
        const EI_Control *iControl,
        EI_DrawContext   *iContext);
```

iControl	a pointer to a user control
iContext	a pointer to a drawing context

A user control drawing function is called whenever the user control to which it is attached needs to be drawn. A pointer to the user control is passed in iControl, and iContext contains a drawing context that can be used to draw the user control.

Electric Image Animation System

EI_UserClickFunction

```
int MyUserClickFunction(
        EI_Control     *iControl,
        EI_DrawContext *iContext,
        int             iModifierKeys,
        int             iClickH,
        int             iClickV);
```

iControl	a pointer to a user control
iContext	a pointer to a drawing context
iModifierKeys	an integer value
iClickH	an integer value
iClickV	an integer value
return value	a boolean value

When a user control is clicked, its click function gets called. A pointer to the user control is passed in iControl, and iContext contains a pointer to a drawing context that can be used for drawing purposes while you track the mouse's movements. The initial location of the mouse is passed in iClickH and iClickV.

Your click function should retain control as long as necessary (usually until the mouse button is released), tracking the mouse and updating the display appropriately.

If the user control click function returns a true result, then the user control's hit function (if any) will be called; otherwise, if the click function returns false, the hit function will not be called. This allows your user control to communicate whether or not a click should count as a "hit."

EI_UserMouseMovedFunction

```
void MyUserMouseMovedFunction(
        EI_Control*     iControl,
        EI_DrawContext* iContext,
        int             iModifierKeys,
        int             iMouseH,
        int             iMouseV,
        int             iMoveType)
```

iControl	a pointer to a user control
iContext	a pointer to a drawing context
iModifierKeys	an integer value
iMouseH	an integer value
iMouseV	an integer value
iMoveType	an integer value

When the user moves the cursor into a user control, the control's mouse moved function gets called. A pointer to the control is passed in iControl, and iContext contains a pointer to a drawing context that can be used for drawing. The cursor's location is passed in iMouseH and iMouseV, and the current state of the modifier keys is passed in iModifierKeys.

The value of iMoveType tells you whether the mouse has entered the control, is moving within the control, or exiting the control, and can be one of the values cEI_MouseEnter, cEI_MouseWithin, or cEI_MouseExit.
You can use a mouse moved function to perform such tasks as custom cursor changing or rollover highlighting.

EI_GLDrawFunction

```
void MyGLDrawFunction(const EI_Control *iControl);
```

iControl	a pointer to an OpenGL control

The EI_GLDrawFunction is analogous to the user control draw function, and is used to draw an OpenGL control. One difference between the OpenGL draw function and the user control draw function is that no EI_DrawContext is passed to the OpenGL draw function. Because the OpenGL draw function is expected to draw the control using the OpenGL API, an EI_DrawContext is unnecessary.

NOTE: Using an EI_DrawContext to draw in a dialog box at the location of an OpenGL control is not supported. To draw using the OpenGL API, your draw function must call EI_BeginGLDrawing before the first OpenGL call, and EI_EndGLDrawing after the last OpenGL call. It is not necessary for you to call any of the OpenGL finishing functions (like glFinish or glFlush), because that is taken care of for you by EI_EndGLDrawing. For example:

```
void MyGLDrawFunction(EI_Control *iControl)
{
        if (EI_BeginGLDrawing(iControl)) {
                glClearColor(1.0f, 1.0f, 0.0f, 1.0f);
                glClear(GL_COLOR_BUFFER_BIT);

                /* Draw a smooth shaded polygon */
                glBegin(GL_POLYGON);
                glColor3d(1.0, 0.0, 0.0);
                glVertex3d( 0.8,  0.8, 0.0);
                glColor3d(0.0, 1.0, 0.0);
                glVertex3d( 0.8, -0.8, 0.0);
                glColor3d(0.0, 0.0, 1.0);
                glVertex3d(-0.8, -0.8, 0.0);
                glColor3d(1.0, 0.0, 1.0);
                glVertex3d(-0.8,  0.8, 0.0);
                glEnd();

                EI_EndGLDrawing(iControl);
        }
}
EI_GLClickFunction
        int MyGLClickFunction(
        EI_Control *iControl,
        int         iModifierKeys,
        int         iClickH,
        int         iClickV);
```

iControl	a pointer to an OpenGL control
iModifierKeys	an integer value
iClickH	an integer value
iClickV	an integer value
return value	a boolean value

The OpenGL click function is analogous to the user control click function. However, like the OpenGL draw function, the OpenGL click function is not passed an EI_DrawContext because the click function is expected to do its drawing using the OpenGL API. The other arguments (iModifierKeys, iClickH and iClickV) have the same meaning that they do for the user control. Your OpenGL click function should retain control as long as necessary (usually until the mouse button is released), tracking the mouse and updating the display appropriately.

Also like the OpenGL draw function, if the click function is going to do any drawing, that drawing must be bracketed between calls to EI_BeginGLDrawing and EI_EndGLDrawing. Note that EI_BeginGLDrawing and EI_EndGLDrawing bracket a single frame, so if your click function draws multiple frames (for example, because it is tracking the mouse and updating the display as the mouse moves), then each frame must be bracketed by a call to EI_BeginGLDrawing and EI_EndGLDrawing.

If the OpenGL control click function returns a true result, then the OpenGL control's hit function (if any) will be called; otherwise, if the click function returns false, the hit function will not be called. This allows your OpenGL control to communicate whether or not a click should count as a "hit."

EI_GLMouseMovedFunction

```
void MyGLMouseMovedFunction(
        EI_Control* iControl,
        int         iModifierKeys,
        int         iCursorH,
        int         iCursorV,
        int         iMoveType)
```

iControl	a pointer to a user control
iModifierKeys	an integer value
iMouseH	an integer value
iMouseV	an integer value
iMoveType	an integer value

The OpenGL mouse moved function is analogous to the user control mouse moved function. However, no EI_DrawContext is passed to an OpenGL mouse moved function, because the OpenGL mouse moved function is expected to do its drawing using the OpenGL API. The other arguments have the same meaning as they do for the user control mouse moved function.

You can use a mouse moved function to perform such tasks as custom cursor changing or rollover highlighting.

EI_TimerFunction

```
void MyTimerFunction(EI_Timer *iTimer);
```

iTimer	a pointer to a timer object

The timer function is called whenever the corresponding timer expires. The timer may expire once (if it was scheduled using EI_StartOneShotTimer) or many times (if it was scheduled using EI_StartPeriodicTimer). The timer action may take any action you like. For example, here is how to use a timer to time out a dialog box:
static EI_Timer *sTimer = NULL;

```
void TimeOutFunction(EI_Timer* iTimer)

{
        EI_Dialog             *dialog;

        /* Extract the dialog from the timer’s extra data. */
        dialog = (EI_Dialog*) EI_GetTimerExtraData(iTimer);

        /* Stop the dialog as if the user had canceled it. */
        EI_StopDialog(dialog, 0);
}

void InitDialog(EI_Dialog *iDialog)

{
        /* ... */

        /* Create a new timer. */
        sTimer = EI_MakeTimer();

        /* Initialize its timer function and extra data. */
        EI_SetTimerFunction(sTimer, TimeOutFunction);
        EI_SetTimerExtraData(sTimer, (void*) iDialog);

        /*     Set it to go off 60 seconds from now. */
        EI_StartOneShotTimer(sTimer, 60 * 1000);
}

void CleanUpDialog(EI_Dialog *iDialog)

{
        /*     All done with the timer. */
        EI_DestroyTimer(sTimer);

        /* ... */
}
```

Note that it is valid to either start or stop any timer when a timer function executes.

Cursors

The functions described in this section allow you to load and display custom cursors. The functions described in this section are declared in the file EI_Cursor.h.

EI_MakeCursor

```
EI_Cursor* EI_MakeCursor(int iCursorID);
```

iCursorID	an integer value
return value	a pointer to an EI_Cursor

EI_MakeCursor loads the cursor with resource ID iCursorID. If the cursor cannot be created (for example because the resource cannot be found), EI_MakeCursor returns NULL.

Cursors are created on the Macintosh and must be of type ‘crsr’. These resources can be created with a resource editing tool such as ResEdit or Resorcer.

NOTE: Use IDs in the range 12000 to 18000 for your cursors, or you may collide with cursors defined by Electric Image.

EI_DestroyCursor

```
void EI_DestroyCursor(EI_Cursor *iCursor);
```

iCursor	a pointer to an EI_Cursor

EI_DestroyCursor destroys a cursor that was previously created using EI_MakeCursor.

EI_SetCursor

```
void EI_SetCursor(EI_Cursor *iCursor);
```

iCursor	a pointer to an EI_Cursor

EI_SetCursor sets the system cursor to *iCursor. You can also pass NULL in iCursor, which will cause the system cursor to be reset to the standard arrow.

Drawing

The functions described in this section allow you to perform drawing to a drawing context. The context can be a screen context (if it is a context passed to one of your control callback functions) or it can be an off-screen image buffer context (if you created it yourself by calling EI_MakeContextForImageBuffer). The functions described in this section are declared in the file EI_DrawContext.h.

EI_MakeContextForImageBuffer

```
EI_DrawContext *EI_MakeContextForImageBuffer(EI_ImageBuffer *iImage);
```

iImage	a pointer to an image buffer
return value	a pointer to a drawing context object

EI_MakeContextForImageBuffer makes a drawing context for the given image buffer. Any drawing done using this context will draw into the image buffer. This can be handy for performing drawing off-screen that is later copied to the screen using the function EI_DrawImage. When you are done using the draw context you must make sure to destroy the draw context using EI_DestroyContext.

EI_MakeContextForDialog

```
EI_DrawContext *EI_MakeContextForDialog(EI_Dialog *iDialog);
```

iDialog	a pointer to a dialog box
return value	a pointer to a drawing context object

EI_MakeContextForDialog makes a drawing context for the given dialog box. Any drawing done using this context will draw into the dialog box. When you are done using the draw context you must make sure to destroy the draw context using EI_DestroyContext.

EI_DestroyContext

```
void EI_DestroyContext(EI_DrawContext  *iContext);
```

iContext	a pointer to a drawing context object

EI_DestroyContext destroys a drawing context object that was previously created with EI_MakeContextForImageBuffer or EI_MakeContextForDialog. Call this function when you are done using the drawing context object.

NOTE: Do not call EI_DestroyContext on any context that you did not create. Specifically, do not call EI_DestroyContext on a context passed to your control callback functions.

EI_SetClip

```
void EI_SetClip(
        EI_DrawContext *ioContext,
        const EI_Rect  *iClipRect);
```

ioContext	a pointer to a drawing context object
iClipRect	a pointer to a rectangle

EI_SetClip sets the clipping area of the given context to the rectangle *iClipRect. It is not possible to set the clipping area of a context to a non-rectangular area.

EI_GetClip

```
void EI_GetClip(
        const EI_DrawContext *iContext,
        EI_Rect              *oClipRect);
```

iContext	a pointer to a drawing context object
oClipRect	a pointer to a rectangle

EI_GetClip sets *iClipRect to the current clipping rectangle of the context.

EI_SetPenMode

```
void EI_SetPenMode(
        EI_DrawContext *ioContext,
        EI_PenMode      iPenMode);
```

ioContext	a pointer to a drawing context object
iPenMode	a pen mode value

EI_SetPenMode sets the pen drawing mode to iPenMode. The pen mode may be either cEI_PenColor or cEI_PenXOR.

EI_GetPenMode

```
EI_PenMode EI_GetPenMode(const EI_DrawContext *iContext);
```

iContext	a pointer to a drawing context object
return value	a pen mode value

EI_GetPenMode returns the context's pen drawing mode.

EI_MoveTo

```
void EI_MoveTo(
        EI_DrawContext *ioContext,
        int             iPenH,
        int             iPenV);
```

ioContext	a pointer to a drawing context object
iPenH	an integer value
iPenV	an integer value

EI_MoveTo moves the context's pen to the location (iPenH, iPenV).

EI_LineTo

```
void EI_LineTo(
        EI_DrawContext *ioContext,
        int             iPenH,
        int             iPenV);
```

ioContext	a pointer to a drawing context object
iPenH	an integer value
iPenV	an integer value

EI_LineTo draws a line to the context from the context's current pen location to the location (iPenH, iPenV). The line is drawn using the foreground color and the pen drawing mode, and the pen is left at location (iPenH, iPenV).

EI_GetPenLocation

```
void EI_GetPenLocation(
        const EI_DrawContext *iContext,
        int                  *oPenH,
        int                  *oPenV);
```

iContext	a pointer to a drawing context object
oPenH	a pointer to an integer value
oPenV	a pointer to an integer value

EI_GetPenLocation sets *oPenH and *oPenV to the context's pen location.

EI_SetForeColor

```
void EI_SetForeColor(
        EI_DrawContext *ioContext,
        const EI_Color *iForeColor);
```

ioContext	a pointer to a drawing context object
iForeColor	a pointer to a color

EI_SetForeColor sets the context's foreground color to the color *iForeColor.

EI_GetForeColor

```
void EI_GetForeColor(
        const EI_DrawContext *iContext,
        EI_Color             *oForeColor);
```

iContext	a pointer to a drawing context object
oForeColor	a pointer to a color

EI_GetForeColor copies the context's current foreground color to *oForeColor.

EI_SetBackColor

```
void EI_SetBackColor(
        EI_DrawContext *ioContext,
        const EI_Color *iBackColor);
```

ioContext	a pointer to a drawing context object
iBackColor	a pointer to a color

EI_SetBackColor sets the context's background color to the color *iBackColor.

EI_GetBackColor

```
void EI_GetBackColor(
        const EI_DrawContext *iContext,
        EI_Color             *oBackColor);
```

iContext	a pointer to a drawing context object
oBackColor	a pointer to a color

EI_GetBackColor copies the context's current background color to *oBackColor.

EI_FrameRect

```
void EI_FrameRect(
        EI_DrawContext *ioContext,
        const EI_Rect  *iFrame);
```

ioContext	a pointer to a drawing context object
iFrame	a pointer to a rectangle

EI_FrameRect draws a rectangle frame to the context. The rectangle's frame will be one pixel thick, and will be drawn in the foreground color using the pen drawing mode.

EI_PaintRect

```
void EI_PaintRect(
        EI_DrawContext *ioContext,
        const EI_Rect  *iFrame);
```

ioContext	a pointer to a drawing context object
iFrame	a pointer to a rectangle

EI_PaintRect draws a filled rectangle to the context. The rectangle will be filled with the foreground color using the pen drawing mode.

EI_EraseRect

```
void EI_EraseRect(
        EI_DrawContext *ioContext,
        const EI_Rect  *iFrame);
```

ioContext	a pointer to a drawing context object
iFrame	a pointer to a rectangle

EI_EraseRect erases a rectangle to the context. The rectangle will be filled with the background color using the pen drawing mode.

EI_FrameOval

```
void EI_FrameOval(
        EI_DrawContext *ioContext,
        const EI_Rect  *iFrame);
```

ioContext	a pointer to a drawing context object
iFrame	a pointer to a rectangle

EI_FrameOval draws an ellipse to the context. The ellipse drawn will fit just inside the given rectangular frame. The ellipse will be one pixel thick, and will be drawn in the foreground color using the pen drawing mode.

EI_PaintOval

```
void EI_PaintOval(
        EI_DrawContext *ioContext,
        const EI_Rect  *iFrame);
```

ioContext	a pointer to a drawing context object
iFrame	a pointer to a rectangle

EI_FrameOval draws a filled ellipse to the context. The ellipse drawn will fit just inside the given rectangular frame. The ellipse will be filled with the foreground color using the pen drawing mode.

EI_DrawImage

```
void EI_DrawImage(
        EI_DrawContext        *ioContext,
        const EI_ImageBuffer *iImage,
        const EI_Rect         *iSourceRect,
        const EI_Rect         *iDestRect);
```

ioContext	a pointer to a drawing context object
iImage	a pointer to an image buffer
iSourceRect	a pointer to a rectangle defined in *iImage's coordinate space
iDestRect	a pointer to a rectangle defined in *ioContext's coordinate space

EI_DrawImage draws the pixels from *iImage to the context. The pixels are taken from the rectangular area defined by *iSourceRect, and are copied to the rectangular area defined by *iDestRect.

If the image buffer is from a disk image object, and the disk image has a mask associated with it, only pixels that are included by the mask are drawn to the drawing context.

EI_CaptureImage

```
void EI_CaptureImage(
        const EI_DrawContext *iContext,
        EI_ImageBuffer        *iImage,
        const EI_Rect         *iSourceRect,
        const EI_Rect         *iDestRect);
```

iContext	a pointer to a drawing context object
iImage	a pointer to an image buffer
iSourceRect	a pointer to a rectangle defined in *iContext's coordinate space
iDestRect	a pointer to a rectangle defined in *iImage's coordinate space

EI_CaptureImage is the inverse of EI_DrawImage. EI_CaptureImage copies pixels from the context to *iImage. The pixels are taken from the rectangular area defined by *iSourceRect, and are copied to the rectangular area defined by *iDestRect.

EI_SetTextSize

```
void EI_SetTextSize(
        EI_DrawContext *ioContext,
        int             iSize);
```

ioContext	a pointer to a drawing context object
iSize	an integer value

EI_SetTextSize sets the font size of the context to iSize.

EI_GetTextSize

```
int EI_GetTextSize(
        const EI_DrawContext *iContext);
```

iContext	a pointer to a drawing context object
return value	an integer value

EI_GetTextSize returns the font size of the context.

EI_SetTextStyle

```
void EI_SetTextStyle(
        EI_DrawContext *ioContext,
        int             iStyleFlags);
```

ioContext	a pointer to a drawing context object
iStyleFlags	an integer value

EI_SetTextStyle sets the font style flags of the context to iStyleFlags. The style flags can be any bit-wise combination of cEI_TextBold, cEI_TextItalic, cEI_TextUnderline, cEI_TextOutline, and cEI_TextShadow. To remove all stylistic variation from the context, pass 0 as iStyleFlags.

EI_GetTextStyle

```
int EI_GetTextStyle(const EI_DrawContext *iContext);
```

iContext	a pointer to a drawing context object
return value	an integer value

EI_GetTextStyle returns the font style flags of the context.

EI_MeasureText

```
int EI_MeasureText(
        const EI_DrawContext *iContext,
        const char           *iText,
        int                   iNumChars);
```

iContext	a pointer to a drawing context object
iText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iText
return value	an integer value

EI_MeasureText returns the pixel width of the string passed in iText and iNumChars. The pixel width accounts for the font size and font style of the context, and is the distance the pen would move if this string was drawn using EI_DrawText.

NOTE: EI_MeasureText does not require that the text argument be a C string, as do other string functions in the API.

EI_GetFontMetrics

```
void EI_GetFontMetrics(
        const EI_DrawContext *iContext,
        int                  *oAscent,
        int                  *oDescent,
        int                  *oLeading);
```

iContext	a pointer to a drawing context object
oAscent	a pointer to an integer value
oDescent	a pointer to an integer value
oLeading	a pointer to an integer value

EI_GetFontMetrics returns height information about the context's font, taking into account the font size and font style. The ascent is the distance that glyphs in the font rise above the base line, the descent is the distance that glyphs sink below the baseline, and the leading is the normal distance between the descent of one line of text and the ascent of the next line of text. This implies that the height of a single line of text can be calculated as the ascent + the descent + the leading.

Any one of oAscent, oDescent and oLeading can be NULL, and if so, they are skipped. For example, if you are only interested in the ascent, you can pass NULL as oDescent and oLeading.

EI_DrawText

```
void EI_DrawText(
        EI_DrawContext *ioContext,
        const char     *iText,
        int            iNumChars);
```

ioContext	a pointer to a drawing context object
iText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iText

EI_DrawText draws the string given by iText and iNumChars to the context. This string is drawn at the current pen location, and uses the current foreground color. After EI_DrawText completes, the context's pen is move to the point just after the last character drawn.

NOTE: EI_DrawText does not require that the text argument be a C string, as do other string functions in the API.

EI_DrawTextToFit

```
void EI_DrawTextToFit(
        EI_DrawContext *ioContext,
        const char     *iText,
        int             iNumChars,
        int             iMaxPixelWidth,
        int             iAllowCondensed);
```

iContext	a pointer to a drawing context object
iText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iText
iMaxPixelWidth	an integer value
iAllowCondensed	a boolean value

EI_DrawTextToFit behaves like EI_DrawText, but allows you to specify a maximum pixel width for the drawn string. If the string's pixel width is less than or equal to iMaxPixelWidth, EI_DrawTextToFit draws the text and returns. If the string's pixel width is greater than iMaxPixelWidth, then it will try to shrink the text to make it fit. First, if iAllowCondensed is true, EI_DrawTextToFit will draws the text in a compressed mode, where characters are moved closer together. If iAllowCondensed is false, or if condensing did not make the pixel width short enough, EI_DrawTextToFit will truncate characters off of the end of the string, replacing them with an ellipsis ("...") to make the string fit.

Like EI_DrawText, EI_DrawTextToFit draws the text in the context's foreground color using the context's font size and font style. Drawing begins at the current pen location, and after drawing the pen is left just past the last character drawn.

NOTE: EI_DrawTextToFit does not require that the text argument be a C string, as do other string functions in the API.

EI_DrawTextInRect

```
void EI_DrawTextInRect(
        EI_DrawContext *ioContext,
        const char     *iText,
        int             iNumChars,
        const EI_Rect  *iBounds);
```

ioContext	a pointer to a drawing context object
iText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iText
iBounds	a pointer to a rectangle

EI_DrawTextInRect draws text to the context within a rectangular area. If the text is too long to fit on one line, the text is word-wrapped as necessary. In no case does drawing occur outside of *iBounds. If the text is long enough to wrap below the bottom of *iBounds, it will be clipped.

EI_DrawTextInRect draws the text in the context's foreground color using the context's font size and font style, but does not use the context's pen location.

NOTE: EI_DrawTextInRect does not require that the text argument be a C string, as do other string functions in the API.

EI_HiliteRect

```
void EI_HiliteRect(
        EI_DrawContext *ioContext,
        const EI_Rect  *iBounds,
        const EI_Color *iHiliteColor,
        const EI_Color *iBackColor);
```

ioContext	a pointer to a drawing context object
iBounds	a pointer to a rectangle
iHiliteColor	a pointer to a color
iBackColor	a pointer to a color

EI_HiliteRect highlights the rectangle by swapping the colors of pixels matching *iHiliteColor and *iBackColor. Pixels that are colored *iHiliteColor are changed to *iBackColor, and pixels that are colored *iBackColor are changed to *iHiliteColor. This has the effect of “highlighting” the rectangle in the given color.

Image Buffers

The functions described in this section allow you to create and destroy image buffer objects. The functions and macros described in this section are declared in the file EI_ImageBuffer.h.

EI_MakeImageBuffer

```
EI_ImageBuffer *EI_MakeImageBuffer(const EI_Rect *iBounds);
```

iBounds	a pointer to a rectangle
return value	a pointer to an image buffer object

EI_MakeImage creates and returns a new image buffer object, using the boundary rectangle *iBounds. The boundary rectangle not only defines the pixel dimensions of the image buffer, but also the coordinate space of the image buffer. For example if the boundary rectangle is (50, 50, 100, 100), then the coordinates of the upper left pixel of the image buffer are (50, 50).

The image buffer created always uses 32-bit pixels.

EI_DestroyImageBuffer

```
void EI_DestroyImageBuffer(EI_ImageBuffer *iImage);
```

iImage	a pointer to an image buffer object

EI_DestroyImageBuffer destroys an image buffer. Call EI_DestroyImageBuffer after you are through using the image buffer.

NOTE: Do not call EI_DestroyImageBuffer on an image buffer that you did not create. In particular, do not call EI_DestroyImageBuffer on the image buffer returned by calling EI_GetDiskImageBuffer.

EI_GetBufferRowBytes

```
int EI_GetBufferRowBytes(const EI_ImageBuffer *iImage);
```

iImage	a pointer to an image buffer object
return value	an integer value

EI_GetBufferRowBytes returns the size, in bytes, of a single row of pixels from the image buffer.

NOTE: Do not assume that this value is equal to the number of pixels in a row times the number of bytes in a pixel. An operating system may add padding bytes to each scan line so that it will be aligned for better performance.

EI_GetBufferBaseAddress - Not 64-bit compatible

```
unsigned char *EI_GetBufferBaseAddress(EI_ImageBuffer *iImage);
```

iImage	a pointer to an image buffer object
return value	a pointer to raw pixel memory

EI_GetBufferBaseAddress returns a pointer to the image buffer’s raw pixel memory. This pointer is actually a pointer to the upper left pixel in the image buffer.

To access and set pixel memory, use the pixel access macros.

EI_GetBufferBounds

```
void EI_GetBufferBounds(
        const EI_ImageBuffer *iImage,
        EI_Rect              *oRect);
```

iImage	a pointer to an image buffer object
oRect	a pointer to a rectangle

EI_GetBufferBounds sets *oRect to the boundary rectangle of the image buffer.

Pixel Access Macros

The pixel access macros provided by the API allow you to get and set the color components of the pixels in an image buffer.

NOTE: If you have a pointer to pixel memory, you must dereference the pointer when using it with the pixel access macros or you will get a compiler error. The macros all assume they are working on pixel values, not pointers to pixel values.

For example:

```
EI_32BitPixel          *pixelPtr;
        unsigned char          r, g, b, a;

        //      Set pixelPtr to point into an image buffer.

        EI_GET_PIXEL_ALL(*pixelPtr, r, g, b, a);
```

EI_MAKE_PIXEL

```
EI_MAKE_PIXEL(
        iRed,
        iGreen,
        iBlue
        iAlpha)
```

iRed	a red value
iGreen	a green value
iBlue	a blue value
iAlpha	an alpha value
return value	a 32 bit pixel

The macro EI_MAKE_PIXEL combines its four arguments to create a 32 bit pixel value. The values of the arguments should be unsigned values in the range 0 to 255.

The best use of EI_MAKE_PIXEL is for creating compile-time constants. For example:
#define RED_PIXEL EI_MAKE_PIXEL(0, 0xFF, 0, 0)

For setting components of pixel variables or pixel memory, use the other macros described below.

Electric Image Animation System

EI_GET_PIXEL_ALL

```
EI_GET_PIXEL_ALL(
        iPixel,
        oRed,
        oGreen,
        oBlue,
        oAlpha)
```

iPixel	a 32 bit pixel value
oRed	an integer value
oGreen	an integer value
oBlue	an integer value
oAlpha	an integer value

The macro EI_GET_PIXEL_ALL breaks up the pixel value iPixel into its separate components and stores the results in oRed, oGreen, oBlue and oAlpha.

EI_SET_PIXEL_ALL

```
EI_SET_PIXEL_ALL(
        oPixel,
        iRed,
        iGreen,
        iBlue,
        iAlpha)
```

oPixel	a 32 bit pixel value
iRed	an integer value
iGreen	an integer value
iBlue	an integer value
iAlpha	an integer value

The macro EI_SET_PIXEL_ALL combines the pixel components passed in iRed, iGreen, iBlue and iAlpha into the pixel variable oPixel.

EI_GET_PIXEL_RED

```
EI_GET_PIXEL_RED(
        iPixel)
```

iPixel	a 32 bit pixel value
return value	an integer value

The macro EI_GET_PIXEL_RED returns the red component of the given pixel value.

EI_SET_PIXEL_RED

```
EI_SET_PIXEL_RED(
        ioPixel,
        iRed)
```

ioPixel	a 32 bit pixel value
iRed	an integer value

The macro EI_SET_PIXEL_RED sets the red component of the given pixel value without changing any of the other components.

EI_GET_PIXEL_GREEN

```
EI_GET_PIXEL_GREEN(
        iPixel)
```

iPixel	a 32 bit pixel value
return value	an integer value

The macro EI_GET_PIXEL_GREEN returns the green component of the given pixel value.

EI_SET_PIXEL_GREEN

```
EI_SET_PIXEL_GREEN(
        ioPixel,
        iGreen)
```

ioPixel	a 32 bit pixel value
iGreen	an integer value

The macro EI_SET_PIXEL_GREEN sets the green component of the given pixel value without changing any of the other components.

EI_GET_PIXEL_BLUE

```
EI_GET_PIXEL_BLUE(
        iPixel)
```

iPixel	a 32 bit pixel value
return value	an integer value

The macro EI_GET_PIXEL_BLUE returns the blue component of the given pixel value.

EI_SET_PIXEL_BLUE

```
EI_SET_PIXEL_BLUE(
        ioPixel,
        iBlue)
```

ioPixel	a 32 bit pixel value
iBlue	an integer value

The macro EI_SET_PIXEL_BLUE sets the blue component of the given pixel value without changing any of the other components.

EI_GET_PIXEL_ALPHA

```
EI_GET_PIXEL_ALPHA(
        iPixel)
```

iPixel	a 32 bit pixel value
return value	an integer value

The macro EI_GET_PIXEL_ALPHA returns the red component of the given pixel value.

NOTE: The alpha component is ignored when the image buffer is displayed; unless you are performing pixel operations yourself that use the alpha component, you will probably not need to use this macro.

EI_SET_PIXEL_ALPHA

```
EI_SET_PIXEL_ALPHA(
        iPixel,
        iAlpha)
```

ioPixel	a 32 bit pixel value
iAlpha	an integer value

The macro EI_SET_PIXEL_ALPHA sets the alpha component of the given pixel value without changing any of the other components.

NOTE: The alpha component is ignored when the image buffer is displayed; unless you are performing pixel operations yourself that use the alpha component, you will probably not need to use this macro.

Sample Code

This section presents sample code showing how to use the image buffer functions and macros to render to an image buffer.

The first example shows how to render to the entire image buffer.

```
void RenderWholeBuffer(EI_ImageBuffer *iBuffer)
```

```
{
        EI_Rect                                 bounds;
        int                                             rowBytes, width, height, x, y;
        char*                                           rowPtr;
        EI_32BitPixel                   *pixelPtr;
        unsigned char                   red, green, blue;

        EI_GetBufferBounds(iBuffer, &bounds);
        width = bounds.right - bounds.left;
        height = bounds.bottom - bounds.top;

        rowBytes = EI_GetBufferRowBytes(iBuffer);

        rowPtr = EI_GetBufferBaseAddress(iBuffer);

        for (y = 0; y < height; ++y) {

                pixelPtr = (EI_32BitPixel*) rowPtr;
                for (x = 0; x < width; ++x) {

                /* Compute red, green, and blue values as desired */

                        EI_SET_PIXEL_RED(*pixelPtr, red);
                        EI_SET_PIXEL_GREEN(*pixelPtr, green);
                        EI_SET_PIXEL_BLUE(*pixelPtr, blue);

                        ++pixelPtr;
                }

                rowPtr += rowBytes;
        }
}
```

The function begins by getting the boundary rectangle, row bytes value and the pointer to the first pixel in the image buffer. The outer loop increments the row pointer by adding rowBytes to it, which gives a pointer to the next row of pixels. The inner loop begins by copying the row pointer to a pixel pointer (casting to EI_32BitPixel*), then it sets the red, green and blue values of each pixel independently.

To render to an area within an image buffer (rather than the whole image buffer), the code is very similar:

```
void RenderPartialBuffer(EI_ImageBuffer *iBuffer,
                                                                        const EI_Rect *iRenderBounds)

{
        EI_Rect                                 renderBounds, bounds;
        int                                             rowBytes, width, height, x, y;
        char*                                           rowPtr;
        EI_32BitPixel                   *pixelPtr;
        unsigned char                   red, green, blue;

        EI_GetBufferBounds(iBuffer, &bounds);

        if (EI_IntersectRects(&bounds, iRenderBounds,
                        &renderBounds)) {

                width = renderBounds.right - renderBounds.left;
                height = renderBounds.bottom - renderBounds.top;
        rowBytes = EI_GetBufferRowBytes(iBuffer);
        rowPtr = EI_GetBufferBaseAddress(iBuffer) +
                    (renderBounds.top - bounds.top) * rowBytes +
                    (renderBounds.left - bounds.left) *
                        sizeof(EI_32BitPixel);

        for (y = 0; y < height; ++y) {
            pixelPtr = (EI_32BitPixel*) rowPtr;
            for (x = 0; x < width; ++x) {

        /* Compute red, green, and blue values as desired */

                EI_SET_PIXEL_RED(*pixelPtr, red);
                EI_SET_PIXEL_GREEN(*pixelPtr, green);
                EI_SET_PIXEL_BLUE(*pixelPtr, blue);

                ++pixelPtr;
            }
            rowPtr += rowBytes;
        }
    }
}
```

There are only two significant differences between this function and the previous function. The first is the call to EI_IntersectRects, which makes sure that the area being rendered is really within the bounds of the image buffer.

The second difference is that the beginning row pointer is no longer just the pointer to the top left pixel. Rather, this pixel is offset downward by the distance from the top of the image buffer to the top of the render area, and leftward by the distance from the left of the image buffer to the left of the render area. Notice that the distance is computed by getting the image buffer’s boundary rectangle and subtracting it from the render area bounds. The image buffer’s boundary rectangle does not necessarily have its origin at (0, 0).

Note that in both examples, x and y do not necessarily correspond to pixel coordinates; they are instead just counters that tell the function how many rows and columns to render. If you need counters that correspond to pixel coordinates, you can increment from the top to the bottom and from the left to the right of the render area bounds.

Disk Images

The functions described in this section allow you to work with disk images stored in resource files. The functions described in this section are declared in EI_DiskImage.h.

EI_MakeDiskImage

```
EI_DiskImage *EI_MakeDiskImage(
        EI_ResourceFile *iResourceFile,
        UInt32           iType,
        int              iID);
```

iResourceFile	a pointer to a resource file
iType	a resource type
iID	a resource ID

EI_MakeDiskImage reads a disk image from the given resource file, using the type iType and ID iID. If EI_MakeDiskImage cannot create the disk image, it returns NULL.

The allowed types include 'argb', 'PICT', and 'cicn.'

EI_DestroyDiskImage

```
void EI_DestroyDiskImage(EI_DiskImage *iImage);
```

iImage	a pointer to a disk image

EI_DestroyDiskImage destroys a disk image object that was created using EI_MakeDiskImage. Call EI_DestroyDiskImage when you are done using the disk image object.

EI_GetDiskImageBuffer - Not 64-bit compatible

```
const EI_ImageBuffer *EI_GetDiskImageBuffer(EI_DiskImage *iImage);
```

iImage	a pointer to a disk image
return value	a pointer to an image buffer object

EI_GetDiskImageBuffer returns the image buffer associated with the disk image object. You can draw this image buffer to a drawing context using EI_DrawImage.

NOTE: Because EI_GetDiskImageBuffer returns a const EI_ImageBuffer pointer, you are not allowed to modify the image buffer associated with a disk image in any way. The pixels of a disk image's image buffer should be considered read-only. In particular, you cannot draw to this image buffer by creating an EI_DrawContext for it, and you cannot destroy this image buffer by calling EI_DestroyImageBuffer.

Rectangles

EI_GetRectWidth

```
int EI_GetRectWidth(const EI_Rect *iRect);
```

iRect	a pointer to a rectangle
return value	an integer value

EI_GetRectWidth returns the width of the rectangle.

EI_GetRectHeight

```
int EI_GetRectHeight(const EI_Rect *iRect);
```

iRect	a pointer to a rectangle
return value	an integer value

EI_GetRectHeight returns the height of the rectangle.

Electric Image Animation System

EI_SetRect

```
void EI_SetRect(
        EI_Rect *oRect,
        int      iLeft,
        int      iTop,
        int      iRight,
        int      iBottom);
```

oRect	a pointer to a rectangle
iLeft	an integer value
iTop	an integer value
iRight	an integer value
iBottom	an integer value

EI_SetRect sets all of the members of a rectangle at once. It is equivalent to the following code fragment:

```
oRect->left     = iLeft;
oRect->top      = iTop;
oRect->right    = iRight;
oRect->bottom   = iBottom;
```

EI_IsRectEmpty

```
int EI_IsRectEmpty(const EI_Rect *iRect);
```

iRect	a pointer to a rectangle
return value	a boolean value

EI_IsRectEmpty returns true if the rectangle is empty and false if the rectangle is non-empty.

EI_IsPtInRect

```
int EI_IsPtInRect(
        const EI_Rect *iRect,
        int            iX,
        int            iY);
```

iRect	a pointer to a rectangle
iX	an integer value
iY	an integer value
return value	a boolean value

EI_IsPtInRect returns true if the point given by (iX, iY) is contained within *iRect.

EI_UnionRects

```
void EI_UnionRects(
        const EI_Rect *iRect1,
        const EI_Rect *iRect2,
        EI_Rect *oResult);
```

iRect1	a pointer to a rectangle
iRect2	a pointer to a rectangle
oResult	a pointer to a rectangle

EI_UnionRects sets *oResult to the union of *iRect1 and *iRect2. The union of two rectangles is defined as the smallest rectangle that encloses both rectangles.

EI_IntersectRects

```
int EI_IntersectRects(
        const EI_Rect *iRect1,
        const EI_Rect *iRect2,
        EI_Rect *oResult);
```

iRect1	a pointer to a rectangle
iRect2	a pointer to a rectangle
oResult	a pointer to a rectangle
return value	a boolean value

EI_IntersectRects sets *oResult to the intersection of *iRect1 and *iRect2, and returns true if *oResult is non-empty. The intersection of two rectangles is defined as the largest rectangle that contains area in both rectangles.

If you pass NULL as oResult, EI_IntersectRects will still compute whether or not the intersection is empty and return the appropriate value.

EI_InsetRect

```
void EI_InsetRect(EI_Rect *ioRect,
        int                iInsetX,
        int                iInsetY);
```

ioRect	a pointer to a rectangle
iInsetX	an integer value
iInsetY	an integer value

EI_InsetRect shrinks *ioRect by moving the left and right sides inwards by iInsetX pixels, and by moving the top and bottom sides inwards by iInsetY pixels. If either value is negative, the corresponding sides of *ioRect will move outwards rather than inwards.

EI_OffsetRect

```
void EI_OffsetRect(
        EI_Rect  *ioRect,
        int       iOffsetX,
        int       iOffsetY);
```

ioRect	a pointer to a rectangle
iOffsetX	an integer value
iOffsetY	an integer value

EI_OffsetRect moves *ioRect iOffsetX pixels horizontally and iOffsetY pixels vertically.

EI_AreRectsEqual

```
int EI_AreRectsEqual(
        const EI_Rect *iRect1,
        const EI_Rect *iRect2);
```

iRect1	a pointer to a rectangle
iRect2	a pointer to a rectangle
return value	a boolean value

EI_AreRectsEqual returns true if the rectangles are identical and false if they are not identical.

File References

The functions described in this section allow you to manipulate directory reference and file reference objects. The functions described in this section are declared in the file EI_FileRef.h.

EI_MakeDirectoryRef

```
EI_DirectoryRef *EI_MakeDirectoryRef(void);
```

return value	a pointer to a directory reference object

EI_MakeDirectoryRef creates and returns an empty directory reference object. When you are done using this directory reference, you must destroy it by calling EI_DestroyDirectoryRef.

EI_DestroyDirectoryRef

```
void EI_DestroyDirectoryRef(EI_DirectoryRef *iDirectoryRef);
```

iDirectoryRef	a pointer to a directory reference object

EI_DestroyDirectoryRef destroys a directory reference object that was created using any of the functions which create new directory reference objects. These functions are

- •EI_MakeDirectoryRef
- •EI_GetHomeDirectory
- •EI_GetPreferencesDirectory
- •EI_GetTemporaryDirectory

EI_AppendDirectoryName

```
void EI_AppendDirectoryName(
        EI_DirectoryRef *ioDirectoryRef,
        const char      *iDirectoryName);
```

ioDirectoryRef	a pointer to a directory reference object
iDirectoryName	a pointer to a C string

EI_AppendDirectoryName adds the string given by iDirectoryName to the path name list in the directory reference.

EI_RemoveLastDirectoryName

```
void EI_RemoveLastDirectoryName(EI_DirectoryRef *ioDirectoryRef);
```

ioDirectoryRef	a pointer to a directory reference object

EI_RemoveLastDirectoryName removes the last name in a directory reference's path name list. This has the effect of changing the directory reference so that it now refers to the directory that previously enclosed it.

EI_GetNumDirectoryNames

```
int EI_GetNumDirectoryNames(const EI_DirectoryRef *iDirectoryRef);
```

iDirectoryRef	a pointer to a directory reference object

EI_GetNumDirectoryNames returns the number of names in the directory reference's path name list.

EI_GetDirectoryNameSize

```
int EI_GetDirectoryNameSize(
        const EI_DirectoryRef  *iDirectoryRef,
        int                     iNameIndex);
```

iDirectoryRef	a pointer to a directory reference object
iNameIndex	a path name list index
return value	an integer value

EI_GetDirectoryNameSize returns the number of characters in the name from the directory reference's path name list whose index is given by iNameIndex. The number of characters includes the terminating zero.

EI_GetDirectoryName

```
void EI_GetDirectoryName(
        const EI_DirectoryRef  *iDirectoryRef,
        int                    iNameIndex,
        char                   *iText,
        int                    *ioNumChars);
```

iDirectoryRef	a pointer to a directory reference object
iNameIndex	a path name list index
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetDirectoryName copies the text from the name from the directory reference’s path name list whose index is given by iNameIndex to the buffer pointed at by oText. If the text is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetDirectoryName will copy as many characters as will fit. including the terminating zero. EI_GetDirectoryName will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_GetDirectoryPathNameSize

```
int EI_GetDirectoryPathNameSize(const EI_DirectoryRef *iDirectoryRef);
```

iDirectoryRef	a pointer to a directory reference object
return value	an integer value

EI_GetDirectoryPathNameSize creates a platform-specific path name from the directory reference object and returns the number of characters in the path name, including the terminating zero.

EI_GetDirectoryPathName

```
void EI_GetDirectoryPathName(
        const EI_DirectoryRef  *iDirectoryRef,
        char                   *iText,
        int                    *ioNumChars);
```

iDirectoryRef	a pointer to a directory reference object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetDirectoryPathName creates a platform-specific path name from the directory reference object, then copies it to the buffer pointed at by oText. If the path name is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetDirectoryPathName will copy as many characters as will fit, including the terminating zero. EI_GetDirectoryPathName will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_GetHomeDirectory

```
EI_DirectoryRef *EI_GetHomeDirectory(void);
```

return value	a pointer to a directory reference object

EI_GetHomeDirectory creates and returns a directory reference object that is initialized to point to the home directory. When you are done using this directory reference, you must destroy it by calling EI_DestroyDirectoryRef.

EI_GetPreferencesDirectory

```
EI_DirectoryRef *EI_GetPreferencesDirectory(void);
```

return value	a pointer to a directory reference object

EI_GetPreferencesDirectory creates and returns a directory reference object that is initialized to point to the preferences directory. When you are done using this directory reference, you must destroy it by calling EI_DestroyDirectoryRef.

EI_GetTemporaryDirectory

```
EI_DirectoryRef *EI_GetTemporaryDirectory(void);
```

return value	a pointer to a directory reference object

EI_GetTemporaryDirectory creates and returns a directory reference object that is initialized to point to the temporary items directory. When you are done using this directory reference, you must destroy it by calling EI_DestroyDirectoryRef.

EI_CreateDirectory

```
int EI_CreateDirectory(const EI_DirectoryRef *iDirectoryRef);
```

iDirectoryRef	a pointer to a directory reference object
return value	an error code

EI_CreateDirectory creates a directory in the user's file system that matches the directory reference *iDirectoryRef. EI_CreateDirectory will also create intermediate parent directories (for example, if the parent directory of the directory indicated by *iDirectoryRef does not exist, the parent directory will be created first). The function result is an error code; zero indicates success, non-zero values indicate failure.

If the directory indicated by *iDirectoryRef already exists, EI_CreateDirectory will do nothing and return zero.

EI_MAKE_FILE_TYPE

```
EI_MAKE_FILE_TYPE(
        iA,
        iB,
        iC,
        iD)
```

iA	a character constant
iB	a character constant
iC	a character constant
iD	a character constant
return value	a file type value

The macro EI_MAKE_FILE_TYPE combines its four character constant arguments into a single file type. Use this macro to construct file type values. For example:

```
const UInt32 cTextFileType =
                EI_MAKE_FILE_TYPE(‘T’, ‘E’, ‘X’, ‘T’);
```

EI_MakeEmptyFileRef

```
EI_FileRef *EI_MakeEmptyFileRef(void);
```

return value	a pointer to a file reference object

EI_MakeEmptyFileRef creates and returns an empty file reference object. When you are done using this file reference, you must destroy it by calling EI_DestroyFileRef.

EI_MakeFileRef

```
EI_FileRef                      *EI_MakeFileRef(
        const EI_DirectoryRef   *iDirectory,
        const char              *iName,
        UInt32                  iFileType,
        const char              *iExtension);
```

iDirectory	a pointer to a directory reference object
iName	a pointer to a C string
iFileType	a file type value
iExtension	a pointer to a C string
return value	a pointer to a file reference object

EI_MakeFileRef creates and returns a completely initialized file reference object. The file reference will copy the directory reference pointed at by iDirectory to use as its own, so you can destroy iDirectory without affecting the new file reference. For example:

```
EI_DirectoryRef *dirRef;
EI_FileRef      *fileRef;

dirRef = EI_GetHomeDirectory();
fileRef = EI_MakeFileRef(dirRef, “Some File”,
                                                            cTextFileType, “txt”);
EI_DestroyDirectoryRef(dirRef);

/*      Use fileRef here.      */
```

Note that the extension string does not include a period (“.”). Also, if an extension string appears in the name, it will override the file reference’s own extension string. For example:

```
fileRef = EI_MakeFileRef(dirRef, “Some File.rsc”,
                                                            cTextFileType, “txt”);
```

This file reference object will refer to a file called “Some File.rsc,” not “Some File.txt,” or “Some File.rsc.txt.” When you are done using this file reference, you must destroy it by calling EI_DestroyFileRef.

EI_DestroyFileRef

```
void EI_DestroyFileRef(EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object

EI_DestroyFileRef destroys a file reference object that was created using any of the functions which create new file reference objects. These functions are

•EI_MakeEmptyFileRef
•EI_MakeFileRef
•EI_AskUserForNewFile
•EI_AskUserForExistingFile

EI_GetFileDirectory

```
EI_DirectoryRef *EI_GetFileDirectory(EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	a pointer to a directory reference object

EI_GetFileDirectory returns a pointer to the directory reference object used by *iFileRef. This directory reference object is the directory reference object used internally by *iFileRef, so changing the directory reference object (for example, by adding or removing names) will change the location of the file indicated by *iFileRef.

NOTE: Do not call EI_DestroyDirectoryRef on the directory reference returned by EI_GetFileDirectory.

EI_SetFileRefType

```
void EI_SetFileRefType(
        EI_FileRef *iFileRef,
        UInt32      iFileType);
```

iFileRef	a pointer to a file reference object
iFileType	a file type value

EI_SetFileRefType sets the file type bytes of the given file type. Calling EI_SetFileRefType will have no effect on the corresponding file in the user's file system. Therefore you should set the file type before using the file reference object to create or open a file (by calling EI_OpenFileStream or EI_OpenResourceFile).

EI_GetFileRefType

```
UInt32 EI_GetFileRefType(const EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	a file type value

EI_GetFileRefType returns the file reference object's file type bytes.

EI_SetFileRefExtension

```
void EI_SetFileRefExtension(
        EI_FileRef *iFileRef,
        const char *iExtension);
```

iFileRef	a pointer to a file reference object
iExtension	a pointer to a C string

EI_SetFileRefExtension sets the file reference object's extension string to the string given by iExtension. Note that iExtension should not contain a period character ("."). Calling EI_SetFileRefExtension will have no effect on the corresponding file in the user's file system. Therefore you should set the file extension before using the file reference object to create or open a file (by calling EI_OpenFileStream or EI_OpenResourceFile).

EI_GetFileRefExtensionSize

```
int EI_GetFileRefExtensionSize(const EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	an integer value

EI_GetFileRefExtensionSize returns the number of characters in the file reference object's extension string, including a terminating zero.

EI_GetFileRefExtension

```
void EI_GetFileRefExtension(
        const EI_FileRef *iFileRef,
        char             *oText,
        int              *ioNumChars);
```

iFileRef	a pointer to a file reference object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetFileRefExtension copies the file reference object's extension string to the buffer pointed at by oText. If the extension string is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetFileRefExtension will copy as many characters as will fit, including a terminating zero. EI_GetFileRefExtension will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_SetFileName

```
void EI_SetFileName(
        EI_FileRef *iFileRef,
        const char *iName);
```

iFileRef	a pointer to a file reference object
iName	a pointer to a C string

EI_SetFileName sets the file reference object's file name to the string given by iName.

EI_GetFileNameSize

```
int EI_GetFileNameSize(const EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	an integer value

EI_GetFileNameSize returns the number of characters in the file reference object's file name, including a terminating zero.

EI_GetFileName

```
void EI_GetFileName(
        const EI_FileRef *iFileRef,
        char             *iText,
        int              *ioNumChars);
```

iFileRef	a pointer to a file reference object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetFileName copies the file reference object's file name to the buffer pointed at by oText. If the file name is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetFileName will copy as many characters as will fit, including a terminating zero. EI_GetFileName will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_GetFilePathNameSize

```
int EI_GetFilePathNameSize(const EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	an integer value

EI_GetFilePathNameSize creates a platform-specific path name from the file reference object and returns the number of characters in the path name.

EI_GetFilePathName

```
void EI_GetFilePathName(
        const EI_FileRef *iFileRef,
        char             *oText,
        int              *ioNumChars);
```

iFileRef	a pointer to a file reference object
oText	a pointer to a character buffer
ioNumChars	a pointer to the number of characters in the buffer pointed to by oText

EI_GetFilePathName creates a platform-specific path name from the file reference object, then copies it to the buffer pointed at by oText. If the path name is too long to fit into the buffer, as indicated by the value of *ioNumChars, EI_GetFilePathName will copy as many characters as will fit, including a terminating zero. EI_GetFilePathName will also adjust the value of *ioNumChars to indicate how many characters were actually copied.

EI_AskUserForNewFile

```
EI_FileRef *EI_AskUserForNewFile(
        const char *iPromptText,
        int         iNumChars);
```

iPromptText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iPromptText
return value	a pointer to a file reference object

EI_AskUserForNewFile presents the user with a platform-specific file dialog box and lets the user name a new file. The string passed in iPromptText and iNumChars is displayed as a prompt in the dialog box.
When you are done using the file reference object, you must destroy it by calling EI_DestroyFileRef.

EI_AskUserForExistingFile

```
EI_FileRef *EI_AskUserForExistingFile(
        const char *iPromptText,
        int         iNumChars);
```

iPromptText	a pointer to a character buffer
iNumChars	the number of characters in the buffer pointed to by iPromptText
return value	a pointer to a file reference object

EI_AskUserForExistingFile presents the user with a platform-specific file dialog box and lets the user pick an existing file. The string passed in iPromptText and iNumChars is displayed as a prompt in the dialog box. When you are done using the file reference object, you must destroy it by calling EI_DestroyFileRef.

EI_DeleteFile

```
int EI_DeleteFile(const EI_FileRef *iFileRef);
```

iFileRef	a pointer to a file reference object
return value	an error code

EI_DeleteFile deletes the file indicated by *iFileRef from the user's file system.

Resource Files

The functions described in this section let you retrieve resources from resource files. The functions described in this section are declared in the file EI_ResourceFile.h.

EI_OpenResourceFile

```
EI_ResourceFile* EI_OpenResourceFile(const EI_FileRef *iResourceFileRef);
```

iResourceFileRef	a file reference object
return value	a pointer to a resource file object

EI_OpenResourceFile attempts to open the resource file indicated by the file reference object. If the resource file cannot be opened, EI_OpenResourceFile returns NULL.

The resource file object does not retain any connection to the file reference object, so you can destroy it as soon as the resource file object is created. For example:

```
EI_FileRef      *fileRef;
EI_ResourceFile *rsrcFile;

/*      Create fileRef here.    */

rsrcFile = EI_OpenResourceFile(fileRef);
EI_DestroyFileRef(fileRef);

/*      Use rsrcFile to read resources here.  */
```

EI_CloseResourceFile

```
void EI_CloseResourceFile(EI_ResourceFile *iResourceFile);
```

iResourceFile	a pointer to a resource file object

EI_CloseResourceFile closes a resource file that was previously opened with EI_OpenResourceFile. You must use EI_CloseResourceFile when you are done using the resource file. When you call EI_CloseResourceFile, all resources previously obtained from the resource file using the EI_GetResource function are disposed.

EI_GetResource

```
void *EI_GetResource(
       EI_ResourceFile *iResourceFile,
       UInt32  iResourceType,
       int     iResourceID,
       SInt32  *oResourceSize);
```

iResourceFile	a pointer to a resource file object
iResourceType	a resource type
iResourceID	a resource ID
oResourceSize	a pointer to an integer value
return value	a pointer to a resource

EI_GetResource reads resource data from a resource file. The resource of type iResourceType and with ID iResourceID is read into a memory and a pointer to the resource memory is returned. If oResourceSize is not NULL, *oResourceSize is set to the size in bytes of the resource data.

If the resource cannot be read into memory for any reason, EI_GetResource returns NULL.

EI_ReleaseResource

```
void EI_ReleaseResource(void *iResource);
```

iResource	a pointer to a resource

EI_ReleaseResource disposes of the resource pointed at by iResource. When you are done with a resource, you should call EI_ReleaseResource.

NOTE: If you do not call EI_ReleaseResource, the resource will be disposed when you call EI_CloseResourceFile.

File I/O

The functions described in this section allow you to perform file I/O. The functions described in this section are declared in the file EI_Stream.h.

EI_OpenFileStream

```
EI_Stream *EI_OpenFileStream(
        const EI_FileRef *iFileRef,
        EI_StreamAccess   iAccess);
```

iFileRef	a pointer to a file reference object
iAccess	a stream access value
return value	a stream object

EI_OpenFileStream opens the file indicated by *iFileRef for I/O using the access value iAccess. The possible values of iAccess, and their behavior, are

cEI_ReadOnly	If the file indicated by *iFileRef does not exist, the function result is NULL. Otherwise the file is opened for read access and a stream object is constructed and returned.
cEI_WriteOnly	If the file does not exist, the file is created. Then the file is opened for write access and a stream object is constructed and returned.
cEI_ReadWrite	If the file does not exist, the file is created. Then the file is opened for read-write access and a stream object is constructed and returned.

In any case, if the file cannot be opened or the stream object cannot be created, EI_OpenFileStream returns NULL.

Once you have finished using the stream object, you must close it by calling EI_CloseStream.

EI_CloseStream

```
void EI_CloseStream(EI_Stream *iStream);
```

iStream a pointer to a stream object

EI_CloseStream closes the file associated with the stream object, and then destroys the stream object.

EI_SetStreamPosition

```
int EI_SetStreamPosition(
        EI_Stream *iStream,
        slong iPosition);
```

iStream	a pointer to a stream object
iPosition	a stream position
return value	an error code

EI_SetStreamPosition sets the position of the stream to iPosition. The position of the stream is the position from within the stream at which the next read or write operation occurs. Valid values for the stream's position are between 0 and the stream's size, inclusive. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_GetStreamPosition

```
slong EI_GetStreamPosition(EI_Stream *iStream);
```

iStream	a pointer to a stream object
return value	a stream position

EI_GetStreamPosition returns the stream’s current position. If an error occurs, the return value will be negative.

EI_GetStreamSize

```
slong EI_GetStreamSize(EI_Stream *iStream);
```

iStream	a pointer to a stream object
return value	a stream size

EI_GetStreamSize returns the number of bytes contained in the stream. If an error occurs, the return value will be negative.

```
EI_ReadStream
       int EI_ReadStream(
       EI_Stream *iStream,
       char      *oBuffer,
       SInt32    *ioCount);
```

iStream	a pointer to a stream object
oBuffer	a pointer to a data buffer
ioCount	a pointer to the number of bytes in the buffer pointed to by oBuffer
return value	an error code

EI_ReadStream reads raw data from the stream to the buffer pointed to by oBuffer. The number of bytes to read is passed in *ioCount; if an error occurs, *ioCount will be set to the actual number of bytes read. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteStream

```
int EI_WriteStream(
       EI_Stream *iStream,
       char      *iBuffer,
       SInt32    *ioCount);
```

iStream	a pointer to a stream object
iBuffer	a pointer to a data buffer
ioCount	a pointer to the number of bytes in the buffer pointed to by iBuffer
return value	an error code

EI_WriteStream writes raw data from the stream to the buffer pointed to by oBuffer. The number of bytes to write is passed in *ioCount; if an error occurs, *ioCount will be set to the actual number of bytes read. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteUInt8

```
int EI_WriteUInt8(
        EI_Stream     *iStream,
        unsigned char iData);
```

iStream	a pointer to a stream object
iData	an unsigned char value
return value	an error code

EI_WriteUInt8 writes an unsigned char value to the stream as an 8 bit value. The function result is an error code; zero indicates success, non-zero values indicate failure.

NOTE: EI_WriteUInt8 does not perform byte-swapping; it is provided for completeness.

EI_WriteSInt16

```
int EI_WriteSInt16(
        EI_Stream *iStream,
        short      iData);
```

iStream	a pointer to a stream object
iData	a short value
return value	an error code

EI_WriteSInt16 writes a short value to the stream as a 16 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteUInt16

```
int EI_WriteUInt16(
        EI_Stream      *iStream,
        unsigned short iData);
```

iStream	a pointer to a stream object
iData	an unsigned short value
return value	an error code

EI_WriteUInt16 writes an unsigned short value to the stream as a16 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteSInt32

```
int EI_WriteSInt32(
        EI_Stream *iStream,
        SInt32     iData);
```

iStream	a pointer to a stream object
iData	a long value
return value	an error code

EI_WriteSInt32 writes a long value to the stream as a 32 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteUInt32

```
int EI_WriteUInt32(
        EI_Stream *iStream,
        UInt32     iData);
```

iStream	a pointer to a stream object
iData	an unsigned long value
return value	an error code

EI_WriteUInt32 writes an unsigned long value to the stream as a 32 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteFloat32

```
int EI_WriteFloat32(
        EI_Stream *iStream,
        float      iData);
```

iStream	a pointer to a stream object
iData	a short value
return value	an error code

EI_WriteFloat32 writes a float value to the stream as a 32 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_WriteFloat64

```
int EI_WriteFloat64(
        EI_Stream *iStream,
        double     iData);
```

iStream	a pointer to a stream object
iData	a short value
return value	an error code

EI_WriteFloat64 writes a double value to the stream as a 64 bit value (byte-swapped if necessary). The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadUInt8

```
int EI_ReadUInt8(
        EI_Stream     *iStream,
        unsigned char *oData);
```

iStream	a pointer to a stream object
oData	a pointer to an unsigned char value
return value	an error code

EI_ReadUInt8 reads an 8 bit unsigned char value from the stream and assigns it to *oData. The function result is an error code; zero indicates success, non-zero values indicate failure.

NOTE: EI_ReadUInt8 does not perform byte-swapping; it is provided for completeness.

EI_ReadSInt16

```
int EI_ReadSInt16(
        EI_Stream *iStream,
        short     *oData);
```

iStream	a pointer to a stream object
oData	a pointer to a short value
return value	an error code

EI_ReadSInt16 reads a 16 bit short value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadUInt16

```
int EI_ReadUInt16(
        EI_Stream      *iStream,
        unsigned short *oData);
```

iStream	a pointer to a stream object
oData	a pointer to an unsigned short value
return value	an error code

EI_ReadUInt16 reads a 16 bit unsigned short value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadSInt32

```
int EI_ReadSInt32(
        EI_Stream *iStream,
        SInt32    *oData);
```

iStream	a pointer to a stream object
oData	a pointer to a long value
return value	an error code

EI_ReadSInt32 reads a 32 bit long value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadUInt32

```
int EI_ReadUInt32(
        EI_Stream *iStream,
        Uint32    *oData);
```

iStream	a pointer to a stream object
oData	a pointer to an unsigned long value
return value	an error code

EI_ReadUInt32 reads a 32 bit unsigned long value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadFloat32

```
int EI_ReadFloat32(
        EI_Stream *iStream,
        float      *oData);
```

iStream	a pointer to a stream object
oData	a pointer to a float value
return value	an error code

EI_ReadFloat32 reads a 32 bit float value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

EI_ReadFloat64

```
int EI_ReadFloat64(
        EI_Stream *iStream,
        double    *oData);
```

iStream	a pointer to a stream object
oData	a pointer to a double value
return value	an error code

EI_ReadFloat64 reads a 64 bit double value from the stream and assigns it to *oData, performing byte-swapping if necessary. The function result is an error code; zero indicates success, non-zero values indicate failure.

Color Picking

The functions described in this section allow you to bring up a color picker dialog box programmatically. The functions described in this section are declared in the file EI_ColorPicker.h.

EI_PickColor

```
int EI_PickColor(
        const char     *iPromptString,
        const EI_Color *iOriginalColor,
        EI_Color       *oNewColor);
```

iPromptString	a pointer to a C string
iOriginalColor	a pointer to a color
oNewColor	a pointer to a color
return value	a boolean value

EI_PickColor presents the color picker dialog box to the user and allows the user to pick a new color. The string given by iPromptString is displayed in the color picker to prompt the user. The original color presented in the dialog box is given by *iOriginalColor. If the user clicks the OK button, the new color is copied to *oNewColor and the function returns true. If the user clicks the Cancel button, the function returns false and *oNewColor is unaffected.

When you present the color picker dialog box with EI_PickColor, color picker does not allow the user to edit an alpha value.

EI_PickColorWithAlpha

```
int EI_PickColorWithAlpha(
        const char     *iPromptString,
        const EI_Color *iOriginalColor,
        unsigned char  iOriginalAlpha,
        EI_Color       *oNewColor,
        unsigned char  *oNewAlpha);
```

iPromptString	a pointer to a C string
iOriginalColor	a pointer to a color
iOriginalAlpha	an unsigned char value
oNewColor	a pointer to a color
oNewAlpha	a pointer to an unsigned char value
return value	a boolean value

EI_PickColorWithAlpha is similar to EI_PickColor, but allows the user to edit an alpha value as well as a color. The string given by iPromptString is displayed in the color picker to prompt the user. The original color presented in the dialog box is given by *iOriginalColor, with the alpha passed in iOriginalAlpha. If the user clicks the OK button, the new color is copied to *oNewColor, the new alpha is copied to *oNewAlpha, and the function returns true. If the user clicks the Cancel button, the function returns false and *oNewColor and *oNewAlpha are unaffected.

QuickTime Support

The functions described in this section allow you to use QuickTime API calls in your dialog box. The way to do this is to use a user control, and write a user control draw function and a user control click function that calls through to the QuickTime API. The strategy taken is to allow conversion from EI API data types to QuickTime data types. The functions described in this section are declared in the file EI_QuickTime.h.

NOTE: When you open a QuickTime movie, you must make sure to set the current GrafPort and GDevice. See the description of EI_DialogToWindowPtr and EI_ImageBufferToGWorld for more.

EI_GetDialogFrameOffset

```
void EI_GetDialogFrameOffset(
       const EI_Dialog *iDialog,
       int              *oOffsetX,
       int              *oOffsetY);
```

iDialog	a pointer to a dialog box object
oOffsetX	a pointer to an integer value
oOffsetY	a pointer to an integer value

EI_GetDialogFrameOffset returns in *oOffsetX and *oOffsetY the dialog's frame offset. The EI API draws a dialog box's frame and title bar within an operating system window. When you make API drawing calls, the API adjusts the coordinates of the drawing operation to account for the width and height of the dialog box's frame. However, if you are going to display a QuickTime movie in a dialog box you will need to perform the coordinate offset yourself. The values returned in *oOffsetX and *oOffsetY are the horizontal and vertical offsets needed to adjust a location in API coordinates to the underlying window's coordinates. For example, to display a QuickTime movie in a user control you will need to call SetMovieBox:

```
EI_Dialog   *myDialog;
Movie        myMovie;
EI_Control  *control;
EI_Rect      bounds;
Rect         qtRect;
int          offsetX, offsetY;

/* Initialize myDialog, myMovie here... */

/* Get the control's bounds. */
control = EI_FindControlByID(myDialog, cMyUserControlID);
EI_GetControlBounds(control, &bounds);

/* Copy it into a QuickTime (Mac) Rect:
qtRect.left = bounds.left;
qtRect.top = bounds.top;
qtRect.right = bounds.right;
qtRect.bottom = bounds.bottom;

/*      Get the dialog's frame offset and offset the QuickTime
Rect. */
EI_GetDialogFrameOffset(myDialog, &offsetX, &offsetY);
MacOffsetRect(&qtRect, (short) offsetX, (short) offsetY);

/* Now set the movie's bounds. */
SetMovieBox(myMovie, &qtRect);
```

Note that if you set up a movie for display into an EI_ImageBuffer, you do not need to perform any offset operation.

EI_FileRefToFSSpec - *Not 64-bit compatible*

```
int EI_FileRefToFSSpec(
        const EI_FileRef *iFileRef,
        FSSpec           *oFSSpec);
```

iFileRef	a pointer to a file reference object
oFSSpec	a pointer to an FSSpec structure
return value	an error code

EI_FileRefToFSSpec converts a file reference object to an FSSpec which can be used in QuickTime calls. The return value is 0 if the conversion completed successfully, or non-zero if the conversion failed.

The FSSpec returned in *oFSSpec and the file reference are not linked in any way; specifically, you can destroy the file reference *iFileRef and still use *oFSSpec.

EI_DialogToWindowPtr - *Not 64-bit compatible*

```
int EI_DialogToWindowPtr(
        EI_Dialog *iDialog,
        WindowPtr *oWindowPtr);
```

iDialog	a pointer to a dialog box object
oWindowPtr	a pointer to a WindowPtr
return value	an error code

EI_DialogToWindowPtr converts a dialog box object to a WindowPtr which can be used in QuickTime calls. The return value is 0 if the conversion completed successfully, or non-zero if the conversion failed.

The EI API does not make any guarantees about how the current GrafPort and GDevice are set at any given time. Since QuickTime will initialize the movie’s GWorld based on the current GrafPort and GDevice when the movie is opened, you should explicitly set them before creating the movie. Using EI_DialogToWindowPtr, you can set the current GrafPort to the dialog box. For example:

```
EI_Dialog   *myDialog;
EI_FileRef *movieFileRef;
FSSpec      movieSpec;
OSErr       errCode;
short       movieResFile;
WindowPtr   window;

/*      Get a reference to a movie file and convert it to
        an FSSpec. */
movieFileRef = EI_AskUserForExistingFile
                                             (“Choose a movie file”);
if (EI_FileRefToFSSpec(movieFileRef, &movieSpec) != 0) {
        EI_DestroyFileRef(movieFileRef);
        return;
}

EI_DestroyFileRef(movieFileRef);

/*      Convert the dialog box to a WindowPtr and set the
        current GrafPort to it. */
```

```
if (EI_DialogToWindowPtr(myDialog, &window) != 0)
       return;
SetGWorld(GetWindowPort(window), GetMainDevice());

/*     Now open the movie. */
errCode = OpenMovieFile(&movieSpec, &movieResFile,
                                                     fsRdPerm);
```

Because the WindowPtr returned in *oWindowPtr refers to the same object referred to by *iDialog, you cannot use *oWindowPtr after destroying *iDialog.

EI_ImageBufferToGWorld - Not 64-bit compatible

```
int EI_ImageBufferToGWorld(
       EI_ImageBuffer *iImageBuffer,
       GWorldPtr *oGWorld);
```

iImageBuffer	a pointer to an image buffer
oGWorld	a pointer to a GWorldPtr
return value	an error code

EI_ImageBufferToGWorld converts an image buffer object to a GWorldPtr which can be used in QuickTime calls. The return value is 0 if the conversion completed successfully, or non-zero if the conversion failed.
The EI API does not make any guarantees about how the current GrafPort and GDevice are set at any given time. Since QuickTime will initialize the movie's GWorld based on the current GrafPort and GDevice when the movie is opened, you should explicitly set them before creating the movie. Using EI_ImageBufferToGWorld, you can set the current GrafPort to the image buffer. For example:

```
EI_Dialog  *myDialog;
EI_FileRef *movieFileRef;
FSSpec      movieSpec;
OSErr       errCode;
short       movieResFile;
GWorldPtr    myGWorld;

/*     Get a reference to a movie file and convert it to
       an FSSpec. */
movieFileRef = EI_AskUserForExistingFile
                                        ("Choose a movie file");
if (EI_FileRefToFSSpec(movieFileRef, &movieSpec) != 0) {
       EI_DestroyFileRef(movieFileRef);
       return;
}

EI_DestroyFileRef(movieFileRef);

/*     Convert the image buffer to a GWorldPtr and set the
       current GrafPort to it. */
if (EI_ImageBufferToGWorld(myImageBuffer, &myGWorld) != 0)
       return;
SetGWorld(myGWorld, GetGWorldDevice(myGWorld));

/*     Now open the movie. */
errCode = OpenMovieFile(&movieSpec, &movieResFile,
                                              fsRdPerm);
```

Because the GWorldPtr returned in *oGWorld refers to the same object referred to by *iImageBuffer, you cannot use *oGWorld after destroying *iImageBuffer.

Timers

The functions described in this section allow you to write simple callback functions which can get called periodically when the system is idle. One use of the timer system is to write a timer function that can call the QuickTime function MoviesTask. The functions described in this section are declared in the header file EI_Timer.h.

EI_MakeTimer

```
EI_Timer *EI_MakeTimer(void);
```

return value	a pointer to a timer object

EI_MakeTimer creates and returns a timer object. If the timer cannot be created, the function result is NULL. When you are through with the timer object, you must destroy it by calling EI_DestroyTimer.

EI_DestroyTimer

```
void EI_DestroyTimer(EI_Timer *iTimer);
```

iTimer	a pointer to a timer object

EI_DestroyTimer destroys a timer object previously created with EI_MakeTimer. If the timer is currently scheduled for execution, it is stopped before being destroyed.

EI_SetTimerFunction - *Not 64-bit compatible*

```
void EI_SetTimerFunction(
        EI_Timer           *ioTimer,
        EI_TimerFunction   iFunction);
```

ioTimer	a pointer to a timer object
iFunction	a pointer to a timer function

EI_SetTimerFunction sets the timer object’s timer function. When the timer object is scheduled for execution (by calling either EI_StartPeriodicTimer or EI_StartOneShotTimer), the timer function will be called when the timer fires.

EI_SetTimerExtraData - *Not 64-bit compatible*

```
void EI_SetTimerExtraData(
        EI_Timer  *ioTimer,
        void      *iExtraData);
```

ioTimer	a pointer to a timer object
iExtraData	a pointer

EI_SetTimerExtraData allows you to associate extra data with a timer object. The extra data is analogous to the extra data used with control objects and dialog box objects. A timer object's extra data can be retrieved by calling EI_GetTimerExtraData.

EI_GetTimerExtraData - *Not 64-bit compatible*

```
void *EI_GetTimerExtraData(const EI_Timer *iTimer);
```

iTimer	a pointer to a timer object
return value	a pointer

EI_GetTimerExtraData returns the extra data pointer associated with the timer object *iTimer. If the timer object has no extra data pointer, EI_GetTimerExtraData returns NULL.

EI_StartPeriodicTimer

```
int EI_StartPeriodicTimer(
        EI_Timer *iTimer,
        int      iMilliseconds);
```

iTimer	a pointer to a timer object
iMilliseconds	an integer value
return value	an error code

EI_StartPeriodicTimer schedules a timer object for future execution. When a timer object is scheduled using EI_StartPeriodicTimer, the timer object's timer function will be called once every iMilliseconds milliseconds. If the timer is successfully scheduled, EI_StartPeriodicTimer returns zero; otherwise a non-zero value is returned. EI_StartPeriodicTimer can be useful for scheduling repetitive actions that must occur frequently. For example, here is how you could write a timer function to call the QuickTime function MoviesTask:

```
static EI_Timer *sQTTimer = NULL;

/*      The timer function just calls MoviesTask. */
void MyTimerFunction(EI_Timer *iTimer)

{
        MoviesTask(NULL, DoTheRightThing);
}

void MyInitTimer(void)

{
        /*      Create the timer. */
        sQTTimer = EI_MakeTimer();

        /*      Set the timer function. */
        EI_SetTimerFunction(sQTTimer, MyTimerFunction);

        /*      Set up timer for approximately 30 calls/second */
        EI_StartPeriodicTimer(sQTTimer, 33);
}

void MyCleanUpTimer(void)

{
        /*      No need to stop the timer; EI_DestroyTimer does it
                for us. */
        EI_DestroyTimer(sQTTimer);
}
```

EI_StartOneShotTimer

```
int EI_StartOneShotTimer(
        EI_Timer *iTimer,
        int      iMilliseconds);
```

iTimer	a pointer to a timer object
iMilliseconds	an integer value
return value	an error code

EI_StartOneShotTimer schedules a timer object for future execution. When a timer object is scheduled using EI_StartOneShotTimer, the timer object's timer function will be called exactly once iMilliseconds milliseconds into the future, after which the timer will become unscheduled as if you had called EI_StopTimer. If the timer is successfully scheduled, EI_StartOneShotTimer returns zero; otherwise a non-zero value is returned.

EI_StopTimer

```
int EI_StopTimer(EI_Timer *iTimer);
```

iTimer	a pointer to a timer object
return value	an error code

EI_StopTimer stops a previously scheduled timer. After calling EI_StopTimer, the timer object is no longer scheduled for execution. If the timer is successfully stopped, EI_StopTimer returns zero; otherwise a non-zero value is returned.

Index