palmsource™

User Interface

Exploring Palm OS®

Written by Jean Ostrem
Edited by Greg Wilson
Engineering contributions by Laz Marhenke, Dianne Hackborn, Joe Onorato, Dan Sandler, Mathias Agopian, David Fedor, Ezekiel Sanborn de Asis, JB Parrett, and Raj Mojumder

Exploring Palm OS: User Interface
Document Number 3109-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

Part II: Reference

16 Control Reference 277

29 Scalable Font Reference 587

About This Document

This book describes how to programmatically work with the user interface (UI) elements that are part of the Palm OS® UI Library. It introduces each of the user interface elements and describes the Palm OS APIs that aid in working with the user interface.

This book focuses only on the tasks you perform by writing C code. Specifically, it does *not* cover the following:

- How to use a resource editor to create UI elements. See the documentation that came with your development environment, or see the book *Resource File Formats*, which describes the XML format used to define resources.
- How to design an interface that is user-friendly and conforms to Palm OS guidelines. See the book *Palm OS User Interface Guidelines* for this type of information.

IMPORTANT: The *Exploring Palm OS* series is intended for developers creating native applications for Palm OS Cobalt. If you are interested in developing applications that work through PACE and that also run on earlier Palm OS releases, read the latest versions of the *Palm OS Programmer's API Reference* and *Palm OS Programmer's Companion* instead.

Who Should Read This Book

You should read this book if you are a Palm OS software developer who writes applications, pinlets, or any other type of executable that displays a user interface.

Because virtually all Palm OS developers require some knowledge of how to present a user interface, this book is intended for developers with all levels of familiarity with Palm OS, from novice to expert. Novice programmers should first read *Exploring Palm OS: Programming Basics* to gain an understanding of the basic structure of a Palm OS application.

Expert Palm OS programmers will find that much of the Palm OS UI library is familiar and may want to just skim this book. Differences between the Palm OS Garnet API set and the Palm OS Cobalt API set are outlined in *Exploring Palm OS: Porting Applications to Palm OS Cobalt*.

What This Book Contains

This book contains the following information:

- Part I contains conceptual information and how-to information:
  - Chapter 1, "The Display," on page 3 describes display-level concepts and the basic display layout that users see. These are concepts that apply in all of the chapters that follow.
  - Chapter 2, "Working with Forms and Dialogs," on page 15 describes how to work with forms and dialogs.
  - Chapter 3, "Working with Controls," on page 41 describes how to work with controls: buttons, push buttons, check boxes, sliders, and so on.
  - Chapter 4, "Working with Menus," on page 73 describes how to work with menu bars and menu items.
  - Chapter 5, "Displaying Text," on page 79 describes how to display text in text fields, labels, and form titles. It also describes how to work with fonts.
  - Chapter 6, "Working with Tables and Lists," on page 109 describes how to work with tables and with lists.
  - Chapter 7, "Creating Custom UI Elements (Gadgets)," on page 121 describes how to create your own user interface elements.
  - Chapter 8, "Drawing," on page 123 describes how to do custom drawing to the screen.
  - Chapter 9, "Working with Bitmaps," on page 147 describes how to display bitmaps.
  - Chapter 10, "Modifying the UI Color List," on page 163 describes the UI color list and how to change it.

  - Chapter 11, "Integrating with the Application Launcher," on page 167 describes how to display your application in the application launcher.

- Part II contains reference information on the various portions of the UI Library.

Changes to This Book

3109-003

- The default winding rule is odd. See "Take Advantage of Winding Rule" on page 141.
- `FrmGetBitmapHandle()` and `GcLoadBitmap()` both accept PNGs as well as BitmapType structures. Clarified when to use `FrmGetBitmapHandle()`. See Chapter 9, "Working with Bitmaps," on page 147.
- Removed the warning about accessing `FieldAttrType` structure members directly.

3109-002

- Modified descriptions of `CatMgrSelectFilter()`, `FldReleaseStorage()`, `TableDrawItemFuncType()`, `TblSelectItem()`, and made minor corrections to other table function descriptions.
- Modified descriptions of the drawing transformation functions and gradient functions. Added sample code to `GcRotate()`.
- Minor editorial corrections.

3109-001

- Initial version.

The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm

operating system, and contains both conceptual and reference documentation for the pertinent technology.

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*

Additional Resources

- Documentation

  PalmSource publishes its latest versions of documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions

(FAQs), sample code, white papers, and the development documentation at

http://www.palmos.com/dev/support/kb/

Part I
Concepts

This part contains conceptual and how-to information for the UI Library. It covers:

1

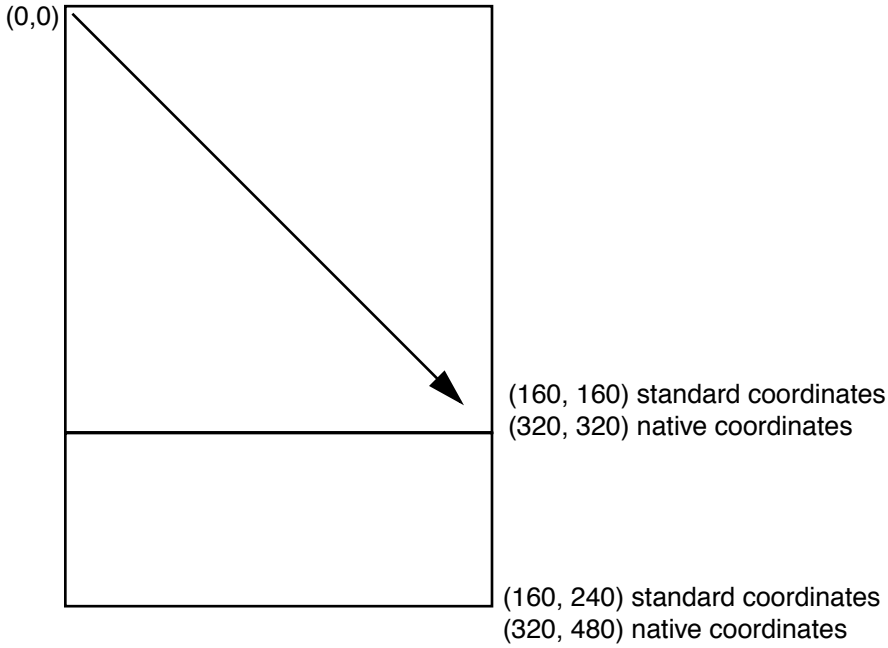
The Display

This chapter describes basic concepts you should be familiar with before you design a user interface. It discusses the display capabilities of Palm OS® and the typical user interface you see on a Palm Powered™ device running Palm OS Cobalt.

Display Coordinate System

As shown in Figure 1.1, the origin is at the top-left of the screen and coordinates grow downward and to the right.

Figure 1.1 Coordinate systems



Palm OS uses one of two coordinate systems, standard or native, depending on the situation.

The **standard coordinate system** that Palm OS uses is always 160 coordinates wide. On a square screen device, the dimensions are 160 X 160. On a device with a dynamic input area, the screen dimensions are 160 X 240 coordinates.

The **native coordinate system** has a one-to-one mapping between coordinates and actual pixels on the screen. Thus, the native coordinate system changes depending on the **display density** of the device. Palm OS Cobalt version 6.0 currently supports only double-density screens, which have 320 X 320 or 320 X 480 pixels. Other versions of Palm OS support single-density screens (160 X 160 or 160 X 240 pixels) and 1.5-density screens (240 X 240 or 240 X 320). Note that double-density screens tend to be the same physical size as single-density screens; they just pack more pixels into the same space (see Figure 1.1).

Palm OS Cobalt uses the standard coordinate system to draw standard user interface elements such as forms, buttons, and menus. If you are doing custom drawing with the Graphics Context

functions described in Chapter 8, "Drawing," on page 123, it uses the coordinate system of the current draw window by default, which in most cases is the standard coordinate system. If you want to use the native coordinate system when drawing, you must explicitly set the drawing functions to use the native coordinate system.

Fonts and bitmaps require special handling that depends on how the font or bitmap is created. See "How Palm OS Displays Bitmapped Fonts" on page 101 and "How Palm OS Displays Bitmaps" on page 154.

When designing a user interface, it is usually better to think in terms of standard coordinates rather than pixels. The operating system takes care of mapping coordinates to physical pixels, and drawing functions work in terms of coordinates. So, for example, if you draw a line of text using one of the system bitmapped fonts, that line of text is 12 standard coordinates high. Depending on the display density, that text might be 12 pixels high, 24 pixels high (on a double-density display), 18 pixels high (on a 1.5-density display,) or some other multiple of 12. You can work in native coordinates if you need finer control, but you'll lose device independence.

Color

All Palm OS Cobalt displays are direct color displays. With **direct color displays**, the value stored into each pixel location specifies the amount of red, green, and blue components directly. For example, a 16-bit direct color display could have 5 bits of each pixel assigned as the red component, 6 bits as the green component, and 5 bits as the blue component.

Earlier releases supported color tables of 1, 2, 4, and 8 bits per pixel. Palm OS Cobalt supports these display depths only in software. If an application changes the display depth, the system creates a color lookup table to translate between the emulated bit depth and the actual bit depth. The lookup table has no effect on the display hardware, since the hardware derives the color from the red, green, and blue bits stored in each pixel location of the frame buffer.

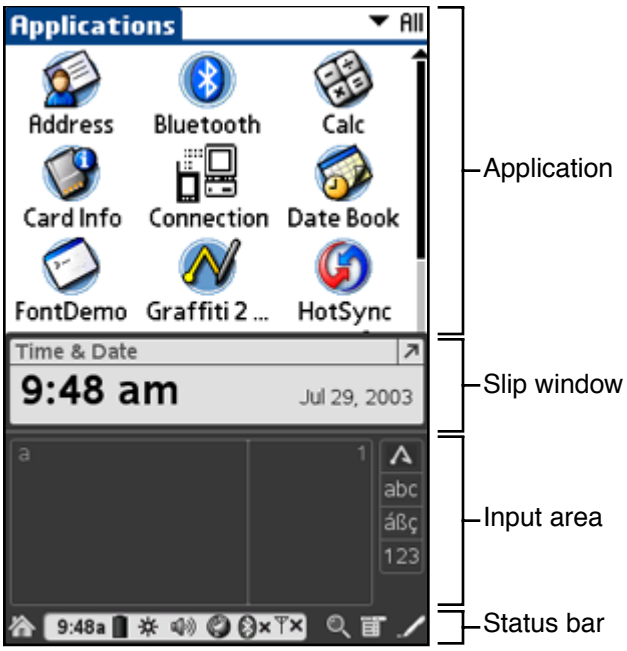
Display Layout

The Palm OS user interface consists of drawing regions called **windows**. A **form** uses a window to display other user interface elements. Menus and pop-up lists are also types of windows.

As shown in Figure 1.2, more than one process may display a window on the screen at the same time. In Figure 1.2, there are four different windows:

- The **status bar**, which runs in the system process
- The **input area** displays a user interface provided by a **pinlet**, which runs in the background process
- The **slip window** displays a user interface provided by a **slip**, which also runs in the background process
- The application window displayed by the application, which runs in the application process

Figure 1.2 Palm OS Cobalt user interface



TIP: Not all devices have all of these aspects of the user interface. For example, a device with a built-in keyboard would not have the input area.

Status Bar

The status bar shows the status of important aspects of the system, for example, signal strength, battery life, roaming status, and the presence of awaiting messages. It is also the way users access the Launcher application and the Find facility and open or close the dynamic input area.

The status bar is a full-width window located at the bottom of the display immediately below the input area. The status bar is always present on devices that have screens that are 160 X 240 standard coordinates (320 X 480 double density displays) unless the application specifically hides it. On traditional square screen devices (160 X 160 standard coordinates), the status bar is only displayed when the Launcher application is active. Other applications may choose to display the status bar as well.

Input Area

The **input area** is the area of the display where the user enters textual data. There are two kinds of input areas: static and dynamic. A **static input area** is one that is silk-screened onto the device. A **dynamic input area** is one that is implemented in software. Users can collapse a dynamic input area when they want to see more of the application and expand it when they want to enter more data.

A **pinlet** is a character input system that displays its user interface in the input area. Its job is to receive the pen events in the input area and translate them into characters that are sent to the application (or the currently active window).

The book *Exploring Palm OS: Input Services* explains how to write a pinlet.

Slip Window

A **slip window** displays when the user taps the icon in the status bar. For example, if the user taps a network connection icon, a slip window displays a user interface that allows the user to connect to the Internet. **Slips** run in the background process and display a user interface in the slip window.

Window Sizing and Placement

On a rectangular screen, the sizing and placement of windows is dynamic. The user can close the input area giving more space to the other displayed windows.

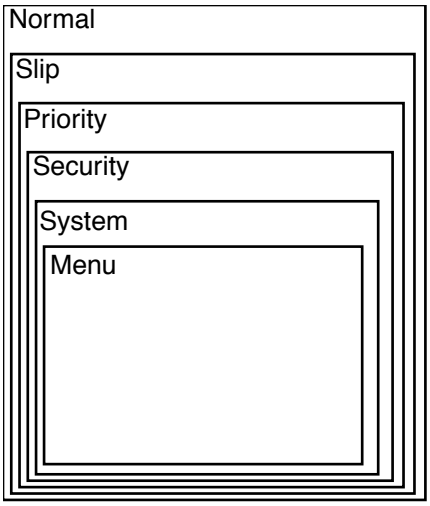
The Palm OS Window Manager determines the size and location of all windows. To do so, it uses two attributes that are contained in the `WINDOW_CONSTRAINTS_RESOURCE` that you add to each form. These attributes are the window layer and the constraints.

The **window layer** specifies which window may appear on top of another window if a sizing conflict occurs. The possible window layers are:

- Menu — Used for menus
- System — Used for the dynamic input area, the status bar, and catastrophic system messages such as the low battery alert dialog
- Security — Used for password dialogs and other security-related dialogs.
- Priority — Used for Palm OS system dialogs
- Slip — Used for slip windows
- Normal — Used for all application windows other than menus

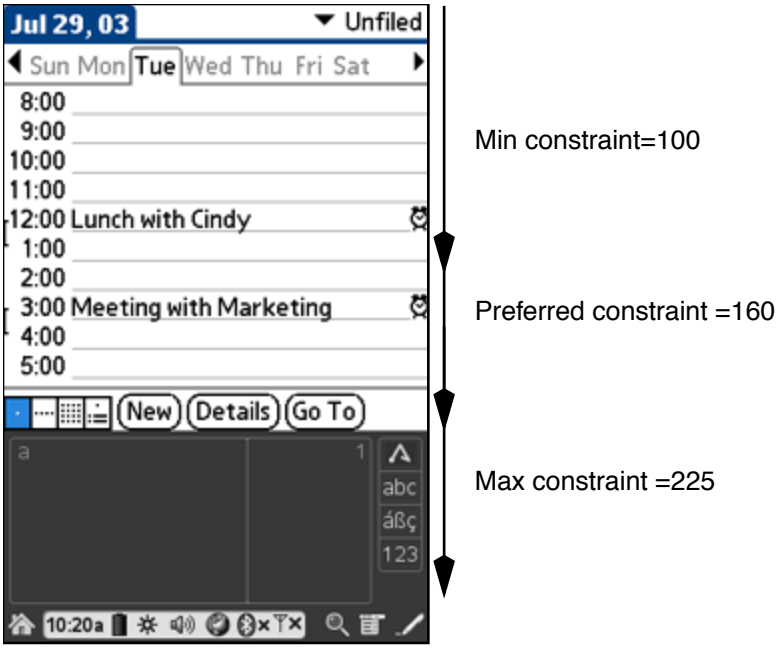
Windows in a higher layer obscure the windows in the lower layer if necessary. See Figure 1.3.

Figure 1.3 Window layers



The **constraints** specify the location and possible sizes a window may have. You specify minimum, maximum, and preferred horizontal and vertical size (see Figure 1.4).

Figure 1.4 Example constraints



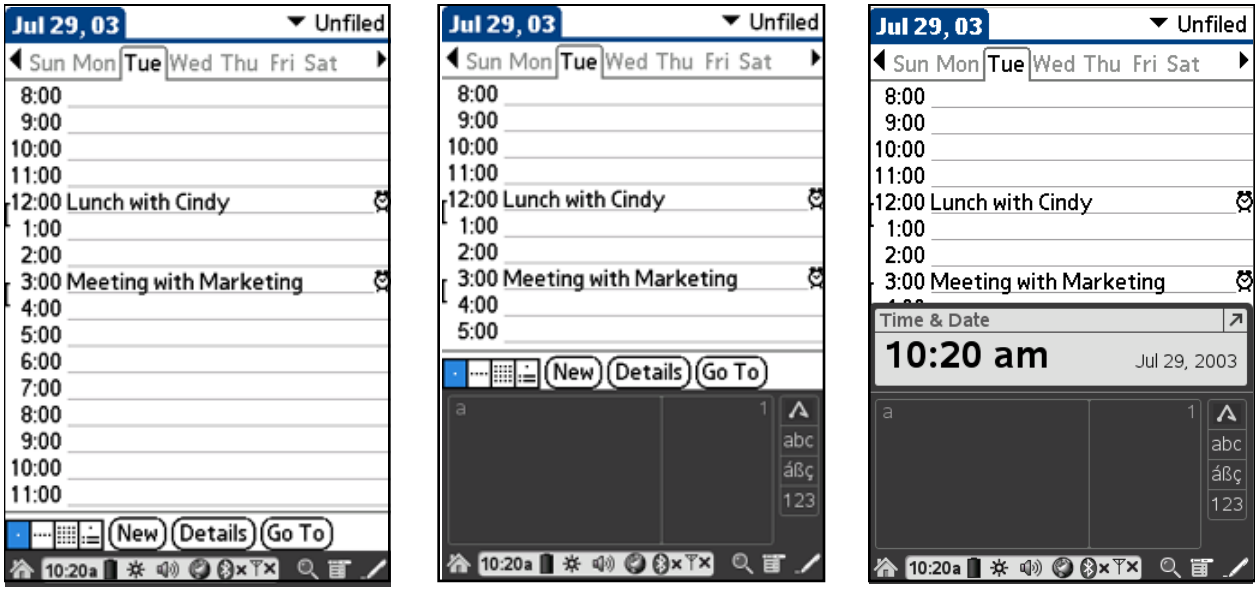
The Window Manager decides how to size the windows from the bottom of the display up:

- The status bar is always displayed and is always 15 standard coordinates high.
- The input area is displayed at the pinlet's preferred height if possible. If the application's preferred height conflicts with the pinlet's preferred height, the pinlet constraint overrides the application's constraint.

  If the pinlet's preferred height does not allow enough space for the application's minimum height, the pinlet's minimum height is used. If there still isn't enough room for the application, the input area overlaps the application window.
- The slip window is displayed above the input area and is always shown at the slip's preferred height. It obscures the application window because it is in a different window layer than the application.

Figure 1.5 shows some examples of how the windows dynamically resize.

Figure 1.5 Sizing and placement examples



App is maximum size.

User opens input area. App resizes to fit remaining space.

User launches slip. Slip overlaps app.

When the Window Manager has determined the size of all visible windows, it sends the `winResizedEvent` to each of them specifying their new bounds. They respond by rearranging the user interface to fit within the available bounds. Chapter 2, "Working with Forms and Dialogs," on page 15 provides more information about setting constraints for a form and responding to the `winResizedEvent`.

Receiving Events

Each of the threads displaying a user interface has its own event queue and receives only the events for that thread. Because each thread has a separate event queue, a window displayed by that thread receives only the events (pen taps, and so on) intended for that thread. If the user taps a day in the Date Book, the event is sent to the application, but if the user taps an icon in the status bar, the status bar receives the event.

The application being displayed is considered to have the **main event queue**. The status bar, the pinlet, and any other process displaying a window are considered to have secondary event queues.

Modal Windows

A window in any thread can be modal. **Modal windows** (or **modal dialogs**) tend to be shorter than their modeless counterparts and should have a different look.

If a window is modal, it receives the events intended for it and for all windows in all layers below its own layer. If you create a modal window in the normal layer, the modal window receives all pen taps in the application's form. For example, if the user is displaying the Details dialog of a Date Book application and taps in the title bar of the main Date Book form, the modal dialog receives the event. (Note that the event received is a pen event, not a title select event.)

If the modal window is in the priority layer, it receives all events for all windows in the priority layer, the slip layer, and the normal layer.

The dynamic input area and status bar windows are in the system layer; therefore, as long the modal dialog is not in the system or menu layer, the status bar and dynamic input area still receive their events.

Input Focus

Because each thread displaying a window has a separate event queue, Palm OS needs a way to determine which event queue should receive character input. For this reason, windows have a notion of **input focus**. The window with the input focus is the one that receives the key events. In general, the user decides which window has the focus by tapping the pen in that window.

The events `winFocusGainedEvent` and `winFocusLostEvent` are sent to windows as they gain and lose the input focus.

Most threads should not care if they have or do not have the focus. Threads only receive the events intended for them, so you simply respond to the events you receive.

Note the following about the input focus events:

- `winFocusGainedEvent` and `winFocusLostEvent` are only sent regarding focusable windows. Menus and pop-up lists are not focusable windows, so the display of a menu does not generate one of these events. The input area's window is also not focusable.
- When a form is first displayed, it requests input focus. That request might be denied if another thread requests the input focus for its window and its window is in a higher layer than the form's window is. The form will still be displayed, but it will not receive the `keyDownEvent`s until the user gives that form the input focus.
- Some slip windows are not forms and do not use the UI Library. If such a slip window is opened, it does not request the input focus if it does not need to receive `keyDownEvents`. For example, if you open the Time and Date slip, the application still has the input focus.
- You may or may not receive a `winEnterEvent` or `winExitEvent` around the same time as a

`winFocusGainedEvent` and `winFocusLostEvent`. The two types of events work independently of each other.

`winFocusGainedEvent` and `winFocusLostEvent` are sent asynchronously and are only about which window will receive `keyDownEvents`. `winEnterEvent` and `winExitEvent` are about which window receives the next pen or key event. They exist for backward compatibility and are only sent within the same thread for windows created using the UI Library.

2

Working with Forms and Dialogs

This chapter describes how to programmatically work with forms and dialogs. It covers:

A **form** is a container for your application's user interface (see Figure 2.1). An application must have at least one form, and most applications have more than one.

Figure 2.1 Form

A **dialog** or **modal dialog** is a special type of form (see Figure 2.2). Dialogs look different from forms in that they are often shorter and use a different title bar; however, the main difference between a dialog and a form is in the way it handles events.

Figure 2.2 Modal dialog

In either case, the main purposes of a form or dialog are to:

- Display other UI elements
- Handle the events that an application receives

Specifying Constraints

As mentioned in "Display Layout" on page 6, when you create a form resource, you should also specify a its **constraints**. Palm OS® uses the constraints to determine how to create and size windows for your form. Because the way you write code for a form depends significantly on how you set its constraints, this section provides guidelines for setting the constraints resource. The rest of this chapter focuses on how to write code to work with forms.

See your resource editor's documentation for specific information on how to set the constraints resource and all other resources.

The resource file that defines your form should contain a `WINDOW_CONSTRAINTS_RESOURCE` with the same ID as the form or dialog. See Listing 2.1.

Listing 2.1 Sample constraints resource

```
<WINDOW_CONSTRAINTS_RESOURCE RESOURCE_ID="1800">
   <VERSION> 1 </VERSION>
   <WINDOW_CREATE_FLAGS> 0x00000004 </WINDOW_CREATE_FLAGS>
   <WINDOW_X_POS> 0 </WINDOW_X_POS>
   <WINDOW_Y_POS> 0 </WINDOW_Y_POS>
   <WINDOW_WIDTH_MIN> 160 </WINDOW_WIDTH_MIN>
   <WINDOW_WIDTH_MAX> 0x7fff </WINDOW_WIDTH_MAX>
   <WINDOW_WIDTH_PREF> 160 </WINDOW_WIDTH_PREF>
   <WINDOW_HEIGHT_MIN> 160 </WINDOW_HEIGHT_MIN>
   <WINDOW_HEIGHT_MAX> 0x7fff </WINDOW_HEIGHT_MAX>
   <WINDOW_HEIGHT_PREF> 240 </WINDOW_HEIGHT_PREF>
</WINDOW_CONSTRAINTS_RESOURCE>
```

The constraints resource contains two types of information: the window type (which includes the layer) and the size constraints.

Window Type

The `WINDOW_CREATE_FLAGS` attribute is a bit mask that contains the window type and the window layer. The `WinFlagsType` enum specifies the values that this attribute can have. In most cases, you'll only need to set one flag:

- The back-buffer flag `0x00000004`

You don't need to set the modal flag in a constraints resource. The window's modal flag is set according to the form resource's modal flag. You set the window layer only very rarely. All application windows appear in the normal layer, which is the default layer. If you're creating a resource for a pinlet, the window layer in the `WINDOW_CREATE_FLAGS` attribute is ignored. The system automatically displays the pinlet's window in the system layer.

The back-buffer flag controls which type of window is created for the form. The window type is important because it controls whether the form follows the Palm OS Cobalt rules for drawing or the legacy rules. In Palm OS Cobalt, drawing is not done directly to the screen as it was in previous releases. Instead, you draw to a **graphics context**, which is described in more detail in Chapter 8, "Drawing," on page 123. The window type determines when this graphics context is available.

Palm OS Cobalt supports three window types:

- **Legacy windows**. Legacy windows are used for forms that do not have a constraints resource. They work just like windows on previous Palm OS releases. Because they do not have constraints, they do not resize or move, and users cannot open or close the dynamic input area when a legacy window is displayed.

  Legacy windows only exist if an application's resources aren't changed when it is ported to Palm OS Cobalt. Because the dynamic input area is disabled when legacy windows are displayed, PalmSource does not recommend that you leave your application in this state.

- **Update-based windows**. Update-based windows are created for forms that use the default window creation flags (specifically, that don't set the back-buffer bit). Update-based windows follow the Palm OS Cobalt drawing rules.

  For update-based windows, the graphics context is only available at certain times. The system notifies you that the graphics context is available by sending the `frmUpdateEvent`. Therefore, if you use update-based windows, you must only draw in response to this event.

IMPORTANT: A pinlet or any form displayed outside of the application process must use an update-based window.

- **Transitional windows.** Transitional windows are created if you set the back-buffer bit in `WINDOW_CREATE_FLAGS`. The graphics context is created once and is then stored in the back buffer shared between the application process and the system process. (The system process is where rendering is performed). For a transitional window, you must wait to draw until you receive the first `winResizedEvent`, which specifies the size of the form. After that, you may draw at any time.

You'll probably want to use transitional windows for ported applications, at least until you've discovered all of the places where your application draws to the screen.

For new applications, you may choose to use either transitional windows or update-based windows. Transitional windows are more efficient when rendering is done entirely in software, but update-based windows are more efficient on devices that use graphics accelerators. Note that the back buffer is allocated once and shared by all applications, so there is no size penalty for using transitional windows.

Regardless of which style of window you use, it's a good idea to try to follow the drawing rules used for update-based windows.

Size Constraints

Follow these guidelines when setting the size constraints:

- Set `WINDOW_X_POS` and `WINDOW_Y_POS` to the location of the top-left corner of the window. In most cases, you can use `winUndefConstraint` (`-32768`), which has the system place the window for you. This is particularly important for modal dialogs that are shorter than the available space.
- The constraints you specify should be the ideal size without taking into consideration the device size.
  - For the minimum, specify the minimum space that is realistically necessary to use the form and read its contents. Keep in mind that some pinlets, such as Asian

handwriting recognition pinlets, may need more space than the Latin-based ones do. It is strongly suggested that you set the minimum height to between 100 and 160 coordinates.

– For the preferred size, specify the amount of space needed to read the entire contents of the form. For example, if the main element on the form is a list that always contains ten items, the preferred size would be to show all ten items. For a form whose main element is a large text field, the preferred size would be large enough for the text field to show its maximum number of characters.

  Of course, the amount of space needed to display the entire contents of the form may not be known until runtime. If so, you can either set the preferred size to the constant `winMaxConstraint` (`32767`), which means make the window as large as possible, or set new size constraints at runtime using `WinSetConstraints()`.

– The maximum size is often the same as the preferred size. If the form does not contain any variable length items such as tables or editable text fields, the maximum size should be the size required to read the entire contents of the form. If the form contains variable length items, set the maximum to the constant `winMaxConstraint` (`32767`) because the user can use that extra space for editing.

– A dialog is often already shorter than the available space, so the minimum, maximum, and preferred sizes are often the same.

Resizing only happens on devices that use a dynamic input area. Devices that use a static input area do not resize application windows.

Displaying a Form

Displaying a form involves several steps:

1. Call `FrmGotoForm()`. Pass it the resource ID of the form to be loaded and a pointer to the database that contains the resource. For example:

```
if ((error = SysGetModuleDatabase(SysGetRefNum(), &gDbID,
   &gDbP)) < errNone)
   return error;
FrmGotoForm(gDbP, MainForm);
```

This function adds a `frmLoadEvent` to the event queue followed by a `frmOpenEvent`.

2. Respond to the `frmLoadEvent` by initializing the form and settings its event handler. See "Initializing a Form" on page 21 and "Setting a Form's Event Handler" on page 22.
3. Respond to the `winResizedEvent` by rearranging the elements on the form. See "Laying Out a Form or Dialog" on page 23.
4. Respond to the `frmOpenEvent` by performing any initialization that must occur before the form is displayed. See "Opening a Form" on page 27.
5. Respond to the `frmUpdateEvent` by drawing the form. See "Drawing or Updating a Form" on page 27.

Steps 3 through 5 may not require you to write any code. The default behavior is sufficient for many forms.

Displaying a dialog is very different from displaying a form. See "Displaying Dialogs" on page 29.

Initializing a Form

Initialize the form in response to a `frmLoadEvent` by doing the following:

- Call `FrmInitForm()` to initialize the form's internal data structure and obtain a pointer to it.
- Call `FrmSetActiveForm()` make the form become the active form.

These steps are typically performed in the `AppHandleEvent()` function as shown in Listing 2.2.

Listing 2.2 Initializing a form

```
static Boolean AppHandleEvent(EventType *eventP)
{
   uint16_t formId;
   FormType *frmP;

   if (eventP->eType == frmLoadEvent) {
      formId = eventP->data.frmLoad.formID;
      frmP = FrmInitForm(gDbID, formId);
      FrmSetActiveForm(frmP);
      ...
      return true;
   }
   return false;
}
```

The `FrmInitForm()` function initializes the internal `FormType` data structure for the form. This data structure is passed to all form functions.

The `FrmSetActiveForm()` function requests that the form be granted input focus. The request is granted asynchronously and can fail if a process with a window in a higher layer requests the input focus at the same time. If the form receives the input focus as a result of this request when initializing a form, it receives these events:

- A `winEnterEvent`
- A `winFocusGainedEvent`

See "Input Focus" on page 12 for more information about these events.

Setting a Form's Event Handler

You typically assign a custom event handler to a form in response to the `frmLoadEvent`. As you learned in *Exploring Palm OS: Programming Basics*, the standard application event loop calls the system function `FrmDispatchEvent()` to send events to a form. `FrmDispatchEvent()` does the following:

- If your form has an event handler set, it calls that event handler.

- If your form's event handler returns `false`, it then calls `FrmHandleEvent()` to let the built-in code provide default behavior.

To set the form's event handler, call `FrmSetEventHandler()` in the `AppHandleEvent()` function as shown in Listing 2.3.

Listing 2.3 Setting the event handler

```
static Boolean AppHandleEvent(EventPtr eventP)
{
   uint16_t formId;
   FormType *frmP;

   if (eventP->eType == frmLoadEvent) {
      ...
      case MainForm:
         FrmSetEventHandler(frmP, MainFormHandleEvent);
         break;
      case EditForm:
         FrmSetEventHandler(frmP, EditFormHandleEvent);
         break;
      // one case statement per form typically.
      return true;
   }
   return false;
}
```

Laying Out a Form or Dialog

Before your form is first displayed or when the user performs an action that causes the form to be resized, the application receives a `winResizedEvent`. The event structure contains the bounds into which the form should be drawn.

If the form is a dialog that remains a constant size, the system takes care of the `winResizedEvent` for you as long as you have specified `winUndefConstraint` for the position of the dialog. The system moves the dialog so that it is always at the bottom of the area allocated to the application.

For forms and full-screen dialogs, the system can take care of much of the resizing for you if you set up the form properly, but you may need to adjust it afterwards. To set up the form for automatic

resizing, create an array of `FormLayoutType` structures that defines a layout rule for each element within the form. Pass the array to `FrmInitLayout()` in response to the `frmLoadEvent`. See Listing 2.4.

IMPORTANT: The `FormLayoutType` structure must remain in memory for the life of the form.

Listing 2.4 Automatic form layout

```
struct FormLayoutType editFormLayout[] = {
   { sizeof(FormLayoutType), EditFormSelectorTrigger, 0,
      frmFollowTop },
   { sizeof(FormLayoutType), EditFormTextField, 0,
      frmFollowTopBottom },
   { sizeof(FormLayoutType), EditFormScrollBar, 0,
      frmFollowTopBottom },
   { sizeof(FormLayoutType), EditFormDoneButton, 0,
      frmFollowBottom },
   { sizeof(FormLayoutType), EditFormDetailsButton, 0,
      frmFollowBottom },

   // NOTE: Don't forget the shift indicator! Use the special
   // value frmLayoutGraffitiShiftID as its ID.
   { sizeof(FormLayoutType), frmLayoutGraffitiShiftID, 0,
      frmFollowBottom },
   { 0, 0, 0, 0 } // always end the array with a 0 element.
};

if (eventP->eType == frmLoadEvent) {
   formId = eventP->data.frmLoad.formID;
   frmP = FrmInitForm(gDbID, formId);
   FrmSetActiveForm(frmP);
   switch (formId) {
      case EditForm:
         FrmSetEventHandler(frmP, EditFormHandleEvent);
         FrmInitLayout(frmP, editFormLayout);
         break;
   return true;
}
```

The layout rule specifies which sides of the form the element is anchored to. If the element is anchored to one side, it moves. If the element is anchored to opposite sides, it resizes. For example,

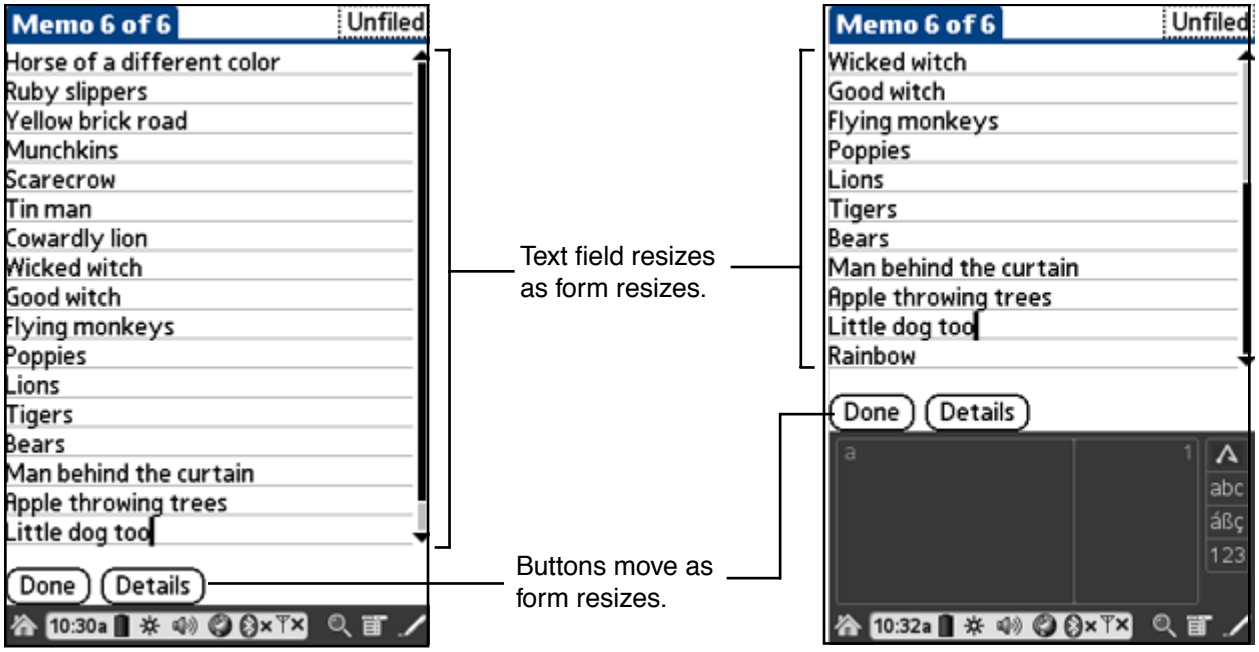
`frmFollowBottom` specifies that the element is always at a fixed location relative to the bottom of the form. If the form's bottom changes (for example, if the form resizes because the user closes the dynamic input area), the element moves. If the element is anchored to both the top and bottom with `frmFollowTopBottom`, the element resizes. Similarly, if you anchor an element to the left or right side of the form with `frmFollowLeft` or `frmFollowRight`, the element moves if the form is resized horizontally. If you use `frmFollowLeftRight`, the element resizes with the form. Note that the horizontal resizing rule is independent of the vertical resizing rule. For example, you can have a command button use the following rule:

```
frmFollowBottom | frmFollowLeftRight
```

Such a button resizes horizontally but moves if the form is resized vertically.

Figure 2.3 shows how Listing 2.4 resizes its form.

Figure 2.3 Resizing a form



For some forms, you do not need to respond to `winResizedEvent` at all if you call `FrmInitLayout()`. `FrmHandleEvent()` by default calls `FrmPerformLayout()`, which applies the layout rules to all of the elements in the form. However, `FrmPerformLayout()` only changes an element's bounds. Specifically, you will need to handle the following:

- Scroll bars attached to tables or fields. You need to call `SclSetScrollBar()` with the new scroll values for the field or table so that the scroll bar automatically appears or disappears as necessary.
- Tables. You may want to increase the number of rows a table displays.
- In any element that scrolls (such as a field, a list, or a table), you may need to make sure that the current selection is still visible when the form shrinks.

In each of these cases, call `FrmPerformLayout()` first, then adjust the elements, and then return `true` to specify that you've handled the event. Listing 2.5 shows an example of adjusting the number of table rows in response to `winResizedEvent`.

Listing 2.5 Responding to winResizedEvent

```
FormType *frmP;
WinHandle winH;
TableType *tableP;
int16_t rowHeight;
int16_t numRows;
RectangleType *tableBounds;

case winResizedEvent:

   frmP = FrmGetActiveForm();
   winH = FrmGetWindowHandle( frmP );
   if( winH == eventP->data.winResized.window )
   {
      FrmPerformLayout(frmP,
      &(eventP->data.winResized.newBounds));

      // Fix up the resized table
      tableP = FrmGetObjectPtr(frmP,
         FrmGetObjectIndex(frmP, MyTable));
      TblGetBounds(tableP, &tableBounds);
```

```
        rowHeight = TblGetRowHeight(tableP, 0);
        numRows = bounds.extent.y / rowHeight;

        // keep track of visible rows in a global
        gNumRowsVisibleInTable = numRows;
        // Load records into the table
        LoadRecordsIntoTable();

        handled = true;
    }
    break;
```

If you ever need to reposition an element on the form using `FrmSetObjectBounds()`, you cannot use `FrmInitLayout()` or `FrmPerformLayout()` to resize or reposition that element. Instead, you must leave it out of the array that you pass to `FrmInitLayout()`, respond to `winResizedEvent` as shown above, and manually reposition and resize that element.

Opening a Form

When the system is ready to open a form, it sends the `frmOpenEvent`. This is a request for the application to perform any initialization that needs to happen before a form is displayed. You might, for example, retrieve a database record that the form should display or set the initial selection for a group of push buttons on the form.

IMPORTANT: Do not call `FrmDrawForm()` in response to `frmOpenEvent`.

Drawing or Updating a Form

Palm OS Cobalt sends a `frmUpdateEvent` when it is ready for the application to draw. This event contains a rectangle specifying the region that needs to be drawn. All drawing is clipped to this rectangle.

Most applications do not need to respond to `frmUpdateEvent`. The default event handler redraws all user interface elements that a

form contains. You only need to respond to this event if you are performing custom drawing. Respond by doing the following:

1. Call `FrmDrawForm()`.
2. Draw using the functions described in Chapter 8, "Drawing," on page 123.

Note that because the clipping rectangle is set to the invalidated region when `frmUpdateEvent` is sent, `FrmDrawForm()` only redraws what is necessary to redraw.

If you are performing custom drawing, always respond to `frmUpdateEvent` for any type of window because a window from another process can obscure and then expose your window at any time. The `frmUpdateEvent` is the signal that your window needs to be redrawn.

Invalidating a Form

If you want to refresh the contents of the form, do not directly redraw. Instead, invalidate the form by calling `WinInvalidateWindow()` or `WinInvalidateRect()`. These two functions mark all or a portion of the form as dirty. The next time drawing occurs, the form receives the `frmUpdateEvent` specifying the area that needs to be redrawn.

As an optimization, you can use `WinInvalidateRectFunc()` instead. This function takes a pointer to a callback function, which the system calls instead of sending a `frmUpdateEvent`.

It is not an error to directly redraw into a transitional or legacy window. It is an error to directly redraw into an update-based window. Because the update-based window does not have a graphics context until it receives a `frmUpdateEvent`, redrawing directly into an update-based windows causes an application crash.

Closing a Form

A form is closed by one of two functions:

- `FrmGotoForm()` closes the current form by adding a `frmCloseEvent` to the event queue before adding the `frmLoadEvent` and `frmOpenEvent` for the new form.

- `FrmCloseAllForms()` posts a `frmCloseEvent` for each open form. It should be called before an application shuts down.

Respond to `frmCloseEvent` by undoing anything you did in response to `frmOpenEvent`. Deallocate memory you allocated, write changes to a database record to the database, and so on.

TIP: Return `false` from your `frmCloseEvent` handler to make sure that the default event handler deletes the form.

Displaying Dialogs

Displaying a dialog is different from displaying a form. Depending on the dialog, it can be much simpler than displaying a form. Specifically, if you display an alert, a progress dialog, or a tips dialog, there are special functions that you may use for convenience.

This section first describes how to display modal dialogs in general and then describes how to display special types of dialogs.

Displaying Any Modal Dialog

A dialog displays on top of another form. You do not call `FrmGotoForm()` to display a dialog because `FrmGotoForm()` closes the current form before it opens the new one. Instead, you should call `FrmDoDialog()` to display a dialog.

The differences between displaying a form and displaying a dialog are:

- You do not receive `frmLoadEvent` or `frmOpenEvent` if you call `FrmDoDialog()`. Perform any necessary initialization, including calling `FrmInitForm()` and setting the form's event handler, before you call that function. `FrmDoDialog()` calls `FrmSetActiveForm()` for you.
- You also do not receive the `frmCloseEvent`. Instead, `FrmDoDialog()` returns when the dialog is closed. The return value is the value of the button that was tapped to dismiss the dialog. Therefore, you must call `FrmDeleteForm()` when you are done displaying the dialog.

- `FrmDoDialog()` runs its own event loop. You can still set an event handler for the dialog, and `FrmDoDialog()` will call it before calling `FrmHandleEvent()`.

Listing 2.6 Using FrmDoDialog

```
frmP = FrmInitForm(gDbP, MyDialogForm);
FrmSetEventHandler(frmP, MyDialogEventHandler);
/* initialize the dialog's controls here if necessary */

/* open the dialog, and wait until a button is pressed to
close it. */
whichButton = FrmDoDialog(frmP);

if (whichButton == DetailsOKButton) {
  /* get data from controls on the form to save/apply changes
  */
}
FrmDeleteForm(frmP);
```

If you need more control over the dialog and how it behaves, call `FrmPopupForm()` instead of `FrmDoDialog()`. `FrmPopupForm()` is like `FrmGotoForm()` except that it does not close the current form. Your dialog will receive the `frmLoadEvent` and `frmOpenEvent` and you should otherwise follow the instructions in "Displaying a Form" on page 20. When your dialog should be dismissed, perform any necessary cleanup and then call `FrmReturnToForm()` with a form ID of 0 to return to the previously displayed form.

Alert Dialogs

An alert dialog is a modal dialog that is primarily informational (see Figure 2.4). It might prompt the user to confirm an action, or it might display an error message.

Figure 2.4 Alert dialog

The following functions display alerts:

- `FrmAlert()` displays a simple alert that contains a message string and has several command buttons.
- `FrmCustomAlert()` displays an alert with a customized string. You create the string using the strings ^1, ^2, and ^3 as placeholders. When you call `FrmCustomAlert()`, you pass strings to substitute for those placeholders.
- `FrmCustomResponseAlert()` displays an alert with a text field as well as command buttons. You pass a pointer to a callback function that verifies the text the user enters in the text field. This is typically used for password dialogs.

All of these functions return the item number of the button that the user tapped to dismiss the alert. Buttons are numbered from left to right starting with button 0. The functions handle all steps for you, from calling `FrmInitForm()` to calling `FrmDeleteForm()`.

Help Dialogs

A help or tips dialog displays when the user taps the "i" button on a modal dialog. You typically only provide a string resource containing the help text and the system handles displaying the dialog for you.

If you want to display a help dialog through some other means, use the `FrmHelp()` function. As with the alert functions, `FrmHelp()` initializes the form for you before it opens and deletes it when the user closes it. You only need to make the one function call.

Progress Dialogs

If your application performs a lengthy process, such as data transfer during a communications session, consider displaying a progress

dialog to inform the user of the status of the process. The Progress Manager provides the mechanism to display progress dialogs.

You display a progress dialog by calling `PrgStartDialog()`. Then, as your process progresses, you call `PrgUpdateDialog()` to update the dialog with new information for the user. In your event loop you call `PrgHandleEvent()` to handle the progress dialog update events queued by `PrgUpdateDialog()`. The `PrgHandleEvent()` function makes a callback to a `PrgCallbackFunc()` function that you supply to get the latest progress information.

Note that whatever operation you are doing that is the lengthy process, you do the work either inside your normal event loop or in a separate thread, not in the callback function. That is, you call `EvtGetEvent()` and do work when you get a `nilEvent`. Each time you get a `nilEvent`, do a chunk of work, but be sure to continue to call `EvtGetEvent()` frequently (like every half second), so that pen taps and other events get noticed quickly enough.

The dialog can display a few lines of text that are automatically centered and formatted. You can also specify an icon that identifies the operation in progress. The dialog has two optional buttons:

- A Cancel or OK button. The label of this button is automatically controlled by the Progress Manager and depends on the current progress state (no error, error, or user canceled operation).
- An optional skip button. You can enable the use of a skip button by setting a flag in the structure returned by `PrgCallbackFunc()`. If the user taps the skip button, you should skip the current stage of the process and move on to the next stage. Of course, you only enable the button if stages in the process can be skipped.

Progress Callback Function

When you want to update the progress dialog with new information, you call the function `PrgUpdateDialog()`. To get the current progress information to display in the progress dialog, `PrgHandleEvent()` calls a function that you supplied in your call to `PrgStartDialog()`.

The system passes the callback function one parameter, a pointer to a `PrgCallbackData` structure. To learn what type of information is passed in this structure, see Chapter 26, "Progress Manager Reference," on page 559.

Your callback function should return a Boolean. Return `true` if the progress dialog should be updated using the values you specified in the `PrgCallbackData` structure. If you specify `false`, the dialog is still updated, but with default status messages. (Returning `false` is not recommended.)

In the callback function, you should set the value of the `PrgCallbackData`'s `textP` buffer to the string you want to display in the progress dialog when it is updated. You can use the value in the `stage` field to look up a message in a string resource. You also might want to append the text in the `message` field to your base string. Typically, the `message` field would contain more dynamic information that depends on a user selection, such as a phone number, device name, or network identifier, etc.

For example, the `PrgUpdateDialog()` function might have been called with a `stage` of 1 and a *`messageP`* parameter value of a phone number string, "555-1212". Based on the stage, you might find the string "Dialing" in a string resource, and append the phone number, to form the final text "Dialing 555-1212" that you place in the text buffer `textP`.

Keeping the static strings corresponding to various stages in a resource makes it easier to localize your application. More dynamic information can be passed in using the *`messageP`* parameter to `PrgUpdateDialog()`.

NOTE: The callback function is called only if the parameters passed to `PrgUpdateDialog()` have changed from the last time it was called. If `PrgUpdateDialog()` is called twice with exactly the same parameters, the callback function is called only once.

Working With UI Elements

One of the main purposes of a form is to display other user interface elements (sometimes called objects) such as command buttons, text

fields, and tables. Specifics on programmatically working with each type of element are given in most of the remaining chapters of this book.

You use the form to gain initial access to an element on that form. You also use form functions to control when the element is displayed. Both of these topics are described in this section.

Accessing an Element

The elements that a form contains are accessed through the form. Functions tend to reference a UI element in one of the following ways:

- By the resource ID that you specified when you created the element in a resource file.
- By the element's index, which is the index into the form's internal list of elements. If you have a resource ID, you can obtain the element's index using the function `FrmGetObjectIndex()`.
- By a pointer to the element's internal data structure. If you have an element's index, you can obtain its pointer using the function `FrmGetObjectPtr()`.

Most functions use the pointer reference to an element. For this reason, many applications define a function similar to the one shown in Listing 2.7.

Listing 2.7 Obtaining a pointer to a UI element

```
void *GetObjectPtr(uint16_t objectID)
{
  FormType *frmP;

  frmP = FrmGetActiveForm();
  return FrmGetObjectPtr(frmP,
      FrmGetObjectIndex(frmP, objectID));
}

void SomeOtherFunction(void)
{
  ControlType *controlP =
    (ControlType *)GetObjectPtr(MainPopupTrigger);
  ListType *listP = (ListType *)GetObjectPtr(MainPopupList);
```

```
   ...
}
```

Dynamically Displaying Elements

If you need to have a user interface element appear or disappear at runtime, do the following:

1. When creating your resource file, define all of the elements that may conceivably be displayed on the form at any time during the life of the application.
2. If an element should not be displayed when the form is first displayed, set its usable attribute to `false` in the resource file.
3. When an event occurs that should change the form's display, use `FrmShowObject()` to show any previously undisplayed element. Use `FrmHideObject()` to remove any element that should no longer be displayed.

For update-based windows, both `FrmShowObject()` and `FrmHideObject()` invalidate the area of the form where that element appears. That is, they asynchronously trigger a `frmUpdateEvent`. When your application receives the `frmUpdateEvent`, the new show and hide settings take effect. For all other windows, `FrmHideObject()` and `FrmShowObject()` are immediate.

TIP: In Palm OS Cobalt, hiding an element means filling its region with the color set for `UIFormFill`. This is true regardless of the type of window used for the form (legacy, transitional, or update-based).

Some applications must create their displays at runtime. These applications can use the dynamic UI APIs:

- `CtlNewControl()`
- `CtlValidatePointer()`
- `FldNewField()`
- `FrmNewBitmap()`
- `FrmNewForm()`

- FrmNewFormWithConstraints()
- FrmNewGadget()
- FrmNewGsi()
- FrmNewLabel()
- FrmRemoveObject()
- FrmValidatePtr()
- LstNewList()

You can use the FrmNewForm() function to create new forms dynamically. Palm's UI guidelines encourage you to keep modal dialogs at the bottom of the screen, using the entire screen width. This isn't enforced by the function, but is strongly encouraged in order to maintain a look and feel that is consistent with the built-in applications.

The FrmNewLabel(), FrmNewBitmap(), FrmNewGadget(), LstNewList(), FldNewField() and CtlNewControl() functions can be used to create new elements on forms.

It is fine to add new items to an active form, but doing so is very likely to move the form structure in memory; therefore, any pointers to the form or to controls on the form might change. Make sure to update any variables or pointers that you are using so that they refer to the form's new memory location, which is returned when you create the element.

If you're not absolutely sure that you need to change your UI dynamically, don't do it—unexpected changes to an application's interface are likely to confuse or frustrate the end user.

The FrmRemoveObject() function removes a dynamically created element from a form. This function doesn't free memory referenced by the element (if any) but it does shrink the form chunk. For best efficiency when removing items from forms, remove items in order of decreasing index values, beginning with the item having the highest index value. When removing items from a form, you need to be mindful of the same concerns as when adding items: the form pointer and pointers to controls on the form may change as a result of any call that moves the form structure in memory.

When creating forms dynamically, or just to make your application more robust, use the FrmValidatePtr() function to ensure that

your form pointer is valid and the form it points to is valid. This routine can catch lots of bugs for you—use it!

Summary of Form and Dialog Functions

Form Functions

Loading a Form

FrmGotoForm()
FrmPopupForm()

Initialization

FrmInitForm()
FrmInitFormWithFlags()
FrmInitLayout()
FrmSetActiveForm()

Event Handling

FrmHandleEvent()
FrmSetEventHandler()
FrmDispatchEvent()

Displaying a Modal Dialog

FrmDoDialog()
FrmCustomAlertWithFlags()
FrmAlert()
FrmCustomResponseAlert()
FrmHelp()
FrmCustomResponseAlertWithFlags()
FrmAlertWithFlags()
FrmHelpWithFlags()
FrmNewGsi()
FrmUIAlert()
FrmCustomAlert()

Updating the Display

FrmDrawForm()
FrmHideObject()
FrmShowObject()
FrmUpdateScrollers()
FrmRemoveObject()
FrmUpdateForm()
WinInvalidateRect()
WinInvalidateRectFunc()
WinInvalidateWindow()

Form Attributes

FrmVisible()
FrmGetHelpID()
FrmGetDefaultButtonID()
FrmSetHelpID()
FrmSetDefaultButtonID()

Form Functions

Accessing a Form Programmatically

FrmGetActiveForm()
FrmGetActiveFormID()
FrmGetFirstForm()
FrmGetFormId()
FrmGetFormPtr()
FrmGetWindowHandle()
FrmValidatePtr()
FrmGetFormWithWindow()
ECFrmValidatePtr()

Obtaining Focus

FrmGetFocus()
FrmSetFocus()
FrmGetActiveField()

Accessing Objects Within a Form

FrmGetObjectId()
FrmGetObjectIndex()
FrmGetObjectType()
FrmGetObjectPosition()
FrmGetObjectPtr()
FrmGetNumberOfObjects()
FrmGetObjectIndexFromPtr()
FrmGetObjectIdFromObjectPtr()
FrmGetObjectUsable()

Title and Menu

FrmCopyTitle()
FrmGetTitle()
FrmPointInTitle()
FrmSetTitle()
FrmSetMenu()
FrmGetMenuBarID()

Labels

FrmCopyLabel()
FrmSetCategoryLabel()
FrmGetLabel()
FrmNewLabel()
FrmGetLabelFont()
FrmSetLabelFont()

Controls

FrmSetControlValue()
FrmGetControlGroupSelection()
FrmGetControlValue()
FrmSetControlGroupSelection()

Gadgets

FrmGetGadgetData()
FrmSetGadgetData()
FrmNewGadget()
FrmSetGadgetHandler()

Form Functions

Bitmaps

FrmNewBitmap()

Coordinates and Boundaries

FrmGetObjectBounds()
FrmSetObjectBounds()
FrmSetObjectPosition()
FrmGetFormBounds()
FrmGetObjectInitialBounds()
FrmGetFormInitialBounds()

Resizing a Form

FrmPerformLayout()
frmLayoutRule()

Removing a Form From the Display

FrmCloseAllForms()
FrmEraseForm()
FrmSaveAllForms()
FrmReturnToForm()

Releasing a Form's Memory

FrmDeleteForm()

Pen Tracking

FrmAmIPenTracking()
FrmSetPenTracking()

Dynamically Creating a Form

FrmNewForm()
FrmNewFormWithConstraints()
FrmRestoreActiveState()
FrmSaveActiveState()

Progress Manager Functions

PrgHandleEvent()
PrgStartDialog()
PrgStopDialog()
PrgUpdateDialog()
PrgUserCancel()

3

Working with Controls

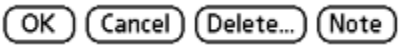
Controls allow for user interaction when you add them to the forms in your application. Events in controls are handled by `CtlHandleEvent()`. This chapter describes the following types of controls:

All of these except the category controls and the scroll bars are managed by the functions described in Chapter 16, "Control Reference," on page 277. The category controls are a special case of the selector trigger and pop-up trigger controls that are only used to display database categories. Scroll bars are managed using a separate scroll bar API that is described in Chapter 31, "Scroll Bar Reference," on page 615.

Command Buttons

Command buttons (see Figure 3.1) display a text or graphic label in a rounded rectangle. The button frame can be changed although doing so is discouraged.

Figure 3.1 Buttons



When the user taps a command button with the pen, the command button highlights until the user releases the pen or drags it outside the bounds of the command button.

Applications use command buttons to allow the user to perform an action. Applications respond to the `ctlSelectEvent` for a command button to perform its action. You may return either `true` or `false` after the `ctlSelectEvent` because Palm OS® does not handle this event.

Table 3.1 shows the system events generated when the user interacts with the button and `CtlHandleEvent()`'s response to the events.

Table 3.1 Event flow for command buttons

User Action	System Response	CtlHandleEvent() Response
Pen goes down on a button.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with button's ID number.	Inverts the button's display.
Pen is lifted from button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlSelectEvent` to the event queue.
Pen is lifted outside button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlExitEvent` to the event queue.

Repeating Buttons

A **repeating button** can look like a command button. In contrast to a command button, however, the repeating button is selected repeatedly until the pen is lifted. The most common use of repeating buttons are for the scroll buttons displayed in the bottom right corner of many forms.

Applications respond to the `ctlRepeatEvent` for repeating buttons to perform the action of the button. Return `false` after handling the event to ensure that more repeat events are sent.

Table 3.2 shows the system events generated when the user interacts with the repeating button and `CtlHandleEvent()`'s response to the events.

Table 3.2 Event flow for repeating buttons

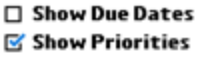
User Action	System Response	CtlHandleEvent() Response
Pen goes down on a repeating button.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with button's ID number.	Adds the `ctlRepeatEvent` to the event queue.
Pen remains on repeating button.	`ctlRepeatEvent`	Tracks the pen for a period of time, then sends another `ctlRepeatEvent` if the pen is still within the bounds of the control.
Pen is dragged off the repeating button.		No `ctlRepeatEvent` occurs.
Pen is dragged back onto the button.	`ctlRepeatEvent`	See above.
Pen is lifted from button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlExitEvent` to the event queue.

Check Boxes

Check boxes (see Figure 3.2) display a setting, either on (checked) or off (unchecked). Touching a check box with the pen toggles the setting. The check box appears as a square, which contains a check mark if the check box's setting is on. A check box can and should have a text label attached to it; selecting the label also toggles the check box. A check box may also be assigned a group ID to make the

group of check boxes mutually exclusive; however, doing so is strongly discouraged. Use push buttons for a mutually exclusive set of options.

Figure 3.2 Check boxes



Applications generally manage check boxes in the following ways:

- Respond to the `ctlSelectEvent` for the check box. Upon receiving this event, retrieve the value of the check box using `CtlGetValue()` and perform an action based on that value. You may return either `true` or `false` after the `ctlSelectEvent` because Palm OS does not handle this event.
- If you want to set the initial value of a check box when the form is first displayed, use `CtlSetValue()` or `FrmSetControlValue()` in response to the `frmOpenEvent`.

Table 3.3 shows the system events generated when the user interacts with the check box and `CtlHandleEvent()`'s response to the events.

Table 3.3 Event flow for check boxes

User Action	Event Generated	CtlHandleEvent() Response
Pen goes down on check box.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with check box's ID number.	Tracks the pen until the user lifts it.

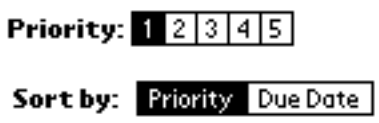
Table 3.3 Event flow for check boxes *(continued)*

User Action	Event Generated	CtlHandleEvent() Response
Pen is lifted from check box.	`penUpEvent` with the x and y coordinates stored in `EventType`.	• If the check box is unchecked, a check appears. • If the check box is already checked and is grouped, there is no change in appearance. • If the check box is already checked and is ungrouped, the check disappears. Adds the `ctlSelectEvent` to the event queue.
Pen is lifted outside box.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlExitEvent` to the event queue.

Push Buttons

Push buttons (see Figure 3.3) have square corners. Touching a push button with the pen inverts the bounds. If the pen is released within the bounds, the button remains inverted. Push buttons are intended to be used to present several mutually exclusive options. When you create the push button resources, assign a nonzero group ID to all push buttons within a group. Doing so ensures that only one push button can be selected at a time.

Figure 3.3 Push buttons



Applications manage push buttons using the functions `FrmGetControlGroupSelection()` and `FrmSetControlGroupSelection()` to retrieve and set which push button in the sequence is selected.

Table 3.4 shows the system events generated when the user interacts with the push button and `CtlHandleEvent()`'s response to the events.

Table 3.4 Event flow for push buttons

User Action	System Response	CtlHandleEvent() Response
Pen goes down on a push button.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with push button's ID number.	If push button is grouped and highlighted, no change. If push button is ungrouped and highlighted, it becomes unhighlighted.
Pen is lifted from push button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlSelectEvent` to the event queue.

Sliders

Sliders (see Figure 3.4) represent a value that falls within a particular range. For example, a slider might represent a value that can be between 0 and 10.

Figure 3.4 Slider

There are four attributes that are unique to slider controls:

- The minimum value the slider can represent
- The maximum value the slider can represent
- The current value
- The page jump value, or the amount by which the value is increased or decreased when the user clicks to the left or right of the slider thumb

Palm OS supports two types of sliders: regular slider and feedback slider. Sliders and feedback sliders look alike but behave differently. Specifically, a regular slider control does not send events while the user is dragging its thumb. A feedback slider control sends an event each time the thumb moves one pixel, whether the pen has been lifted or not.

Applications generally respond to the `ctlSelectEvent` for sliders or `ctlRepeatEvent` for feedback sliders. Upon receiving either one of these events, check the value of the slider using `CtlGetValue()` and perform an action based on that value. You may return either `true` or `false` after the `ctlSelectEvent` because Palm OS does not handle this event. Return `false` for the `ctlRepeatEvent`.

Table 3.5 shows the system events generated when the user interfaces with a slider and how `CtlHandleEvent()` responds to the events.

Table 3.5 Event flow for sliders

User Action	System Response	CtlHandleEvent() Response
Pen tap on slider's background.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with slider's ID number.	Adds or subtracts the slider's page jump value from its current value, and adds a `ctlSelectEvent` with the new value to the event queue.
Pen goes down on the slider's thumb.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with slider's ID number.	Tracks the pen.

Table 3.5 Event flow for sliders *(continued)*

User Action	System Response	CtlHandleEvent() Response
Pen drags slider's thumb to the left or right.		Continues tracking the pen.
Pen is lifted from slider.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlSelectEvent` with the slider's ID number and new value if the coordinates are within the bounds of the slider. Adds the `ctlExitEvent` if the coordinates are outside of the slider's bounds.

Table 3.6 shows the system events generated when the user interacts with a feedback slider and `CtlHandleEvent()`'s response to the events.

Table 3.6 Event flow for feedback sliders

User Action	System Response	CtlHandleEvent() Response
Pen tap on slider's background.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with slider's ID number.	Adds or subtracts the slider's page jump value from its current value and then sends a `ctlRepeatEvent` with the slider's new value.

Table 3.6 Event flow for feedback sliders *(continued)*

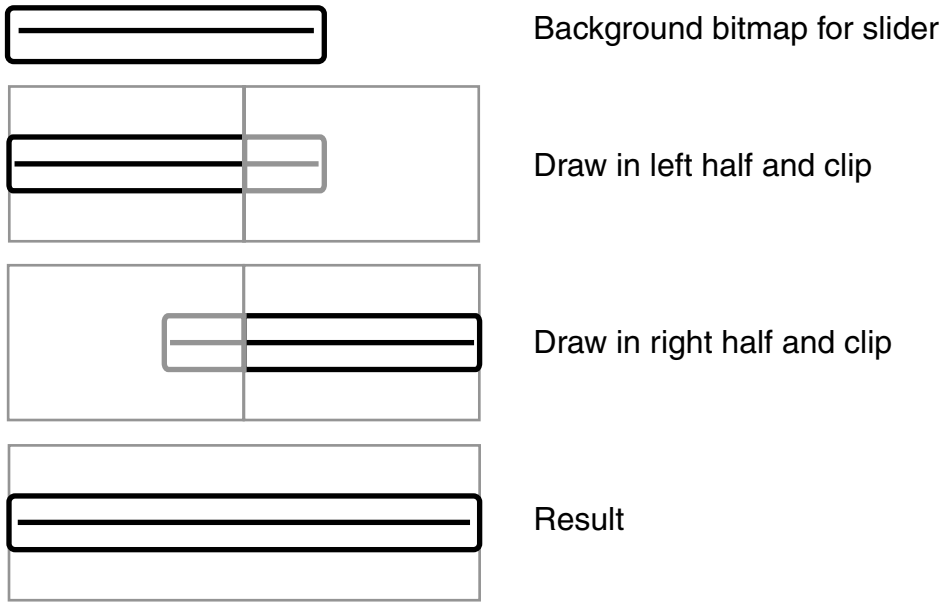
User Action	System Response	CtlHandleEvent() Response
	`ctlRepeatEvent`	Adds or subtracts the slider's page jump value from its current value repeatedly until the thumb reaches the pen position or the slider's minimum or maximum. Then sends a `ctlSelectEvent` with slider's ID number and new value.
Pen goes down on the slider's thumb.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with slider's ID number.	Tracks the pen and updates the display.
Pen drags slider's thumb to the left or right.	`ctlRepeatEvent` with slider's ID number and new value.	Tracks the pen. Each time pen moves to the left or right, sends another `ctlRepeatEvent` if the pen is still within the bounds of the control.
Pen is dragged off the slider vertically.		`ctlRepeatEvent` with the slider's ID number and old value.
Pen is dragged back onto the slider.		`ctlRepeatEvent` with the slider's ID number and new value.
Pen is lifted from slider.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlExitEvent` to the event queue.

Sliders are drawn using two bitmaps: one for the slider background, and the other for the thumb. You may use the default bitmaps to

draw sliders, or you may specify your own bitmaps when you create the slider.

The background bitmap you provide can be smaller than the slider's bounding rectangle. This allows you to provide one bitmap for sliders of several different sizes. If the background bitmap isn't as tall as the slider's bounding rectangle, it's vertically centered in the rectangle. If the bitmap isn't as wide as the slider's bounding rectangle, the bitmap is drawn twice. First, it's drawn left-justified in the left half of the bounding rectangle and clipped to exactly half of the rectangle's width. Then, it's drawn right-justified in the right half of the bounding rectangle and clipped to exactly half of the rectangle's width. (See Figure 3.5.) Note that this means that the bitmap you provide must be at least half the width of the bounding rectangle.

Figure 3.5 Drawing a slider background



Selector Triggers

A **selector trigger** (see Figure 3.6) displays a text label surrounded by a gray rectangular frame. If the text label changes, the width of the control expands or contracts to the width of the new label.

Figure 3.6 Selector trigger

When the selector trigger is tapped, it should display a dialog from which the user selects a new value to display in the trigger. Applications are responsible for implementing most of this behavior:

- Upon receiving the `frmOpenEvent` for the form that contains the selector trigger, set the initial value of the trigger using `CtlSetLabel()`.
- Upon receiving a `ctlSelectEvent` for a selector trigger, call `FrmDoDialog()` as described in "Displaying Dialogs" on page 29. If the user then dismisses the dialog by tapping the OK button, use `CtlSetLabel()` to set the new value of the selector trigger.

Some forms that display category information for a single record use a selector trigger to do so. This selector trigger displays a pop-up list when tapped instead of a dialog. Such controls are considered exceptions to the rule of how selector triggers should behave and are not meant to be imitated. See "Category Controls" on page 54 for information on how to work with a category selector trigger.

Note that selector triggers, unlike the button controls and slider controls, do not have a numeric value associated with them, so you should not use `CtlSetValue()` or `CtlGetValue()` on them.

Table 3.7 shows the system events generated when the user interacts with the selector trigger and `CtlHandleEvent()`'s response to the events.

Table 3.7 Event flow for selector triggers

User Action	System Response	CtlHandleEvent() Response
Pen goes down on a selector trigger.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.

Table 3.7 Event flow for selector triggers *(continued)*

User Action	System Response	CtlHandleEvent() Response
	`ctlEnterEvent` with selector trigger's ID number.	Inverts the button's display.
Pen is lifted from the selector trigger.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlSelectEvent` to the event queue.

Pop-Up Triggers

A **pop-up trigger** (see Figure 3.7) displays a text label and a graphic element (always on the left) that signifies the control initiates a pop-up list. If the text label changes, the width of the control expands or contracts to the width of the new label plus the graphic element.

Figure 3.7 Pop-up trigger

To create a pop-up list in a resource editor, you must create both a pop-up trigger resource and a list resource. Do the following:

1. Add a pop-up trigger to your form. Use 0 as the width to have the pop-up trigger automatically resize based on the label.
2. Add a list at the same coordinates.
3. Specify that the list is unusable.
4. Set the List ID in the pop-up trigger to match the ID of the list resource.

Applications generally manage pop-up triggers in the following ways:

- Set the initial label of the pop-up trigger when the form is first opened. Respond to the `frmOpenEvent` by calling `CtlSetLabel()`.
- If the list items were statically created in the resource editor, respond to the `popSelectEvent` for a pop-up trigger and perform an action based on the selection passed in the event

structure. Return `false` to allow the system to set the pop-up trigger's label to the text of the selected list item.

- If you're dynamically creating the list elements at run-time, the system cannot set the trigger label for you. Respond to the `popSelectEvent` to both perform an action based on the selection and to set the trigger label using `CtlSetLabel()`. Return `true` to bypass the system default behavior. Otherwise, the application will crash.

IMPORTANT: Do *not* return `false` in response to the `ctlSelectEvent` for a pop-up trigger. If you do, the list won't pop up.

Some forms use a pop-up list to display database categories. See "Category Controls" on page 54 for information on how to work with such a list.

Table 3.8 shows the system events generated when the user interacts with the pop-up trigger and `CtlHandleEvent()`'s response to the events. Because pop-up triggers are used to display lists, also see "Lists" on page 112.

Table 3.8 Event flow for pop-up triggers

User Action	System Response	CtlHandleEvent() Response
Pen goes down on the pop-up trigger.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlEnterEvent` to the event queue.
	`ctlEnterEvent` with pop-up trigger's ID number.	Inverts the trigger's display.
Pen is lifted from button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlSelectEvent` to the event queue.

Table 3.8 Event flow for pop-up triggers *(continued)*

User Action	System Response	CtlHandleEvent() Response
	`ctlSelectEvent` with pop-up trigger's ID number.	Adds a `winEnterEvent` for the list's window to the event queue. Control passes to `FrmHandleEvent()`, which displays the list and adds a `popSelectEvent` to the event queue. Control then passes to `LstHandleEvent()`.
Pen is lifted outside button.	`penUpEvent` with the x and y coordinates stored in `EventType`.	Adds the `ctlExitEvent` to the event queue.

Category Controls

Palm OS databases use **categories** to allow the user to group records logically into manageable lists. In the user interface, categories typically appear:

- In a pop-up list in a form's title bar if the form displays multiple database records.
- In a pop-up list in dialogs (such as the Details dialog) that allow you to edit a single database record.
- In a selector trigger in a form's title bar if the form allows you to edit a single database record.

The Category Manager allows you to easily create and work with these standard controls used for categories. The Category Manager contains two separate sets of APIs: one for non-schema Palm OS databases, and one for schema databases. This section describes how to use the Category Manager functions to manage category controls. For information on manipulating categories in a database, see *Exploring Palm OS: Memory, Databases, and Files*.

Creating the Category Controls

You create a category pop-up list the same way you create any pop-up list. In your resource editor, do the following:

1. Create a pop-up trigger resource with a width of 0.
2. Add a list resource at the same coordinates as the pop-up trigger. Set the list to unusable.
3. Assign the list ID attribute of the pop-up trigger to be the resource ID of the list you created in the previous step.

To crate a category selector trigger, do the following:

1. Create a selector trigger with a width of 0.
2. Add a list at the same coordinates as the pop-up trigger. Set the list to unusable.

Because selector triggers are not normally associated with pop-up lists, the selector trigger does not have a list ID attribute for you to assign. The pop-up list for the selector trigger is managed through the Category Manager.

Category Controls for Non-schema Database Databases

For the most part, you can handle category controls for classic Palm OS databases using only these calls:

- Call `CategoryInitialize()` when you create a new database as described in "Initializing Categories in a Non-schema Database" below).
- Call `CategorySetTriggerLabel()` to set the category pop-up or selector trigger's label when the form is opened (as described in "Initializing the Category Trigger for Non-schema Databases").
- Call `CategorySelect()` when the user selects the category trigger (as described in "Managing a Category Pop-up List for a Non-schema Database").

You typically don't need to use the other Category Manager functions unless you want more control over what happens when the user selects the category trigger.

Initializing Categories in a Non-schema Database

Before you can use the category functions on a non-schema database, you must set up the database appropriately. The category functions expect to find information at a certain location. If the information is not there, the functions will fail.

Category information is stored in the `AppInfoType` structure within the non-schema database's application info block. The application info block may contain any information that your database needs. If you want to use the Category Manager, the first field in the application info block must be an `AppInfoType` structure.

The `AppInfoType` structure maps category names to category indexes and category unique IDs. Category names are displayed in the user interface. Category indexes are used to associate a database record with a category. That is, the database record's attributes contain the index of the category to which the record belongs. Category unique IDs are used when synchronizing the database with the desktop computer.

To initialize the `AppInfoType` structure, you call `CategoryInitialize()`, passing a string list resource containing category names. This function creates as many category indexes and unique IDs as are necessary. You only need to make this call when the database is first created or when you newly assign the application info block to the database.

The string list resource is an `appInfoStringsRsc('tAIS')` resource. It contains predefined categories that new users see when they start the application for the first time. Note that the call to `CategoryInitialize()` is the only place where you use an `appInfoStringsRsc`. Follow these guidelines when creating the resource:

- Place any categories that you don't want the user to be able to change at the beginning of the list. For example, it's common to have at least one uneditable category named Unfiled, so it should be the first item in the list.
- The string list must have 16 entries. Typically, you don't want to predefine 16 categories. You might define one or two and leave the remaining entries blank. The unused slots should have 0 length.

- Keep in mind that there is a limit of 16 categories. That includes both the predefined categories and the categories your users will create.
- Each category name has a maximum length defined by the `dmCategoryLength` constant (currently, 16 bytes).
- Don't include strings for "All" or "Edit Categories." While these two items often appear in category lists, they are not categories, and they are treated differently by the category functions.

Listing 3.1 shows an example function that creates and initializes a database with an application info block. Notice that because the application info block is stored with the database, you allocate memory for it using `DmNewHandle()`, not with `MemHandleNew()`.

Listing 3.1 Creating a database with an app info block

```
typedef struct {
   AppInfoType appInfo;
   uint16_t myCustomAppInfo;
} MyAppInfoType;

DatabaseID gDbID;
DmOpenRef gDbP;

Err CreateAndOpenDatabase(DmOpenRef *dbPP, uint16_t mode)
{
   status_t error = errNone;
   DmOpenRef dbP;
   MemHandle appInfoHandle;
   DatabaseID dbID;
   MyAppInfoType *appInfoP;
   DmDatabaseInfoType myDbInfo;

   // Create the database.
   error = DmCreateDatabase (MyDBName, MyDBCreator, MyDBType, false);
   if (error < 0) return error;

   // Open the database.
   dbP = DmOpenDatabaseByTypeCreator(MyDBType, MyDBCreator, mode);
   if (!dbP) return (dmErrCantOpen);

   // Get database ID so we can initialize app info block.
   if (DmGetOpenInfo(dbP, &dbID, NULL, NULL, NULL))
      return dmErrInvalidParam;
```

```
   // Allocate app info in storage heap.
   appInfoHandle = DmNewHandle(dbP, sizeof(MyAppInfoType));
   if (!appInfoHandle) return dmErrMemError;

   // Associate app info with database.
   error = DmDatabaseInfo(dbID, &myDbInfo);
   if (error < 0) return error;
   myDbInfo->pAppInfoHandle = appInfoHandle;
   DmSetDatabaseInfo(dbID, &myDbInfo);

   // Initialize app info block to 0.
   appInfoP = MemHandleLock(h);
   DmSet(appInfoP, 0, sizeof(MyAppInfoType), 0);

   // Get a DmOpenRef to the resource database that contains the app info
   // strings resource.
   if (!gDbP) {
      if ((error = SysGetModuleDatabase(SysGetRefNum(), &gDbID, &gDbP)) <
         errNone)
         return error;
   }
   // Initialize the categories.
   CategoryInitialize ((AppInfoPtr)appInfoP, gDbP, MyLocalizedAppInfoStr);

   // Unlock the app info block.
   MemPtrUnlock(appInfoP);

   // Set the output parameter and return.
   *dbPP = dbP;
   return error;
}
```

Initializing the Category Trigger for Non-schema Databases

When a form is opened, you need to set the text that the category trigger (whether pop-up or selector trigger) should display. To do this, use `CategoryGetName()` to look up the name in the `AppInfoType` structure and then use `CategorySetTriggerLabel()` to set the label.

For the main form of the application, it's common to store the index of the previously selected category in a preference and restore it when the application starts up again.

Forms that display information from a single record may show that record's category in a selector trigger or pop-up list. You can retrieve the record's category using `DmGetRecordCategory()`.

Listing 3.2 shows how to set the trigger label to match the category for a particular database record.

Listing 3.2 Setting the category trigger label (non-schema database)

```
uint8_t category;
char categoryName [dmCategoryLength];
ControlType *ctl;

// If current category is All, we need to look
// up category.
if (CurrentCategory == dmAllCategories)
   DmGetRecordCategory (AddrDB, CurrentRecord, &category);
else
   category = CurrentCategory;
CategoryGetName (AddrDB, category, categoryName);
ctl = FrmGetObjectPtr(frm,
   FrmGetObjectIndex(frm, objectID));
CategorySetTriggerLabel (ctl, categoryName);
```

Managing a Category Pop-up List for a Non-schema Database

When the user taps the category pop-up trigger, call `CategorySelect()`. That is, call `CategorySelect()` in response to a `ctlSelectEvent` when the ID stored in the event matches the ID of the category's trigger. The `CategorySelect()` function displays the pop-up list, manages the user selection, displays the Edit Categories modal dialog as necessary, and sets the trigger label to the item the user selected.

Listing 3.3 shows a typical call to `CategorySelect()`:

Listing 3.3 Calling CategorySelect()

```
categoryEdited = CategorySelect (AddrDB, frm,
  ListCategoryTrigger, ListCategoryList, true, &category,
  CategoryName, 1, NULL, categoryDefaultEditCategoryString);
```

This example uses the following as parameters:

- `AddrDB` is the database with the categories to be displayed.
- `frm`, `ListCategoryTrigger`, and `ListCategoryList` identify the form, pop-up or selector trigger resource, and list resource.
- `true` indicates that the list should contain an "All" item. The "All" item should appear only in forms that display multiple records. It should not appear in forms that display a single record because selecting it would have no meaning.
- `category` and `CategoryName` are pointers to the index and name of the currently selected category. When you call this function, these two parameters should specify the category currently displayed in the pop-up trigger. Unfiled is the default.
- The number 1 is the number of uneditable categories. `CategorySelect()` needs this information when the user chooses the Edit Categories list item. Categories that the user cannot edit should not appear in the Edit Categories dialog.

  Because uneditable categories are assumed to be at the beginning of the category list, passing 1 for this parameter means that `CategorySelect()` does not allow the user to edit the category at index 0.
- `NULL` is passed because this call uses the default title for the Edit Categories list item. If you want to change the name of this list item to something else, you must pass the resource database that contains the string resource you want to use for this list item.
- `categoryDefaultEditCategoryString` is a constant that means include an Edit Categories item in the list and use the default string for its name ("Edit Categories" on US English ROMs).

  To use a different name (for example, if you don't have enough room for the default name), pass the ID of a string resource containing the desired name and pass a reference to its resource database in the parameter before it.

  In some cases, you might not want to include the Edit Categories item. If so, pass the constant `categoryHideEditCategory`.

The `CategorySelect()` return value is somewhat tricky: `CategorySelect()` returns `true` if the user edited the category list, `false` otherwise. That is, if the user chose the Edit Categories item and added, deleted, or changed category names, the function returns `true`. If the user never selects Edit Categories, the function returns `false`. In most cases, a user simply selects a different category from the existing list without editing categories. In such cases, `CategorySelect()` returns `false`.

This means you should not rely solely on the return value to see if you need to take action. Instead, you should store the value that you passed for the category index and compare it to the index that `CategorySelect()` passes back. For example:

Listing 3.4 CategorySelect() return value

```
int16_t category;
Boolean categoryEdited;

category = CurrentCategory;

categoryEdited = CategorySelect (AddrDB, frm,
  ListCategoryTrigger, ListCategoryList, true, &category,
  CategoryName, 1, NULL, categoryDefaultEditCategoryString);

if ( categoryEdited || (category != CurrentCategory)) {
  /* user changed category selection or edited category list.
     Do something. */
}
```

If the user has selected a different category, you probably want to do one of two things:

- Update the display so that only records in that category are displayed. See the function `ListViewUpdateRecords()` in the Address Book example application for sample code.
- Change the current record's category from the previous category to the newly selected category. See the function `EditViewSelectCategory()` in the Address Book example application for sample code.

Note that the `CategorySelect()` function handles the results of the Edit Categories dialog for you. It adds, deletes, and renames items in the database's `AppInfoType` structure. If the user deletes a

category that contains records, it moves those records to the Unfiled category. If the user changes the name of an existing category to the name of another existing category, it prompts the user and, if confirmed, moves the records from the old category to the new category. Therefore, you never have to worry about managing the category list after a call to `CategorySelect()`.

Category Controls for Schema Databases

The category rules differ between non-schema databases and schema databases in the following ways:

- In schema databases, you can have up to 255 categories. In non-schema databases, the limit is 16.
- In schema databases, a database record can be assigned to multiple categories. In non-schema databases, a record can only belong to one category.
- In schema databases, the category information is stored in a private structure within the database. In non-schema databases, you must define an `AppInfoType` structure and store it in the database.
- In schema databases, the Database Manager keeps track of whether a category can be modified or not. In non-schema databases, the Database Manager has no notion of whether the category can be edited.

For the most part, you can handle category controls for schema Palm OS databases using only these calls:

- Call `CatMgrInitialize()` when you create a new database as described in "Initializing Categories in a Schema Database" below).
- Call `CatMgrSetTriggerLabel()` to set the category pop-up or selector trigger's label when the form is opened (as described in "Initializing the Category Trigger for Non-schema Databases").
- Call `CatMgrSelectFilter()` when the user selects the pop-up list on a form that uses categories to filter the display (as described in "Managing a Category Pop-up List for a Non-schema Database").

- Call CatMgrSelectEdit() when the user selects a selector trigger or pop-up list on a form that changes a database record's category.

You typically don't need to use the other Category Manager functions unless you want more control over what happens when the user selects the category trigger.

Initializing Categories in a Schema Database

Before you can use the category functions, you must set up the schema database appropriately. The category functions expect to find information at a certain location. If the information is not there, the functions will fail.

Category information is stored in a private structure within the schema database. This structure maps category names to category unique IDs. Category names are displayed in the user interface. Category IDs are used when storing category information in each database record and when synchronizing the schema database with the desktop computer.

To initialize this private structure, you call CatMgrInitialize(), passing a string list resource containing category names. This function creates as many category IDs as are necessary. You only need to make this call when the database is first created.

The string list resource is an `appInfoStringsRsc ('tAIS')` resource. It contains predefined categories that new users see when they start the application for the first time. Note that the call to `CatMgrInitialize()` is the only place where you use an `appInfoStringsRsc`. Follow these guidelines when creating the resource:

- The string list may contain up to 255 entries. Typically you don't want to predefine 255 categories.
- Each category name has a maximum length defined by the `catCategoryNameLength` constant (currently 31+1 bytes).
- It's common to have at least one category named Unfiled as the first item in the list. Records that aren't a member of any category are displayed under "Unfiled."

- Don't include strings for the "All" or "Edit Categories" items that you see in a categories pop-up list. When displaying the category list, the category manager will automatically generated list entries for these items.
- If you want any of the predefined categories to be uneditable by the user, call `CatMgrSetEditable()` and pass false as the editable attribute.

Listing 3.1 shows an example function that creates and initializes a schema database with category information.

Listing 3.5 Creating a schema database with categories

```
DatabaseID gDbID;
DmOpenRef gDbP;

Err CreateAndOpenDatabase(DmOpenRef *dbPP, uint16_t mode)
{
   status_t error = errNone;
   DmOpenRef dbP;
   DatabaseID dbID;
   DmDatabaseInfoType *myDbInfo;
   MemHandle myAppInfoStringsH;

   // Create the database.
   error = DbCreateDatabase (MyDBName, MyDBCreator, MyDBType, numSchemas,
      schemaList, &dbID);
   if (error < 0) return error;

   // Open the database.
   dbP = DbOpenDatabase(dbID, mode, dbShareNone);
   if (!dbP) return (dmErrCantOpen);

   // Get a DmOpenRef to the resource database that contains the app info
   // strings resource. Lock down the resource.
   if (!gDbP) {
      if ((error = SysGetModuleDatabase(SysGetRefNum(), &gDbID, &gDbP)) <
         errNone)
         return error;
   }
   myAppInfoStringsH = DmGetResource(gDbP, appInfoStringRsc,
      MyLocalizedAppInfoStr);

   // Initialize the categories.
   CatMgrInitialize(dbP, myAppInfoStringsH);
```

```
   // Set the output parameter and return.
   *dbPP = dbP;
   return error;
}
```

Initializing the Category Trigger for a Schema Database

When a form is opened, you need to set the text that the category trigger (whether pop-up or selector trigger) should display. To do this, use `CatMgrSetTriggerLabel()`.

For the main form of the application, it's common to store the ID of the previously selected category in a preference and restore it when the application starts up again.

Forms that display information from a single record from a schema database may show that record's category membership in a selector trigger or pop-up list. You can retrieve the record's category using `DbGetCategory()`. Keep in mind that the record may belong to many categories. The Category Manager functions display multiple categories as a comma-separated list.

Listing 3.2 shows how to set the trigger label to match the category for a particular database record.

Listing 3.6 Setting the category trigger label (schema database)

```
CategoryID categoryIDs[];
uint32_t numCategories;
ControlType *ctl;
char *nameP;

// Allocate memory for the nameP string, which will
// contain the trigger's label.
nameP = (char *)MemPtrNew(catCategoryNameLength);

// If current category is All or Multiple, we need to look
// up category membership
if ((CurrentCategory == catIDAll) ||
   (CurrentCategory == catIDMultiple))
   DbGetCategory (schemaDB, CurrentRecord, &numCategories,
      &categoryIDs);
else
   category = CurrentCategory;
```

```
ctl = FrmGetObjectPtr(frm,
   FrmGetObjectIndex(frm, objectID));
CategorySetTriggerLabel (schemaDB, categoryIDs,
   numCategories, ctl, nameP);

// Upon frmCloseEvent, free nameP and call
// DbReleaseStorage() if categoryIDs is not NULL.
```

Managing a Category Pop-up List for a Schema Database

When the user taps the category pop-up trigger on a form that manages a schema database, call `CatMgrSelectFilter()` if the pop-up list filters the display of multiple records or `CatMgrSelectEdit()` if the pop-up list chooses a new category for a single record. That is, call one of these functions in response to a `ctlSelectEvent` when the ID stored in the event matches the ID of the category's trigger. These functions display the pop-up list, manage the user selection, display the Edit Categories modal dialog as necessary, and set the trigger label to the item the user selected.

Listing 3.7 shows a typical call to `CatMgrSelectEdit()`. The call to `CatMgrSelectFilter()` is similar.

Listing 3.7 Calling CatMgrSelectEdit()

```
uint8_t category;
CategoryID inCategoryIDs[], outCategoryIDs[];
uint32_t numInCategories, numOutCategories;
ControlType *ctl;
char *labelString;

//Allocate space for labelString. outCategoryIDs space is
// allocated for you.
labelString = (char *)MemPtrNew(catCategoryNameLength);

categoryEdited = CatMgrSelectEdit (schemaDB, frm,
   ListCategoryTrigger, labelString, ListCategoryList, true
   inCategoryIDs, numInCategories, outCategoryIDs,
   &numOutCategories, true, NULL);

// Upon frmCloseEvent, deallocate labelString and call
// CatMgrFreeSelectedCategories() to free the category
// list allocated by CatMgrSelectEdit().
```

This example uses the following as parameters to `CatMgrSelectEdit()`:

- `schemaDB` is the schema database with the categories to be displayed.
- `frm`, `ListCategoryTrigger`, and `ListCategoryList` identify the form, pop-up or selector trigger resource, and list resource.
- `labelString` contains the text displayed in the category label. This string must be at least `catCategoryNameLength`. If you've got a lot of space to display the name, you might allocate a larger string.
- `true` indicates that a record can belong to multiple categories. The Category Manager adds a "Multiple" item to the list. If the user selects this item, it displays a dialog from which the user selects multiple categories.

  To disallow this behavior, pass `false` instead. Note that `CatMgrSelectFilter()` does not take this parameter. It does not allow the selection of multiple categories to display.
- `inCategoryIDs` contains a list of the category IDs to which the record currently belongs and `numInCategories` contains the number of elements in `inCategoryIDs`.
- Upon return, `outCategoryIDs` contains a list of the category IDs that the user selected, and `numOutCategories` contains the number of elements in `outCategoryIDs`. If the user does not change the selection, `outCategoryIDs` is `NULL`.
- `true` indicates that the pop-up list should contain the Edit Categories list item.
- `NULL` indicates that the default string should be used for the Edit Categories list item.

When the pop-up list is displayed, `inCategoryIDs` represents the record's category membership. If the user selects items in the list, this category membership changes, the function returns `true`, and `outCategoryIDs` reflects the new set of categories to which the record should now belong. This list supersedes the current category membership list. Your application is responsible for updating the record to reflect the new membership. In the case of

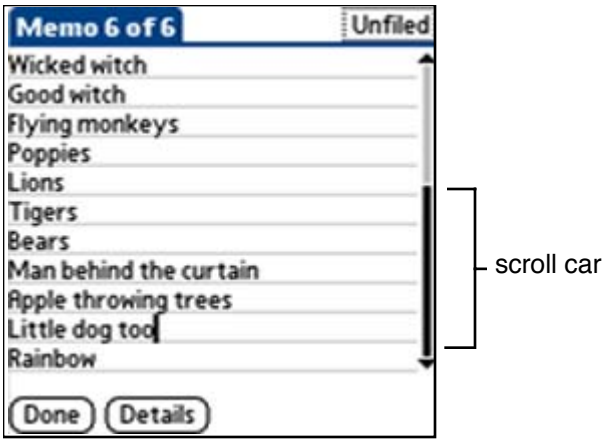
`CatMgrSelectFilter()`, your application is responsible for updating the display to reflect the new selection.

Note that the `CatMgrSelectEdit()` and `CatMgrSelectFilter()` functions handle the results of the Edit Categories dialog for you. They add, delete, and rename items in the database's private category structure. If the user deletes a category that contains records, they remove membership in that category from any existing records. If the user changes the name of an existing category to the name of another existing category, they prompt the user and, if confirmed, moves the records from the old category to the new category. Therefore, you never have to worry about managing the category list after a call to `CatMgrSelectEdit()` or `CatMgrSelectFilter()`.

Scroll Bars

A **scroll bar** is a control that is used to scroll user interface elements that are longer or wider than the screen. You can attach scroll bars to fields or tables and the system sends the appropriate events when the end user interacts with the scroll bar (see Figure 3.8).

Figure 3.8 Scroll bar



To do to include a scroll bar in your user interface, do the following:

1. Create a scroll bar UI resource in your resource file.

   Provide the ID and the bounds for the scroll bar rectangle. The height has to match the UI element you want to attach it to. The width should be 7 standard coordinates.

2. Provide a minimum and maximum value as well as a page size.

   - Minimum is usually 0.
   - Maximum is usually 0 and set programmatically.
   - The page size determines how many lines the scroll bar moves when the text scrolls.

3. In the UI element that is to be scrolled, set the attribute that indicates that it has a scroll bar attached to it. (You can also do this programmatically.)

There are two ways in which the scroll bar and the user interface element that it's attached to need to interact:

- When the user adds or removes text, the scroll bar needs to know about the change in size.

  If the "has scroll bar" attribute is set for a field, you'll receive a `fldChangedEvent` whenever the field's size changes. Your application should handle these events by computing new values for the scroll bar's minimum, maximum, and current position and then use `SclSetScrollBar()` to update it.

  If a table has a scroll bar attached, you should keep track of when the table's size changes. Whenever it does, you should compute new values for the scroll bar's minimum, maximum, and current position and then use `SclSetScrollBar()` to update it.

  You should also call `SclSetScrollBar()` when the form is initialized to set the current position of the scroll bar.

- When the user moves the scroll bar, the text needs to move accordingly. This can either happen dynamically (as the user moves the scroll bar) or statically (after the user has released the scroll bar).

The system sends the following scroll bar events:

- `sclEnterEvent` is sent when a `penDownEvent` occurs within the bounds of the scroll bar.
- `sclRepeatEvent` is sent when the user drags the scroll bar.
- `sclExitEvent` is sent when the user lifts the pen. This event is sent regardless of previous `sclRepeatEvents`.

Applications that want to support immediate-mode scrolling (that is, scrolling happens as the user drags the pen) need to watch for occurrences of `sclRepeatEvent`. In response to this event, call the scrolling function associated with the UI element (`FldScrollField()` or your own scrolling function in the case of tables).

Applications that don't support immediate-mode scrolling should ignore occurrences of `sclRepeatEvent` and wait only for the `sclExitEvent`.

Summary of Control Functions

Control Functions	
Displaying a Control	
CtlShowControl() CtlDrawCheckboxControl()	`CtlDrawControl()`
Enabling/Disabling	
CtlSetEnabled() CtlHideControl()	CtlEnabled() CtlEraseControl()
Control's Value	
FrmGetControlValue() CtlGetValue() FrmGetControlGroupSelection() CtlGetSliderValues()	FrmSetControlValue() CtlSetValue() FrmSetControlGroupSelection()
Label	
`CtlSetLabel()`	`CtlGetLabel()`

Control Functions

Event Handling

CtlHandleEvent()

Setting up controls

CtlSetGraphics()
CtlSetSliderValues()

Debugging

CtlHitControl()
CtlValidatePointer()

Dynamically Creating a Control

CtlNewGraphicControl()
CtlNewSliderControl()
CtlNewControl()

Category Manager Functions

Category Controls for Non-schema Databases

CategoryCreateList()
CategoryEdit()
CategoryFind()
CategoryFreeList()
CategoryGetName()
CategoryGetNext()
CategoryInitialize()
CategorySelect()
CategorySetName()
CategorySetTriggerLabel()
CategoryTruncateName()

Category Controls for Schema Databases

CatMgrCreateList()
CatMgrEdit()
CatMgrFind()
CatMgrFreeList()
CatMgrFreeSelectedCategories()
CatMgrGetName()
CatMgrGetNext()
CatMgrInitialize()
CatMgrSelectEdit()
CatMgrSelectFilter()
CatMgrSetName()
CatMgrSetTriggerLabel()
CatMgrTruncateName()

Category Manager Functions	
Obtaining Category Information for Schema Databases	
CatMgrAdd()	CatMgrGetUnfiledItemLabel()
CatMgrGetAllItemLabel()	CatMgrNumCategories()
CatMgrGetEditable()	CatMgrRemove()
CatMgrGetID()	CatMgrSetEditable()

Scroll Bar Functions	
SclSetScrollBar()	SclGetScrollBar()
SclHandleEvent()	SclDrawScrollBar()

4

Working with Menus

A menu bar is displayed whenever the user taps a Menu icon or whenever the user taps in a form's title bar. The menu bar, a horizontal list of menu titles, appears at the top of the screen in its own window, above all application windows. Pressing a menu title highlights the title and "pulls down" the menu below the title (see Figure 4.1).

Figure 4.1 Menu

User actions have the following effect on a menu:

When...	Then...
User drags the pen through the menu	Command under the pen is highlighted
Pen is released over a menu item	That item is selected and the menu bar and menu disappear
Pen is released outside both the menu bar and the menu	Both menu and menu bar disappear and no selection is made
Pen is released in a menu title	Menu bar and Menu remain displayed until a selection is made from the menu.
Pen is tapped outside menu and menu bar	Both menu and menu bar are dismissed

A menu has the following features:

- Item separators, which are lines to group menu items.
- Menu shortcuts; the shortcut labels are right justified in menu items.
- A menu remembers its last selection; the next time a menu is displayed the prior selection appears highlighted.

Menu Events

All applications that support menus should handle the `menuEvent`. This event is sent when the user selects a menu command from a menu. The application should respond by performing the requested action.

General event handling for menus is performed by `MenuHandleEvent()`. Table 4.1 describes how user actions get translated into events and what `MenuHandleEvent()` does in response.

Table 4.1 Event flow for menus

User Action	Event Generated	MenuHandleEvent() Response
Pen enters menu bar.	`penDownEvent` with the x and y coordinates. `winEnterEvent` identifying menu's window is generated when a pull-down menu's window is opened.	Tracks the pen without blocking. Responds to `penMoveEvent` as appropriate. If this is the first time the menu has been opened, adds a `menuOpenEvent` containing how the menu was opened.
User selects a menu item.	`penUpEvent` with the x and y coordinates.	Adds a `menuEvent` with the item's ID to the event queue.

Edit Menu

Forms that display editable text fields should provide the system default Edit menu. This menu is shown in Figure 4.1 on page 73. To provide the system default edit menu, do the following:

1. Create a menu resource in your application's resource file and give it the ID 10000.
2. Add the menu commands to the menu in the order shown in Figure 4.1 on page 73.
3. Associate the menu resource with the form's menu bar resource.

As long as you have given your Edit menu the ID 10000 and have the commands listed in the same order, Palm OS® handles all of the `menuEvent`s for your Edit menu. You do not have to write any code.

Menu Command Shortcuts

As an alternative to selecting a menu command through the user interface, users can instead write a menu shortcut in the input area if the current handwriting recognition engine supports it.

IMPORTANT: Only use a single character from A to Z (case-insensitive) to define a shortcut. Do not use accented characters.

The user can draw a menu command stroke followed by another character. If the next character matches one of the shortcut characters for an item on the active menu, a `menuEvent` with that menu item is generated. To support this behavior, you simply specify a shortcut character when you create a menu item resource. The default behavior of Palm OS handles this shortcut appropriately.

Drawing the menu command stroke displays the command tool bar (see Figure 4.2). This tool bar is the width of the screen. The command tool bar displays a status message on the left and buttons on the right. After drawing the command stroke, the user has the choice of writing a character or of tapping one of the buttons on the command tool bar. Both of these actions cause the status message to

be briefly displayed and (in most cases) a `menuEvent` to be added to the event queue.

Figure 4.2 Command tool bar

The buttons displayed on the tool bar depend on the user context. If the focus is in an editable field, the UI Library displays buttons for cut, copy, and paste on the command tool bar. If there is an action to undo, the UI Library also displays a button for undo.

The active application may also add its own buttons to the tool bar. To do so, respond to the `menuCmdBarOpenEvent` and use `MenuCmdBarAddButton()` to add the button. Listing 4.1 shows some code from the Memo Pad application that adds to the command tool bar a button that displays the security dialog and then prevents the UI Library from adding other buttons.

Listing 4.1 Responding to menuCmdBarOpenEvent

```
else if (event->eType == menuCmdBarOpenEvent) {

   MenuCmdBarAddButton(menuCmdBarOnLeft, myAppDB,
      BarSecureBitmap, menuCmdBarResultMenuItem,
      ListOptionsSecurityCmd, 0);

   // Tell the field package to not add buttons
   // automatically; we've done it all ourselves.
   event->data.menuCmdBarOpen.preventFieldButtons = true;

   // Don't set handled to true; this event must
   // fall through to the system.
}
```

The system contains bitmaps that represent such commands as beaming and deleting records. If your application performs any of these actions, it should use the system bitmap. Table 4.2 shows the system bitmaps and the commands they represent. If you use any of these, you should use them in the order shown, from right to left. That is, `BarDeleteBitmap` should always be the rightmost of these bitmaps, and `BarInfoBitmap` should always be the leftmost.

Table 4.2 System command tool bar bitmaps

Bitmap	Command
`BarDeleteBitmap`	Delete record.
`BarPasteBitmap`	Paste clipboard contents at insertion point.
`BarCopyBitmap`	Copy selection.
`BarCutBitmap`	Cut selection.
`BarUndoBitmap`	Undo previous action.
`BarSecureBitmap`	Show Security dialog.
`BarBeamBitmap`	Beam current record.
`BarInfoBitmap`	Show Info dialog (Launcher).

You should limit the buttons displayed on the command tool bar to 4 or 5. There are two reasons to limit the number of buttons. You must leave room for the status message to be displayed before the action is performed. Also, consider that the tool bar is displayed only briefly. Users must be able to instantly understand the meaning of each of the buttons on the tool bar. If there are too many buttons, it reduces the chance that users can find what they need.

Note that the UI Library already potentially displays 4 buttons by itself. If you want to suppress this behavior and display your own buttons when a field has focus, set the `preventFieldButtons` flag of the `menuCmdBarOpenEvent` to `true` as is shown in Listing 4.1.

Dynamic Menus

You can add, hide, or unhide menu items while the menu resource is being loaded.

A `menuOpenEvent` is sent when the menu resource is loaded. In response to this event, you can call `MenuAddItem()` to add a menu item to one of the pull-down menus, `MenuHideItem()` to hide a menu item, or `MenuShowItem()` to display a menu item.

You might receive `menuOpenEvent` several times during an application session. The menu resource is loaded each time the menu is made the active menu. A menu becomes active the first time the user either requests that the menu be displayed or enters the command keystroke on the current form. That menu remains active as long as the form with which it is associated is active. A menu loses its active status under these conditions:

- When `FrmSetMenu()` is called to change the active menu on the form.
- When a new form, even a modal form or alert panel, becomes active.

Suppose a user selects your application's About item from the Options menu then clicks the OK button to return to the main form. When the About dialog is displayed, it becomes the active form, which causes the main form's menu state to be erased. This menu state is not restored when the main form becomes active again. The next time the user requests the menu, the menu resource is reloaded, and a new `menuOpenEvent` is queued.

You should only make changes to a menu the first time it is loaded after a form becomes active. You should not add, hide, or show items based on user context. Such practice is discouraged in the Palm OS user interface guidelines.

Summary of Menu Functions

Menu Functions	
MenuDispose()	MenuDrawMenu()
MenuEraseStatus()	MenuInit()
MenuHandleEvent()	MenuGetActiveMenu()
MenuSetActiveMenu()	MenuSetActiveMenuRscID()
MenuAddItem()	MenuCmdBarAddButton()
MenuCmdBarDisplay()	MenuCmdBarGetButtonData()
MenuHideItem()	MenuShowItem()

5

Displaying Text

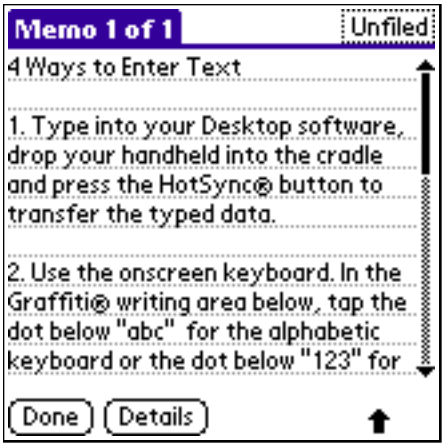
This chapter describes user interface elements that display text or that assist in the display of text. It covers:

See the book *Exploring Palm OS: Text and Localization* for information about manipulating the text strings that are stored in these elements.

Text Fields

A **text field** displays one or more lines of text. Figure 5.1 shows a form whose main element is an underlined, left-justified, multi-line field containing data.

Figure 5.1 Text field



The text field supports these features:

- Proportional bitmapped fonts (only one font per field)
- Drag-selection
- Scrolling for multiline fields
- Cut, copy, and paste
- Left and right text justification
- Tab stops
- Insertion point positioning with pen (the insertion point is positioned by touching the pen between characters)
- Double-tap to select a word

The text field does *not* support overstrike input mode; horizontal scrolling; numeric formatting; or special keys for page up, page down, left word, right word, home, end, left margin, or right margin.

Events in fields are handled by `FldHandleEvent()`. Table 5.1 provides an overview of how `FldHandleEvent()` deals with the different events

Table 5.1 Event flow for fields

User Action	Event Generated	FldHandleEvent() Response
Pen goes down on a field.	`penDownEvent` with the x and y coordinates stored in `EventType`.	Adds the `fldEnterEvent` to the event queue.
	`fldEnterEvent` with the field's resource ID.	Sets the insertion point position to the position of the pen and tracks the pen without blocking until it is released. Responds to the `penMoveEvent` as appropriate. Drag-selection and drag-scrolling are supported.

Table 5.1 Event flow for fields *(continued)*

User Action	Event Generated	FldHandleEvent() Response
		Double-tapping in a field selects the word at that location, and triple-tapping selects the line.
Pen is lifted.	`penUpEvent` with the x and y coordinates.	Nothing happens; a field remains selected until another field is selected or the form that contains the field is closed.
Enters character into selected field.	`keyDownEvent` containing character value in `EventType`.	Character added to field's text string. If there is an active FEP, keys are passed to it first so that it has a chance to convert characters.
Presses up arrow key	`keyDownEvent` containing `chrUpArrow`.	Moves insertion point up a line.
Presses down arrow	`keyDownEvent` containing `chrDownArrow`.	Moves insertion point down a line; the insertion point doesn't move beyond the last line that contains text.
Presses left arrow	`keyDownEvent` containing `chrLeftArrow`.	Moves insertion point one character position to the left. When the left margin is reached, move to the end of the previous line.
Presses right arrow	`keyDownEvent` containing `chrRightArrow`.	Moves insertion point one character position to the right. When the right margin is reached, move to the start of the next line.

Table 5.1 Event flow for fields *(continued)*

User Action	Event Generated	FldHandleEvent() Response
Cut command	`keyDownEvent` containing `vchrFieldCut`.	Cuts the current selection to the text clipboard.
Copy command	`keyDownEvent` containing `vchrFieldCopy`.	Copies the current selection to the text clipboard.
Paste command	`keyDownEvent` containing `vchrFieldPaste`.	Inserts clipboard text into the field at insertion point.
Undo command	`keyDownEvent` containing `vchrFieldUndo`.	Undoes the previous operation.

Depending on the attributes you set when you create the field resource, a field may be either editable or noneditable and either single-line or multi-line. The following sections describe each type of field.

Editable Text Fields

An editable text field has an underline to indicate where the user can tap to enter text. If the field has the focus, it displays a blinking insertion point. If you create an editable text field, you should include the following elements in your form:

- An Edit menu so that the user can cut, copy, and paste.
- A shift indicator to indicate the current input mode on displays with static input areas. See *Exploring Palm OS: Input Services* for more information. Be sure to set the text field's auto-shift attribute so that the first word of a sentence is automatically capitalized.

The editable attribute affects how you retrieve or change the text and how the memory associated with the text field is freed. A text field stores two types of information about the text string that it displays:

- A pointer to the text string.
- A handle to the pointer.

The text handle is used for editable text strings because the field must resize the string while the user edits it and the string may move in memory as a result of the resizing.

Use `FldGetTextHandle()` to retrieve the text of an editable text field. If you want to change the text, use any of the following functions:

- `FldInsert()` and `FldDelete()` for small changes.
- `FldSetTextHandle()` for larger changes.

Before you use `FldSetTextHandle()` to change the text, you must remove the association between the handle and the text field. If you change the text while it is being used by a field, the field's internal structures specifying the text length, allocated size, and word wrapping information can become out of sync. To avoid this problem, remove the text handle from the field, change the text, and then set the field's text handle again. See Listing 5.1.

Listing 5.1 Editing text with FldSetTextHandle()

```
static void SetFieldText(FieldType *fieldP, char *string)
{
   MemHandle textH;
   char *textP;

   // Get handle to field's text and remove associate between
   // field and text.
   textH = FldGetTextHandle(fieldP);
   FldSetTextHandle(fieldP, NULL);

   // Lock the handle and change the text. This code assumes
   // that textP is large enough to handle the string.
   textP = (char *)MemHandleLock(textH);
   strcpy(textP, string);

   // Reassociate the text with the field.
   MemHandleUnlock(textH);
   FldSetTextHandle(fieldP, textH);
}
```

Editable text fields also support an edit in-place feature. With edit in-place, you can point the text field at a string within a non-schema database record or a column within a schema database row. As the user edits the text field, the database record is automatically updated. Use `FldSetText()` for non-schema database records and `FldSetTextColumn()` for schema database rows.

NOTE: `FldSetTextColumn()` locks the entire database row, not just the column it displays. If you need to make a change to a value in another column, release the lock using `FldReleaseStorage()` and then use `FldReturnStorage()` to reassociate the field with the data it displays.

When the form is closed, all memory associated with all of the user interface elements on the form, including the text field and its text string, is freed. However, if the text field is pointing to a string in a database record, the string is *not* freed. If a text field is pointing to a string in the dynamic heap and you want to preserve the string after the form is closed, remove the association between the text handle and the text field before the field is freed. Do so by calling `FldSetTextHandle()` with `NULL` for the text handle in response to the `frmCloseEvent`.

Noneditable Text Fields

Noneditable text fields are not underlined and do not display a blinking insertion point. They are generally used to display complicated labels or labels that might change at run time.

As explained in the previous section, the text field contains both a handle and a pointer to the text string. Noneditable text fields often just use the pointer to the string and leave the handle as `NULL` because the memory resizing issues do not apply.

Use `FldGetTextPtr()` to retrieve the text of a non-editable text field and `FldSetTextPtr()` to set it. When you use `FldSetTextPtr()` to change the text, there is no association between the text handle and the text string pointer stored in the field. Because of this, the field does not free the string. If the string is in the dynamic heap, you must free it yourself. See Listing 5.2.

Listing 5.2 Freeing the string set by FldSetTextPtr()

```
case frmCloseEvent:
   strP = FldGetTextPtr(fldP);
   // Set the field text to NULL before freeing the string
   // to ensure that it is not being displayed when you
   // free it.
   FldSetTextPtr(fldP, NULL);
   MemPtrFree(strP);
```

Single-line Text Fields

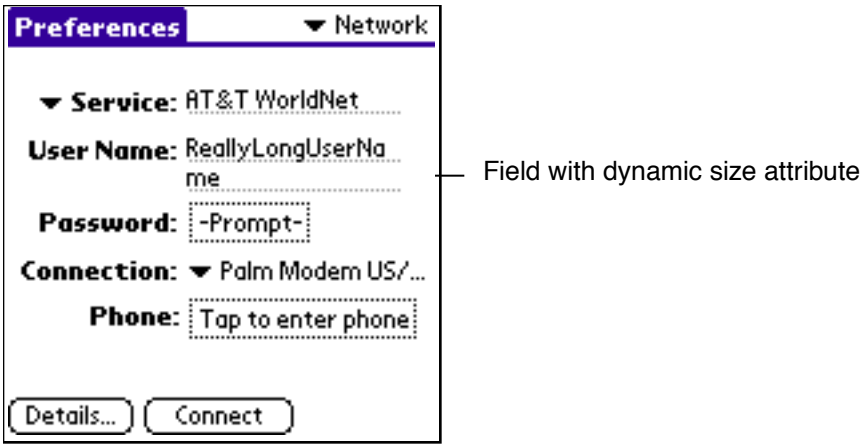
Single-line text fields do not wrap when the text exceeds the width of the field. Instead, the field displays as many characters as possible. Use single-line text fields only for fields likely to contain very small amounts of information, such as a monetary figure.

Multi-line Text Fields

Multi-line text fields display more than one line of text. Text fields handle all word wrapping for you.

You can also set a multi-line text field to be dynamically sized. If you do this, the number of lines of text that the field displays can grow as the user adds more text to the field. Depending on the form layout, it might be advantageous to only display a single line of text initially, but to set the dynamic size attribute instead of the single line attribute. If you do, the field can grow to display extra lines when the text exceeds the bounds of the field (see Figure 5.2). Dynamically sized fields send `fldHeightChangedEvent` when they need to be resized. Respond to this event to resize the field and, if necessary, reposition the rest of the elements on the form.

Figure 5.2 Dynamically sized field



Note that the `fldHeightChangedEvent` is completely independent of the `winResizedEvent`. The `winResizedEvent` is sent when the entire form is being resized. `fldHeightChangedEvent` is sent when only the dynamically sized field is being resized.

If the text field contains a large amount of text, attach a scroll bar to it. Palm OS® displays the scroll bar only when the text field contains more text than it displays, and Palm OS scrolls the field for you. Your application is responsible for updating the display of the scroll bar, however. See "Scroll Bars" on page 68.

Labels

Most resource editors allow you to create a label resource. Labels are ideal for small, static amounts of text that you need to display to identify other elements on the form.

You generally don't interact with a label as a programmatic entity. However, if you need to do so, use the following functions:

- FrmGetLabel() retrieves the label.
- `FrmGetLabelFont()` retrieves the bitmapped font used for a label.
- `FrmSetLabelFont()` sets the bitmapped font used for a label.

- `FrmCopyLabel()` changes the label. You may only pass a string that is the same size or shorter than the string initially allocated in the resource. If the new string is shorter, hide the label using `FrmHideObject()` before calling `FrmCopyLabel()` and then display it afterwards using `FrmShowObject()`.

If you need to change the label frequently, you may find it more convenient to use a noneditable text field rather than a label. See "Noneditable Text Fields" on page 84.

TIP: Palm OS Cobalt labels can handle line breaks.

Form Titles

The title displayed in the form is not a separate resource. It is an attribute that you specify when you create the form. If you need to change the form title at runtime, use the following functions:

- `FrmGetTitle()` retrieves the form's title.
- `FrmSetTitle()` sets the title of the form. This function does not copy the string that you pass to it, so you must ensure that it exists for as long as the form is displayed.
- `FrmCopyTitle()` copies the string into the form's title. If you use this function, you must set the initial form title in the resource file to the largest string you might display in the title. If the new string is shorter, hide the label using `FrmHideObject()` before calling `FrmCopyTitle()` and then display it afterwards using `FrmShowObject()`.

Fonts

Palm OS supports both bitmapped and scalable TrueType fonts. A **bitmapped font** is one that provides a separate bitmap for each glyph in each size and style. The Palm OS UI library, including all form controls, use bitmapped fonts for labels and other text.

A **scalable font** defines the shape of each glyph but not the size. Given a typeface definition, a scalable-font system can produce glyphs at any size. Scalable fonts are used only through the graphics

context `GcDrawTextAt()` call. See Chapter 8, "Drawing," on page 123 for more information on drawing with the graphics context.

Scalable font support begins in Palm OS Cobalt; however, it is optional. A given Palm OS Cobalt device might not include the scalable font engine in its ROM. If so, the functions used to access scalable fonts still work, but they return the bitmapped fonts.

Each font is associated with a particular character encoding. The font contains **glyphs** that define how to draw each character in the encoding. A bitmapped font provides a separate bitmap for each glyph in each size and style. A scalable font provides a **font family** that specifies how the glyphs look. Each font family supports one or more font **styles** or **faces** such as regular or bold. The font engine determines how to scale these fonts to different heights.

This section covers the following information about both bitmapped and scalable fonts:

- About Font Resources
- Built-in Fonts
- Setting the Font Programmatically
- Selecting Which Font to Use
- Obtaining Font Dimensions
- Creating and Using Custom Bitmapped Fonts
- Creating and Using Custom Scalable Fonts
- How Palm OS Displays Bitmapped Fonts

About Font Resources

For bitmapped fonts, Palm OS supports two types of resources: a single-density font resource `'NFNT'` and an extended font resource (`'nfnt'`) that supports multiple display densities. Both of these resources are the same 68K-based resources defined in all versions of Palm OS. ARM-based versions of these resources are not supported in Palm OS Cobalt. See "Bitmapped Font Resources" on page 213 for the exact layout of these resources.

The built-in bitmapped fonts provided with the system use extended font resources, meaning that they contain a separate set of

glyphs for each screen density. Thus, the single-density versions of the glyphs are used on standard 160 X 160 displays, the double-density versions of the glyphs are used for 320 X 320 displays, and so on.

You access any bitmapped font resource, whether `'NFNT'` or `'nfnt'`, using a `FontID` constant. This constant is essentially an index into the system's list of bitmapped fonts.

Scalable font resources are `'fttf'` resources. This type of resource creates a wrapper around a TrueType font specification. Because TrueType fonts are scalable by their nature, they already support multiple display densities.

Scalable fonts are accessed by creating a font handle using an HTML Cascading Style Sheet (CSS) style font specification string or a font family name. See the description of the `GcCreateFont()` function for an example of a CSS-style font specification string.

Built-in Fonts

There are several bitmapped and scalable fonts built into Palm OS.

The `Font.h` file defines constants that can be used to access the built-in bitmapped fonts programmatically. These constants are defined on all versions of Palm OS no matter what language or character code; however, they may point to different fonts. For example, `stdFont` on a Japanese system may be quite different from `stdFont` on a Latin system.

Table 5.2 lists and describes the built-in bitmapped fonts that may be used to display text.

Table 5.2 Built-in text bitmapped fonts

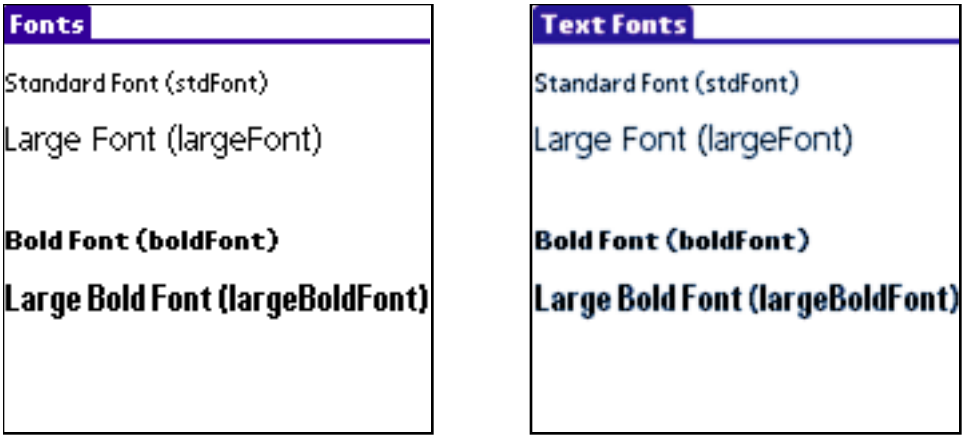
Constant	Description
`stdFont`	A small standard font used to display user input. This font is small to display as much text as possible.
`largeFont`	A larger font provided as an alternative for users who find the standard font too small to read.

Table 5.2 Built-in text bitmapped fonts *(continued)*

Constant	Description
`boldFont`	Same size as `stdFont` but bold for easier reading. Used for text labels in the user interface.
`largeBoldFont`	Same size as `largeFont` but bold.

Figure 5.3 shows what each of the fonts in Table 5.2 looks like.

Figure 5.3 Built-in text fonts



Low-density fonts High-density fonts

Palm OS also defines the bitmapped fonts listed in Table 5.3. These fonts do not contain most letters of the alphabet. They are used only for special purposes.

Table 5.3 Built-in symbol bitmapped fonts

Constant	Description
`symbolFont`	Contains many special characters such as arrows, shift indicators, and so on.
`symbol11Font`	Contains the check boxes, the large left arrow, and the large right arrow.

Table 5.3 Built-in symbol bitmapped fonts

Constant	Description
`symbol7Font`	Contains the up and down arrows used for the repeating button scroll arrows and the dimmed version of the same arrows.
`ledFont`	Contains the numbers 0 through 9, –, ., and the comma (,). Used by the Calculator application for its numeric display.

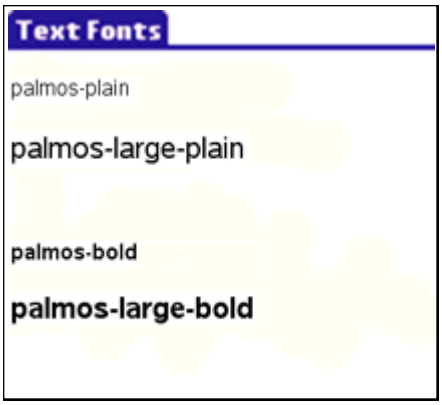
The system may also contain system-defined scalable fonts, which are accessed by passing a standard name to the function `GcCreateFont()`. Table 5.4 lists the strings that you can use. Note that these are not constants; they are strings that can be included as part of the CSS-style font specification you pass to `GcCreateFont()`. (CSS is an HTML term that stands for Cascading Style Sheet.)

Table 5.4 Built-in scalable fonts

Font name (string)	Description
palmos-plain	Default font used if no font is specified.
palmos-bold	Bold version of the palmos-plain font.
palmos-large-plain	palmos-plain with a larger point size.
palmos-large-bold	palmos-bold with a larger point size.

Figure 5.4 shows what each of the fonts in Table 5.4 looks like.

Figure 5.4 System-defined scalable text fonts



Note that if a device supports scalable fonts, these strings return TrueType fonts. They do not return the same fonts as the constants listed in Table 5.2. However, if a Palm OS Cobalt device does not support scalable fonts, these strings do return the same bitmapped fonts as the font constants listed in Table 5.2.

The fonts listed in Table 5.4 are all variations on a single font family. There may be other font families installed in the device. To obtain a full list, use code like that shown in Listing 5.3.

Listing 5.3 Iterating through all scalable fonts

```
int32_t numFamily, curFamily;
int32_t numStyle, curStyle;
FontFamily family;
FontStyle style;

numFamily = GcCountFontFamilies();
for (curFamily = 0; curFamily < numFamily; curFamily++) {
   GcGetFontFammily(curFamily, &family);
   numStyle = GcCountFontStyles(family);
   for (curStyle = 0; curStyle < numStyle; curStyle++) {
      GcGetFontStyle(family, curStyle, &style);
   }
}
```

Setting the Font Programmatically

To set the font that a user interface element uses for its label or for its textual contents, you use different functions depending on the

element. Table 5.5 shows which functions set fonts for which user interface elements. Note that user interface elements only support bitmapped fonts.

Table 5.5 Setting the font

UI Element	Function
Field	`FldSetFont()`
Field within a table	`TblSetItemFont()`
Command button, push button, pop-up trigger, selector trigger, or check box	`CtlSetFont()`
Label resource	`FrmSetLabelFont()`
List items	`LstSetFont()`
All other text (text drawn directly on the screen)	`FntSetFont()`

If you are drawing text directly to the screen, perform the following steps:

1. Call one of the following functions to create a handle to a specific font:
   - `GcCreateFont()` creates a font out of one of the strings in Table 5.4 on page 91 or out of a CSS-style font specification.
   - `GcCreateFontFromFamily()` creates a font using a specified family. You can then set the size and style later.
   - `GcCreateFontFromID()` creates a font handle for a bitmapped font. Use this function to have your text more closely match the built-in system fonts.

IMPORTANT: It's best to create a font handle before you receive the `frmUpdateEvent`. Creating a font handle can be slow.

2. Obtain a handle to the current graphics context using `GcGetCurrentContext()`.

3. Use the GcSetFont() function to set the draw state's font to the handle of the font you created.
4. Draw the text using GcDrawTextAt().
5. When you are finished drawing release the graphics context to free memory associated with it.
6. When you are finished with the font, call GcReleaseFont() to free the memory associated with the font handle.

Listing 5.4 shows drawing a text string in the `stdFont`.

Listing 5.4 Setting a font for custom-drawn text

```
// Make this call outside of the event loop, probably
// in your AppStart() function.
GcHandle gcStdFontH = GcCreateFontFromID(stdFont);


// In the EventLoop, do the following:
GcContext gcH;
char *string = "Hi, Mom!"

case frmUpdateEvent:
   gcH = GcGetCurrentContext();
   if (gcH) {
      GcSetCoordinateSystem(gcH, kCoordinatesNative);
      GcSetFont(gcH, gcStdFontH);
      GcDrawTextAt(gcH, 160, 100, string, strlen(string));
      GcReleaseContext(gcH);
}
break;


// Release the font when you are done with it. Possibly, in
// the AppStop() function.
GcReleaseFont(gcStdFontH);
```

Selecting Which Font to Use

The default fonts used to display normal text and bold text vary based on the handheld's character encoding. Devices with the Palm OS Latin encoding typically use `stdFont` and `boldFont`, while Japanese devices use `largeFont` and `largeBoldFont` as the

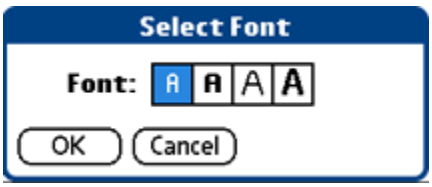
default. When your application starts up for the first time, it should respect the system defaults. Use the `FntGetDefaultFontID()` function to determine what the default fonts are (see Listing 5.5).

Listing 5.5 Determining the default system fonts

```
FontID textFont = FntGetDefaultFontID(defaultSystemFont);
FontID labelFont = FntGetDefaultFontID(defaultBoldFont);
```

In general, where users can enter text, you should allow them to select the font through the Select Font dialog (see Figure 5.5).

Figure 5.5 Select Font dialog



The `FontSelect()` function displays the Select Font dialog. This function takes as an argument a `FontID`, which specifies the value that is initially selected in the dialog. It returns the `FontID` that the user selected.

```
newFontID = FontSelect(textFont);
```

Because the default fonts vary based on the character encoding, the font size choices displayed in the Font Select dialog also vary based on character encoding. For this reason, you must call `FntGetDefaultFontID()` to obtain the default system font and pass the returned value to `FontSelect()` when you call it for the first time. On subsequent calls to `FontSelect()`, you can pass the user's current font choice.

If you are displaying text in a scalable font and want to respect the system default font, you can use the scalable font counterpart to the default system font. See Listing 5.6.

Listing 5.6 Setting a default scalable font

```
GcFontHandle gcLabelFontH;
FontID labelFont = FntGetDefaultFontID(defaultBoldFont);
```

```
switch (labelFont) {
   case stdFont:
      gcLabelFontH = GcCreateFont("palmos-plain");
      break;
   case boldFont:
      gcLabelFontH = GcCreateFont("palmos-bold");
      break;
   case largeFont:
      gcLabelFontH = GcCreateFont("palmos-large-plain");
      break;
   case largeBoldFont:
      gcLabelFontH = GcCreateFont("palmos-large-bold");
      break;
   default:
      gcLabelFontH = GcCreateFont("palmos-plain");
      break;
}
```

Obtaining Font Dimensions

Use functions in the Font Manager to obtain information about a bitmapped font and how it is drawn to the screen. Figure 5.6 shows graphically the characteristics of a font. Table 5.6 describes the types of information that can be retrieved with the Font Manager.

Figure 5.6 Font characteristics

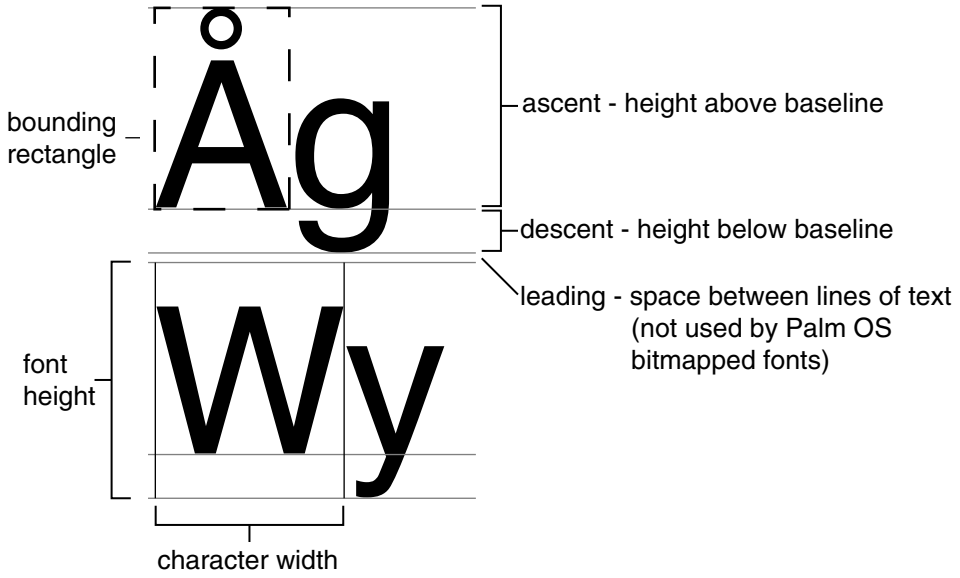


Table 5.6 Obtaining bitmapped font information

Characteristic	Function
Ascent + leading	`FntBaseLine()`
Descent	`FntDescenderHeight()`
Leading + font height	`FntLineHeight()` `FntCharHeight()`
Maximum width of a character in the font	`FntAverageCharWidth()`
Width of a specific character	`FntCharWidth()`
Character displayed at particular location	`FntWidthToOffset()`

NOTE: All dimensions are given in terms of the active coordinate system. Note also that because Palm OS does not support kerning of bitmapped fonts, `FntAverageCharWidth()` actually returns the maximum width of any character in the font.

The functions listed in Table 5.6 all work on the current font, that is, the font listed in the Window Manager draw state. To change the current font, use `FntSetFont()`. For example, to determine the line height of the font used for text in a field, do the following:

Listing 5.7 Obtaining characteristics of a field font

```
FieldType *fieldP;
FontID oldFont;
int16_t lineHeight;

oldFont = FntSetFont(FldGetFont(fieldP));
lineHeight = FntLineHeight();
FntSetFont(oldFont);
```

With a scalable font, all of the information about the font's dimensions is dependent on the current size you've set for the font.

Table 5.7 shows the functions that retrieve information about a scalable font.

Table 5.7 Obtaining scalable font information

Characteristic	Function
Point size	`GcGetFontSize()`
Ascent	`ascent` field of structure returned by `GcGetFontHeight()`
Descent	`descent` field of structure returned by `GcGetFontHeight()`
Leading	`leading` field of structure returned by `GcGetFontHeight()`
Font's bounding box (dimensions for a typical character)	`GcGetFontBoundingBox()`
Width of a specific character or a set of characters	`GcFontStringWidth()`

NOTE: All dimensions are given in terms of the native coordinate system.

Creating and Using Custom Bitmapped Fonts

You can create your own bitmapped font (`'NFNT'`) or extended font (`'nfnt'`) resource and use it only within your application.

Both the font and extended font resources are only large enough to support a font for the Palm Latin character encoding (256 characters). Defining a custom font for larger character sets is not supported.

Once you have defined a custom font in a resource file, you must assign it a font ID before you can use the font in your application.

The font ID is different from the font's resource ID. It's a number between 0 and 255. The font ID you use must be greater than or equal to `fntAppFontCustomBase`. All IDs less than that are reserved for system use. The function `FntDefineFont()` assigns a font ID to a font resource. The font resource must be locked for the entire time that the font is in use. It's a good idea to load the font resource, lock it, and assign a font ID in your application's `AppStart()` function. Unlock and release the font resource in the `AppStop()` function.

Listing 5.8 shows code that loads a font resource, assigns a font ID to that resource, and then draws characters to the screen using the new font.

Listing 5.8 Loading and using a custom font

```
#define customFontID ((FontID) fntAppFontCustomBase)

MemHandle customFontH;
FontType *customFontP;

Err AppStart(void)
{
   ...
   // Load the font resource and assign it a font ID.
   customFontH = DmGetResource(globalResDBPtr, fontRscType,
      MyCoolFontRscID);
   customFontP = (FontType *)MemHandleLock(customFontH);
   FntDefineFont(customFontID, customFontP);
   ...
}

void AppStop(void)
{
   ...
   //Release the font resource when app quits.
   MemHandleUnlock(customFontH);
   DmReleaseResource(customFontH);
   ...
}

void DrawCharsInNewFont(void)
{
   FontID oldFont = FntSetFont(customFontID);
   char *msg = "Look, Mom. It's a new font!");
   WinDrawChars(msg, strlen(msg), 28, 0);
```

```
   FntSetFont(oldFont);
}
```

To use a extended font, you use essentially the same code as above, except that you must change the resource type used in the `DmGetResource()` call to `fontExtRscType`:

```
customFontH = DmGetResource(globalResDBPtr, fontExtRscType,
   MyCoolExtFontRscID);
// rest as shown above.
```

Note that you still use a pointer to a `FontType` structure to access the extended font.

It's possible to create an extended font resource that contains only double-density glyphs and not low-density glyphs. You could define an `'NFNT'` resource for the low-density glyphs and an extended font resource with just the double-density glyphs. Then you could load and use the `'NFNT'` resource on all devices with low-density screens and load and use the extended font resource when the display is not low-density. If you do this, you must carefully check the display density before deciding which resource to load (see Listing 5.9).

Listing 5.9 Conditionally loading a font resource

```
#define customFontID ((FontID) fntAppFontCustomBase)
MemHandle fontH = NULL;
FontType *fontP;
uint16_t winVersion;
Err error;

error = FtrGet(sysFtrCreator, sysFtrNumWinVersion,
   &winVersion);
// If winVersion is >= 4, the high-density feature set
// is present. Check what type of display we are on
// and load the appropriate font resource.
if (!error && (winVersion >= 4)) {
   uint32_t density;
   error = WinScreenGetAttribute(winScreenDensity, &density);
   if (!error && (density != kDensityLow)) {
         // load and use the extended font resource.
         fontH = DmGetResource(globalResDBPtr,
            fontExtRscType, MyNewFontRscID);
```

```
            fontP = MemHandleLock(fontH);
      }
}

if (!fontH) {
  // Either the feature set is not present or we're on a
  // low-density screen. Load and use the 'NFNT' resource.
  fontH = DmGetResource(globalResDBPtr, fontRscType,
          MyOldFontRscID);
  fontP = (FontType *)MemHandleLock(fontH);
}

FntDefineFont(customFontID, fontP);
```

Creating and Using Custom Scalable Fonts

You can also create your own scalable fonts resources. Unlike bitmapped fonts, custom scalable fonts are available system-wide. Create a database of type `'ofnt'`. The XRD file that defines the database's resources should contain resource definitions like that shown in Listing 5.10.

Listing 5.10 Scalable font resource definition

```
<RAW_RESOURCE RESOURCE_ID="2000">
  <RES_TYPE> 'fttf' </RES_TYPE>
  <DATA_FILE> "../MyNewTrueTypeFont.ttf" </DATA_FILE>
</RAW_RESOURCE>
```

When Palm OS boots, the font engine searches for and loads all databases of type `'ofnt'` that it finds in the system; therefore, installing a new font database requires rebooting before you can access the fonts in it.

Because the font database is already loaded, you simply access your new fonts in the same manner as the fonts that came with the device. See "Selecting Which Font to Use" on page 94.

How Palm OS Displays Bitmapped Fonts

To support multiple display densities, Palm OS uses an extended bitmapped font resource. An extended font resource contains a separate set of glyphs for each supported density. At runtime, Palm

OS determines the current display density and then draws text using the glyphs that match the display density. On a double-density display, text is drawn using the double-density glyphs. On a low-density display, text is drawn using the low-density glyphs.

Palm OS uses extended font resources for each of the built-in fonts. If your application uses only these fonts, your text is drawn by default using high-density glyphs on high-density displays. You do not have to make any changes to your code for this to occur.

You only need to be concerned about the font density if you need to display more text than normal on a single-density screen or if you use custom fonts:

- When creating a custom bitmapped font, you'll want to create an extended font resource. See "Creating and Using Custom Bitmapped Fonts" on page 98.
- If an extended font resource is not available or does not contain glyphs that match the screen density, Palm OS uses the following selection algorithm when selecting the font, from high to low priority:
  a. Select the font with the correct density.
  b. Select the font whose density is one-half of the correct density.
  c. Select the font with the closest density, with a tie going to the lower-density font.
- When drawing text in a custom font to an off-screen window, you must take care. Off-screen windows also have a display density. If you create a low-density off-screen window and draw text to it, use a low-density font or an extended font with low-density glyphs. If the resource contains only high-density glyphs, the rendering system shrinks the high-density font bitmaps. This results in poor quality text when the off-screen bitmap is subsequently transferred, and pixel-doubled, to a high-density display. If a low-density font is available, the rendering system substitutes a low-density font when drawing text to the window in an attempt to produce the best possible aesthetic result.

  Because Palm OS includes glyphs for each supported density for each of the built-in fonts, this is only a potential problem if you are using a custom font.

The font scaling behavior can be overridden to allow your application to display more text at one time; `WinSetScalingMode()` can force the rendering system to use the low-density font, unscaled, for subsequent text drawing.

To draw text using high-density coordinates, set the high-density coordinate system by calling `WinSetCoordinateSystem()` before using the Font Manager functions to position text or extract font metrics.

Summary of Text Display Functions

Field Functions

Obtaining User Input

`FldGetTextHandle()`
`FldGetSelection()`
`FldSetDirty()`
`FldDirty()`

Editing Text

`FldInsert()`
`FldSetSelection()`
`FldReplaceText()`
`FldSetTextHandle()`
`FldDelete()`

Edit in Place

`FldSetText()`
`FldSetTextColumn()`
`FldGetTextColumn()`
`FldReleaseStorage()`
`FldReturnStorage()`

Cut/Copy/Paste

`FldCopy()`
`FldPaste()`
`FldCut()`
`FldUndo()`

Displaying Text

`FldSetTextPtr()`
`FldGetTextPtr()`

Field Functions

Scrolling

FldScrollField()
FldScrollable()
FldSetScrollPosition()
FldGetScrollPosition()
FldGetVisibleLines()
FldGetScrollValues()
FldGetNumberOfBlankLines()

Field Attributes

FldGetAttributes()
FldSetFont()
FldGetFont()
FldSetMaxChars()
FldGetMaxChars()
FldSetMaxVisibleLines()
FldSetAttributes()
FldGetBounds()
FldSetUsable()

Text Attributes

FldCalcFieldHeight()
FldGetTextHeight()
FldGetTextAllocatedSize()
FldGetTextLength()
FldSetTextAllocatedSize()
FldWordWrap()
FldGetLineInfo()

Working With the Insertion Point

FldGetInsPtPosition()
FldSetInsPtPosition()
FldSetInsertionPoint()
FldGrabFocus()
FldReleaseFocus()

Releasing Memory

FldCompactText()
FldFreeMemory()

Updating the Display

FldDrawField()
FldMakeFullyVisible()
FldSetBounds()
FldEraseField()
FldRecalculateField()

Event Handling

FldHandleEvent()
FldSendChangeNotification()
FldSendHeightChangeNotification()

Field Functions

Dynamic UI

FldNewField()

Form Label and Title Functions

FrmGetLabel()
FrmGetLabelFont()
FrmSetLabelFont()
FrmCopyLabel()
FrmGetTitle()
FrmSetTitle()
FrmCopyTitle()

Bitmapped Font Functions

Changing the Bitmapped Font

FontSelect()
FldSetFont()
CtlSetFont()
LstSetFont()
FntSetFont()
TblSetItemFont()
FrmSetLabelFont()

Accessing the Bitmapped Font Programmatically

FntGetFont()
FntGetDefaultFontID()
FntGetFontPtr()

Wrapping or Truncating Text

FntWordWrap()
FntTruncateString()
FntWordWrapReverseNLines()

String Width

FntCharsInWidth()
FntLineWidth()
FntCharsWidth()
FntWidthToOffset()

Character Width

FntAverageCharWidth()
FntCharWidth()

Bitmapped Font Functions

Height

FntCharHeight()
FntLineHeight()
FntBaseLine()
FntDescenderHeight()

Scrolling

FntGetScrollValues()

Creating a Bitmapped Font

FntDefineFont()
FntIsAppDefined()

Scalable Font Functions

Creating the Scalable Font Handle

GcCreateFont()
GcCheckFont()
GcCreateFontFromFamily()
GcReleaseFont()
GcCreateFontFromID()

Drawing with a Scalable Font

GcSetFont()
GcDrawTextAt()

Font Style

GcSetFontFace()
GcGetFontFace()
GcSetFontStyle()
GcGetFontFamilyAndStyle()
GcApplyFontSpec()

String Width

GcFontStringBounds()
GcFontStringBytesInWidth()
GcFontStringCharsInWidth()
GcFontStringWidth()
GcFontStringEscapement()

Dimensions

GcGetFontSize()
GcSetFontSize()
GcGetFontBoundingBox()
GcGetFontHeight()

Scalable Font Functions	
Counting Fonts	
GcCountFontFamilies()	GcCountFontStyles()
GcGetFontFamily()	GcGetFontStyle()
Setting the Font Rendering State	
GcSetFontAntialiasing()	GcGetFontAntialiasing()
GcSetFontHinting()	GcGetFontHinting()
GcSetFontTransform()	GcGetFontTransform()

6

Working with Tables and Lists

Tables and lists are two types of user interface elements that present lists of data. This chapter describes how to use both types of user interface elements:

Tables

Tables support multi-column displays. Examples are:

- The List view of the To Do application
- The Day view in the Date Book
- The main view and the Edit view of the Address Book application

When you create a table resource, you set the table to display a certain number of rows and columns and you set the overall bounds of the table. The rest of the table initialization occurs at runtime in response to `frmOpenEvent`.

It's important to remember that the number of rows and columns is fixed at compile time and cannot be changed at runtime. For example, the Address Book main view can display at most 14 table rows or contacts, but you may have several hundred contacts stored in the Address Book database. Many beginning Palm OS® programmers assume that the table contains rows for each of the hundreds of contacts but is only displaying 14. In reality, the table contains only those 14 rows that are being displayed. It's the application's responsibility to repopulate the table rows with data as the user scrolls the table up and down. Note that this is quite

different from a text field, which conceptually holds the entire text of a document whether it can display the full text or not.

Each table is made up of the following:

- Columns - Each column contains a user interface element of a particular type. The available types of elements are listed in `TableItemStyleType`.
- Rows - Each row shows a different set of data, for example, a database record or database row.
- Items - Items are the intersection of row and column, or a table cell.

  For each item, the table stores three values: an integer value, a pointer value, and a font ID. Most of the table item styles, however, use only one of these three values. See the `TableItemStyleType` description to find out what values are applicable to what types of items.

When the user presses the stylus down on a table, it highlights the tapped item and that tapped item only. If you want the entire row to highlight, create a single-column table.

Initializing a Table

In response to the `frmOpenEvent`, you should call `TblSetItemStyle()` to set the item style for each row and column in the table, as shown in Listing 6.1.

Listing 6.1 Initializing a table

```
for (row = 0; row < rowsInTable; row++) {
  TblSetItemStyle (table, row, completedColumn,
    checkboxTableItem);
  TblSetItemStyle (table, row, priorityColumn,
    numericTableItem);
  TblSetItemStyle (table, row, descColumn, textTableItem);
  TblSetItemStyle (table, row, dueDateColumn,
    customTableItem);
  TblSetItemStyle (table, row, categoryColumn,
    customTableItem);
}
```

Following this loop to initialize the style, you should call `TblSetItemPtr()` or `TblSetItemInt()`, depending on the style you set, to initialize the each item in each visible row.

You only need to set the item style once. You will need to set the item's value each time the user scrolls the table.

Table Callbacks

There are three possible callback functions that you might have to write to support tables:

- `TableLoadDataFuncType()` — Implement this callback if you have items that display editable text fields. It is called whenever the item with the editable text field is drawn or selected.

  The purpose of this function is to return a handle to the text that needs to be displayed.

- `TableDrawItemFuncType()` — Implement this callback if you've chosen a custom style. The custom styles mean that you want to control all aspects of table drawing.

  Most developers who use tables find that they need to use the custom table item and perform all drawing themselves.

- `TableSaveDataFuncType()` — This function is called only for the currently selected field right before the memory associated with the table is freed. You might implement it to free or to store the text associated with that field.

Table Events

The table generates the event `tblSelectEvent`. This event contains:

- The table's ID number
- The row of the selected table
- The column of the selected table

When `tblSelectEvent` is sent to a table, the table generates an event to handle any possible events within the item's UI element.

Lists

The **list** appears as a vertical list of choices in a box. The current selection of the list is inverted.

Figure 6.1 List

TIP: A list is not the same as a pop-up list, although the same set of functions is used to manipulate both. Pop-up lists are associated with pop-up triggers. See "Pop-Up Triggers" on page 52 for more information about pop-up triggers.

A list is meant for static data. Users can choose from a predetermined number of items. Examples include:

- The time list in the time edit window of the datebook
- The Category pop-up list (see "Category Controls" in this chapter)

If there are more choices than can be displayed, the system draws small arrows (scroll indicators) in the right margin next to the first and last visible choice. When the user taps a scroll indicator, the list is scrolled. When the user scrolls down, the last visible item becomes the first visible item if there are enough items to fill the list. If not, the list is scrolled so that the last item of the list appears at the bottom of the list. The reverse is true for scrolling up. Scrolling doesn't change the current selection.

Bringing the pen down on a list item unhighlights the current selection and highlights the item under the pen. Dragging the pen through the list highlights the item under the pen. Dragging the pen above or below the list causes the list to scroll if it contains more choices than are visible.

When the pen is released over an item, that item becomes the current selection. When the pen is dragged outside the list, the item that was highlighted before the `penDownEvent` is highlighted again if it's visible. If it's not, no item is highlighted.

An application can use a list in two ways:

- Initialize a structure with all data for all entries in the list (either in the resource file or in code at form initialization time) and let the list manage its own data.
- Provide list drawing functions but don't keep any data in memory. The list picks up the data as it's drawing.

  Not keeping data in memory avoids unacceptable memory overhead if the list is large and the contents of the list depends on choices made by the user. An example would be a time conversion application that provides a list of clock times for a number of cities based on a city the user selects. Note that only lists can pick up the display information on the fly like this; tables cannot.

  See "Custom Drawing List Items" on page 114 for sample code of how to draw list items.

List Events

Although lists generate events, you typically do not have to respond to events for a list. Instead, you check the value of the current list selection using `LstGetSelection()` or `LstGetSelectionText()` when some other event occurs.

NOTE: You cannot call `LstGetSelectionText()` if you call `LstSetListChoices()` and pass `NULL` or something other than the actual list items as the *`itemsText`* parameter. See "Custom Drawing List Items" for details.

The `LstHandleEvent()` function handles list events. Table 6.1 provides an overview of how `LstHandleEvent()` deals with the different events. For the event flow for pop-up lists, see Table 3.8 on page 53.

Table 6.1 Event flow for lists

User Action	Event Generated	LstHandleEvent() Response
Pen enters a list box.	penDownEvent with the x and y coordinates stored in EventType.	Adds the lstEnterEvent to the event queue. Begins tracking the pen without blocking. Responds to penMoveEvent and penUpEvent as appropriate.
Pen is lifted from the list box.	penUpEvent with the x and y coordinates stored in EventType.	Adds the lstSelectEvent with list's ID number and number of selected item to the event queue.
Pen is lifted outside the list box.	penUpEvent with the x and y coordinates stored in EventType.	Adds lstExitEvent to event queue.

Custom Drawing List Items

When the items in a list must be generated at runtime and can be quickly generated in any order, use a callback function to draw the contents of the list instead of providing the list with static strings.

In the `frmOpenEvent`, set the draw function and the number of items in the list. See Listing 6.2.

Listing 6.2 Setting up a custom drawn list

```
case frmOpenEvent:
   lstP = (ListType *)FrmGetObjectPtr(frmP,
FrmGetObjectIndex(frmP, MainTimesList);
   LstSetDrawFunction(lstP, ListDrawCallback);
   LstSetListChoices(lstP, NULL, 24);
   break;
```

Notice that this example passes `NULL` for the *`itemsText`* parameter of `LstSetListChoices()`. For static lists, the list contains a pointer to an array of strings that it displays for each item. When your application calls `LstSetListChoices()`, this parameter sets

the value of the list's pointer. Because the application is going to draw the list items, it sets the pointer to `NULL`. This means that the application cannot call `LstGetSelectionText()` to retrieve the list selection because the list does not know what the text is.

The callback is called for each line in the list that needs to be drawn. Fonts and colors have already been set.

Listing 6.3 Custom list drawing callback

```
void ListDrawCallback(int16_t itemNum, RectangleType *bounds,
char **itemsText)
{
   char buffer[bufferStringLength];

   // set the buffer based on the itemNum.
   GetListItemText(itemNum, buffer, bufferStringLength);

   WinDrawChars(buffer, strlen(buffer), bounds->topLeft.x,
      bounds->topLeft.y);
}
```

One of the benefits of having a draw function is that you do not need to worry about keeping track of memory that would have to be allocated for the list's contents. However, the drawing callback function will be called often. In addition to being called to draw each item, it is also called (with different foreground and background colors) to highlight or un-highlight items as the user taps them, so it needs to be fast. If it isn't fast enough, consider creating the list of strings beforehand. Another option is to have your callback function generate each item as necessary but store them in *`itemsText`* for fast access in case they are needed again.

Because calling `LstGetSelectionText()` is often not possible when you provide a custom drawing function, it's convenient to have a single function that copies the text for a given item number into a buffer. You can then call that function in both the drawing callback and in response to `lstSelectEvent`. For example, Listing 6.4 uses the same `GetListItemText()` function as is used in Listing 6.3.

Listing 6.4 Responding to lstSelectEvent for custom drawn list

```
case lstSelectEvent:
   GetListItemText(eventP->data.lstSelect.selection,
      lstSelection, lstSelectionSize);
   break;
```

Using Lists in Place of Tables

Lists really consist of single-column rows of text, but it is possible to imitate a multi-column display if you provide a custom list drawing function. Many programmers choose to use lists instead of tables for multi-column displays because lists are generally easier to program than tables are. Doing so is acceptable, but it is somewhat problematic because the list always displays a rectangular border around the list. If you choose to use lists to display multi-column data that would normally be displayed in a table, you must suppress the drawing of the list border. The safest way to do so is to set the draw window's clipping rectangle to the bounds of the list before drawing the list, as shown in Listing 6.5. See *Exploring Palm OS: User Interface Guidelines* for more information.

Listing 6.5 Suppressing the list border

```
void DrawFormWithNoListBorder(FormType *frmP,
uint16_t listIndex)
{
   RectangleType *clip;
   RectangleType *newClip;
   ListType *listP = FrmGetObjectPtr(frmP, listIndex);

   // Hide the list and then draw the rest of the
   // form.
   FrmHideObject(frmP, listIndex);
   FrmDrawForm (frmP);

   // Set the clipping rectangle to the list boundaries and
   // draw the list. This suppreses the list border.
   WinGetClip(&clip);
   FrmGetObjectBounds(frmP, listIndex, &newClip);
   WinSetClip(&newClip);
   LstSetSelection(listP, noListSelection);
   FrmShowObject(frmP, listIndex);
```

```
   // Reset the clipping rectangle.
   WinSetClip(&clip);
}

Boolean MyFormHandleEvent(EventPtr eventP)
{
   Boolean handled = false;
   FormType *frmP;
   UInt16 listIndex;

   switch (eventP->eType) {
      case frmUpdateEvent:
         frmP = FrmGetActiveForm();
         listIndex = FrmGetObjectIndex(frmP, MyListRscID);
         DrawFormWithNoListBorder(frmP, listIndex);
         handled = true;
         break;
      ...
   }
}
```

Summary of Table and List Functions

Table Functions

Drawing Tables

TblDrawTable()
TblSetCustomDrawProcedure()
TblSetLoadDataProcedure()

Updating the Display

TblInvalidate()
TblRedrawTable()
TblReleaseFocus()
TblRemoveRow()
TblMarkTableInvalid()
TblUnhighlightSelection()
TblGrabFocus()
TblUnhighlightSelection()
TblMarkRowInvalid()
TblSelectItem()

Table Functions

Retrieving Data

TblGetItemPtr()
TblFindRowData()
TblGetSelection()
TblSetSaveDataProcedure()
TblGetRowData()
TblGetItemInt()
TblGetCurrentField()

Displaying Data

TblSetItemInt()
TblSetItemPtr()
TblSetRowData()
TblSetItemStyle()
TblSetRowID()

Retrieving a Row

TblFindRowID()
TblGetRowID()

Table Information

TblEditing()
TblGetItemBounds()
TblGetNumberOfRows()
TblHasScrollBar()
TblGetBounds()
TblGetLastUsableRow()
TblSetBounds()

Row Information

TblGetRowHeight()
TblRowSelectable()
TblRowUsable()
TblSetRowStaticHeight()
TblSetRowHeight()
TblSetRowSelectable()
TblSetRowUsable()
TblRowInvalid()

Masked Records

TblRowMasked()
TblSetColumnMasked()
TblSetRowMasked()

Column Information

TblGetColumnSpacing()
TblGetColumnWidth()
TblSetColumnUsable()
TblSetColumnSpacing()
TblSetColumnWidth()
TblSetColumnEditIndicator()

Table Functions

Removing a Table From the Display

TblEraseTable()

Event Handling

TblHandleEvent()

Private Record Functions

SecSelectViewStatus()
SecVerifyPW()

List Functions

Displaying a List

LstDrawList()
LstPopupList()
LstSetDrawFunction()
LstNewList()

Updating the Display

LstMakeItemVisible()
LstSetListChoices()
LstSetSelection()
LstScrollList()
LstSetHeight()
LstSetTopItem()
LstSetPosition()

List Data and Attributes

LstGetNumberOfItems()
LstGetSelection()
LstGetSelectionText()
LstSetHeight()
LstSetIncrementalSearch()
LstSetScrollArrows()
LstGetTopItem()
LstGetVisibleItems()
LstGetFont()
LstSetFont()
LstGetItemsText()

Removing a List From the Display

LstEraseList()

List Functions
Event Handling
LstHandleEvent()

7

Creating Custom UI Elements (Gadgets)

A gadget resource lets you implement a custom UI element. A **gadget** is best thought of as simply a reserved rectangle at a set location on the form. You must provide all drawing and event handling code. There is no default behavior for a gadget.

If the gadget is an extended gadget, you can write a callback function (`FormGadgetHandlerType()`) to provide drawing and event handling code for the gadget. A pointer to the gadget is passed to the callback, so you can use the same function for multiple gadgets. Use `FrmSetGadgetHandler()` to set the callback function.

When the form receives certain requests to draw itself, delete itself, or to hide or show a gadget, it calls the gadget handler function you provide. When the form receives events intended for the gadget, it passes those to the gadget handler function as well.

Listing 7.1 shows an example of a gadget handler function.

Listing 7.1 Example gadget

```
Boolean GadgetHandler (struct FormGadgetType *gadgetP,
uint16_t cmd, void *paramP)
{
   Boolean handled = false;

   switch (cmd) {
      case formGadgetDrawCmd:
      // Sent to active gadgets any time form is drawn or
      // redrawn.
         DrawGadget();
         gadgetP->attr.visible = true;
         handled = true;
         break;

      case formGadgetHandleEventCmd:
```

```
      //Sent when form receives a gadget event.
      //paramP points to EventType structure.
         if (paramP->eType == frmGadgetEnterEvent) {
         // penDown in gadget's bounds.
            GadgetTapped ();
            handled = true;
         }
         if (paramP->eType == frmGadgetMiscEvent) {
            //This event is sent by your application
            //when it needs to send info to the gadget
         }
         break;
      case formGadgetDeleteCmd:
         //Perform any cleanup prior to deletion.
         break;
      case formGadgetEraseCmd:
        //FrmHideObject() takes care of this event if you
        //return false.
         handled = false;
         break;
   }
   return handled;
}
```

8

Drawing

The Palm OS® Cobalt rendering model includes support for complex path and shape construction, arbitrary two-dimensional transformations, a state stack, color gradients, alpha blending, and anti-aliasing. You interact with this new drawing model using the Graphics Context functions.

Previous versions of Palm OS had a more simplistic drawing model with a substantially different idea of the drawing state. The functions for this old drawing model are supported in Palm OS Cobalt and work as before. This chapter focuses first on the new drawing model and how to work with it. Later, it describes some compatibility issues that you might find when porting an older application to Palm OS Cobalt.

The topics covered are:

Drawing bitmaps to the screen and creating bitmaps programmatically are closely related topics that are not covered in this chapter. See Chapter 9, "Working with Bitmaps," on page 147.

Conceptual Overview

This section introduces you to the features of the new drawing model. It covers:

Graphics Context

The new drawing API does not draw directly to a window on the screen. Instead, it draws to a **graphics context**, which stores the current rendering state.

The Palm OS Cobalt rendering system has more control over the size and placement of windows and thus when to draw into them. When the rendering system needs to refresh the screen, it creates graphics contexts, tells all of the windows on the screen to update themselves, and then destroys the graphics contexts when all drawing is finished. Your application sees this request to draw as a `frmUpdateEvent`. Thus, to guarantee that your window has a graphics context that is ready to accept drawing, your application should only draw in response to an `frmUpdateEvent`.

For performance reasons, none of the Graphics Context functions perform any error checking on the parameters you pass. (Error checking is performed by the rendering system.) You must verify that you have a graphics context when you try to acquire one.

WARNING! Drawing to a `NULL` graphics context causes an application crash.

The rendering system runs in a different process than the application. To avoid IPC overhead, drawing operations are buffered and sent to the rendering system in batches. The rendering system sends its buffered commands at the end of each event loop. Because of this, drawing is likely not to have finished when a drawing function returns. In the rare cases where you need to wait for the IPC to complete before your application continues, there are two functions, `GcFlush()` and `GcCommit()`, that allow you to do so; however, they are slow and should be used only for debugging.

Path-Based Drawing

The new Graphics Context drawing functions behave very much like the Postscript language. Instead of drawing simple shapes directly to a window, you define a complex path and then paint it onto the screen. A **path** consists of shapes made out of lines, arcs, and curves. A single path can have numerous disjointed shapes in it. Each shape is called a **subpath**.

Think of drawing with these functions as painting a portrait. First you sketch lightly in pencil what you want to paint. Then you fill the sketch with paint.

The path has a notion of a **current point**. All drawing functions draw from the current point to the points you specify in the function.

To begin a path, you generally use `GcMoveTo()` to define the starting point. All drawing functions that you call from then on connect to the end point of the previous call until you call `GcMoveTo()` again. The `GcMoveTo()` function is like lifting the pencil and moving it to the location you specify.

When you have defined the path and are ready to paint it onto the screen, you call `GcPaint()`. `GcPaint()` fills the shapes contained in the path onto the screen and then clears the path from the rendering state. It does not clear any other aspect of the rendering state.

By default, `GcPaint()` expects to fill the current path. If you want `GcPaint()` to trace the path, you should call `GcStroke()` first. `GcStroke()` turns the path definition into the outline of the shape rather than the shape itself. (This is called a **stroked path**.)

Alpha Blending

When you set the color in the rendering state, you may specify an alpha transparency level. An alpha level of 0 is fully transparent, and an alpha level of 255 is fully opaque.

If you specify an alpha level somewhere between fully transparent and fully opaque, the rendering system composites pixels in the path with the current value of those pixels on the screen using standard alpha blending. This is a change from earlier Palm OS releases, where any current drawing simply overwrote any previous drawing.

Striking Rule

The **striking rule** controls how the system determines which pixels get painted. The coordinate system and antialiasing both affect the striking rule.

You define a path in terms of coordinates. The coordinates you specify are floating-point values, allowing you to specify sub-coordinate boundaries. The border of a path is always interpreted to have 0 width. The rendering system draws the path using the coordinates you've specified, and then it looks at which pixels are covered by the path. For stroked paths, it takes the pen width into consideration. The pen width is drawn such that 50% of the width is on one side of the path and 50% is on the other side of the path.

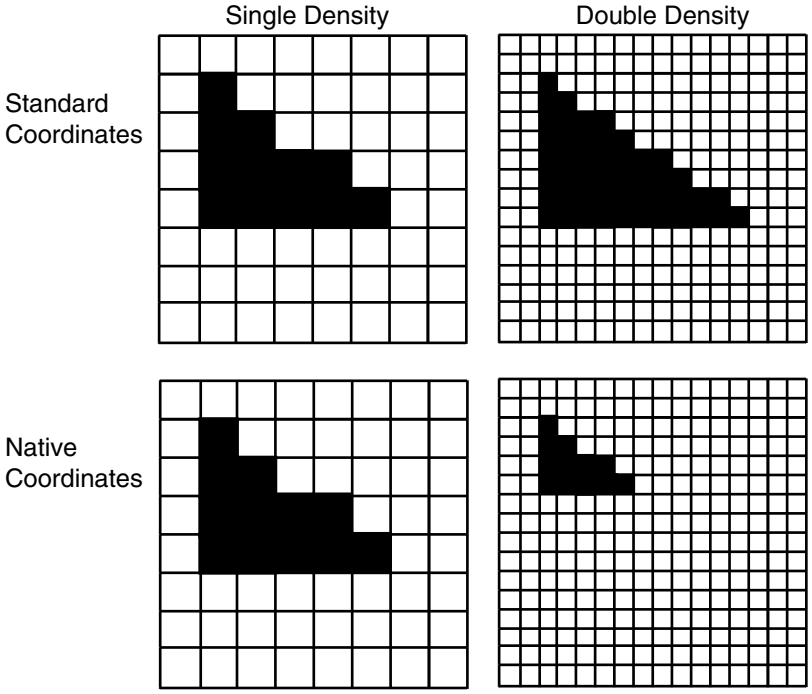
As described in Chapter 1, "The Display," on page 3, the **coordinate system** specifies how the values that you pass to the path definition functions are interpreted. The rendering model uses the current draw window's coordinate system by default, so you can set it to use standard coordinates, native coordinates, or a specific screen density.

Figure 8.1 shows the result of executing the code in Listing 8.1 for both standard coordinates and native coordinates on a single-density and a double-density display.

Listing 8.1 Drawing a triangle

```
GcMoveTo(gc, 1.0, 1.0);
GcLineTo(gc, 1.0, 5.0);
GcLineTo(gc, 6.0, 5.0);
GcClosePath(gc);
GcPaint(gc);
```

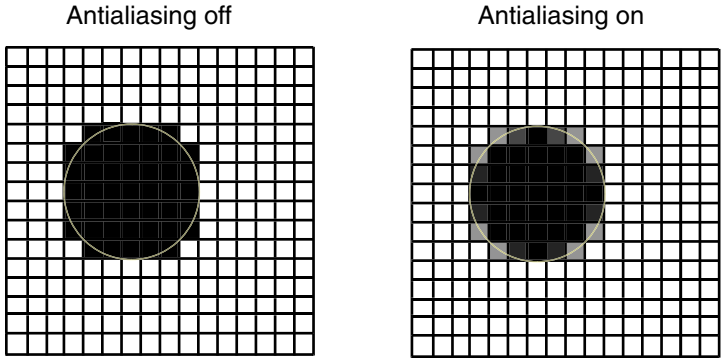
Figure 8.1 Standard and native coordinate systems



TIP: It is acceptable to specify coordinates that lie outside of screen boundaries. Any drawing outside of the current draw window's boundaries is simply clipped.

Antialiasing specifies whether the rendering system tries to smooth out rough lines. When antialiasing is off, the system decides whether to paint a pixel as follows: If the center point of the pixel falls within the line or shape being drawn, that pixel is painted with the current color. When antialiasing is on, each pixel that contains part of the shape gets a color, whether that pixel's center point is in the shape or not. The alpha is determined by how much of the shape is in the pixel. If only 10% of the shape is in the pixel, then the pixel gets a value equal to 10% of the color, and so on. This has the effect of making curved lines appear smoother (see Figure 8.2).

Figure 8.2 Antialiasing



Basic Drawing Steps

This section introduces you to the basic things you need to know to work with the new Graphics Context API. Read it to get started.

Listing 8.2 shows a simple drawing example.

Listing 8.2 Drawing a button frame

```
GcHandle gc;
case frmUpdateEvent:
   gc = GcGetCurrentContext();
   if (gc) {
      GcSetColor(gc, 0, 0, 0, 255);
      GcRoundRect(gc, 0, 0, 50, 15, 3, 3);
      GcPaint(gc);

      GcReleaseContext(gc);
   }
break;
```

The basic steps for drawing with the new functions are:

1. Call `GcGetCurrentContext()`.

   This function obtains a handle to the graphics context that you pass to the rest of the functions.
2. Set the rendering state.
3. Define the path.
4. Paint the path onto the screen.

5. Call `GcReleaseContext()`.

   This function signals that you are done drawing and the graphics context memory can be freed. If you do not release the graphics context, your user interface will eventually freeze.

IMPORTANT: Do not change the draw window in between calls to `GcGetCurrentContext()` and `GcReleaseContext()`.

The next several sections discuss these steps in more detail.

Setting the Rendering State

The rendering state contains the attributes listed in Table 8.1.

Table 8.1 Rendering State

State	How to Change	Description
Color	`GcSetColor()`	The color `GcPaint()` uses when stroking or filling. The default is black.
Coordinate system	`GcSetCoordinateSystem()`	Determines how the coordinates you pass to the path definition functions are converted to pixels. The default is the current draw window's coordinate system.
Pen width	`GcSetPenSize()`	For stroked paths only, the width of each line drawn. The default is 1 coordinate.
Line cap	`GcSetCaps()`	For stroked paths only, specifies what the ends of lines look like. The default is `kButtCap`.
Join style	`GcSetJoin()`	For stroked paths only, specifies what the connection between two lines looks like. The default is a bevel join (`kBevelJoin`).

Table 8.1 Rendering State *(continued)*

State	How to Change	Description
Font	GcSetFont()	For text paths only, the font in which the text is drawn. The default is the Palm OS plain scalable font.
Transformations	GcReflect(), GcRotate(), GcScale(), GcShear(), GcTranslate(), GcTransform()	Specify changes to the coordinate system that affect subsequent path definitions. The default is no transformation.
Antialiasing	GcSetAntialiasing()	Specifies whether antialiasing is on or off. The default is off.
Text antialiasing	GcSetFontAntialiasing()	Specifies whether antialiasing is on or off for text. The default is on.
Text hinting	GcSetFontHinting()	If enabled, characters are always drawn to exact pixel boundaries. The default is on.
Text transformation	GcSetFontTransform()	Specify changes to the text definition. The default is no transformation.
Gradient	GcInitGradient(), GcSetGradient()	Specifies a gradient. The default is no gradient.

Before changing the state, it is a good idea to preserve the current state with GcPushState(). Remember that the path is part of the current state, so when you call GcPopState() to restore the previous state, you'll clear any path definition you've just performed.

For performance reasons, you should specify the rendering state first, then define the path. Don't change the state in the middle of the path unless you are applying one of the transformations. The transformation is the only part of the state that should apply to individual points and not the path as a whole. It is best to leave most state settings alone unless you know that your application absolutely must have a specific value.

However, you should also be aware that some of the functions that draw standard user interface elements to the screen change the coordinate system and other aspects of the draw state. Therefore, it is a good idea to preserve the state before drawing standard user interface elements and pop it afterwards.

Defining a Path

As stated previously, a path is a series of connected or disconnected lines or shapes.

The following functions are path definition primitives:

- `GcMoveTo()` — Move the current point to the location.
- `GcLineTo()` — Draw a straight line from the current point.
- `GcClosePath()` — Draw a straight line from the current point to the start of the most recent subpath. This function is most often used to close the path so that it is a shape that can be filled.
- `GcBezierTo()` — Draw a Bezier curve from the current point.

In addition, the graphics context defines the following convenience functions built from the primitives. Always use the convenience functions where possible. They are optimized for performance.

- `GcRect()` — Draws a rectangle.
- `GcRoundRect()` — Draws a rectangle with rounded corners.
- `GcRectI()`— Draws a rectangle using integer coordinates.
- `GcArcTo()` — Draws an arc connected to the current point.
- `GcArc()` — Draws an arc connected to the current point.

Painting the Path

When you have defined the path and are ready to paint it onto the screen, you call `GcPaint()`.

By default, `GcPaint()` expects to fill the current path with the current color. If you want to draw the outline of the path, you

should call GcStroke() first. GcStroke() turns the path definition into the outline of the shape rather than the shape itself.

You must only paint closed paths. If you've defined an open path such as two lines connected to each other, use GcStroke() to specify that these two lines are what you want filled. The results are undefined if the graphics context tries to fill an open path.

There are two special cases of painting paths to the screen: text and bitmaps.

If you want to paint text to the screen, you specify the path and paint it to the screen in a single function: GcDrawTextAt(). This function clears any other path that you had defined previously, so it is best to call this outside of any other path definition.

"Displaying a Bitmap on the Screen" on page 151 describes how to paint a bitmap to the screen.

Specifying Clipping Regions

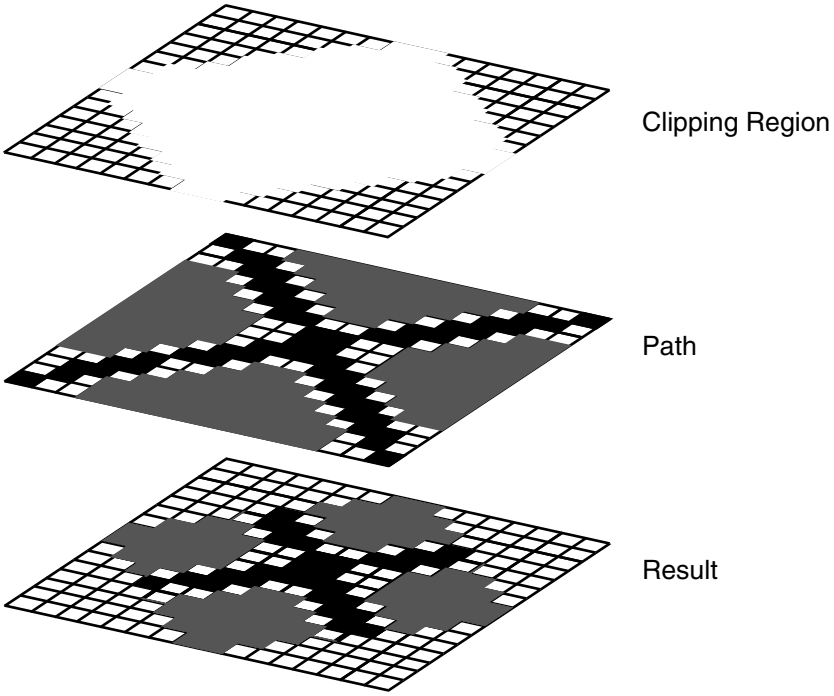
The rendering system allows you to create a **clipping region**. When a clipping region is defined, GcPaint() is constrained to painting only into that region. Any portions of the path outside of that region are discarded.

To specify a clipping region, do the following:

1. Call GcGetCurrentContext().
2. Call GcBeginClip().
3. Call one or more of the path definition functions to specify the clipping region.
4. Call GcPaint(), which fills the specified region with white.
5. Call GcEndClip().
6. Call the path definition functions to create the path you want to draw.
7. Call GcPaint().This call to GcPaint() only paints the areas of the path that are contained in the region you defined in Step 3.
8. Call GcReleaseContext().

Figure 8.3 shows an example of specifying a diamond shaped clipping region and then drawing a pattern that is constrained to the screen.

Figure 8.3 Clipping region



The clipping region remains in effect until you call `GcPopState()`. If you define a new clipping region before calling `GcPopState()`, all painting is constrained to the intersection of those clipping regions. If you want to paint to two disjointed areas of the screen, include them both in the path before calling `GcEndClip()`.

Clipping regions are usually rectangular, but they do not have to be. Although you can specify any complex shape as the clipping region, you should use a simple rectangle whenever possible. Doing so allows the system to perform some optimizations when rendering. Note that clipping regions are never antialiased.

Where and When to Draw

The **draw window** is the window that receives the painted pixels. Drawing may be performed into on-screen windows, off-screen windows, or bitmaps.

On-Screen Windows

As described in "Window Type" on page 17, an on-screen window may be either an update-based window, a transitional window, or a legacy window. If you are drawing to an update-based window, you may only draw in response to a `frmUpdateEvent` because that is the only time you are guaranteed to have a graphics context. For transitional windows, drawing only in response to update events is not a requirement, but it is a good idea to still try to follow this rule.

Off-Screen Windows

To perform double-buffering, you can create an **off-screen window** and draw to it. You may draw to off-screen windows at any time without producing an error.

Size can be an issue with off-screen windows. They allocate a region of memory proportional to the size of the window. You should ensure that the window's dimensions are only as large as necessary. For example, if all you want to do is draw a 10-pixel long horizontal black line, you can create a window with the width of 10, the height of 1, and the bit depth of 1.

After you create the off-screen window, set the draw window to be the off-screen window, and then obtain the graphics context and perform the drawing operations. See Listing 8.3.

Listing 8.3 Creating an off-screen window

```
WinHandle win;
GcHandle gc;
uint32_t depth = 16;

WinPushDrawState();
//Set the bit depth.
WinScreenMode(winScreenModeSet, NULL, NULL, &depth, NULL);
```

```
// Create off-screen window using native format.
WinSetCoordinateSystem(kCoordinatesNative);
win = WinCreateOffscreenWindow(width, height, nativeFormat,
   &error);

//Set the draw window, get the graphics context, and draw.
WinSetDrawWindow(win);
gc = GcGetCurrentContext();
if (gc) {
   //draw
   GcReleaseContext(gc);
}
WinPopDrawState();
```

Bitmaps

You can programmatically create a bitmap, draw into it, and then later paint it onto the screen or store it in a database for future use. See Chapter 9, "Working with Bitmaps," on page 147 for more information.

Compatibility with the Old Drawing System

If you are porting a 68K-based application that does drawing, it probably uses the `WinDraw...` or `WinPaint...` functions (such as `WinDrawLine()` or `WinPaintLine()`).

The old window drawing functions are still supported for backward compatibility and work as you would expect.

The Graphics Context functions and the Window Manager drawing functions live in two separate worlds. They have separate notions of the drawing state. Do not mix the two. Do not use the functions described in "Window Drawing Functions and Macros" on page 757 in between calls to `GcGetCurrentContext()` and `GcReleaseContext()`. Do not expect the Window Manager functions that set the draw state to affect the Graphics Context (see Listing 8.4).

Listing 8.4 Do not mix Window drawing and Graphics Context drawing

```
GcHandle gc;
RGBColorType *textColor, *oldTextColor;

textColor->r = 255;
textColor->g = textColor->b = 0;

// CAUTION! WinSetTextColoRGB() does not affect
// GcDrawTextAt().
WinSetTextColorRGB(textColor, oldTextColor);
gc = GetCurrentContext();
if (gc) {
   GcDrawTextAt(gc, 200, 200, "I am not red.", 15);
}
```

In particular, be careful not to confuse `WinSetCoordinateSystem()` and `GcSetCoordinateSystem()`. If you call `GcSetCoordinateSystem()`, it changes only the coordinate system used by the graphics context, not by the Window Manager. It is a good idea to always make both calls to be sure that the window's coordinate system is the same coordinate system used by the graphics context.

Note that it is perfectly acceptable to call Window Manager functions that do not set or depend on the draw state; for example, Window Manager functions that convert a set of coordinates from one system to another, such as `WinConvertRectangle()`, are acceptable.

Making Library Calls

Sometimes, you may unintentionally mix the two drawing systems. For example, if you make a call to a library, you might not know how that library draws to the screen or even if it does. If you need to make a library call while you have the graphics context, preserve your state before you do so, and pop the state upon return. See Listing 8.5.

Listing 8.5 Making a library call

```
GcHandle gc = GetCurrentContext();
if (gc) {
   GcSetColor(gc, 50, 50, 50, 255);
   GcSetCoordinates(gc, kCoordinatesStandard);
   GcRect(gc, 10, 10, 10, 10);
   GcPushState(gc); // preserves path and state.
   // Call to library goes here.
   GcPopState(gc);
   // more drawing
   GcReleaseContext(gc);
}
```

If you are calling Palm OS user interface functions, it is also a good idea to preserve the state before the call and pop it after the call. Some Palm OS functions change the graphics context state, in particular the coordinate system.

Color Table Compatibility

In earlier releases of Palm OS, the system supported screens that had **color tables** or **palettes**. Each entry in the color table specified a color that the screen could display.

All Palm OS Cobalt displays are direct color displays. **Direct color displays** do not rely on a color lookup table because the value stored into each pixel location specifies the amount of red, green, and blue components directly. For example, a 16-bit direct color display could have 5 bits of each pixel assigned as the red component, 6 bits as the green component, and 5 bits as the blue component. With this type of display, the application is no longer limited to drawing with a color that is in the color lookup table.

When the screen is a direct color display, the color lookup table for the screen is present only for compatibility with the old indexed mode color calls. The lookup table has no effect on the display hardware, since the hardware derives the color from the red, green, and blue bits stored in each pixel location of the frame buffer.

You can still change the color table used by the current draw windows with the `WinPalette()` function. However, `WinPalette()` no longer affects what is already drawn onto the screen. It only affects future drawing.

Drawing Tips

This section provides the following tips for efficient use of the Graphics Context API:

Draw to Exact Pixel Boundaries

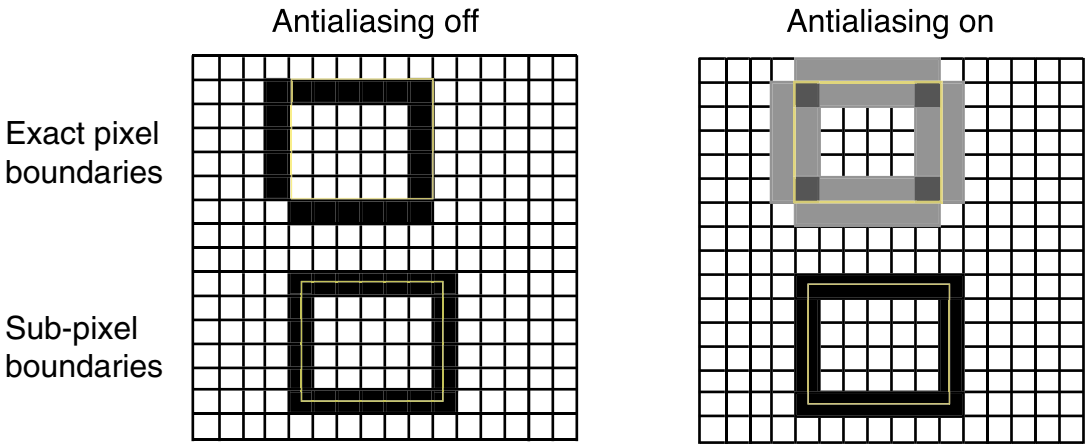
If you are drawing user interface items, your drawing will look better if it takes up entire pixels rather than partial pixels. Generally, using whole integer coordinates helps limit the number of partial pixels for filled paths. For stroked paths, you must take the pen width into account and make sure that the majority of the pen is on whole pixel boundaries. Remember that 50% of the pen width lies on one side of your path and 50% on the other side. If the pen width is odd (for example 1 pixel), then you should specify coordinates that end in 0.5. If the pen width is even, you can use whole integer coordinates.

Suppose you have the code shown in Listing 8.6 to draw a rectangle frame. This may result in a frame that is one pixel off from where you expect, as shown in the top two rectangles in Figure 8.4. The rendering system paints whichever pixels have their center points covered. The hardware determines which exact pixels are chosen, the ones on the left side of your path or the ones on the right. If antialiasing is turned on, the effect is even worse. None of the pixels are wholly part of the path, so they all get only a percentage of the color.

Listing 8.6 Drawing to exact pixel boundaries (incorrect)

```
GcHandle gc = GetCurrentContext();
if (gc) {
   GcSetColor(gc, 255, 255, 255, 255);
   GcSetPenSize(gc, 1.0);
   GcSetJoin(gc, kMiterJoin, 1);
   //BAD CODE! This may result in a rect one pixel off
   // from where you expect
   GcRect(gc, 4.0, 2.0, 10.0, 7.0);
   GcStroke(gc);
   GcPaint(gc);
   GcReleaseContext(gc);
}
```

Figure 8.4 Stroked paths with odd pen width



If you change your rectangle coordinates so that they all end in 0.5 as shown in Listing 8.7, the pen draws on exact pixel boundaries, and you'll get the desired rectangle regardless of the antialiasing setting, as shown in the bottom two rectangles of Figure 8.4.

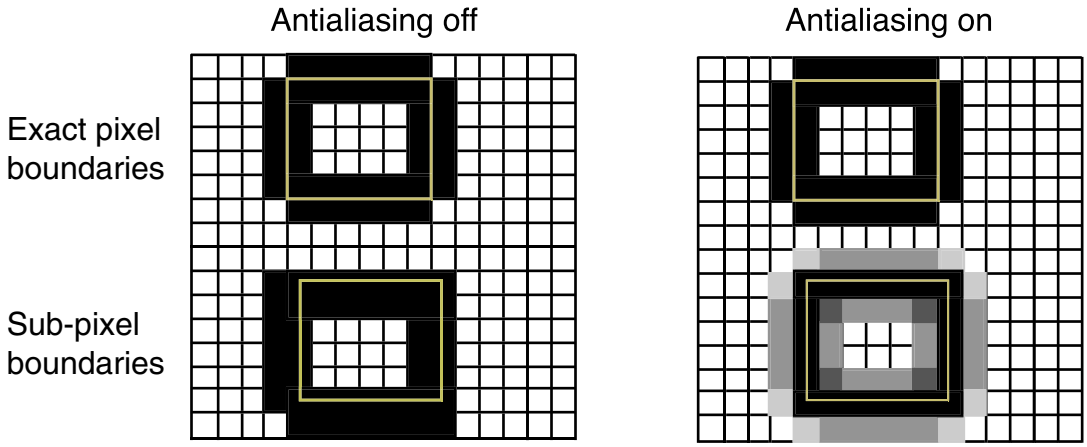
Listing 8.7 Drawing to exact pixel boundaries (correct)

```
GcHandle gc = GetCurrentContext();
if (gc) {
   GcSetColor(gc, 255, 255, 255, 255);
   GcSetPenSize(gc, 1.0);
   GcSetJoin(gc, kMiterJoin, 1);
   GcRect(gc, 4.5, 8.5, 10.5, 13.5);
```

```
   GcStroke(gc);
   GcPaint(gc);
   GcReleaseContext(gc);
}
```

If the pen width is even, however, it is better to specify whole integer coordinates. Then the system paints all of the pixels on the left side of your path and all of the pixels on the right side of your path as shown in the top two rectangles of Figure 8.5. When the pen width is even, the path looks bad when you specify sub-pixel coordinates.

Figure 8.5 Stroked paths with even pen width



Reuse the Current Path

The current path is stored as part of your drawing state and is cleared by `GcPaint()`. You can take advantage of this to perform multiple operations on this path without having to redefine it each time. For example, Listing 8.8 shows an efficient way to fill and then outline a rounded rectangle.

Listing 8.8 Reusing the path

```
void DrawOutlinedRoundRect(GcHandle ctxt)
{
   // First fill the rectangle, pushing it on the state
   // stack so we can recover it after painting.
   GcSetAntialiasing(ctxt, kEnableAntialiasing);
```

```
   GcSetColor(ctxt, 255, 0, 0, 255);
   GcRoundRect(ctxt, 0, 0, 10, 10, 3, 3);
   GcPushState(ctxt);
   // For even more efficient code, turn off antialiasing.
   GcSetAntialiasing(ctxt, kDisableAntialiasing);
   GcPaint(ctxt);

   // Now restore the previous rectangle, stroke to make
   // its outline and paint that path.
   GcPopState(ctxt);
   GcSetColor(0, 255, 0, 255);
   GcStroke(ctxt);
   GcPaint(ctxt);
}
```

Take Advantage of Winding Rule

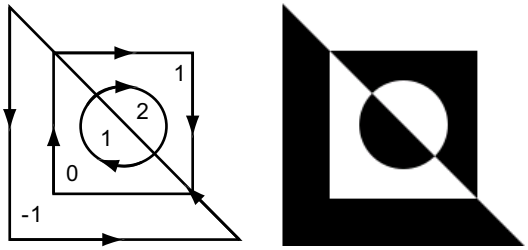
Palm OS Cobalt version 6.0 uses the odd winding rule. It is more efficient to take advantage of the winding rule than it is to define complex clipping regions. In addition, the winding rule supports antialiasing, and the clipping region does not.

The **winding rule** determines which points are part of the result when some parts of the path enclose other parts of it. Each point is assigned a **winding number** based on its location in relation to the the parts of the path. On Palm OS, you draw counterclockwise to decrease the winding number, and draw clockwise to increase it.

The odd winding rule specifies that only points with an odd winding number are considered part of the result. All other points are not part of the result.

Figure 8.6 shows a path that is affected by the winding rule. A triangle drawn counter clockwise is partially intersected by a square drawn clockwise, which wholly contains a a circle that is also drawn clockwise.

Figure 8.6 Winding rule effects

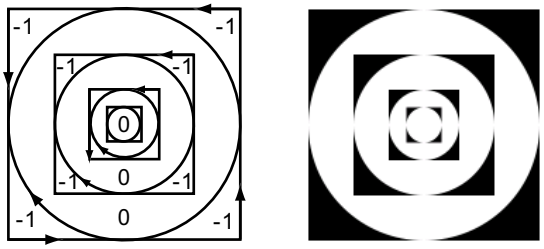


Winding numbers are determined starting from the top-left corner moving toward the bottom right. Because the triangle is drawn counterclockwise, it decrements the winding number so that points only within the triangle have a winding number of –1. Because the square is drawn clockwise, it increments the winding number. Therefore, the intersection of the square and the triangle has a winding number of 0 and points only within the square have a winding number of 1. Points within the circle are either 1 or 2.

The odd winding rule specifies that all points with an odd winding number are part of the result and should be painted. As you can see in Figure 8.6, those portions with a 0 or 2 winding number are not painted.

Figure 8.7 shows a path that is a series of squares and circles that enclose one another. All squares are drawn counterclockwise, and all circles are drawn clockwise. Because of this, all points that are *not* part of the intersection of a square and its immediately enclosed circle have a winding number of –1, but all points that are part of the the circle have a winding number of 0.

Figure 8.7 Winding rule effects



If you intend to take advantage of the winding rule to draw complex shapes, you should note the following:

- All `GcRect()` functions draw clockwise. If you want to draw a rectangle counterclockwise, use `GcLineTo()` to construct it.
- `GcArc()` always draws clockwise. There is currently no way to draw a counterclockwise arc.
- Stroked paths are drawn clockwise. If a stroked path intersects with itself, the intersection has an even winding number and is therefore not painted.
- In the discussions and figures above, the terms "clockwise" and "counterclockwise" are from your point of view *as you are drawing to or looking at the screen*. Mathematically speaking, drawing clockwise first increases the y value as you move from the origin and counterclockwise drawing decreases the y value. On Palm OS screens, the origin is in the top-left corner, and the y value increases downward. Therefore, drawing that looks counterclockwise to you is actually clockwise.

  If you're familiar with the OpenGL or PostScript languages, you may know that the winding number is increased on a counterclockwise path and decreased on a clockwise one. Palm OS follows this same rule if you use the mathematical definitions of clockwise and counterclockwise.

  To avoid confusion, you can do a reflection and translation transformation to move the origin to the lower-left corner of the screen before doing any drawing that is affected by the winding rule.

Avoid Antialiasing

Because antialiasing is a significant performance drain, you should disable it if you know you are not going to need it.

For example, Listing 8.8 on page 140 fills and then outlines a rectangle. It turns off antialiasing for the first path. The outline will cover the edges of the filled path, so there is no need to antialias the filled path.

Avoid Alpha Blending

To perform alpha blending, the rendering system must first read the value of the destination pixels and then blend with the requested color. This can be expensive. Use opaque figures where possible. Note that antialiasing generally uses alpha blending.

Efficient Cap and Join Modes

You should use `kBevelJoin` and `kButtCap` when drawing lines for which you don't have a particular preference about the join and cap styles. When using these styles, the system can optimize the stroking of simple figures to avoid full stroking calculations.

Avoid Transformations

Simple translation transformations are significantly more efficient than general transformations; scaling (with or without translation) transformations are not as efficient but still allow many internal optimizations to occur. In general, the more complicated the transformation (translation being the least complicated, then scaling, and rotation being the most complicated), the more the system will have to go through the full general drawing model instead of being able to use a more optimized code path.

Avoid Flushing the Buffer

As stated in "Graphics Context" on page 124, the rendering system runs in a different process than the application. To reduce IPC overhead, drawing operations are buffered and sent to the rendering system in batches. The rendering system sends its buffered commands at the end of each event loop.

The functions `GcCommit()` and `GcFlush()` break this buffering, causing all currently buffered commands to be sent immediately. In addition, `GcCommit()` waits for those commands to finish drawing. `GcCommit()` and `GcFlush()` can be extremely slow, so it is best to avoid calling them.

Most applications should flush the buffer only during debugging. Remove these calls before you release the application.

You might also flush the buffer if you know that your application is about to perform a lengthy operation that might delay it from reaching the end of the event loop.

Remember that all drawing to off-screen windows or to bitmaps is performed immediately, so there is no need to call `GcCommit()` and `GcFlush()` in those cases.

Summary of Drawing Functions

Graphic Context Functions

Obtaining a context

`GcGetCurrentContext()`
`GcCreateBitmapContext()`
`GcReleaseContext()`

Setting the State

`GcPopState()`
`GcPushState()`
`GcSetAntialiasing()`
`GcInitGradient()`
`GcSetCaps()`
`GcSetGradient()`
`GcSetCoordinateSystem()`
`GcSetColor()`
`GcSetJoin()`
`GcSetFont()`
`GcSetPenSize()`

Path Construction

`GcArc()`
`GcArcTo()`
`GcBezierTo()`
`GcClosePath()`
`GcLineTo()`
`GcMoveTo()`
`GcRect()`
`GcRoundRect()`
`GcStroke()`
`GcRectI()`

Painting the path

`GcPaint()`
`GcDrawTextAt()`

Painting bitmaps

`GcDrawBitmapAt()`
`GcPaintBitmap()`
`GcDrawRawBitmapAt()`
`GcReleaseBitmap()`

Graphic Context Functions	
Transformations	
GcReflect()	GcRotate()
GcScale()	GcTranslate()
GcTransform()	GcShear()
Setting the Clipping Rectangle	
GcBeginClip()	GcEndClip()
Flushing the buffer	
GcCommit()	GcFlush()

9

Working with Bitmaps

A **bitmap** is a graphic displayed by Palm OS®. You create a bitmap resource using your resource editor, or you can create a bitmap programmatically.

This chapter covers:

Before you read this chapter, it is strongly suggested that you be familiar with the concepts introduced in Chapter 1, "The Display," on page 3 and Chapter 8, "Drawing," on page 123.

Bitmap Format

Programmatically, a bitmap or bitmap family is represented by a `BitmapType` structure. This structure is simply a header. It is followed by the bitmap data in the same memory block. Bitmaps are also allowed to have their own color tables. When a bitmap has its own color table, it is stored between the bitmap header and the bitmap data.

This section describes how the `BitmapType` stores bitmap data. See "How Palm OS Displays Bitmaps" on page 154 to learn how Palm OS uses this bitmap data to determine what to display on the screen.

Versions of Bitmap Support

The `BitmapType` structure contains one of four versions of bitmap encodings. (The `version` field specifies which one is used.)

- Version 0 encoding is supported by all Palm OS releases. This encoding supports monochrome bitmaps only.
- Version 1 encoding is supported on Palm OS 3.0 and later. This encoding supports 1, 2, and 4-bit grayscale bitmaps and adds support for **bitmap families**.
- Version 2 encoding is supported on Palm OS 3.5 and later. This encoding supports 1, 2, 4, 8 and 16-bit color bitmaps, transparency indices, and RLE compression.

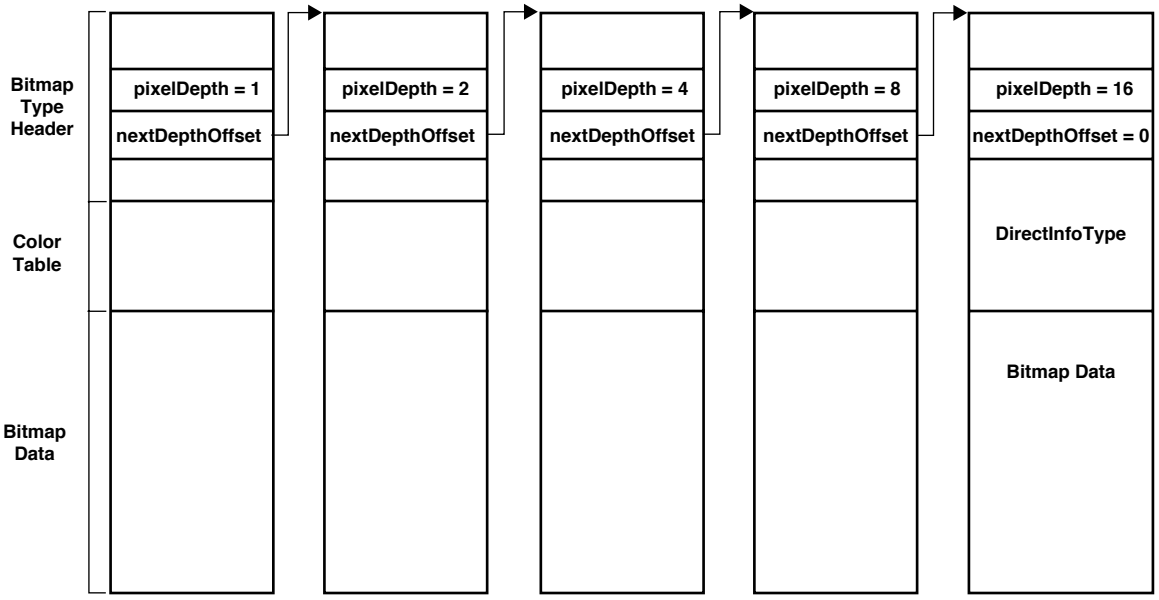
  With a version 2 bitmap, you can specify one index value as a transparent color at creation time. The transparency index is an alternative to masking. The system does not draw bits that have the transparency index value.

  When a bitmap with a transparency index is rendered at a depth other than the one at which it was created, the transparent color is first translated to the corresponding depth color, and the resulting color is named transparent. This may result in a group of colors becoming transparent.
- Version 3 encoding is supported on Palm OS Garnet and later devices with high-density displays. Version 3 adds support for bitmaps of varying densities.

Bitmap Families

A `BitmapType` structure represents either a single bitmap or a bitmap family. A **bitmap family** is a group of bitmaps, each containing the same drawing but at a different bit depth (sometimes called the pixel depth) or density. Figure 9.1 shows a bitmap family for a single display density.

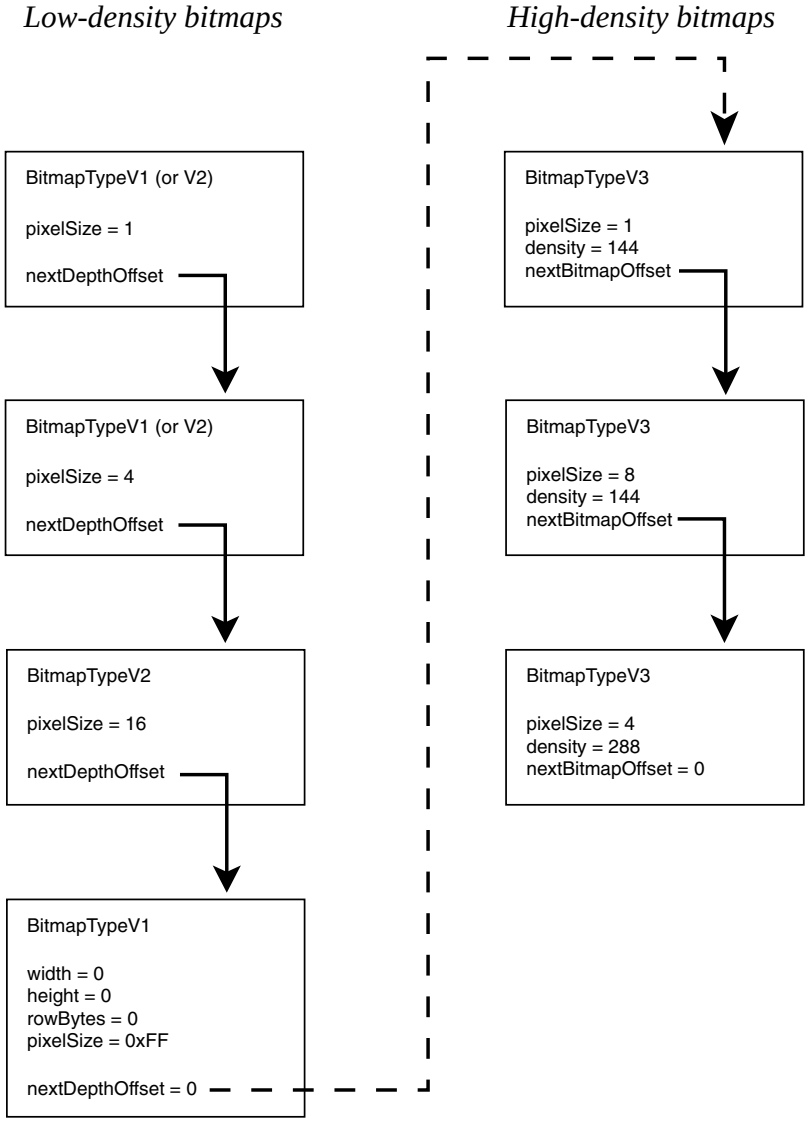
Figure 9.1 Single-density bitmap family



Bitmaps in a bitmap family are grouped by density. For backward compatibility, the linked list of low density bitmaps occur first, and remain ordered from low to high bit depths. If the family contains high-density bitmaps, the high-density bitmaps follow the low-density bitmaps, again ordered from low to high bit depths. If the family contains multiple densities, then the density sets are ordered from low to high density.

Figure 9.2 illustrates the process of traversing the bitmaps in a bitmap family.

Figure 9.2 Linked list of bitmaps in a bitmap family



Color Tables and Bitmaps

As mentioned previously, bitmaps can have their own color tables attached to them. If a bitmap has its own color table, the system must create a conversion table to convert the draw window before it can draw the bitmap. This conversion is a drain on performance, so using custom color tables with bitmaps is strongly discouraged.

You can still change the color table used by the current draw windows with the `WinPalette()` function before drawing the bitmap and change it back to the default afterwards. Note that `WinPalette()` no longer affects what is already drawn onto the screen. It only affects future drawing.

PNG Files

In addition to the bitmap format described in this section, Palm OS supports the display of PNG formatted files. To add a PNG file to your resource, use `RAW_RESOURCE` as shown in Listing 9.1.

Listing 9.1 Creating a PNG resource

```
<RAW_RESOURCE RESOURCE_ID="1000" COMMENT="PNG file">
   <RES_TYPE>'pngf'</RES_TYPE>
   <DATA_FILE>"./MyPictures/Picture1.png"</DATA_FILE>
</RAW_RESOURCE>
```

To display the PNG on the screen, you must create a `GcBitmapHandle` for the resource using `FrmGetBitmapHandle()` or `GcLoadBitmap()`. This is described further in the next section. Note that you must create a `GcBitmapHandle`; the `BitmapType` structures do not accept PNG files.

Displaying a Bitmap on the Screen

There are two ways to statically display a bitmap on a screen:

- If the bitmap should always be displayed at a fixed location, in your resource file create a form bitmap resource in addition to the bitmap. The **form bitmap** specifies where the bitmap should be displayed.
- If the bitmap might move or you do not know the location where it should be displayed until runtime, use the `GcDrawBitmapAt()` function to display it. Because this is a graphics context function, as described in Chapter 8, "Drawing," on page 123, you need to first obtain access to the graphics context. See Listing 9.2.

Listing 9.2 Drawing a bitmap

```
GcHandle gc;
FormType *formP;
GcBitmapHandle gcBitmap;
...

//Obtain the graphics context and draw the bitmap.
while (/* some condition */) {
   //Load the bitmap from the resource file and convert it.
   gcBitmap = FrmGetBitmapHandle(formP, myAppDB, bitmapRsc,
      rscID, 0);
   if (!gcBitmap)
      return memErrNotEnoughSpace;

   GcGetCurrentContext(gc);
   if (gc) {
      GcDrawBitmapAt(gc, gcBitmap, NULL, x, y);
      GcReleaseContext(gc);
   }
}
```

Before you can call `GcDrawBitmapAt()`, you must obtain a `GcBitmapHandle` for the bitmap. In most cases, you can do so using `FrmGetBitmapHandle()`, which loads the specified bitmap resource from the database, locks it, and converts it into a format that can be displayed on the screen more efficiently. For example, it converts big-endian bitmaps to little-endian format. This function also caches the bitmap handle and returns the cached handle on subsequent calls, ensuring that the bitmap is only loaded and converted once. The memory associated with the bitmap cache is freed when the form is freed.

If you're performing animation, doing a slide show, or otherwise drawing a lot of bitmaps to the form, don't use `FrmGetBitmapHandle()`. If you do, you may quickly fill the bitmap cache. Instead, load the bitmap (or PNG) from the resource file, and create the `GcBitmapHandle` using `GcLoadBitmap()`. In this case when you are finished with the bitmap, you'll need to call `GcReleaseBitmap()` to remove the memory associated with the `GcBitmapHandle` and then free the bitmap.

Alternatively, you can use `GcDrawRawBitmapAt()`, which converts a `BitmapType` to a `GcBitmapHandle`, draws the bitmap

on the screen, and then removes the memory associated with the `GcBitmapHandle`. You are responsible for loading the bitmap from the resource and releasing it when you are finished.

Creating a Bitmap Programmatically

If you want to create or modify a bitmap programmatically, do the following:

1. Use `BmpCreate()` to create a `BitmapType` structure.
2. Pass the `BitmapType` to `GcCreateBitmapContext()`. Doing so creates a graphics context representing the bitmap.
3. Draw to the graphics context returned from the previous step using the Graphics Context path definition and drawing functions described in Chapter 8, "Drawing," on page 123.
4. Release the graphics context when you are finished (`GcReleaseContext()`).

Listing 9.3 Programmatically creating a bitmap

```
BitmapType *bmpP;
GcHandle gc;

bmpP = BmpCreate(10, 10, 8, NULL, &error);
if (bmpP) {
  gc = GcCreateBitmapContext(bmpP);
  if (gc) {
      GcSetColor(gc, 0, 0, 0, 255);
      GcRoundRect(gc, 0, 0, 50, 15, 3, 3);
      GcPaint(gc);
      /* etc */
      GcReleaseContext(gc);
  }
}
```

A few items of note about programmatically creating a bitmap:

- `BmpCreate()` always creates a version 2 bitmap, which is a low-density bitmap. To create a high-density bitmap, you need to call `BmpCreateBitmapV3()` after calling `BmpCreate()`.

- Even if you are creating a low-density bitmap, the coordinates that you pass to the graphics context drawing functions are interpreted as native coordinates by default. If you prefer to work in standard coordinates, call `GcSetCoordinateSystem()`.

How Palm OS Displays Bitmaps

This section describes how Palm OS draws bitmaps. It covers how Palm OS chooses a bitmap from the bitmap family and how it deals with differences between the bitmap's density and the draw window's density.

Displaying Bitmaps from a Bitmap Family

The algorithm that is used to determine which bitmap to display depends upon the density of the draw window. The **draw window** is not always the screen. It can be an off-screen window created with the `WinCreateOffscreenWindow()` function or a bitmap context created with `GcCreateBitmapContext()`.

If the draw window is single density, single-density bitmaps are always favored over double-density bitmaps, regardless of source bitmap depth. If the draw window is double density, however, the color domain match (color vs. grayscale) is favored over a double-density bitmap with a color domain mismatch. Listing 9.4 shows the algorithm used to determine which bitmap to display.

Listing 9.4 Algorithm for choosing which bitmap to display

```
If draw window is low density {
    Favor low-density over double-density
    If draw window is color {
        Favor color bitmap
    } else {
        Favor grayscale, picking greatest depth
          less than or equal to draw window's
          depth
    }
} else {
        If draw window is color {
            Favor color
```

```
        } else {
            Favor grayscale
        }
    }
}
```

Table 9.1 provides the results of applying this algorithm. The two left columns represent the draw window's depth and density. The third column lists the bitmap selection preferences, ordered from best to worst (a 'd' in this third column indicates double-density).

Table 9.1 Double-density algorithm results

Draw Window		Bitmap selection preferences
Depth	**Density**	
1	Single	1, 2, 4, 8, 16
2	Single	2, 1, 4, 8, 16
4	Single	4, 2, 1, 8, 16
8	Single	8, 16, 4, 2, 1
16	Single	16, 8, 4, 2, 1
1	Double	1d, 2d, 4d, 1, 2, 4, 8d, 16d, 8, 16
2	Double	2d, 1d, 4d, 2, 1, 4, 8d, 16d, 8, 16
4	Double	4d, 2d, 1d, 4, 2, 1, 8d, 16d, 8, 16
8	Double	8d, 16d, 8, 16, 4d, 2d, 1d, 4, 2, 1
16	Double	16d, 8d, 16, 8, 4d, 2d, 1d, 4, 2, 1

Drawing High-Density Bitmaps

The rendering system uses the `density` field in the source and destination bitmaps (where the destination bitmap is the bitmap representing the draw window) to determine an appropriate scaling factor. Because standard density bitmaps must be scaled for high-density displays, some handhelds with high-density screens may use graphic accelerators. Nevertheless, the software rendering

system incorporates pixel-scaling logic for when the destination is an off-screen window.

The coordinate system affects the placement and dimensions of graphic primitives. It does not affect bitmap contents, however. You can create bitmaps that contain either low- or high-density bitmap data. Palm OS uses the coordinate system to determine where to place the top left corner of the bitmap on the screen, while the rendering system uses the bitmap structure's `density` field to determine if it needs to stretch or shrink the bitmap data as it blits.

When scaling down from a density of `kDensityDouble` to `kDensityLow`, the software shrinks the bitmap data. The result is almost always a poorer quality image when compared with a bitmap originally generated with a density of `kDensityLow`.

The following examples demonstrate the above concepts.

- *An application draws a low-density bitmap to a double-density screen.*

  The source data is a 16 by 16 bitmap. The application calls

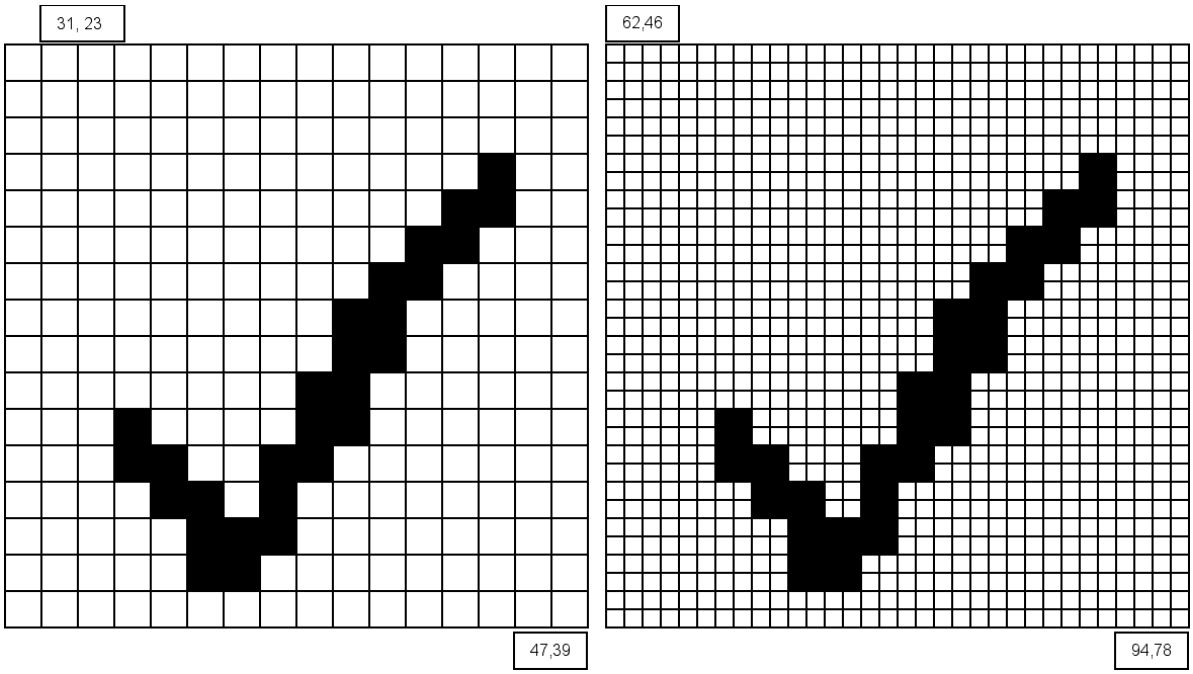
```
GcSetCoordinateSystem(gc, kCoordinatesStandard);
GcDrawBitmapAt(gc, bitmapH, NULL, 31, 23);
```

  with the intention of placing the bitmap on the screen beginning at standard coordinates (31, 23). Because the device has a double-density screen, the scaling factor is 2.0. The coordinates are converted to native coordinates (62, 46).

  The rendering system recognizes the bitmap as low density, based upon the version of its `BitmapType` structure, and pixel-doubles the source data when blitting to the double-density screen.

  Figure 9.3 shows the source data on the left, with low-density window coordinates for the top-left and bottom-right corners. The illustration on the right shows the result as displayed on the screen, with top-left coordinates scaled and bitmap data pixel-doubled.

Figure 9.3 Low-density bitmap on a double-density screen



Note that it's not necessary to use standard coordinates to display low-density bitmaps. The coordinate system only determines *where* the bitmap is displayed, not *how* the bitmap is displayed. The advantage to using standard coordinates is that you only have to compute one set of coordinates for all supported displays.

- *An application draws a double-density bitmap to a double-density screen.*

  The source data is a 32 by 32 double-density bitmap:

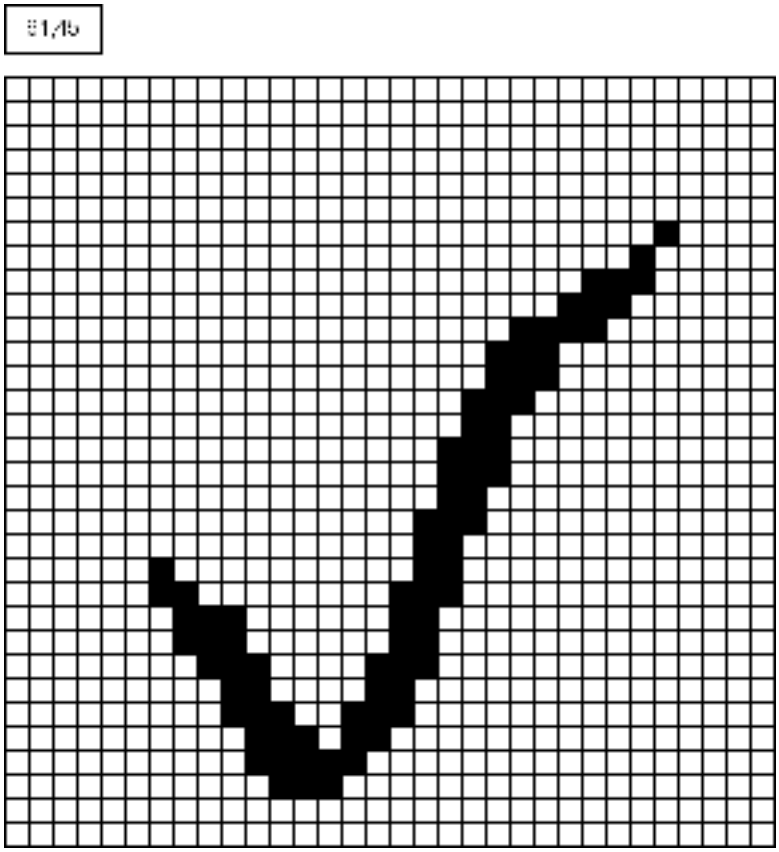
```
GcDrawBitmapAt(gc, bitmapH, NULL, 61, 45);
```

By default, the Graphics Context functions use native coordinates, which in this case are double-density coordinates. The double-density coordinates (61, 45) allow the application to position the bitmap more precisely on the

screen; these coordinates are equivalent to coordinates (30.5, 22.5) in the standard coordinate system.

Because the screen bitmap has the same density, the rendering system copies the source data to the screen unchanged.

Figure 9.4 Double-density bitmap on a double-density screen



- *An application draws a double-density bitmap to a low-density screen.*

  If an application includes only high-density bitmaps, the rendering system needs to shrink them when drawing them

to the screen. The application can determine the screen density like this:

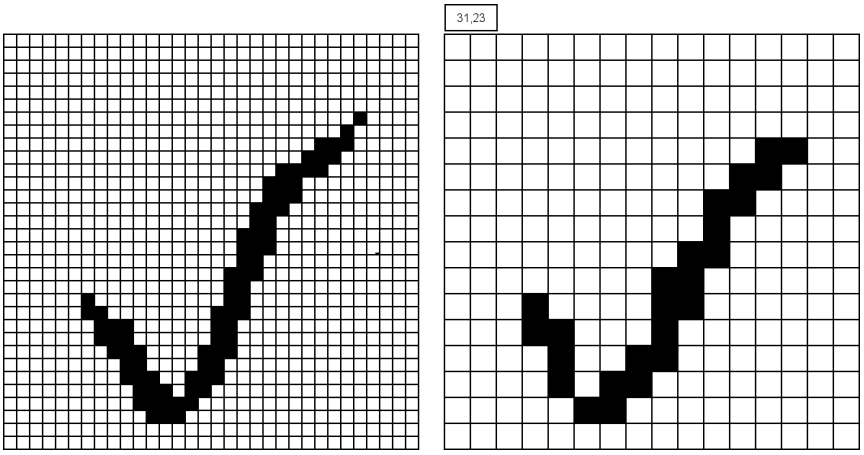
```
uint32_t density;
err = WinScreenGetAttribute(winScreenDensity, &density);
```

Understanding that the destination is low density, the application uses the standard coordinate system:

```
GcSetCoordinateSystem(gc, kCoordinatesStandard);
GcDrawBitmapAt(gc, bitmapH, NULL, 31, 23);
```

Because the destination window is low density, and because the passed coordinates are standard coordinates, Palm OS does not scale the passed coordinates. The rendering system, however, recognizes that the source bitmap has a density of `kDensityDouble` and shrinks the data to one-half the original size when blitting it to the low-density screen.

Figure 9.5 Double-density bitmap on a low-density screen



The result, shown in Figure 9.5 on the right, is poor. Because of this, for an application to look good on both low and high-density screens it should include both low and high-density bitmaps.

Summary of Bitmap Support

Bitmap Functions	
Obtaining bitmap data	
BmpGetBits()	
Working with bitmap families	
BmpGetNextBitmap()	BmpGetNextBitmapAnyDensity()
Creating and deleting bitmap	
BmpCreate() BmpDelete()	BmpCreateBitmapV3()
Obtaining size information	
GcGetBitmapDensity() GcGetBitmapDepth() GcGetBitmapHeight() GcGetBitmapWidth()	BmpGetSizes() BmpSize() BmpGetDimensions()
Obtaining attributes	
BmpGetVersion() BmpGetTransparentValue() BmpGetBitDepth() BmpGetPixelFormat()	BmpGetCompressionType() BmpGetDensity() GcIsBitmapAlphaOnly()
Setting attributes	
BmpSetDensity()	BmpSetTransparentValue()
Compressing bitmaps	
BmpCompress()	
Obtaining color table information	
BmpColortableSize() ColorTableEntries()	BmpGetColortable()

Bitmap Functions	
Drawing bitmaps	
GcCreateBitmapContext()	GcDrawBitmapAt()
GcDrawBitmapAt()	GcReleaseBitmap()
GcReleaseContext()	GcPaintBitmap()

10

Modifying the UI Color List

The UI color list contains the colors used by the various user interface elements. Each UI color is represented by a symbolic color constant. See Table 10.1 for a list of colors used.

Each bit depth has its own list of UI colors, allowing for a different color scheme in monochrome, grayscale, and color modes. This is important because even with a default monochrome look and feel, highlighted field text is black-on-yellow in color and white-on-black in other modes.

To obtain the color list, the system first tries to load it from the synchronized preferences database using the value `sysResIDPrefUIColorTableBase` plus the current screen depth. The use of a preference allows for the possibility that individual users could customize the look using a third party "personality" or "themes" editor. If the preference is not defined, it loads the default color table from the system color table resource, using the current screen depth to look up the appropriate table.

Using a list allows easy variation of the colors of UI elements to either personalize the overall color scheme of a given Palm Powered™ device or to adjust it within an application. Defining these as color classes ensures that the user interface elements are consistent with each other.

Table 10.1 UI elements and colors

UI Object	Symbolic Colors Used
Forms	`UIFormFrame`, `UIFormFill`
Modal dialogs	`UIDialogFrame`, `UIDialogFill`
Alert dialogs	`UIAlertFrame`, `UIAlertFill`

Table 10.1 UI elements and colors *(continued)*

UI Object	Symbolic Colors Used
Buttons (push button, repeating button, check boxes, and selector triggers)	`UIObjectFrame`, `UIObjectFill`, `UIObjectForeground`, `UIObjectSelectedFill`, `UIObjectSelectedForeground`
Fields	`UIFieldBackground`, `UIFieldText`, `UIFieldTextLines`, `UIFieldTextHighlightBackground`, `UIFieldTextHighlightForeground`
Menus	`UIMenuFrame`, `UIMenuFill`, `UIMenuForeground`, `UIMenuSelectedFill`, `UIMenuSelectedForeground`
Tables	Uses `UIFieldBackground` for the background, other colors controlled by the UI element in the table cell.
Lists and pop-up triggers	`UIObjectFrame`, `UIObjectFill`, `UIObjectForeground`, `UIObjectSelectedFill`, `UIObjectSelectedForeground`
Labels	Labels on a control and noneditable fields use `UIObjectForeground`, text written to a form using `WinDrawChars()` or `WinPaintChars()` uses the current text setting in the Window Manager draw state, text written to a form using `GcDrawTextAt()` uses the current color in the graphics context render state.
Scroll bars	`UIObjectFill`,`UIObjectForeground`, `UIObjectSelectedFill`, `UIObjectSelectedForeground`

Table 10.1 UI elements and colors *(continued)*

UI Object	Symbolic Colors Used
Insertion point	`UIFieldCaret`
Front-end processor (currently only used on Japanese systems)	`UIFieldFepRawText`, `UIFieldFepRawBackground`, `UIFieldFepConvertedText`, `UIFieldFepConvertedBackground`, `UIFieldFepUnderline`

Should your application need to change the colors used by the UI color list, it can do so with `UIColorSetTableEntry()`. If you need to retrieve a color used, it can do so with `UIColorGetTableEntryIndex()` or `UIColorGetTableEntryRGB()`.

If you change the UI color list, your changes are in effect only while your application is active. The UI color list is reset as soon as control switches to another application. When control switches back to your application, you'll have to call `UIColorSetTableEntry()` again.

Summary of UI Color Functions

UI Color List Functions	
UIColorGetTableEntryIndex() UIColorSetTableEntry()	`UIColorGetTableEntryRGB()`

11

Integrating with the Application Launcher

The Application Launcher is the screen from which most applications are launched. Users navigate to the Launcher by tapping the Applications icon in the status bar. They then launch a specific application by tapping its icon.

To integrate well with the Application Launcher, you must provide application icons and a version string as described in the following sections. In rare cases, you may need to provide a default application category as well.

Icons in the Launcher

Applications installed on the Palm Powered™ device (resource databases of type `'appl'`) appear in the Application Launcher automatically. Specifically, the Launcher displays an application icon and an application name.

Your application needs to have two icons and an icon name:

- A large `APP_ICON_BITMAP_RESOURCE`, with an ID of 1000. This icon should be 22 X 22 standard coordinates.
- A smaller icon, also of type `APP_ICON_BITMAP_RESOURCE`, with an ID of 1001. This icon should be 15 X 9 standard coordinates.

- The application icon name (optional) is type APP_ICON_NAME_RESOURCE. It is used by the Launcher screen and in the Button Assignment preferences panel.

  The application icon name is technically optional, but if you want the name to appear with the icon in the Launcher's main view, you must supply it.

  Note: If you use an application icon name, make it short!

Application Version String

The Launcher displays a version string from each application's APP_VERSION_RESOURCE, ID 1000. This short string (usually 3 to 7 characters) is displayed in the Info dialog.

A version string should have the format:

major.minor.[*stage.change*]

where *major* is the major version number, *minor* is a minor version number, *stage* is a letter denoting a development stage (a for alpha b for beta or d for developer release) and *change* is the build number. Remove the *stage* and *change* numbers for the final release.

The Default Application Category

The Launcher divides applications into categories. You can store an application's category in an APP_DEFAULT_CATEGORY_RESOURCE with the ID 1000 in the resource file. The Launcher application installs your application into the specified category.

Most applications should *not* specify an APP_DEFAULT_CATEGORY_RESOURCE. By default, Launcher installs applications in the Unfiled category, and each user chooses where to file the application.

Only specify an APP_DEFAULT_CATEGORY_RESOURCE in these instances:

- Your application is intended for consumers and clearly belongs to one of the Launcher predefined categories (see Table 11.1).

  Always specify the Launcher predefined categories in US English in ASCII characters. Launcher provides the appropriate translations for localized ROMs.

- Your application is intended for a vertical market or you've created a suite of custom applications that work together to provide a complete custom solution.

  In this case, you might define an `APP_DEFAULT_CATEGORY_RESOURCE` with a custom category name. Launcher creates the category if it doesn't already exist in the Launcher database. When you're not identifying one of Launcher's predefined categories, you may identify the category in any language.

Table 11.1 Launcher predefined categories

Default Launcher Category	Description
Games	Any game.
Main	Applications that would be used on a daily basis, such as Date Book or Address Book.
System	Applications that control how the system behaves, such as the Preferences, HotSync®, and Security applications.
Utilities	Applications that help the user with system management.
Unfiled	The default category.

Do not treat the default application category as something analogous to the Microsoft Windows Start menu category. The Launcher uses a non-schema database, so the user is limited to 16 categories (including Unfiled). Obviously, that limit would be quickly reached if each application defines its own category. Only assign a default category where it is a clear benefit to your users.

Part II Reference

This part contains reference material for the UI Library. It covers:

12

Bitmap Reference

This chapter provides information about bitmaps by discussing these topics:

The header files `Bitmap.h` and `CmnBitmapTypes.h` declare the API that this chapter describes. For more information on bitmaps, see Chapter 9, "Working with Bitmaps," on page 147.

Bitmap Structures and Types

BitmapDirectInfoType Struct

Purpose Specifies how colors are stored in a `BitmapTypeV2` direct color bitmap.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct BitmapDirectInfoType {
   uint8_t redBits;
   uint8_t greenBits;
   uint8_t blueBits;
   uint8_t reserved;
   RGBColorType transparentColor;
} BitmapDirectInfoType
```

Fields

`redBits`
Number of bits used by the red component in each pixel.

`greenBits`
Number of bits used by the green component in each pixel.

`blueBits`
Number of bits used by the blue component in each pixel.

`reserved`
Must be zero. Reserved for future use.

`transparentColor`
Contains the red, green, and blue components of the transparent color.

Comments For **direct color bitmaps**, where each pixel is represented by an RGB triplet rather than a palette index, the `BitmapDirectInfoType` structure follows the color table if one is present, or immediately follows the `BitmapType` if a color table is not present. For direct color bitmaps, only 16 bits per pixel is supported, 5 bits for red, 6 bits for green, and 5 bits for blue.

WARNING! This structure is documented so that you can directly access the internals of your own bitmap resources. *Bitmaps created by Palm OS® are not guaranteed to adhere to this structure*; you cannot cast a direct color bitmap data pointer from a bitmap created by the Palm OS to this structure and expect to be able to correctly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

BitmapType Struct

Purpose Represents the portions of a bitmap that are common to all versions of the bitmap resource supported by Palm OS. The `BitmapPtr` type defines a pointer to a `BitmapType` structure.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct BitmapType {
   int16_t widthBE16;
   int16_t heightBE16;
   uint16_t rowBytesBE16;
   uint16_t flagsBE16;
   uint8_t pixelSize;
   uint8_t version;
} BitmapType;
typedef BitmapType *BitmapPtr
```

Fields

`widthBE16`
The width of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`heightBE16`
The height of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`rowBytesBE16`
The number of bytes stored for each row of the bitmap where `heightBE16` is the number of rows, given in big-endian format. Use `BmpGetDimensions()` to obtain the contents of this field.

`flagsBE16`
The bitmap's attributes, given in big-endian format. See Bitmap Flag Constants.

`pixelSize`
The pixel depth. Currently supported pixel depths are 1, 2, 4, 8, and 16-bit. You specify this value when you create the bitmap. Use `BmpGetBitDepth()` to access the contents of this field.

`version`
The version of bitmap encoding used. See "Bitmap Constants" on page 187. The value in this field determines the data structure to use when interpreting the fields following `version`: a value of `BitmapVersionZero` (0) corresponds to `BitmapTypeV0`, `BitmapVersionOne` (1) corresponds to `BitmapTypeV1`, `BitmapVersionTwo` (2) corresponds to `BitmapTypeV2`, and `BitmapVersionThree` (3) corresponds to `BitmapTypeV3`. Use `BmpGetVersion()` to obtain the contents of this field.

Comments

The `BitmapType` structure represents that which is common to all `BitmapTypeV`*x* structures (`BitmapTypeV0`, `BitmapTypeV1`, `BitmapTypeV2`, and `BitmapTypeV3`). The `BitmapType` structures define both the bitmaps representing the window display and bitmap resources that you create using a resource editor and load into your program.

Because `BitmapType` is merely a portion of the `BitmapTypeV`*x* structures, you should never do `sizeof(BitmapType)`.

WARNING! This structure is documented so that you can directly access the internals of your own bitmap resources. *Bitmaps created by Palm OS are not guaranteed to adhere to this structure*; you cannot cast a bitmap created by the Palm OS to this structure and expect to be able to correctly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

Note the following about the `BitmapType` structures:

- None of these fields contains the actual bitmap data. Instead, the bitmap data is stored immediately following the `BitmapTypeV`*x* (which one depends on the value of the `version` field) header structure. If the bitmap has its own color table, the color table is stored in between the header and the data. If the bitmap has a pixel size of 16, and the bitmap is `BitmapTypeV2`, the `BitmapDirectInfoType` structure is stored between the header and the data. You can retrieve a bitmap's data by passing its `BitmapType` structure to `BmpGetBits()`, and you can retrieve its color table with `BmpGetColortable()`.
- Unlike most other user interface structures, the `BitmapType` does not store the bitmap's location on the screen. The `FormBitmapType` or the internal window structure with which this bitmap is associated contains that information.
- A bitmap may be part of a bitmap family. A **bitmap family** is a group of bitmaps, each containing the same drawing but at a different pixel depth (see "Bitmap Families" on page 148). When requested to draw a bitmap family, the operating system chooses a member of the bitmap family based upon the bitmap density and pixel depth; see "Displaying Bitmaps from a Bitmap Family" on page 154 for the algorithm that Palm OS uses to determine which one to choose.

BitmapTypeV0 Struct

Purpose Structure corresponding to the version 0 encoding of a bitmap. Version 0 encoding is supported in Palm OS 1.0 and later.

Declared In CmnBitmapTypes.h

Prototype

```
typedef struct BitmapTypeV0 {
   int16_t widthBE16;
   int16_t heightBE16;
   uint16_t rowBytesBE16;
   uint16_t flagsBE16;
   uint16_t reservedBE16[4];
} BitmapTypeV0;
typedef BitmapTypeV0 *BitmapPtrV0
```

Fields

`widthBE16`
The width of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`heightBE16`
The height of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`rowBytesBE16`
The number of bytes, given in big-endian format, stored for each row of the bitmap where `heightBE16` is the number of rows. Use `BmpGetDimensions()` to access this field.

`flagsBE16`
The bitmap's attributes, given in big-endian format. See Bitmap Flag Constants. Only the `bmpFlagsCompressed` flag is defined for `BitmapTypeV0` structures.

`reservedBE16`
Reserved. These values are set to zero. Note that in the `BitmapTypeV0` structure, the `pixelSize` and `version` fields, defined in `BitmapType`, do not exist. They coincide with the `reserved` array, however, and this array was initialized to zero when the bitmap was created. The operating system recognizes that a `pixelSize` of zero means that the bitmap's depth is 1.

Comments Generally you work with pointers to `BitmapType` structures; if the structure's version is `BitmapVersionZero`, the structure is of type `BitmapTypeV0`.

WARNING! This structure is documented so that you can directly access the internals of your own bitmap resources. *Bitmaps created by Palm OS are not guaranteed to adhere to this structure*; you cannot cast a bitmap created by the Palm OS to this structure and expect to be able to directly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

BitmapTypeV1 Struct

Purpose Structure corresponding to the version 1 encoding of a bitmap. Version 1 encoding is supported in Palm OS 3.0 and later.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct BitmapTypeV1 {
   int16_t widthBE16;
   int16_t heightBE16;
   uint16_t rowBytesBE16;
   uint16_t flagsBE16;
   uint8_t pixelSize;
   uint8_t version;
   uint16_t nextDepthOffsetBE16;
   uint16_t reservedBE16[2];
} BitmapTypeV1;
typedef BitmapTypeV1 *BitmapPtrV1
```

Fields

`widthBE16`
The width of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`heightBE16`
The height of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`rowBytesBE16`
The number of bytes, given in big-endian format, stored for each row of the bitmap where `heightBE16` is the number of rows. Use `BmpGetDimensions()` to access this field.

`flagsBE16`
: The bitmap's attributes, given in big-endian format. See Bitmap Flag Constants. Only the `bmpFlagsCompressed` and `bmpFlagsHasColorTable` flags are defined for `BitmapTypeV1` structures.

`pixelSize`
: The pixel depth. Supported pixel depths are 1, 2, and 4-bit. You specify this value when you create the bitmap. Use `BmpGetBitDepth()` to obtain the contents of this field.

`version`
: The version of bitmap encoding used. This field has a value of `BitmapVersionOne` (1) for `BitmapTypeV1` structures. Use `BmpGetVersion()` to obtain the contents of this field.

`nextDepthOffsetBE16`
: For bitmap families, this field specifies the start of the next bitmap in the family and is given in big-endian format. The value it contains is the number of 4-byte words to the next `BitmapType` from the beginning of this one. If the bitmap is not part of a bitmap family or it is the last bitmap in the family, the `nextDepthOffsetBE16` is 0.

`reservedBE16`
: Reserved.

Comments Generally you work with pointers to `BitmapType` structures; if the structure's version is `BitmapVersionOne`, the structure is of type `BitmapTypeV1`.

WARNING! This structure is documented so that you can directly access the internals of your own bitmap resources. *Bitmaps created by Palm OS are not guaranteed to adhere to this structure*; you cannot cast a bitmap created by the Palm OS to this structure and expect to be able to directly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

BitmapTypeV2 Struct

Purpose Structure corresponding to the version 2 encoding of a bitmap. Version 2 encoding is supported in Palm OS 3.5 and later.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct BitmapTypeV2 {
   int16_t widthBE16;
   int16_t heightBE16;
   uint16_t rowBytesBE16;
   uint16_t flagsBE16;
   uint8_t pixelSize;
   uint8_t version;
   uint16_t nextDepthOffsetBE16;
   uint8_t transparentIndex;
   uint8_t compressionType;
   uint16_t reservedBE16;
} BitmapTypeV2;
typedef BitmapTypeV2 *BitmapPtrV2
```

Fields

`widthBE16`
The width of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`heightBE16`
The height of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`rowBytesBE16`
The number of bytes, given in big-endian format, stored for each row of the bitmap where `heightBE16` is the number of rows. Use `BmpGetDimensions()` to access this field.

`flagsBE16`
The bitmap's attributes, given in big-endian format. See Bitmap Flag Constants. Only the `bmpFlagsCompressed`, `bmpFlagsHasColorTable`, `bmpFlagsHasTransparency`, `bmpFlagsIndirect`, `bmpFlagsForScreen`, and `bmpFlagsDirectColor` flags are defined for `BitmapTypeV2` structures. Note that the `bmpFlagsIndirect and bmpFlagsForScreen` flags are system-only flags that are not used in user-created bitmap resources.

`pixelSize`
: The pixel depth. Supported pixel depths are 1, 2, 4, 8, and 16-bit. You specify this value when you create the bitmap. Use `BmpGetBitDepth()` to obtain the contents of this field.

`version`
: The version of bitmap encoding used. This field has a value of `BitmapVersionTwo` (2) for `BitmapTypeV2` structures. Use `BmpGetVersion()` to obtain the contents of this field.

`nextDepthOffsetBE16`
: For bitmap families, this field specifies the start of the next bitmap in the family and is given in big-endian format. The value it contains is the number of 4-byte words to the next `BitmapType` from the beginning of this one. If the bitmap is not part of a bitmap family or it is the last bitmap in the family, the `nextDepthOffsetBE16` is 0.

`transparentIndex`
: The color index for the transparent color. Only used for version 2 bitmaps and only when the `bmpFlagsHasTransparency` flag is set (see Bitmap Flag Constants). You specify this value when you create the bitmap using a resource editor, or you can set it programmatically with `BmpSetTransparentValue()`. To obtain the value of this field, call `BmpGetTransparentValue()`.

`compressionType`
: The compression type used. Only used when the `bmpFlagsCompressed` flag is set. See `BitmapCompressionType` for possible values. The `BmpCompress()` function sets this field, and the `BmpGetCompressionType()` function obtains its value.

`reservedBE16`
: Reserved.

Comments

Generally you work with pointers to `BitmapType` structures; if the structure's version is `BitmapVersionTwo` (2), the structure is of type `BitmapTypeV2`.

WARNING! This structure is documented so that you can directly access the internals of bitmaps that you create. *Bitmaps created by Palm OS are not guaranteed to adhere to this structure*; you cannot cast a bitmap created by the Palm OS to this structure and expect to be able to directly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

BitmapTypeV3 Struct

Purpose Structure corresponding to the version 3 encoding of a bitmap. Version 3 encoding is supported on high-density devices.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct BitmapTypeV3 {
   int16_t widthBE16;
   int16_t heightBE16;
   uint16_t rowBytesBE16;
   uint16_t flagsBE16;
   uint8_t pixelSize;
   uint8_t version;
   uint8_t size;
   uint8_t pixelFormat;
   uint8_t unused;
   uint8_t compressionType;
   uint16_t densityBE16;
   uint32_t transparentValueBE32;
   uint32_t nextBitmapOffsetBE32;
} BitmapTypeV3;
typedef BitmapTypeV3 *BitmapPtrV3
```

Fields

`widthBE16`
The width of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`heightBE16`
The height of the bitmap in pixels, given in big-endian format. You specify this value when you create the bitmap. Use `BmpGetDimensions()` to access this field.

`rowBytesBE16`
: The number of bytes, given in big-endian format, stored for each row of the bitmap where `heightBE16` is the number of rows. Use `BmpGetDimensions()` to access this field.

`flagsBE16`
: The bitmap's attributes, given in big-endian format. See Bitmap Flag Constants. Note that the `bmpFlagsIndirect`, `bmpFlagsForScreen`, and `bmpFlagsIndirectColorTable` flags are system-only fields that are not used in user-created bitmap resources.

`pixelSize`
: The pixel depth. Currently supported pixel depths are 1, 2, 4, 8, and 16-bit. You specify this value when you create the bitmap. Use `BmpGetBitDepth()` to obtain the value of this field.

`version`
: The version of bitmap encoding used. This field has a value of `BitmapVersionThree` (3) for `BitmapTypeV3` structures. Use `BmpGetVersion()` to obtain the value of this field.

`size`
: The size of this structure, in bytes. This field does not include the size of the color table or the size of the bitmap data. Use `BmpGetSizes()` to obtain the value of this field.

`pixelFormat`
: An enumerated constant representing the format of the pixel data. See `PixelFormatType`.

`unused`
: Not used.

`compressionType`
: The compression type used; 0 if the bitmap is not compressed. Only used when the `bmpFlagsCompressed` flag is set. See `BitmapCompressionType` for possible values. The `BmpCompress()` function sets this field, and the `BmpGetCompressionType()` function obtains its value.

`densityBE16`
: Value that determines how to stretch or shrink the bitmap data, given in big-endian format. For devices with low-density displays, this field is initialized to `kDensityLow`. For

devices with double-density displays, this field is initialized to `kDensityDouble`. See `DensityType` for the full set of values that this field can assume. Set this field with `BmpSetDensity()`, and obtain its value with `BmpGetDensity()`.

`transparentValueBE32`
If this structure represents a bitmap with a bit depth of 8 or less, this field contains the bitmap's transparent index. If the bitmap has a bit depth of 16, the 16-bit transparent RGB color is stored in this field in big-endian format. You specify this value when you create the bitmap using a resource editor, or you can set it programmatically with `BmpSetTransparentValue()` To obtain the value of this field, call `BmpGetTransparentValue()`.

`nextBitmapOffsetBE32`
A 32-bit big-endian value that indicates the number of bytes to the next bitmap in the family. If the bitmap is not part of a bitmap family or it is the last bitmap in the family, the `nextBitmapOffsetBE32` is 0.

Comments

Generally you work with pointers to `BitmapType` structures; if the structure's version is `BitmapVersionThree` (3), the structure is of type `BitmapTypeV3`.

`BmpCreate()` allocates and initializes a `BitmapTypeV2` structure. To create `BitmapTypeV3` bitmap use `BmpCreate` and pass the results to `BmpCreateBitmapV3()`.

In earlier versions of the `BitmapTypeV`*x* structure, the size of compressed bitmap data is stored in a 16-bit field preceding the bitmap data. With the version 3 structure, the size is stored in a 32-bit field.

The `BitmapTypeV3` structure has fields that identify how each pixel is stored (`pixelFormat`) and which color, if any, is "transparent" (`transparentValueBE32`). Because of this, you don't use a `BitmapDirectInfoType` structure in conjunction with a `BitmapTypeV3` structure.

WARNING! This structure is documented so that you can directly access the internals of bitmaps that you create. *Bitmaps created by Palm OS are not guaranteed to adhere to this structure*; you cannot cast a bitmap created by the Palm OS to this structure and expect to be able to directly access the structure's fields. Always use accessor functions to access the contents of user interface structures created by Palm OS.

ColorTableType Struct

Purpose

Defines a color table. Bitmaps can have color tables attached to them; however, doing so is not recommended for performance reasons.

Declared In

`CmnBitmapTypes.h`

Prototype

```
typedef struct ColorTableType {
   uint16_t entryCount;
} ColorTableType
```

Fields

`entryCount`
The number of entries in table. High bits (`entryCount` > 256) reserved.

Comments

The color table entries themselves are of type `RGBColorType`, and there is one per `entryCount`. Use the macro `ColorTableEntries()` to retrieve these entries.

Care should be taken not to confuse a full color table (which includes the count) with an array of RGB color values. Some routines operate on entire color tables; others operate on lists of color entries.

WARNING! PalmSource, Inc. does not support or provide backward compatibility for the `ColorTableType` structure. Never access its structure members directly, or your code may break in future versions. Use `BmpGetColortable()` to access this structure. Use the information above for debugging purposes only.

RGBColorType Struct

Purpose Defines a color. It is used as an entry in the color table. `RGBColorType` structures can also be created manually and passed to several user interface functions.

Declared In `CmnBitmapTypes.h`

Prototype

```
typedef struct RGBColorType {
   uint8_t index;
   uint8_t r;
   uint8_t g;
   uint8_t b;
} RGBColorType
```

Fields

`index`

The index of this color in the color table. Not all functions that use `RGBColorType` use the `index` field.

Indexed color bitmaps support no more than 256 colors. The number of possible RGB colors greatly exceeds this amount. For this reason, some drawing functions use a color look up table (CLUT). If the CLUT is used, the `index` field contains the index of an available color that is the closest match to the color specified by the `r`, `g`, and `b` fields.

`r`

Amount of red (0 to 255).

`g`

Amount of green (0 to 255).

`b`

Amount of blue (0 to 255).

See Also `GcColorType`

Bitmap Constants

Bitmap Version Constants

Purpose Specifies the version of encoding used for `BitmapType`.

Declared In `CmnBitmapTypes.h`

Constants

`#define BitmapVersionZero 0`
Uses the version 0 encoding of a bitmap.

`#define BitmapVersionOne 1`
Uses the version 1 encoding of a bitmap.

`#define BitmapVersionTwo 2`
Uses the version 2 encoding of a bitmap. Version 2 bitmaps either use the transparency index or are compressed. If you programmatically create a bitmap using `BmpCreate()`, a version 2 bitmap is created.

`#define BitmapVersionThree 3`
Uses the version 3 encoding of a bitmap. Version 3 bitmaps are supported only on high-density displays. You can programmatically create a version 3 bitmap using `BmpCreateBitmapV3()`.

Bitmap Flag Constants

Purpose Flags used in the `flagsBE16` field of the `BitmapType` structure.

Declared In `CmnBitmapTypes.h`

Constants

`#define bmpFlagsCompressed 0x8000`
If set, the bitmap is compressed and the `compressionType` field specifies the compression used. If not set, the bitmap is uncompressed. The `BmpCompress()` function sets this flag.

`#define bmpFlagsDirectColor 0x0400`
If set, the bitmap is a direct color (RGB) bitmap.

`#define bmpFlagsForScreen 0x0800`

If set, the bitmap is intended for the display (screen) window. This flag is never set.

Note that this flag is supported for bitmaps created by Palm OS only; this flag is not used in user-created bitmap resources.

`#define bmpFlagsHasColorTable 0x4000`

If set, the bitmap has its own color table. You specify whether the bitmap has its own color table when you create the bitmap.

`#define bmpFlagsHasTransparency 0x2000`

If set, the OS does not draw pixels that have a value equal to the transparency color. If not set, the bitmap has no transparency value. You specify the transparent color when you create the bitmap.

`#define bmpFlagsIndirect 0x1000`

If set, the address to the bitmap's data is stored where the bitmap itself would normally be stored. The actual bitmap data is stored elsewhere. If not set, the bitmap data is stored directly following the bitmap header or directly following the bitmap's color table if it has one.

Note that this flag is supported for bitmaps created by Palm OS only; this flag is not used in user-created bitmap resources.

`#define bmpFlagsIndirectColorTable 0x0200`

If set and if `bmpFlagsHasColorTable` is set, a pointer to the bitmap's color table immediately follows the `BitmapType` structure. If not set, and `bmpFlagsHasColorTable` is `true`, the color table immediately follows the `BitmapType` structure. If `bmpFlagsHasColorTable` is not set, this flag is ignored. The `bmpFlagsIndirect` bit uses similar logic: if both the color table and the bitmap data are indirect, the color table pointer precedes the bitmap data pointer.

Note that this flag is supported for bitmaps created by Palm OS only; this flag is not used in user-created bitmap resources.

`#define bmpFlagsNoDither 0x0100`
: If set, the rendering system does not dither the bitmap. If not set, the source bitmap is dithered if it has a bit depth greater than the destination bitmap.

BitmapCompressionType Typedef

Purpose Specifies possible bitmap compression types. These are the possible values for the `compressionType` field of `BitmapType`. You can compress or uncompress a bitmap using a call to `BmpCompress()`.

Declared In `CmnBitmapTypes.h`

Prototype
```
typedef Enum8 BitmapCompressionType
```

Constants

`BitmapCompressionTypeScanLine = 0`
: Use scan line compression.

`BitmapCompressionTypeRLE`
: Use RLE compression.

`BitmapCompressionTypePackBits`
: Use PackBits compression.

`BitmapCompressionTypeEnd`
: For internal use only.

`BitmapCompressionTypeBest = 0x64`
: For internal use only.

`BitmapCompressionTypeNone = 0xFF`
: No compression is used.

  This value should only be used as an argument to `BmpCompress()`.

DensityType Typedef

Purpose The density of the bitmap (see Chapter 1, "The Display," on page 3 for a definition of display density).

Declared In `CmnBitmapTypes.h`

Prototype
```
typedef Enum16 DensityType
```

Constants

`kDensityLow = 72`
Low (single) density. A low-density screen is 160 X 160 pixels.

`kDensityOneAndAHalf = 108`
"One and a half" density. A one-and-a-half-density display is 240 X 240 pixels; this would most likely be used on a handheld with a 240 X 320 screen where the bottom portion is used as a dynamic input area.

`kDensityDouble = 144`
Double density when compared with `kDensityLow`. A double-density screen is 320 X 320 pixels.

`kDensityTriple = 216`
Triple density when compared with `kDensityLow`. A triple-density screen is 480 X 480 pixels.

`kDensityQuadruple = 288`
Quadruple density when compared with `kDensityLow`. A quadruple-density screen is 640 X 640 pixels.

Comments

IMPORTANT: Not all densities listed in the `DensityType` enum are supported by a given version of Palm OS. For Palm OS Cobalt version 6.0, only `kDensityLow` and `kDensityDouble` are currently supported.

Density is only supported in `BitmapType` structures with a version greater than 2; if a given `BitmapType` structure is version 2 or lower, it is assumed to contain low-density data.

Palm OS uses the density field in the source and destination bitmaps to determine an appropriate scaling factor. When scaling down from a density of `kDensityDouble` to `kDensityLow`, Palm OS must shrink the bitmap data. This will almost always result in a poorer-quality image when compared with a bitmap that was created with a density of `kDensityLow`.

The `kDensityLow` value of 72 is arbitrary. Although this value doesn't necessarily represent pixels per inch, it is useful to think of it that way.

PixelFormatType Typedef

Purpose

Pixel formats defined for use with `BitmapTypeV3` structures.

Declared In `CmnBitmapTypes.h`

Prototype
```
typedef Enum8 PixelFormatType
```

Constants

`pixelFormatIndexed`
Each pixel is represented by a palette index.

`pixelFormat565`
Each pixel is represented by an RGB triplet stored in 16-bits: 5 red bits, 6 green bits, and 5 blue bits.

`pixelFormat565LE`
Similar to `pixelFormat565`, except that the 16 bits of the RGB triplet are stored as little-endian. This pixel format is not supported in user-created bitmaps.

`pixelFormatIndexedLE`
Similar to `pixelFormatIndexed`, except that the pixels within a byte are stored as little-endian.

`pixelFormat5551`
Each pixel is represented by an RGB plus alpha quadruplet stored in 16 bits: 5 red bits, 5 green bits, 5 blue bits, and one bit specifying if the color is opaque or not.

`pixelFormat4444`
Each pixel is represented by an RGB plus alpha quadruplet stored in 16 bits: 4 red bits, 4 green bits, 4 blue bits, and 4 bits of alpha, which specifies how transparent the color is.

Miscellaneous Bitmap Constants

Purpose Other constants defined in `CmnBitmapTypes.h`.

Declared In `CmnBitmapTypes.h`

Constants

```
#define colorTableHeaderSize 2
```
The size in bytes of the color table header. This is used when determining a bitmap's data size.

```
#define kTransparencyNone ((uint32_t)0xFFFFFFFF)
```
A value that specifies that the bitmap does not define a transparency color.

Bitmap Functions and Macros

BmpColortableSize Function

Purpose Returns the size of the bitmap's color table.

Declared In `Bitmap.h`

Prototype

```
uint16_t BmpColortableSize
    (const BitmapType *bitmapP)
```

Parameters

→ *bitmapP*
Pointer to the bitmap. (See BitmapType.)

Returns The size in bytes of the bitmap's color table or 0 if the bitmap does not use its own color table.

See Also BmpSize(), BmpGetColortable()

BmpCompress Function

Purpose Does nothing. Previously called to compress or uncompress a bitmap.

Declared In `Bitmap.h`

Prototype

```
status_t BmpCompress (BitmapType *bitmapP,
    BitmapCompressionType compType)
```

Parameters

→ *bitmapP*
Pointer to the bitmap to compress. (See BitmapType.)

→ *compType*
The type of compression to use. (See BitmapCompressionType.) If set to `BitmapCompressionTypeNone` and *bitmapP* is compressed, this function uncompresses the bitmap.

Returns Always returns `errNone`.

BmpCreate Function

Purpose Creates a bitmap.

Declared In `Bitmap.h`

Prototype

```
BitmapType *BmpCreate (Coord width, Coord height,
   uint8_t depth, ColorTableType *colortableP,
   status_t *error)
```

Parameters

→ *width*
The width of the bitmap in pixels. Must not be 0.

→ *height*
The height of the bitmap in pixels. Must not be 0.

→ *depth*
The pixel depth of the bitmap. Must be 1, 2, 4, 8, or 16. This value is used as the `pixelSize` field of `BitmapType`.

→ *colortableP*
A pointer to the color table associated with the bitmap, or `NULL` if the bitmap does not include a color table. If specified, the number of colors in the color table must match the *depth* parameter. (2 for 1-bit, 4 for 2-bit, 16 for 4-bit, and 256 for 8-bit). 16-bit bitmaps do not use any color table.

← *error*
Contains the error code if an error occurs.

Returns A pointer to the new bitmap structure (see `BitmapType`) or `NULL` if an error occurs. The parameter *error* contains one of the following:

`errNone`
Success.

`sysErrParamErr`
The *width*, *height*, *depth*, or *colortableP* parameter is invalid. See the descriptions above for acceptable values.

`sysErrNoFreeResource`
There is not enough memory available to allocate the structure.

Comments This function creates an uncompressed, non-transparent `BitmapVersionTwo` bitmap with the width, height, and depth that you specify. To create a `BitmapVersionThree` bitmap, use `BmpCreate` and pass the results to `BmpCreateBitmapV3()`.

If you pass a color table, the bitmap's `bmpFlagsHasColorTable` flag is set. For performance reasons, attaching a custom color table to a bitmap is strongly discouraged. An alternative is to use the `WinPalette()` function to change the color table as needed, draw the bitmap, and then undo your changes after you have finished displaying the bitmap.

`BmpCreate()` allocates sufficient memory to hold the bitmap and initializes all of its pixels to white. To change the bitmap's contents, use the drawing functions. See "Creating a Bitmap Programmatically" on page 153.

You cannot use this function to create a bitmap written directly to a database; that is, you must create the bitmap in memory first, then write it to the database.

See Also `BmpDelete()`

BmpCreateBitmapV3 Function

Purpose Creates a version 3 bitmap from an existing bitmap, an existing set of data bits, and, optionally, a color table.

Declared In `Bitmap.h`

Prototype

```
BitmapTypeV3 *BmpCreateBitmapV3
    (const BitmapType *bitmapP, uint16_t density,
    const void *bitsP,
    const ColorTableType *colorTableP)
```

Parameters

→ *bitmapP*
Pointer to a valid bitmap from which the version 3 bitmap is to be created. See `BitmapType`.

→ *density*
Density of the returned bitmap. If 0, the returned bitmap's density is set to the default value of `kDensityLow`.

→ *bitsP*
Pointer to the bitmap image data. Note that the bitmap data can be located in a database, but then the bitmap should be treated as read-only. You must use `DmWrite()` to write to the storage heap; blitting to it causes a system error.

→ *colorTableP*
Pointer to a color table, or `NULL` to use *bitmapP*'s color table, if one exists.

Returns A version 3 bitmap, or `NULL` if the bitmap could not be created from the specified bitmap, bitmap data, and optional color table.

Comments You can use this function when the bitmap data is stored in the storage heap as bands of raster data. Rather than allocating several bitmap structures, one for each band, use this function to allocate a single bitmap, and have the structure point to each band successively. This is typically used with high-density bitmaps that cannot be stored entirely within 64 KB.

WARNING! Due to a limitation in the way that this function is implemented, `BitmapCreateBitmapV3` doesn't work with compressed bitmaps. Don't pass bitmaps to this function that have the `bmpFlagsCompressed` flag set.

After your application is done with the returned bitmap structure, dispose of it by calling `BmpDelete()`.

See Also BmpCreate()

BmpDelete Function

Purpose Deletes a bitmap structure.

Declared In `Bitmap.h`

Prototype
```
status_t BmpDelete (BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to the structure of the bitmap to be deleted. (See `BitmapType`.)

Returns `errNone` upon success, or one of the following values:

`sysErrParamErr`
If the bitmap's `bmpFlagsForScreen` flag is set or the bitmap resides in the storage heap.

Returns one of the memory errors if the freeing pointer fails.

Comments Only delete bitmaps that have been created using `BmpCreate()`.

You cannot use this function on a bitmap located in a database. To delete a bitmap from a database, use the standard Data Manager calls.

BmpGetBitDepth Function

Purpose Retrieves the depth of a bitmap.

Declared In `Bitmap.h`

Prototype
```
uint8_t BmpGetBitDepth (const BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to a bitmap. See `BitmapType`.

Returns The bit depth of the bitmap, as represented by the `pixelSize` field in `BitmapType`. For debug ROMs, this function reports an error and returns 0 if *bitmapP* is `NULL`.

See Also BmpGetDimensions(), BmpGetNextBitmap(), BmpGetSizes()

BmpGetBits Function

Purpose Retrieves the bitmap's data.

Declared In `Bitmap.h`

Prototype
```
void *BmpGetBits (BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to the bitmap's structure. (See `BitmapType`.)

Returns A pointer to the bitmap's data.

Comments This function returns the bitmap's data even if the bitmap's `bmpFlagsIndirect` flag is set. (See Bitmap Flag Constants.)

See Also `BmpGetBits()`

BmpGetColortable Function

Purpose Retrieves the bitmap's color table.

Declared In `Bitmap.h`

Prototype
```
ColorTableType *BmpGetColortable
    (BitmapType *bitmapP)
```

Parameters → *bitmapP*
A pointer to the bitmap. See `BitmapType`.

Returns A pointer to the color table or `NULL` if the bitmap uses the system color table.

See Also `BmpColortableSize()`, `ColorTableEntries()`

BmpGetCompressionType Function

Purpose Gets the compression type of a bitmap.

Declared In `Bitmap.h`

Prototype
```
BitmapCompressionType BmpGetCompressionType
    (const BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to a valid bitmap. See `BitmapType`.

Returns The type of compression used by *bitmapP*. See `BitmapCompressionType` for the values that can be returned from this function. For debug ROMs, this function reports an error and returns `BitmapCompressionTypeNone` if *bitmapP* is `NULL`.

Comments If the bitmap is not compressed, this function returns `BitmapCompressionTypeNone`. If the bitmap version is 0 or 1 (corresponding to `BitmapTypeV0` and `BitmapTypeV1`, respectively), it returns `BitmapCompressionTypeScanLine`.

See Also BmpCompress()

BmpGetDensity Function

Purpose Gets the density of a bitmap.

Declared In `Bitmap.h`

Prototype
```
uint16_t BmpGetDensity (const BitmapType *bitmapP)
```

Parameters

→ *bitmapP*
Pointer to a valid bitmap. See BitmapType.

Returns The density of *bitmapP*; see DensityType for the defined set of density values. For debug ROMs, this function reports an error and returns 0 if *bitmapP* is NULL.

Comments Bitmaps with a version of 0, 1, or 2 (corresponding to BitmapTypeV0, BitmapTypeV1, and BitmapTypeV2, respectively) are assumed to be low density (kDensityLow).

See Also BmpCreateBitmapV3(), BmpSetDensity()

BmpGetDimensions Function

Purpose Retrieves the width, height, and number of data bytes per row of a bitmap.

Declared In `Bitmap.h`

Prototype
```
void BmpGetDimensions (const BitmapType *bitmapP,
   Coord *widthP, Coord *heightP,
   uint16_t *rowBytesP)
```

Parameters

→ *bitmapP*
Pointer to the bitmap. See BitmapType.

← *widthP*
Pointer to bitmap's width in pixels. Use NULL if this information is not wanted.

← *heightP*
Pointer to bitmap's height in pixels. Use NULL if this information is not wanted.

← *rowBytesP*
Pointer to number of bytes per row of bitmap. Use NULL if this information is not wanted.

Returns Nothing. This function reports an error on debug ROMs if *bitmapP* is NULL.

See Also BmpGetBitDepth(), BmpGetNextBitmap(), BmpGetSizes()

BmpGetNextBitmap Function

Purpose Retrieves the next low-density bitmap in a bitmap family.

Declared In Bitmap.h

Prototype

```
BitmapType *BmpGetNextBitmap (BitmapType *bitmapP)
```

Parameters

→ *bitmapP*
Pointer to a bitmap. See BitmapType.

Returns A pointer to the next low-density BitmapType in a bitmap family. It returns NULL if *bitmapP* is the last bitmap. For debug ROMs, this function reports an error and returns 0 if *bitmapP* is NULL.

Comments This function is only supported for low-density bitmap families. For families with both low and high-density bitmaps (BitmapTypeV3), use BmpGetNextBitmapAnyDensity().

In a BitmapTypeV3, a dummy low-density BitmapType precedes any high-density bitmaps. If you pass a pointer to the last valid low-density bitmap within a BitmapTypeV3 structure, BmpGetNextBitmap returns a pointer to the dummy bitmap. If you then pass that pointer to BmpGetDimensions(), it returns the dimensions for the first high-density bitmap rather than the dummy bitmap. To avoid confusion, it is best to use BmpGetNextBitmapAnyDensity when iterating through a family and gathering information about each bitmap.

See Also BmpGetBitDepth(), BmpGetDimensions(), BmpGetSizes()

BmpGetNextBitmapAnyDensity Function

Purpose Gets the next bitmap in the bitmap family, irrespective of density.

Declared In `Bitmap.h`

Prototype

```
BitmapType *BmpGetNextBitmapAnyDensity
    (BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to a valid bitmap. See `BitmapType`.

Returns The next bitmap in a bitmap family, or `NULL` if *bitmapP* is the last bitmap. For debug ROMs, this function reports an error and returns 0 if *bitmapP* is `NULL`.

Comments This function is an extended version of `BmpGetNextBitmap()`. For backward compatibility, `BmpGetNextBitmap()` only returns low-density bitmaps. If the bitmap family contains high-density bitmaps, however, `BmpGetNextBitmapAnyDensity()` skips over the dummy bitmap that separates the low and high-density bitmaps in the linked list and returns a high-density bitmap.

See Also BmpGetDensity(), BmpGetNextBitmap()

BmpGetPixelFormat Function

Purpose Returns the pixel format used by the bitmap.

Declared In `Bitmap.h`

Prototype

```
PixelFormatType BmpGetPixelFormat
    (const BitmapType *bitmapP)
```

Parameters → *bitmapP*
Pointer to the bitmap. See `BitmapType`.

Returns One of the `PixelFormatType` constants.

BmpGetSizes Function

Purpose Retrieves the size of a bitmap and its header structure.

Declared In `Bitmap.h`

Prototype

```
void BmpGetSizes (const BitmapType *bitmapP,
    uint32_t *dataSizeP, uint32_t *headerSizeP)
```

Parameters

→ *bitmapP*
Pointer to the bitmap. See `BitmapType`.

← *dataSizeP*
Pointer to size of bitmap data, not including structures. Use `NULL` if this information is not wanted.

← *headerSizeP*
Pointer to size of bitmap's structures, not including data. Use `NULL` if this information is not wanted.

Returns Nothing.

Comments This function returns the size in bytes of the bitmap data in *dataSizeP*. The size does not include the data structures (`BitmapType`, `BitmapDirectInfoType`, or color table) that are associated with a bitmap. The size of the structures (in bytes) are returned in *headerSizeP*, which includes the size of the `BitmapType`, `BitmapDirectInfoType` (if any), the color table (if any), and the size of the pointer for indirect bitmaps (described in Bitmap Flag Constants).

This function displays an error on debug ROMs if *bitmapP* is `NULL`.

See Also BmpGetBitDepth(), BmpGetDimensions(), BmpGetNextBitmap()

BmpGetTransparentValue Function

Purpose Gets a bitmap's transparent color.

Declared In `Bitmap.h`

Prototype

```
Boolean BmpGetTransparentValue
    (const BitmapType *bitmapP,
    uint32_t *transparentValueP)
```

Parameters

→ *bitmapP*
Pointer to a valid bitmap. See `BitmapType`.

← *transparentValueP*
Pointer to a variable that receives the transparent color, either as a palette index or as a direct color value (an RGB565 value), depending on the bitmap's depth.

Returns `true` if *`bitmapP`* has a transparent color defined, `false` otherwise.

See Also `BmpSetTransparentValue()`

BmpGetVersion Function

Purpose Gets the version of a bitmap.

Declared In `Bitmap.h`

Prototype
```
uint8_t BmpGetVersion (const BitmapType *bitmapP)
```

Parameters → *`bitmapP`*
Pointer to a valid bitmap. See `BitmapType`.

Returns The version of `bitmapP`. See Bitmap Version Constants for the defined bitmap version numbers. For debug ROMs, this function reports an error and returns 0 if *`bitmapP`* is `NULL`.

BmpSetDensity Function

Purpose Sets the density of a version 3 bitmap.

Declared In `Bitmap.h`

Prototype
```
status_t BmpSetDensity (BitmapType *bitmapP,
   uint16_t density)
```

Parameters → *`bitmapP`*
Pointer to a valid version 3 bitmap. See `BitmapTypeV3`.

→ *`density`*
The bitmap's density. This value should be one of the values defined by the `DensityType` enum.

Returns `errNone` upon success, or `sysErrParamErr` either if *`bitmapP`* is `NULL`, if *`density`* is not supported, or if *`bitmapP`* is not a version 3 bitmap.

Comments To allocate a high-density bitmap, first call `BmpCreateBitmapV3()`.

See Also `BmpGetDensity()`

BmpSetTransparentValue Function

Purpose Sets a bitmap's transparent color.

Declared In `Bitmap.h`

Prototype

```
void BmpSetTransparentValue (BitmapType *bitmapP,
    uint32_t transparentValue)
```

Parameters

→ *bitmapP*
Pointer to a valid bitmap. See `BitmapType`.

→ *transparentValue*
Transparent color. This should either be a palette index or a direct color value (an RGB565 value), depending on the bitmap's depth.

Returns Nothing.

Comments If *bitmapP* points to a version 2 bitmap, `BmpSetTransparentValue()` sets the `BitmapTypeV2` structure's `bmpFlagsHasTransparency` flag to `true` and initializes the structure's `transparentIndex` field according to *transparentValue*. For 16-bit bitmaps, this function sets the `transparentColor` field in the `BitmapDirectInfoType` auxiliary structure and sets the `transparentIndex` field to 0.

If *bitmapP* points to a version 3 bitmap, `BmpSetTransparentValue()` sets the `BitmapTypeV3` structure's `bmpFlagsHasTransparency` flag to `true` and sets the `transparentValue` field to the transparent color.

Regardless of the bitmap version, if this function is passed a *transparentValue* set to `kTransparencyNone`, this function sets the bitmap structure's `bmpFlagsHasTransparency` flag to `false` and sets the transparent color field(s) to 0.

This function does nothing if *transparentValue* contains a value that is not valid for the depth of *bitmapP*.

See Also `BmpGetTransparentValue()`

BmpSize Function

Purpose Returns the size of the bitmap.

Declared In `Bitmap.h`

Prototype
```
uint32_t BmpSize (const BitmapType *bitmapP)
```

Parameters → *bitmapP*
A pointer to the bitmap. See `BitmapType`.

Returns The size in bytes of the bitmap, including its header, color table (if any), and `sizeof(BitmapDirectInfoType)` if one exists.

Comments If the bitmap has its `bmpFlagsIndirect` flag set (see Bitmap Flag Constants), the bitmap data is not included in the size returned by this function.

See Also `BmpColortableSize()`, `BmpGetSizes()`

ColorTableEntries Macro

Purpose Returns the color table.

Declared In `CmnBitmapTypes.h`

Prototype
```
#define ColorTableEntries (ctP)
```

Parameters → *ctP*
A pointer to a `ColorTableType` structure.

Returns An array of `RGBColorType` structures, one for each entry in the color table.

Comments You can use this macro to retrieve the RGB values in use by a bitmap. For example:

```
BitmapType *bmpP;
RGBColorType *tableP =
  ColorTableEntries(BmpGetColorTable(bmpP));
```

See Also BmpGetColortable()

13

Bitmapped Font Reference

This chapter provides the following information regarding bitmapped font support:

The header files `Font.h` and `FontSelect.h` declare the API that this chapter describes. For more information on using fonts, see Chapter 5, "Displaying Text," on page 79. For reference information on scalable fonts, see Chapter 29, "Scalable Font Reference," on page 587.

Bitmapped Font Structures and Types

FontDensityTypeType Struct

Purpose An entry in the `densities` array in the `FontTypeV2Type` structure. The `densities` array specifies the location of each set of glyphs within an extended font resource.

Declared In `Font.h`

Prototype

```
typedef struct FontDensityTypeTag {
   int16_t density;
   int16_t reserved;
   uint32_t glyphBitsOffset;
} FontDensityTypeType
```

Fields

`density`
One of the `DensityType` constants.

`reserved`
: Reserved for future use.

`glyphBitsOffset`
: Offset in bytes from the beginning of the font data to the start of the font image for this density.

FontTablePtr Typedef

Purpose A pointer to a pointer to a `FontType` structure.

Declared In `Font.h`

Prototype

```
typedef FontPtr *FontTablePtr
```

FontType Struct

Purpose Defines a font resource's header. The `FontPtr` type defines a pointer to a `FontType` structure.

WARNING! PalmSource, Inc. does not support or provide backward compatibility for the `FontType` structure. Never access its structure members directly, or your code may break in future versions. Use the information below for debugging purposes only.

Declared In `Font.h`

Prototype

```
typedef struct FontTag {
   int16_t fontType;
   int16_t firstChar;
   int16_t lastChar;
   int16_t maxWidth;
   int16_t kernMax;
   int16_t nDescent;
   int16_t fRectWidth;
   int16_t fRectHeight;
   int16_t owTLoc;
   int16_t ascent;
   int16_t descent;
   int16_t leading;
   int16_t rowWords;
} FontType;
typedef FontType *FontPtr
```

Fields

`fontType`
A mask providing the general characteristics of the font. When creating an application-defined font resource, use 0x9000.

`firstChar`
Character code of first glyph in the font.

`lastChar`
Character code of last glyph in the font.

`maxWidth`
The maximum width in standard coordinates of any glyph. In Palm OS®, there is currently no difference between this field and `fRectWidth`.

`kernMax`
This value is not currently used and must be set to 0.

`nDescent`
This value is not currently used and must be set to 0.

`fRectWidth`
A metric of the font image. In Palm OS, this metric is equivalent to the maximum width in standard coordinates of any glyph in the font. Use `FntAverageCharWidth()` to obtain this value.

`fRectHeight`
The height, including ascenders and descenders, of the glyphs in this font. Use `FntCharHeight()` to obtain this value.

`owTLoc`
The offset in 16-bit words from this field to the first byte of the offset/width table. The offset/width table gives the width of each character in the font. Do not access the offset/width table directly. Use `FntCharWidth()` instead.

`ascent`
The distance in standard coordinates from the top of the font rectangle to its baseline. Use `FntBaseLine()` to obtain this value.

`descent`
The distance in standard coordinates from the baseline to the bottom of the font rectangle. Use `FntDescenderHeight()` to obtain this value.

`leading`
: The font's leading, which is the vertical space between lines of text, in standard coordinates. This field is unused in Palm OS and must be set to 0. If your font requires a leading value, add blank space to the bottom of each of your glyphs. The `FntLineHeight()` function returns the size of the font's character cell plus the leading.

`rowWords`
: The number of 16-bit words stored for each row of a glyph's bitmap where `fRectHeight` is the number of rows.

Comments The fields in this structure give general information about the font. Following the structure are several tables that Palm OS uses to draw the font on the screen.

See Also "Font Resource" on page 213

FontTypeV2Type Struct

Purpose Defines the header for an extended font resource, which contains a separate set of glyphs for each screen density.

WARNING! PalmSource, Inc. does not support or provide backward compatibility for the `FontCharTypeV2` structure. Never access its structure members directly, or your code may break in future versions. Use the information below for debugging purposes only.

Declared In `Font.h`

Prototype

```
typedef struct FontTypeV2Tag {
   int16_t fontType;
   int16_t firstChar;
   int16_t lastChar;
   int16_t maxWidth;
   int16_t kernMax;
   int16_t nDescent;
   int16_t fRectWidth;
   int16_t fRectHeight;
   int16_t owTLoc;
   int16_t ascent;
   int16_t descent;
```

```
   int16_t leading;
   int16_t rowWords;
   int16_t version;
   int16_t densityCount;
   int16_t reserved;
   FontDensityTypeType densities[1];
} FontTypeV2Type
```

Fields

`fontType`
: A mask providing the general characteristics of the font. When creating an application-defined extended font resource, use the value `0x9200`.

`firstChar`
: Character code of first glyph in the font.

`lastChar`
: Character code of last glyph in the font.

`maxWidth`
: The maximum width in standard coordinates of any glyph. There is currently no difference between this field and `fRectWidth`.

`kernMax`
: This value is not currently used and must be set to 0.

`nDescent`
: This value is not currently used and must be set to 0.

`fRectWidth`
: A metric of the font image. In Palm OS, this metric is equivalent to the maximum width in standard coordinates of any glyph in the font. Use `FntAverageCharWidth()` to obtain this value.

`fRectHeight`
: The height, including ascenders and descenders, of the glyphs in this font. Use `FntCharHeight()` to obtain this value.

`owTLoc`
: The offset in 16-bit words from this field to the first byte of the offset/width table. The offset/width table gives the width of each character in the font. Do not access the offset/width table directly. Use `FntCharWidth()` instead.

`ascent`
: The distance in standard coordinates from the top of the font rectangle to its baseline. Use `FntBaseLine()` to obtain this value.

`descent`
: The distance in standard coordinates from the baseline to the bottom of the font rectangle. Use `FntDescenderHeight()` to obtain this value.

`leading`
: The font's leading, which is the vertical space between lines of text, in standard coordinates. This field is unused in Palm OS and must be set to 0. If your font requires a leading value, add blank space to the bottom of each of your glyphs. The `FntLineHeight()` function returns the size of the font's character cell plus the leading.

`rowWords`
: The number of 16-bit words stored for each row of a glyph's bitmap where `fRectHeight` is the number of rows.

`version`
: The version of the extended font resource. This value should be set to 1.

`densityCount`
: The number of entries in the `densities` array.

`reserved`
: Reserved for future use.

`densities`
: An array of one or more `FontDensityTypeType` structures identifying the glyphs for each supported density.

See Also "Extended Font Resource" on page 216

Bitmapped Font Constants

FontDefaultType Enum

Purpose Specifies the default fonts defined in the system.

Declared In `Font.h`

Constants

`defaultSmallFont = 0`
The default small font.

`defaultLargeFont`
The default large font.

`defaultBoldFont`
The default bold font.

`defaultSystemFont`
The default font for the system.

Comments Although the standard bitmapped fonts are included on all systems, each locale might choose a different default font. For example, Latin ROMs tend to use `stdFont` as the default system bitmapped font, while Japanese ROMs might use `largeFont` instead.

See Also `FntGetDefaultFontID()`

FontID Typedef

Purpose The IDs of available fonts. A font can either be a system-defined font or an application-defined font.

Declared In `Font.h`

Prototype

```
typedef Enum8 FontID
```

Constants

`stdFont = 0x00`
A small standard font used to display user input. This font is small to display as much text as possible.

`boldFont`
Same size as `stdFont` but bold for easier reading. Used for text labels in the user interface.

`largeFont`
: A larger font provided as an alternative for users who find the standard font too small to read.

`symbolFont`
: Contains many special characters such as arrows, shift indicators, and so on.

`symbol11Font`
: Contains the check boxes, the large left arrow, and the large right arrow.

`symbol7Font`
: Contains the up and down arrows used for the repeating button scroll arrows and the dimmed version of the same arrows.

`ledFont`
: Contains the numbers 0 through 9, –, ., and the comma (,). Used by the Calculator application for its numeric display.

`largeBoldFont`
: Same size as `largeFont` but bold.

`fntAppFontCustomBase = 0x80`
: The first available ID for application-defined fonts.

See Also `FntGetFont()`, `FntSetFont()`

Miscellaneous Font Constants

Purpose Other constants defined in `Font.h`.

Declared In `Font.h`

Constants

`#define checkboxFont symbol11Font`
: A convenience constant that points to the font containing the check box bitmap.

`#define fntTabChrWidth 20`
: The width of the tab character in standard coordinates.

Bitmapped Font Resources

Font Resource

The font resource (`'NFNT'`) represents a version 1 single-density bitmapped font. This resource is the same as the Macintosh `'NFNT'` resource with some restrictions. It contains a header followed by several tables that provide information about each glyph in the font.

Figure 13.1 shows how the font resource is laid out in memory. Table 13.1 describes each table within the font resource.

Figure 13.1 Font resource ('NFNT')

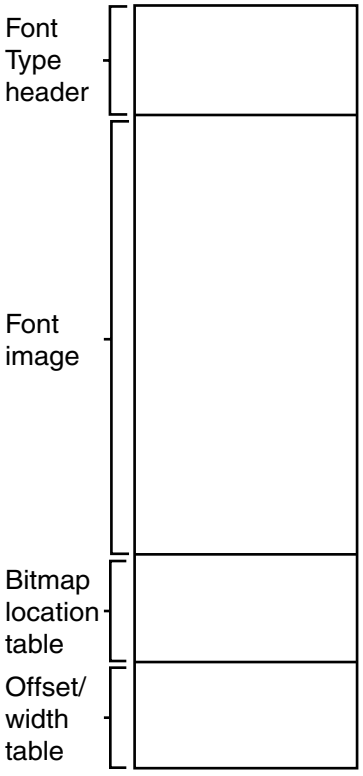


Table 13.1 Font resource description

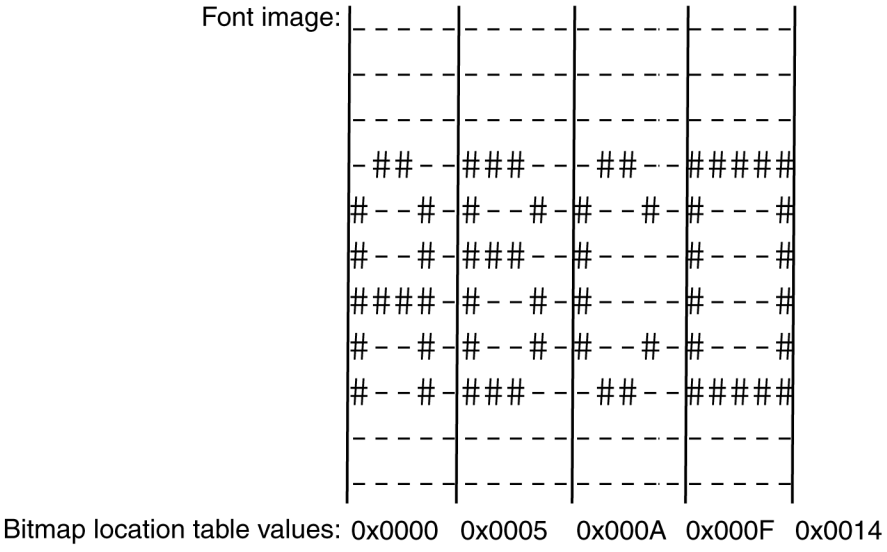
Field	Description
FontType header	Contains general information about the glyphs in the font. See `FontType`.
Font image	A raw bitmap image containing the packed character glyphs from left to right (see Figure 13.2 on page 215). This part of the resource tells Palm OS how to draw each character in the font. The height of the image is `fRectHeight` and the size is `rowWords * 2 * fRectHeight`. Place glyphs sequentially in order of increasing character code. Leave at least a one-pixel wide vertical column of space to the right of each image so that there is space between characters when Palm OS draws text on the screen. If your font requires leading, leave horizontal space at the bottom of the characters as well. The font image must end with the glyph for the missing character symbol.
Bitmap location table	A table of 16-bit words that specify the location of each glyph's entry in the font image. The location is specified as the bit offset from the start of the image to the glyph in the first row of the font image. The last entry in the table contains the offset of the column after the last bitmap. (See Figure 13.2 on page 215.) If you have skipped characters within an encoding, for each glyph that is missing, specify the same value for its location as the entry for the next glyph in the table.

Table 13.1 Font resource description *(continued)*

Field	Description
Offset/width table	A table that specifies how wide each glyph in the font is. On Macintosh systems, this table also specifies how each glyph kerns. Palm OS does not support kerning, as the offset value is ignored. Each entry in the offset/width table is two bytes long. The first byte should be 0, and the second byte should contain the glyph width, which must be greater than or equal to 0. If the glyph at this index does not have a bitmap in the font image, the values should be –1 and –1.

Figure 13.2 shows an example of the font image for a font that defines glyphs for four characters (A, B, C, and the missing character symbol) and the portion of the bitmap location table that provides the offsets for these characters. The last entry in the bitmap location table is the offset to the column after the last bitmap, or 0x0014.

Figure 13.2 Font image and bitmap location table

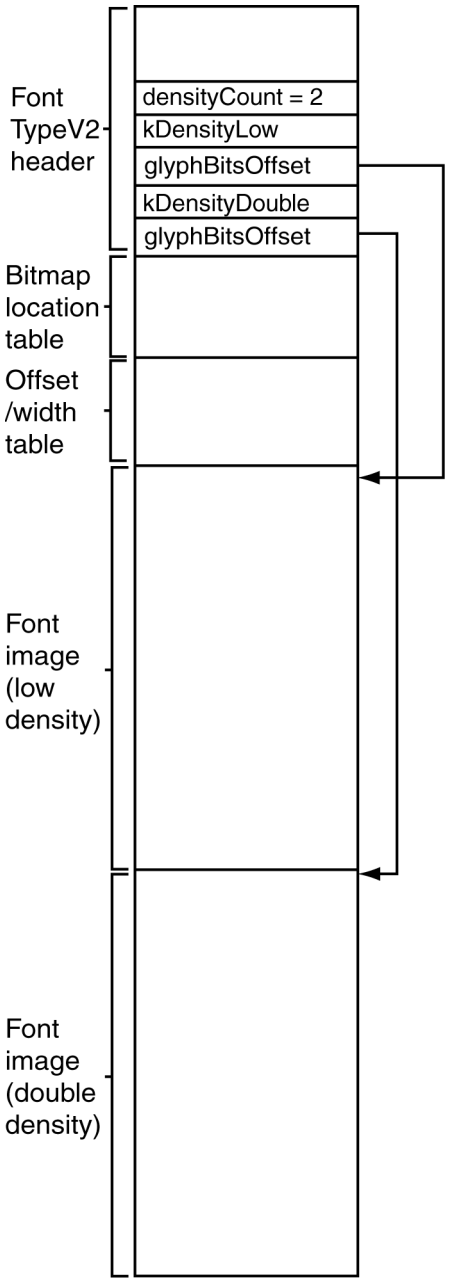


Extended Font Resource

The extended font resource (`'nfnt'`) defines a font that supports multiple screen densities. As shown in Figure 13.3, the extended font resource is essentially:

- A `FontTypeV2Type` header giving all general information about the glyphs in the font. All metrics are in terms of the low-density version of the font.
- Tables for the low-density font. See "Font Resource" on page 213 for a description of these tables.
- The font image (set of glyphs) for each density specified by the font.

Figure 13.3 Extended font resource



Bitmapped Font Functions and Macros

FntAverageCharWidth Function

Purpose Gets the *maximum* character width of the current font.

Declared In `Font.h`

Prototype
```
Coord FntAverageCharWidth (void)
```

Parameters None.

Returns The *maximum* character width (in the active coordinate system).

Comments This function returns the value of the `fRectHeight` field in the `FontType` structure for the current font. Because Palm OS does not support kerning, this value is the maximum width rather than the average width.

FntBaseLine Function

Purpose Gets the distance from the top of the character cell to the baseline for the current font.

Declared In `Font.h`

Prototype
```
Coord FntBaseLine (void)
```

Parameters None.

Returns The ascent of the font (in the active coordinate system).

FntCharHeight Function

Purpose Gets the character height of the current font including accents and descenders.

Declared In `Font.h`

Prototype
```
Coord FntCharHeight (void)
```

Parameters None.

Returns The height (in the active coordinate system) of the characters in the current font.

FntCharsInWidth Function

Purpose Finds the length in bytes of the characters from a specified string that fit within a passed width.

Declared In `Font.h`

Prototype

```
void FntCharsInWidth (const char *string,
    Coord *stringWidthP, size_t *stringLengthP,
    Boolean *fitWithinWidth)
```

Parameters

→ *string*
A pointer to the character string.

↔ *stringWidthP*
The maximum width (in the active coordinate system) to allow. Upon return, contains the actual width allowed. Note that this value does not include any trailing spaces or tabs, which are stripped by this function.

↔ *stringLengthP*
The maximum length of text to allow, in bytes (assumes current font). Upon return, contains the number of bytes of text that can appear within the width. Note that this value does not include any trailing space or tabs, which are stripped by this function.

← *fitWithinWidth*
Upon return, `false` if the string is considered truncated, `true` if it isn't.

Returns Nothing.

Comments Spaces and tabs at the end of a string are ignored and removed. If the string fits within the specified width after spaces and tabs are removed, the *fitWithinWidth* value contains `true`. Characters after a carriage return are ignored, and the string is considered truncated.

This function is specifically designed for the code used to draw text fields. Consider using `FntWidthToOffset()` or `FntTruncateString()` in your application code instead, particularly if you do not want the special processing of trailing spaces, tabs, and carriage returns.

FntCharsWidth Function

Purpose Gets the width of the specified character string. The **missing character symbol** (an open rectangle) is substituted for any character that does not exist in the current font.

Declared In `Font.h`

Prototype
```
Coord FntCharsWidth (const char *chars, size_t len)
```

Parameters

→ *chars*
Pointer to a string of characters.

→ *len*
Length in bytes of the string.

Returns The width of the string (in the active coordinate system).

Comments Like all functions that work with strings, this function returns correct results for strings with multi-byte characters as well as strings with only single-byte characters.

See Also FntCharWidth()

FntCharWidth Function

Purpose Gets the width of the specified character. If the specified character does not exist within the current font, the missing character symbol is substituted.

Declared In `Font.h`

Prototype
```
Coord FntCharWidth (wchar32_t ch)
```

Parameters

→ *ch*
Character whose width is needed.

Returns The width of the specified character (in the active coordinate system).

See Also FntCharsWidth()

FntCharWidthV50 Function

Purpose Gets the width of the specified character. If the specified character does not exist within the current font, the missing character symbol is substituted.

Declared In `Font.h`

Prototype

```
Coord FntCharWidthV50 (char ch)
```

Parameters

→ *ch*
Character whose width is needed.

Returns The width of the specified character.

Comments Do not use this function. Instead use `FntCharWidth()`, which can check the width of either single-byte or multi-byte characters.

FntGetDefaultFontID Function

Purpose Returns the font ID of a default font.

Declared In `Font.h`

Prototype

```
FontID FntGetDefaultFontID
    (FontDefaultType inFontType)
```

Parameters

→ *inFontType*
A `FontDefaultType` constant specifying one of the system default fonts.

Returns The ID of *inFontType*.

Comments Use this function whenever you need to obtain a font ID for one of the system default fonts. The default fonts (and thus, the IDs for the default fonts) vary depending on the system's locale. For example, Japanese systems have a different set of default fonts than systems using the Latin character encoding.

Use this function in place of the constants that specify the IDs of default fonts, as shown in the following table.

In place of this...	...use FntGetDefaultFontID() with this constant...
`stdFont`	`defaultSystemFont` (best for displaying text) or: `defaultSmallFont` (if you want a smaller font)
`largeFont`	`defaultLargeFont`
`largeBoldFont`	`defaultLargeFont`
`boldFont`	`defaultBoldFont`

Note that `defaultSystemFont` and `defaultSmallFont` might return the same font ID or different font IDs, depending on the system locale.

See Also `FontSelect()`, `FntGetFont()`, `FntSetFont()`

FntDefineFont Function

Purpose Makes a custom font available to your application.

Declared In `Font.h`

Prototype

```
status_t FntDefineFont (FontID font,
   const FontType *fontP)
```

Parameters

→ *font*
A value greater than or equal to `fntAppFontCustomBase` that identifies the custom font to the system. Values less than that are reserved for system use. Note that font IDs are 8-bit unsigned values and so must be less than 256. See `FontID`.

→ *fontP*
Pointer to the custom font resource to be used by this function. This resource must remain locked until the calling application undefines the custom font or quits.

Returns `errNone` upon success or one of the following errors:

`memErrNotEnoughSpace`
Insufficient dynamic heap space

Comments The custom font is available only when the application that called this function is running; when the application quits, the custom font is uninstalled automatically.

The font this function specifies is not available at build time; as a result, some UI elements—labels, for example—cannot determine their bounds automatically as they do when using the built-in fonts.

Before you use this function, you must load the font resource from the database and obtain a pointer to it. See "Creating and Using Custom Bitmapped Fonts" on page 98 for more information.

See Also `FontSelect()`, `FntSetFont()`

FntDescenderHeight Function

Purpose Gets the height of a character's descender in the current font. The height of a descender is the distance between the baseline and the bottom of the character cell.

Declared In `Font.h`

Prototype

```
Coord FntDescenderHeight (void)
```

Parameters None.

Returns The height of a descender (in the active coordinate system).

FntGetFont Function

Purpose Gets the font ID of the current bitmapped font.

Declared In `Font.h`

Prototype

```
FontID FntGetFont (void)
```

Parameters None.

Returns The ID of the current font.

Comments The current bitmapped font is the font stored in the Window Manager version of the draw state. It is used when drawing characters directly onto the screen using `WinDrawChars()` or `WinDrawChar()`. Most user interface elements, such as fields,

tables, labels, and buttons, do not use the current font. The graphics context function GcDrawTextAt() uses the scalable font specified in GcSetFont().

See Also FntSetFont(), FntGetFontPtr(), FontID

FntGetFontPtr Function

Purpose Gets a pointer to the current bitmapped font.

Declared In Font.h

Prototype

```
const FontType *FntGetFontPtr (void)
```

Parameters None.

Returns A pointer to the current font.

Comments The current bitmapped font is the font stored in the Window Manager version of the draw state. It is used when drawing characters directly onto the screen using WinDrawChars() or WinDrawChar(). Most user interface elements, such as fields, tables, labels, and buttons, do not use the current font. The graphics context function GcDrawTextAt() uses the scalable font specified in GcSetFont().

See Also FntGetFont()

FntGetScrollValues Function

Purpose Gets the values needed to update a scroll bar based on a specified string and the position within the string.

Declared In Font.h

Prototype

```
void FntGetScrollValues (const char *chars,
    Coord maxWidth, size_t scrollPos,
    uint32_t *linesP, uint32_t *topLine)
```

Parameters

→ *chars*
A null-terminated string.

→ *maxWidth*
The width (in the active coordinate system) of a line of text in the display.

→ *scrollPos*
The byte offset of the first character displayed on the topmost line.

← *linesP*
Number of lines required to display the string.

← *topLine*
The line of text that is the topmost visible line. Line numbering starts with 0.

Returns Nothing. Stores the number of lines of text in `linesP` and the top visible line in `topLine`.

See Also `FldGetScrollValues()`

FntIsAppDefined Macro

Purpose Returns `true` if the font is defined by the application or `false` if it is defined by the system.

Declared In `Font.h`

Prototype
```
#define FntIsAppDefined (fnt)
```

Parameters → *fnt*
The `FontID` of a font.

Returns Boolean that indicates if the font is an application-defined font. Returns `true` if application-defined, `false` if system-defined.

FntLineHeight Function

Purpose Gets the height of a line in the current font. The height of a line is the height of the character cell plus the space between lines (the external leading).

Declared In `Font.h`

Prototype
```
Coord FntLineHeight (void)
```

Parameters None.

Returns The height (in the active coordinate system) of a line in the current font.

FntLineWidth Function

Purpose Gets the width of the specified line of text, taking tab characters into account.

Declared In `Font.h`

Prototype

```
Coord FntLineWidth (const char *chars,
    size_t length)
```

Parameters

→ *chars*
Pointer to a string of characters.

→ *length*
Length in bytes of the string.

Returns The line width (in the active coordinate system).

Comments The function assumes that the characters passed are left-aligned and that the first character in the string is the first character drawn on a line. In other words, this function doesn't work for characters that don't start at the beginning of a line.

FntSetFont Function

Purpose Sets the current font.

Declared In `Font.h`

Prototype

```
FontID FntSetFont (FontID font)
```

Parameters

→ *font*
ID of the font to make the current font.

Returns The ID of the previous font.

Comments If the specified font ID is invalid, this function sets the current font to `stdFont`.

The current font is the font stored in the Window Manager's draw state. It is used when drawing characters directly onto the screen using `WinDrawChars()` or `WinDrawChar()`. Most user interface elements, such as fields, tables, labels, and buttons, do not use the current font. To set the font for one of these elements, check the API for that element.

The graphics context function `GcDrawTextAt()` uses the scalable font specified in `GcSetFont()`. You can use `GcCreateFontFromID()` to use a bitmapped font when drawing into the graphics context.

See Also `FntGetFont()`

FntTruncateString Function

Purpose Returns a string truncated to fit in the specified width.

Declared In `Font.h`

Prototype

```
Boolean FntTruncateString (char *iDstString,
   const char *iSrcString, FontID iFont,
   Coord iMaxWidth, Boolean iAddEllipsis)
```

Parameters

← *iDstString*
Upon return, contains *iSrcString* truncated, if necessary, to fit in the specified width.

→ *iSrcString*
The string to be truncated.

→ *iFont*
The `FontID` of the font to use when checking the width.

→ *iMaxWidth*
The maximum width of the characters to be drawn. The width is given using the active coordinate system.

→ *iAddEllipsis*
`true` if *iDstString* should end in an ellipsis (...) if *iSrcString* is too wide for the space. If that is the case, *iSrcString* is truncated further to fit the ellipsis within the specified width. If `false`, the *iDstString* simply contains the truncated version of *iSrcString* without a trailing ellipsis.

Returns `true` if *iDstString* contains a truncated version of *iSrcString*. `false` if *iDstString* is an exact copy of *iSrcString*.

Comments This function uses `FntWidthToOffset()` to determine if the destination string should be truncated. It differs from `FntWidthToOffset()` in that it returns the truncated string rather than the offset at which the string should be truncated. It differs

from WinDrawTruncChars() in that it does not draw the truncated string onto the screen.

This function may display a fatal error message if either *iDstString* or *iSrcString* are NULL or if *iMaxWidth* is invalid.

FntWCharWidthV50 Function

Purpose Gets the width of the specified character. If the specified character does not exist within the current font, the missing character symbol is substituted.

Declared In Font.h

Prototype
```
Coord FntWCharWidthV50 (wchar32_t iChar)
```

Parameters

→ *iChar*
Character whose width is needed.

Returns The width of the specified character.

Comments Do not use this function. Instead use FntCharWidth(), which works with both single-byte and multi-byte characters in Palm OS Cobalt.

FntWidthToOffset Function

Purpose Given a pixel position, gets the offset of the character displayed at that location.

Declared In Font.h

Prototype
```
size_t FntWidthToOffset (const char *chars,
   size_t length, Coord pixelWidth,
   Boolean *leadingEdge, Coord *truncWidth)
```

Parameters

→ *chars*
Pointer to the character string.

→ *length*
The length in bytes of *chars*.

→ *pixelWidth*
A horizontal offset from the beginning of the string. The offset is given using the active coordinate system.

← *leadingEdge*
Set to `true` if the pixel position *pixelWidth* falls on the left side of the character. Pass `NULL` for this parameter if you don't need this information.

← *truncWidth*
The width of the text up to but not including the returned offset. The width is given using the active coordinate system. Pass `NULL` for this parameter if you don't need this information.

Returns The byte offset into *chars* of the character that contains the offset *pixelWidth*. If *pixelWidth* is past the right edge of the string, the function returns the byte offset past the last character in *chars*, and *truncWidth* contains the width required to display the entire string.

FntWordWrap Function

Purpose Given a string, determines how many bytes of text can be displayed within the specified width with a line break at a tab or space character.

Declared In `Font.h`

Prototype

```
size_t FntWordWrap (const char *string,
   Coord maxWidth)
```

Parameters

→ *string*
A pointer to a null-terminated string.

→ *maxWidth*
The maximum line width given using the active coordinate system.

Returns The length of the line, in bytes. If the entire string cannot be displayed within *maxWidth*, the value that this function returns specifies the offset where the line should be broken, which is typically following a space, tab, or line-feed character.

See Also FldWordWrap()

FntWordWrapReverseNLines Function

Purpose Word wraps a text string backwards by the number of lines specified. The character position of the start of the first line and the number of lines that are actually word wrapped are returned.

Declared In `Font.h`

Prototype

```
void FntWordWrapReverseNLines (const char *pChars,
    Coord maxWidth, uint32_t *linesToScrollP,
    size_t *scrollPosP)
```

Parameters

→ *pChars*
A pointer to a null-terminated string.

→ *maxWidth*
The maximum line width (in active coordinates).

↔ *linesToScrollP*
The number of lines to scroll. Upon return, contains the number of lines that were scrolled.

↔ *scrollPosP*
The byte offset of the first character displayed on the topmost line. Upon return, contains the first character after wrapping.

Returns Nothing.

FontSelect Function

Purpose Displays a dialog from which the user can choose one of the system-supplied fonts, and returns a `FontID` value representing the user's choice.

Declared In `FontSelect.h`

Prototype

```
FontID FontSelect (FontID fontID)
```

Parameters

→ *fontID*
A `FontID` value specifying the font to be highlighted as the default choice in the dialog that this function displays. This value must be one of the following system-supplied constants:

`stdFont`
Standard plain text font.

`boldFont`
Bold version of `stdFont`.

`largeFont`
A large plain text font.

`largeBoldFont`
Larger version of `boldFont`.

Returns A `FontID` value representing the font that the user chose.

Comments When your application launches for the first time, it should determine the system's default font. The default font varies based on locale. You can use `FntGetDefaultFontID()` to determine the default font as follows:

```
fntID =
FntGlueGetDefaultFontID(defaultSystemFont);
```

See Also `FntGetFont()`, `FntSetFont()`

14

Category Manager Reference

This chapter describes the category API as declared in the header file `CatMgr.h`. It discusses the following topics:

For more information on categories, see the section "Category Controls" on page 54.

Category Manager Structures and Types

AppInfoType Struct

Purpose Maps category names to category indexes and unique IDs. The `AppInfoPtr` defines a pointer to an `AppInfoType` structure.

Declared In `CatMgr.h`

Prototype

```
typedef struct AppInfoTag {
   uint16_t renamedCategories;
   char categoryLabels [dmRecNumCategories]
      [dmCategoryLength];
   uint8_t categoryUniqIDs[dmRecNumCategories];
   uint8_t lastUniqID;
   uint8_t padding;
} AppInfoType;
typedef AppInfoType *AppInfoPtr
```

Fields

`renamedCategories`
Used by `CategorySetName()` as a bit field indicating which categories have been renamed. Usually cleared by a conduit.

`categoryLabels`
: An array of strings containing the category names. The maximum size of the array is `dmRecNumCategories`, and the maximum length of each string in the array is `dmCategoryLength`. Both of these constants are defined in `DataMgr.h`.

`categoryUniqIDs`
: Category IDs used for synchronization with the desktop database. Unique IDs generated by the device are between 0 and 127. Unique IDs generated by the desktop computer are between 128 and 255.

`lastUniqID`
: Used for sorting and assigning unique IDs.

`padding`
: Reserved for future use.

Comments

The `AppInfoType` structure is used in non-schema databases. A non-schema database's application info block must either be an `AppInfoType` structure, or it must have an `AppInfoType` structure as its first field if you want to use the functions described in this chapter to manage the user interface elements associated with categories.

Allocate the application info block in the storage heap and use the `DmSetDatabaseInfo()` function to set the database's application info ID to the local ID of this structure. Then, use the `CategoryInitialize()` function to initialize it with a localized list of strings containing the category names.

Category Manager Constants

Category ID Range Constants

Purpose Specify the range of valid values for category IDs in schema databases.

Declared In `CatMgr.h`

Constants

```
#define catCategoryIDNegativeLowerBound
  ((CategoryID)0x80000001)
```
The lowest possible private category ID.

```
#define catCategoryIDNegativeUpperBound
  ((CategoryID)0xfffffffe)
```
The highest possible private category ID.

```
#define catCategoryIDPositiveLowerBound
  ((CategoryID)0x00000010)
```
The lowest possible public category ID.

```
#define catCategoryIDPositiveUpperBound
  ((CategoryID)0x7fffffff)
```
The highest possible public category ID.

Error Code Constants

Purpose Error codes returned by Category Manager functions.

Declared In `CatMgr.h`

Constants

```
#define catmErrAISResourceNotFound (catmErrorClass
  | 9)
```
The app info strings resource used to initialize the categories in the database cannot be found.

```
#define catmErrCantFind (catmErrorClass | 3)
```
The specified category could not be found.

```
#define catmErrCategoryNotFound (catmErrorClass |
  5)
```
The specified category could not be found.

```
#define catmErrIndexOutOfRange (catmErrorClass | 4)
```
An error occurred while iterating through a list of categories.

```
#define catmErrInvalidParam (catmErrorClass | 2)
```
An invalid parameter was passed to a function.

```
#define catmErrInvalidStoragePtr (catmErrorClass |
  7)
```
The category information in the database cannot be accessed.

```
#define catmErrMaxCategoryLimit (catmErrorClass |
  8)
```
An attempt was made to add more categories to a database than is allowed.

```
#define catmErrMemError (catmErrorClass | 1)
```
Not enough memory to perform the requested operation.

```
#define catmErrNameAlreadyExists (catmErrorClass |
  6)
```
An attempt was made to add a new category or rename a category, but a category with that name already exists.

```
#define catmErrNotSchemaDatabase (catmErrorClass |
  10)
```
A schema database Category Manager call was passed the handle to a non-schema database.

```
#define catmErrReadOnlyDatabase (catmErrorClass |
  11)
```
The schema database must be opened for write access for this function.

Non-schema Database Category Constants

Purpose Constants used by the non-schema database Category Manager functions.

Declared In `CatMgr.h`

Constants

```
#define categoryDefaultEditCategoryString 10001
```
Used to show the default "Edit Categories" item.

```
#define categoryHideEditCategory 10000
```
Used to suppress the "Edit Categories" item.

Comments These constants look like system resource IDs, but they are not. To use a non-default string for the "Edit Categories" item you pass a resource ID of a string containing your title. If you want to use the default or hide the item, you pass one of these constants. They are within the system resource ID range (that is, they are greater than 10000) so that they don't conflict with any other possible value for that parameter.

Special Category ID Constants

Purpose Represent items that a user might select in a category pop-up list but that are not actual categories. The `CatMgrSelectEdit()` and `CatMgrSelectFilter()` might return these IDs as the user selection.

Declared In `CatMgr.h`

Constants

`#define catIDAll ((CategoryID)0x01)`
The "All" list item. The record should belong to all categories, or all records should be displayed.

`#define catIDMultiple ((CategoryID)0x02)`
The "Multiple" list item. Users select this list item to assign more than one category to a record.

`#define catIDUnfiled ((CategoryID)0x00)`
The "Unfiled" list item. The name of this category is stored as category ID 0 in each schema database.

See Also `CatMgrGetAllItemLabel()` `CatMgrGetUnfiledItemLabel()`

Miscellaneous Schema Database Category Constants

Purpose Other constants defined in `CatMgr.h`.

Declared In `CatMgr.h`

Constants

`#define catCategoryNameLength 0x20`
The size of a category name. Strings that hold the category name must be at least this size.

`#define CategoryNameReserved 10021`
: Resource ID of an alert displayed if the user attempts to rename uneditable categories.

`#define catNumCategories 0xff`
: The maximum number of categories allowed in a schema database.

Category Manager Functions and Macros

CategoryCreateList Function

Purpose Populates a pop-up list with a non-schema database's categories.

Declared In `CatMgr.h`

Prototype

```
void CategoryCreateList (DmOpenRef db,
   ListType *listP, uint16_t currentCategory,
   Boolean showAll, Boolean showUneditables,
   uint8_t numUneditableCategories,
   DmOpenRef resDbRef, uint32_t editingStrID,
   Boolean resizeList)
```

Parameters

→ *db*
: Open non-schema database containing the category information you want to read.

← *listP*
: Pointer to the `ListType` that is populated with the database's categories.

→ *currentCategory*
: Index of the category to select. The index is the index into the `categoryLabels` array. The default is to have the "Unfiled" category selected.

→ *showAll*
: `true` to include an "All" list item.

→ *showUneditables*
: `true` to show uneditable categories.

→ *numUneditableCategories*
: The number of categories that the user cannot edit. You should store uneditable categories at the beginning of the

`categoryLabels` array. For example, it's common to have an "Unfiled" category at position zero that is not editable. This function displays the uneditable categories at the end of the pop-up list.

→ *`resDbRef`*
Open resource database containing the string resource.

→ *`editingStrID`*
The resource ID of a string resource to use as the Edit Categories list item. To use the default string for the current locale (such as "Edit Categories") pass the constant `categoryDefaultEditCategoryString`.

If you don't want users to edit categories, pass the `categoryHideEditCategory` constant.

→ *`resizeList`*
`true` to resize the list to the number of categories. Set to `true` for pop-ups, `false` otherwise.

Returns Nothing.

Comments The "All" item is first in the list (if the *`showAll`* parameter is `true`), followed by the editable categories in the database and then the categories that cannot be edited. The option to edit categories is last in the list and can be suppressed if desired.

You rarely call this function directly. Instead, most applications use `CategorySelect()`, which calls this function and fully manages the user's selection of a category in the pop-up list. Use `CategoryCreateList()` only if you want more control over the category pop-up list.

This function obtains the *`db`* parameter's `appInfoID`, reads the `AppInfoType` structure at that location, and uses the information in it to initialize the *`listP`*'s items array with the names of the database's categories. You must have already allocated the structure pointed to by *`listP`*. `CategoryCreateList()` does not display the list.

You must balance a call to `CategoryCreateList()` with a call to `CategoryFreeList()`. The `CategoryCreateList()` function locks the resources for the category names. It also allocates the *`listP`* items array. `CategoryFreeList()` unlocks all resources locked by `CategoryCreateList()` and frees all memory allocated by `CategoryCreateList()`.

CategoryEdit Function

Purpose Event handler for the Edit Categories dialog for non-schema databases.

Declared In `CatMgr.h`

Prototype

```
Boolean CategoryEdit (DmOpenRef db,
    uint16_t *categoryP, DmOpenRef resDbRef,
    uint32_t titleStrID,
    uint8_t numUneditableCategories)
```

Parameters

→ *db*
Open non-schema database containing the categories to be edited.

← *category*
Upon return, the index of the last category selected before the dialog was closed.

→ *resDbRef*
Open resource database containing the string resource.

→ *titleStrID*
The resource ID of a string resource to use as the dialog's title. To use the default string ("Edit Categories"), pass the constant `categoryDefaultEditCategoryString`.

→ *numUneditableCategories*
The number of categories that the user cannot edit. You should store uneditable categories at the beginning of the `categoryLabels` array. For example, it's common to have an "Unfiled" category at position zero that is not editable.

Returns `true` if any of the following conditions are `true`:

- The current category is renamed.
- The current category is deleted.
- The current category is merged with another category.

Comments You rarely call this function directly. The `CategorySelect()` function calls it when the user chooses the Edit Category list item.

This function both displays the Edit Categories dialog and handles the result of the user actions. It updates the `AppInfoType` structure's list of categories and reassigns database records to new categories as needed. If a user deletes a category, `CategoryEdit()`

moves all of the records belonging to that category to the Unfiled category. If a category is renamed to be the same as an existing category, this function moves all of the old category's records to the new category.

See Also CategoryEdit(), DmMoveCategory()

CategoryFind Function

Purpose Returns the index of a category in a non-schema database given its name.

Declared In CatMgr.h

Prototype

```
uint16_t CategoryFind (DmOpenRef db,
    const char *name)
```

Parameters

→ *db*
Open non-schema database to search.

→ *name*
Category name. Pass the empty string to find the first unused category.

Returns The index of the category's entry in the `categoryLabels` array (see AppInfoType). Returns `dmAllCategories` if the category does not exist.

CategoryFreeList Function

Purpose Unlocks or frees memory locked or allocated by CategoryCreateList().

Declared In CatMgr.h

Prototype

```
void CategoryFreeList (DmOpenRef db,
    const ListType *listP, Boolean showAll,
    uint32_t editingStrID)
```

Parameters

→ *db*
Open non-schema database containing the categories.

→ *listP*
Pointer to the category list.

→ *showAll*
`true` if the list was created with an "All" category.

→ *editingStrID*
The resource ID that you passed as the `editingStrID` parameter to `CategoryCreateList()`. This function unlocks the resource.

Returns Nothing.

Comments You only need to call this function if you explicitly call `CategoryCreateList()`. Typical applications call `CategorySelect()`, which handles both the creation and deletion of the list.

This function frees the items in the pop-up list *listP*'s items array and it unlocks other resources that `CategoryCreateList()` may have locked.

This function does *not* remove the categories from the passed database, and it does *not* free the `ListType` structure pointed to by *listP*. (Typically, a list is freed when its form is freed.)

CategoryGetName Function

Purpose Returns the name of the specified category in a non-schema database.

Declared In `CatMgr.h`

Prototype
```
void CategoryGetName(DmOpenRef db, uint16_t index,
    char *name)
```

Parameters → *db*
Open non-schema database that contains the categories.

→ *index*
Category index. This is the index into the `categoryLabels` array in the `AppInfoType` structure. You can retrieve this index from a database record's attribute word.

← *name*
Buffer to hold category name. Buffer should be `dmCategoryLength` in size.

Returns Nothing. May display a fatal error message if the index is out of range.

Example You can use this function to find out the name of a given database record's category. Use the DmGetRecordCategory() call to obtain the category index from the given record. For example:

```
DmOpenRef myDB;
uint16_t category;
Char *name;

DmGetRecordCategory (AddrDB, CurrentRecord, &category);
CategoryGetName(myDB, category, name);
```

See Also CategorySetName()

CategoryGetNext Function

Purpose Returns the index of the next category after a given category in a non-schema database.

Declared In CatMgr.h

Prototype

```
uint16_t CategoryGetNext (DmOpenRef db,
    uint16_t index)
```

Parameters

→ *db*
Open non-schema database containing the categories.

→ *index*
Category index.

Returns Category index of next category.

Comments The intended use of this function is to allow your users to cycle through categories. For example, the built-in applications cycle through categories when the user presses the corresponding hard-key button. (See the `ListViewNextCategory()` function in the Address Book sample application for an example.) Note that categories are not displayed in the same order as they are stored.

Do not use this function for searching for a particular category or iterating through a category list.

CategoryInitialize Function

Purpose Initializes the category names, IDs, and flags for a non-schema database.

Declared In `CatMgr.h`

Prototype

```
void CategoryInitialize (AppInfoPtr appInfoP,
    DmOpenRef resDbRef,
    uint16_t localizedAppInfoStrID)
```

Parameters

→ *appInfoP*
Pointer to the locked application info block. See `AppInfoType`.

→ *resDbRef*
Open resource database containing the app info strings resource.

→ *localizedAppInfoStrID*
Resource ID of the localized category names. This must be a resource of the type `appInfoStringsRsc` (`'tAIS'`).

Returns Nothing.

Comments Call this function at database creation time to initialize the database's categories from a list of localized strings.

`CategoryInitialize()` initializes the `AppInfoType` structure that is associated with your database. It does not create the structure. To create the structure, you must allocate it in the storage heap (using `DmNewHandle()`) and associate it with your database using `DmSetDatabaseInfo()`.

CategorySelect Function

Purpose Processes the selection and editing of categories for a non-schema database.

Declared In `CatMgr.h`

Prototype

```
Boolean CategorySelect (DmOpenRef db,
    const FormType *frm, uint16_t ctlID,
    uint16_t lstID, Boolean showAll,
    uint16_t *categoryP, char *categoryName,
    uint8_t numUneditableCategories,
    DmOpenRef resDbRef, uint32_t editingStrID)
```

Parameters

→ *db*
Open database containing the categories.

→ *frm*
Form that contains the category pop-up list.

→ *ctlID*
Resource ID of the pop-up trigger.

→ *lstID*
Resource ID of the pop-up list.

→ *showAll*
`true` to have an "All" list item. In general, if the form displays multiple records, it should have an "All" list item. If the form displays a single database record, it should not.

↔ *categoryP*
Index of the selected category. The index is the index into the `categoryLabels` array.

↔ *categoryName*
Name of the selected category.

→ *numUneditableCategories*
The number of categories that the user cannot edit. You should store uneditable categories at the beginning of the `categoryLabels` array. For example, it's common to have an "Unfiled" category at position zero that is not editable. This function displays the uneditable categories at the end of the pop-up list.

→ *resDbRef*
Open resource database containing the string resource.

→ *editingStrID*
The resource ID of a string resource to use as the Edit Categories list item. To use the default string ("Edit Categories"), pass the constant `categoryDefaultEditCategoryString`.

If you don't want users to edit categories, pass the `categoryHideEditCategory` constant.

Returns `true` if any of the following conditions are true:

- The current category is renamed.
- The current category is deleted.
- The current category is merged with another category.

Comments Call this function when the user taps the category pop-up trigger. This function handles all aspects of displaying the pop-up list and managing the user selection—It creates the pop-up list using `CategoryCreateList()`, displays the pop-up list, calls `CategoryEdit()` if the user selects the Edit Categories item, uses `CategorySetTriggerLabel()` to set the trigger label to the item the user selected, and then calls `CategoryFreeList()` to free the list items array. Your application is responsible for checking the value of *`categoryP`* upon return and updating the display or changing the record's category to the new selection.

CategorySetName Function

Purpose Changes the category name in the `AppInfoType` structure of a non-schema database, or deletes a category.

Declared In `CatMgr.h`

Prototype

```
void CategorySetName(DmOpenRef db,uint16_t index,
    const char *nameP)
```

Parameters

→ *`db`*
Open non-schema database containing the category to change.

→ *`index`*
Index of category to rename.

→ *`nameP`*
The new category name (null-terminated), or `NULL` to delete the category.

Returns Nothing.

Comments The `CategoryEdit()` function calls this function when a user creates a new category or renames an existing category in the Edit Categories dialog. Your application does not have to call it directly.

CategorySetTriggerLabel Function

Purpose Sets the label displayed by the category pop-up trigger for a non-schema database.

Declared In `CatMgr.h`

Prototype

```
void CategorySetTriggerLabel (ControlType *ctl,
    char *name)
```

Parameters

↔ *ctl*
Pointer to control (pop-up or selector trigger) to relabel.

↔ *name*
Pointer to the name of the new category.

Returns Nothing.

Comments The `CategorySetTriggerLabel()` function calls the `CategoryTruncateName()` function to truncate the category name to the maximum length. The maximum length varies, depending upon which ROM is installed in the device.

NOTE: This function passes the *name* parameter to the `CategoryTruncateName()` function, which means that the *name* value must be modifiable. `CategorySetTriggerLabel()` does not make a copy of the string passed, so you must ensure that the string remains valid until the form is closed.

See Also `CtlSetLabel()`

CategoryTruncateName Function

Purpose Truncates a category name from a non-schema database so that it's short enough to display. The category name is truncated if it's longer than *maxWidth*.

Declared In `CatMgr.h`

Prototype

```
void CategoryTruncateName (char *name,
    uint16_t maxWidth)
```

Parameters

↔ *name*
Category name to truncate. Upon return, contains the truncated name.

→ *maxWidth*
Maximum size, in standard coordinates, of truncated category (including ellipsis).

Returns Nothing.

CatMgrAdd Function

Purpose Creates a category and adds it to a schema database.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrAdd (DmOpenRef dbP,
   const char *nameP, CategoryID *catIDP)
```

Parameters

→ *dbP*
Open schema database.

→ *nameP*
The name of the new category.

← *catIDP*
The ID that this function assigned to the new category.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
dbP or *nameP* is `NULL`.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrMaxCategoryLimit`
The database already contains the maximum number of categories allowed.

`catmErrNameAlreadyExists`
nameP is already used by another category.

Comments Typical applications do not call this function. Instead, use `CatMgrInitialize()` to initialize the database with default categories stored in an application's resource database, and use `CatMgrSelectEdit()` and `CatMgrSelectFilter()` to allow the user to edit the categories list. These functions add categories to the schema database as necessary.

Example The following code shows how to create an uneditable category.

```
CategoryID catID;

dbRef = DbOpenDatabase(dbID, dmModeReadWrite, dbShareNone);

err = CatMgrAdd(dbRef, "category name", &catID);
if (err == errNone)
   CatMgrSetEditable(dbref, catID, false);
```

See Also `CatMgrRemove()`

CatMgrCreateList Function

Purpose Populates a pop-up list with a schema database's categories.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrCreateList (DmOpenRef dbRef,
   ListType *listP, CategoryID currentCategory,
   Boolean showAll, Boolean showMultiple,
   Boolean showUneditables, Boolean resizeList,
   Boolean showEditingStr,
   char *customEditingStrP)
```

Parameters

→ *dbRef*
Open schema database containing the category information you want to read.

← *listP*
Pointer to the `ListType` structure that should display the categories.

→ *currentCategory*
ID of the category that should be selected when the list is initially displayed.

→ *showAll*
`true` to include an "All" list item. In general, if the list is used to filter the display of database records, it should have an "All" list item. If the list is a list of categories that might be assigned to a single database record, it should not.

→ *showMultiple*
`true` to show a "Multiple" list item, which is used if multiple categories can be selected. In general, if the list is used to filter the display of database records, it should not have a "Multiple" list item. If the list is a list of categories that might be assigned to a single database record, it can have the list item if you like.

→ *showUneditables*
`true` to show uneditable categories.

→ *resizeList*
`true` to resize the list to the number of categories. Set to `true` for pop-up lists, `false` otherwise.

→ *showEditingStr*
`true` to show an Edit Categories list item.

→ *customEditingStrP*
A string to use as the Edit Categories list item. To use the default string for the current locale, pass `NULL` for this parameter. This parameter is ignored if *showEditingStr* is `false`.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbRef*.

`catmErrNotSchemaDatabase`
dbRef is not a schema database.

`catmErrReadOnlyDatabase`
dbRef is not open for writing.

`dmErrInvalidParam`
dbRef is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
: An error occurred while retrieving category information from the database.

`catmErrIndexOutOfRange`
: An error occurred while retrieving category information from the database.

`catmErrCategoryNotFound`
: Could not find the specified category.

`memErrNotEnoughSpace`
: An error occurred while attempting to resize the list.

`memErrChunkLocked`
: An error occurred while attempting to resize the list.

`memErrInvalidParam`
: An error occurred while attempting to resize the list.

Comments

The "All" item is first in the list (if the *`showAll`* parameter is `true`), followed by the editable categories in the database and then the categories that cannot be edited. The option to edit categories is last in the list and can be suppressed if desired.

You rarely call this function directly. Instead, most applications use `CatMgrSelectEdit()` and `CatMgrSelectFilter()`, which call this function and fully manage the user's selection of categories in the pop-up list. Use `CatMgrCreateList()` only if you want more control over the category pop-up list.

This function obtains the *`dbRef`* parameter's category information and uses it to initialize the *`listP`*'s items array with the names of the database's categories. You must have already allocated the structure pointed to by *`listP`*. `CatMgrCreateList()` does not display the list.

You must balance a call to `CatMgrCreateList()` with a call to `CatMgrFreeList()`. The `CatMgrCreateList()` function locks the resources for the category names. It also allocates the *`listP`* items array. `CatMgrFreeList()` unlocks all resources locked by `CatMgrCreateList()` and frees all memory allocated by `CatMgrCreateList()`.

CatMgrEdit Function

Purpose Event handler for the Edit Categories dialog used for schema databases.

Declared In `CatMgr.h`

Prototype

```
Boolean CatMgrEdit (DmOpenRef dbRef,
   CategoryID *inCurrentCategoriesP,
   uint32_t numInCurrentCategories,
   CategoryID **outCurrentCategoriesPP,
   uint32_t *numOutCurrentCategoriesP,
   char *titleStrP)
```

Parameters

→ *dbRef*
Open schema database containing the categories to be edited.

→ *inCurrentCategoriesP*
An array of category IDs representing the currently selected categories. If this parameter contains only one category, that category is initially selected when the dialog is opened. If it contains multiple categories, there is no initial selection.

→ *numInCurrentCategories*
The number of categories listed in *inCurrentCategoriesP*.

← *outCurrentCategoriesPP*
An array of category IDs from *inCurrentCategoriesP* that the user has edited. Deleted categories do not appear in this list.

← *numOutCurrentCategoriesP*
The number of categories in *outCurrentCategoriesPP*.

→ *titleStrP*
A string to use as the dialog's title. This function makes a copy of the string, so you may free it at any time. To use the default string for the current locale, pass `NULL` for this parameter.

Returns `true` if any of the following conditions are true:

- One of the categories in *inCurrentCategoriesP* is renamed.
- One of the categories in *inCurrentCategoriesP* is deleted.
- One of the categories in *inCurrentCategoriesP* is merged with another category.

Comments You rarely call this function directly. The `CatMgrSelectFilter()` and `CatMgrSelectEdit()` functions call it when the user chooses the Edit Category list item.

This function both displays the Edit Categories dialog and handles the result of the user actions. It update's the schema database's list of categories and reassigns database records to new categories as needed. If the user deletes a category, `CatMgrEdit()` removes it from the membership lists of all database records that belong to it. If a category is renamed to be the same as an existing category, this function moves all of the old category's records to the new category.

The *`inCurrentCategoriesP`* and *`outCurrentCategoriesPP`* parameters are used to manage the display of database records on the form containing the category user interface. Set *`inCurrentCategoriesP`* to the categories currently being displayed. For example, if the form displays one database record and the currently displayed record belongs to three categories, *`inCurrentCategoriesP`* should list the IDs of all three categories and *`numInCurrentCategories`* should be set to three. If the user modifies any of these categories during the Edit Categories session, `CatMgrEdit()` returns `true`. Your application should then check *`outCurrentCategoriesPP`* and, if necessary, update the user interface to reflect the change.

CatMgrFind Function

Purpose Returns the ID of the category with the specified name.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrFind (DmOpenRef dbP,
   const char *nameP, CategoryID *catIDP)
```

Parameters

→ *`dbP`*
Open schema database.

→ *`nameP`*
The name of the category that you want to find.

← *`catIDP`*
The ID of the category with the specified name.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP*.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Could not find the specified category.

See Also CatMgrGetName(), CatMgrSetName()

CatMgrFreeList Function

Purpose Unlocks or frees memory locked or allocated by CatMgrCreateList().

Declared In CatMgr.h

Prototype

```
void CatMgrFreeList (DmOpenRef dbRef,
   ListType *listP, Boolean showAll,
   Boolean showMultiple, Boolean showEditingStr)
```

Parameters

→ *dbRef*
Open schema database.

→ *listP*
Pointer to the category list.

→ *showAll*
`true` if the list was created with an "All" item.

→ *showMultiple*
`true` if the list was created with a "Multiple" item.

→ *showEditingStr*
`true` if the list was created with an "Edit Categories" item.

Returns Nothing.

Comments You only need to call this function if you explicitly call `CatMgrCreateList()`. Typical applications call `CatMgrSelectEdit()` and `CatMgrSelectFilter()`, both of which handle the creation and deletion of the list.

This function frees the items in the pop-up list *`listP`*'s items array and it unlocks other resources that `CatMgrCreateList()` may have locked.

This function does *not* do the following:

- Remove the categories from the passed database.
- Free the `ListType` structure pointed to by *`listP`*. (Typically a list is freed when its form is freed.)
- Free the string you pass as a custom name for the "Edit Categories" list item. You'll need to unlock the associated resource or free that string in your application's code.

CatMgrFreeSelectedCategories Function

Purpose Frees the memory associated with the selected categories list, which is returned by some of the Category Manager functions.

Declared In `CatMgr.h`

Prototype

```
void CatMgrFreeSelectedCategories(DmOpenRef dbRef,
    CategoryID **selectedCategoriesPP)
```

Parameters

→ *`dbRef`*
Open schema database.

→ *`selectedCategoriesPP`*
The parameter whose memory should be freed.

Returns Nothing.

See Also `CatMgrSelectEdit()`, `CatMgrSelectFilter()`

CatMgrGetAllItemLabel Function

Purpose Returns the text of the "All" list item used in the current locale.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrGetAllItemLabel (DmOpenRef dbP,
    char *labelP)
```

Parameters

→ *dbP*
Open schema database.

← *labelP*
A copy of the string used for the "All" item in a category pop-up list.

Returns `errNone` upon success or `catmErrInvalidParam` if *labelP* is `NULL`.

Comments The "All" item label is stored in the system resource database and is localized to all supported languages. In contrast, the "Unfiled" item label is specific to each schema database.

See Also `CatMgrGetUnfiledItemLabel()`

CatMgrGetEditable Function

Purpose Returns whether the user can edit the specified category.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrGetEditable (DmOpenRef dbP,
    CategoryID catID, Boolean *editableP)
```

Parameters

→ *dbP*
Open schema database.

→ *catID*
A category ID.

← *editableP*
`true` if the category with the specified ID is editable, or `false` otherwise.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP* or *catID* is not a valid category ID number.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID *catID*.

See Also CatMgrSetEditable()

CatMgrGetID Function

Purpose Returns the ID of the category at the specified index.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrGetID (DmOpenRef dbP,
   uint32_t index, CategoryID *catIDP)
```

Parameters

→ *dbP*
Open schema database.

→ *index*
The index into the database's list of categories.

← *catIDP*
The ID of the category at the specified index.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP*.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrIndexOutOfRange`
The *`index`* parameter is invalid.

Comments You might use this function in a loop to retrieve all category information stored in a schema database. To retrieve the ID of a specific category, use `CatMgrFind()`.

This function only returns information about public categories. Schema databases might contain private categories. If so, `CatMgrGetID()` ignores those categories.

See Also `CatMgrGetName()`, `CatMgrNumCategories()`

CatMgrGetName Function

Purpose Returns the name of the specified category.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrGetName (DmOpenRef dbP,
   CategoryID catID, char *nameP)
```

Parameters

→ *`dbP`*
Open schema database.

→ *`catID`*
The category ID.

← *`nameP`*
The name of the category with the specified ID. This string is copied from the database, so you are responsible for freeing it.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *`dbP`*.

`catmErrNotSchemaDatabase`
`dbP` is not a schema database.

`catmErrReadOnlyDatabase`
`dbP` is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID *catID*.

See Also CatMgrSetName(), CatMgrFind(), CatMgrGetID()

CatMgrGetNext Function

Purpose Returns the next non-empty category after the specified category.

Declared In CatMgr.h

Prototype

```
status_t CatMgrGetNext (DmOpenRef dbP,
   CategoryID inCatID, CategoryID *outCatIDP)
```

Parameters

→ *dbP*
Open schema database.

→ *inCatID*
A category ID.

← *outCatIDP*
The ID of the next category in the list.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP*.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID *inCatID*.

`catmErrIndexOutOfRange`
An error occurred iterating through the list of categories.

Comments The intended use of this function is to allow your users to cycle through categories. For example, the built-in applications cycle through categories when the user presses the corresponding hard-key button on the device. Note that categories are not displayed in the same order as they are stored.

Do not use this function for searching for a particular category or iterating through the category list.

See Also `CatMgrGetID()`, `CatMgrFind()`

CatMgrGetUnfiledItemLabel Function

Purpose Returns the label used for the "Unfiled" item in a category list.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrGetUnfiledItemLabel (DmOpenRef dbP,
    char *labelP)
```

Parameters

→ *dbP*
Open schema database.

← *labelP*
The text used for the "Unfiled" item within this database.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP*.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID `catIDUnfiled`.

Comments The "Unfiled" item label is specific to each schema database. In contrast, the "All" item label is stored in the system resource database and is localized to all supported languages.

In schema databases, a database record never belongs to the Unfiled category the way that it does in a non-schema database. A schema database record with no category information can be said to be Unfiled.

See Also `CatMgrGetAllItemLabel()`

CatMgrInitialize Function

Purpose Initializes category information for a schema database. This function is typically called right after you create a schema database.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrInitialize (DmOpenRef dbRef,
    MemHandle localizedCategoryNamesStrH)
```

Parameters

→ *dbRef*
Open schema database.

→ *localizedCategoryNamesStrH*
A handle to a string list containing localized category names. This list is initialized from a resource of type `appInfoStringsRsc ('tAIS')`.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP* or *localizedCategoryNamesStrH.*

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrMaxCategoryLimit`
: The *`localizedCategoryNamesStrH`* contains more than the maximum number of categories allowed for a schema database.

`catmErrNameAlreadyExists`
: The database has already been initialized with category information.

Comments

This function creates IDs for each initialized category. The IDs are assigned starting with zero for the first category in the list and incremented for each subsequent category in the list.

The *`localizedCategoryNamesStrH`* parameter contains predefined categories that new users see when they start the application for the first time. Follow these guidelines when creating the resource:

- The string list may contain up to 255 entries. Typically you don't want to predefine 255 categories.
- Each category name has a maximum length defined by the `catCategoryNameLength` constant (currently 31+1 bytes).
- It's common to have at least one category named Unfiled as the first item in the list. No database records ever belong to this category, but it may be used to determine which records to display. Records that do not belong to any category are considered to be unfiled.
- Don't include strings for the "All" or "Edit Categories" items that you see in a categories pop-up list. When displaying the category list, the Category Manager will automatically generated list entries for these items.

See Also `CatMgrGetUnfiledItemLabel()`, `CatMgrAdd()`

CatMgrNumCategories Function

Purpose Returns the number of categories in a schema database.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrNumCategories (DmOpenRef dbP,
   uint32_t *numCategoriesP)
```

Parameters

→ *dbP*
Open schema database.

← *numCategoriesP*
The number of categories stored in the database.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP*.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

See Also `CatMgrGetID()`, `CatMgrGetNext()`

CatMgrRemove Function

Purpose Removes a category from a schema database.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrRemove (DmOpenRef dbP,
   CategoryID catID)
```

Parameters

→ *dbP*
Open schema database.

→ *catID*
The ID of the category to remove.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP* or *catID* is an invalid ID.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`dmErrInvalidCategory`
The category could not be found in the database.

`catmErrMemError`
There is not enough memory to complete the operation.

Comments This function also removes the membership in this category from each database record that is a member of this category. If a record belongs only to this category, it becomes unfiled.

See Also CatMgrAdd()

CatMgrSelectEdit Function

Purpose Displays a pop-up list from which the user can choose category membership for an existing record.

Declared In CatMgr.h

Prototype

```
Boolean CatMgrSelectEdit (DmOpenRef dbRef,
   const FormType *frm, uint16_t ctlID,
   char *ctlLabelBuffer, uint16_t lstID,
   Boolean allowMultiple,
   CategoryID currentCategoriesP[],
   uint32_t numCurrentCategories,
   CategoryID *selectedCategoriesPP[],
   uint32_t *numSelectedCategoriesP,
   Boolean showEditingStr,
   char *customEditingStrP)
```

Parameters → *dbRef*
Open schema database.

→ *frm*
Pointer to the form that contains the category pop-up list.

→ *ctlID*
Resource ID of the selector trigger.

↔ *ctlLabelBuffer*
Stores the string displayed in the selector trigger. This string must have a minimum size of `catCategoryNameLength`, but it can be longer. It must exist for the life of the form.

Upon return, contains the updated string displayed in the selector trigger. This string is a comma-separated list of the currently selected categories, truncated to fit a particular space.

→ *lstID*
Resource ID of the pop-up list.

→ *allowMultiple*
If `true`, a "Multiple" item is added to the list so that the user can select multiple categories for the record. If `false`, the record can only belong to a single category.

→ *currentCategoriesP*
An array of the IDs of categories to which the record belongs.

→ *numCurrentCategories*
The number of categories listed in *currentCategoriesP*.

← *selectedCategoriesPP*
Points to an array of the IDs of categories that the user has selected, or `NULL` if the selection has not changed. If this parameter is defined, your application should update the record so that it belongs to all categories listed in this parameter. This new list should overwrite the current category membership list.

← *numSelectedCategoriesP*
The number of categories listed in the *selectedCategoriesPP* array or `NULL` if *selectedCategoriesPP* is `NULL`.

→ *showEditingStr*
`true` if the list should display an Edit Categories list item, or `false` not to display it.

→ *customEditingStrP*
A string to use as the Edit Categories list item. To use the default string for the current locale, pass `NULL` for this

parameter. This parameter is ignored if *showEditingStr* is `false`.

Returns `true` if the user changes the selected categories or selects new categories to which the record should belong. Returns `false` if the user does not change category membership for this record.

Comments Use this function for forms that use a category selector trigger/pop-up list combination to allow the user to change a category or categories to which the displayed schema database record belongs. Call it when the user taps the category selector trigger. This function handles all aspects of displaying the pop-up list and managing the user selection—It creates the pop-up list using `CatMgrCreateList()`, displays the pop-up list, calls `CatMgrEdit()` if the user selects the Edit Categories item, uses `CatMgrSetTriggerLabel()` to set the selector trigger label to the item the user selected, and then calls `CatMgrFreeList()` to free the list items array.

Your application is responsible for checking the value of *selectedCategoriesPP* upon return and changing the record's category memberships to the new selection. You must then free the selected categories list using the function `CatMgrFreeSelectedCategories()`.

See Also `CatMgrSelectFilter()`, `CatMgrTruncateName()`

CatMgrSelectFilter Function

Purpose Displays a category pop-up list from which the user selects a category whose records should be listed on the form.

Declared In `CatMgr.h`

Prototype

```
Boolean CatMgrSelectFilter (DmOpenRef dbRef,
   const FormType *frm, uint16_t ctlID,
   char *ctlLabelBuffer, uint16_t lstID,
   CategoryID *inCurrentCategoriesP,
   uint32_t numInCurrentCategories,
   CategoryID **outCurrentCategoriesPP,
   uint32_t *numOutCurrentCategoriesP,
   Boolean showEditingStr,
   char *customEditingStrP)
```

Parameters

→ *dbRef*
Open schema database containing the categories to display.

→ *frm*
Pointer to the form that contains the category pop-up list.

→ *ctlID*
Resource ID of the pop-up trigger.

↔ *ctlLabelBuffer*
Stores the string displayed in the pop-up trigger. This string must have a minimum size of `catCategoryNameLength`, but it can be longer. It must exist for the life of the form.

Upon return, contains the updated string displayed in the pop-up trigger. This string is a comma-separated list of the currently selected categories, truncated to fit a particular space.

→ *lstID*
Resource ID of the pop-up list.

→ *inCurrentCategoriesP*
An array of the IDs of categories that are currently displayed.

→ *numInCurrentCategories*
The number of categories specified in *inCurrentCategoriesP*.

← *outCurrentCategoriesPP*
: If the category selection changes, points to an array of the IDs of the newly selected categories. Typically, there is only one category. `NULL` if the selection did not change.

← *numOutCurrentCategoriesP*
: The number of categories in *outCurrentCategoriesPP* or `NULL` if *outCurrentCategoriesPP* is `NULL`.

→ *showEditingStr*
: `true` if the list should display an Edit Categories list item, or `false` not to display it.

→ *customEditingStrP*
: A string to use as the Edit Categories list item. To use the default string for the current locale, pass `NULL` for this parameter. This parameter is ignored if *showEditingStr* is `false`.

Returns `true` if the user changes the selected categories, or `false` otherwise.

Comments Use this function for forms that use a category pop-up list to filter the display. Call it when the user taps the category pop-up trigger. This function handles all aspects of displaying the pop-up list and managing the user selection—It creates the pop-up list using `CatMgrCreateList()`, displays the pop-up list, calls `CatMgrEdit()` if the user selects the Edit Categories item, uses `CatMgrSetTriggerLabel()` to set the trigger label to the item the user selected, and then calls `CatMgrFreeList()` to free the list items array. Your application is responsible for checking the value of *outCurentCategoriesPP* upon return and updating the display or changing the record's category to the new selection. You must then free the selected categories list using the function `CatMgrFreeSelectedCategories()`.

The "All" list item is automatically added. The "Edit Categories" list item is only added if *showEditingStr* is `true`.

If this function returns `true`, the application should update the display so that only the newly selected categories are displayed.

CatMgrSetEditable Function

Purpose Enables or disables the ability to edit a specified category.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrSetEditable (DmOpenRef dbP,
   CategoryID catID, Boolean editable)
```

Parameters

→ *dbP*
Open schema database.

→ *catID*
A category ID.

→ *editable*
`true` to allow the user to modify or delete the specified category; `false` to disallow it.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP* or *catID* is invalid.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID *catID*.

See Also `CatMgrGetEditable()`

CatMgrSetName Function

Purpose Sets the name of the specified category.

Declared In `CatMgr.h`

Prototype

```
status_t CatMgrSetName (DmOpenRef dbP,
   CategoryID catID, const char *nameP)
```

Parameters

→ *dbP*
Open schema database.

→ *catID*
A category ID.

→ *nameP*
A string containing the category's new name. This string is copied into the appropriate portion of the database.

Returns `errNone` upon success or one of the following:

`catmErrInvalidParam`
Could not access *dbP* or *catID* is invalid.

`catmErrNotSchemaDatabase`
dbP is not a schema database.

`catmErrReadOnlyDatabase`
dbP is not open for writing.

`dmErrInvalidParam`
dbP is an invalid database.

`catmErrMemError`
There is not enough memory to complete the operation.

`catmErrCategoryNotFound`
Cannot find the category with ID *catID*.

`catmErrNameAlreadyExists`
The database already contains a category with this name.

See Also `CatMgrGetName()`

CatMgrSetTriggerLabel Function

Purpose Sets the label displayed by the category pop-up trigger for a schema database.

Declared In `CatMgr.h`

Prototype

```
void CatMgrSetTriggerLabel (DmOpenRef dbRef,
   CategoryID categoriesP[],
   uint32_t numCategories, ControlType *ctl,
   char *nameP)
```

Parameters

→ *dbRef*
Open schema database.

→ *categoriesP*
An array of the IDs of categories whose names should be displayed in the trigger.

→ *numCategories*
The number of categories in the *categoriesP* array.

→ *ctl*
Pointer to the control (pop-up or selector trigger) to relabel.

↔ *nameP*
Stores the string displayed in the trigger. This string must have a minimum size of `catCategoryNameLength`, but it can be longer. It must exist for the life of the form.

Upon return, contains the updated string displayed in the trigger. This string is a comma-separated list of the currently selected categories, truncated to fit a particular space.

Returns Nothing.

Comments The `CatMgrSetTriggerLabel()` function looks up the names for the specified categories and then creates a comma-separated list of the category names. It calls the `CatMgrTruncateName()` function to truncate this string to the maximum length. The maximum length varies, depending upon which ROM is installed in the device.

NOTE: This function passes the *nameP* parameter to the `CatMgrTruncateName()` function, which means that the *nameP* value must be modifiable. `CatMgrSetTriggerLabel()` does not make a copy of the string passed, so you must ensure that the string remains valid until the form is closed.

See Also CtlSetLabel()

CatMgrTruncateName Function

Purpose Truncates a list of categories from a schema database so that it's short enough to display. The string is truncated if it's longer than *maxWidth*.

Declared In CatMgr.h

Prototype

```
void CatMgrTruncateName (DmOpenRef dbRef,
    CategoryID categoriesP[],
    uint32_t numCategories, uint16_t maxWidth,
    char *nameP)
```

Parameters

→ *dbRef*
An open schema database.

→ *categoriesP*
An array of the IDs of categories whose names should be displayed in the trigger.

→ *numCategories*
The number of categories in the *categoriesP* array.

→ *maxWidth*
Maximum size, in standard coordinates, of the truncated category string (including ellipsis and commas).

↔ *name*
Stores the string displayed in the pop-up or selector trigger. This string must have a minimum size of catCategoryNameLength, but it can be longer. It must exist for the life of the form.

Upon return, contains the updated string displayed in the trigger. This string is a comma-separated list of the currently selected categories, truncated to fit the space indicated by *maxWidth*.

Returns Nothing.

15
Clipboard Reference

This chapter provides reference material for the clipboard API defined in `Clipboard.h`. It covers:

Clipboard Structures and Types

ClipboardItem Struct

Purpose Private structure that defines an item on the clipboard.

Declared In `Clipboard.h`

Prototype
```
typedef struct ClipboardItemTag ClipboardItem
```

Fields None.

Clipboard Constants

ClipboardFormatType Typedef

Purpose Specifies the type of data to add to the clipboard or retrieve from the clipboard.

Declared In `Clipboard.h`

Prototype
```
typedef Enum8 ClipboardFormatType
```

Constants

`clipboardText`
Textual data. This is the most commonly used clipboard.

`clipboardInk`
Reserved.

`clipboardBitmap`
Bitmap data.

Comments Clipboards for each type of data are separately maintained. That is, if you add a string of text to the clipboard, then add a bitmap, then ask to retrieve a `clipboardText` item from the clipboard, you will receive the string you added before the bitmap; the bitmap does not overwrite textual data and vice versa.

Miscellaneous Constants

Purpose Other constants declared in `Clipboard.h`.

Declared In `Clipboard.h`

Constants

```
#define numClipboardFormats 3
```

The number of possible clipboard formats. See `ClipboardFormatType`.

Clipboard Functions and Macros

ClipboardAddItem Function

Purpose Adds the item passed to the specified clipboard. Replaces the current item (if any) of that type.

Declared In `Clipboard.h`

Prototype

```
void ClipboardAddItem
    (const ClipboardFormatType format,
    const void *ptr, size_t length)
```

Parameters

→ *format*
Text, ink, bitmap, and so on. See `ClipboardFormatType`.

→ *ptr*
Pointer to the item to place on the clipboard.

→ *length*
Size in bytes of the item to place on the clipboard.

Returns Nothing.

Comments The clipboard makes a copy of the data that you pass to this function. Thus, you may free any data that you've passed to the clipboard without destroying the contents of the clipboard. You may also add constant data or stack-based data to the clipboard.

WARNING! You can't add null-terminated strings to the clipboard.

See Also `FldCut()`, `FldCopy()`

ClipboardAppendItem Function

Purpose Appends data to the item on the clipboard.

Declared In `Clipboard.h`

Prototype

```
status_t ClipboardAppendItem
    (const ClipboardFormatType format,
    const void *ptr, size_t length)
```

Parameters

→ *format*
Text, ink, bitmap, and so on. See `ClipboardFormatType`. This function is intended to be used only for the `clipboardText` format.

→ *ptr*
Pointer to the data to append to the item on the clipboard.

→ *length*
Size in bytes of the data to append to the clipboard.

Returns 0 upon success or `memErrNotEnoughSpace` if there is not enough space to append the data to the clipboard.

Comments This function differs from `ClipboardAddItem()` in that it does not overwrite data already on the clipboard. It allows you to create a large text item on the clipboard from several small disjointed pieces. When other applications retrieve the text from the clipboard, it's retrieved as a single unit.

This function simply appends the specified item to the item already on the clipboard without attempting to parse the format. It's assumed that you'll call it several times over a relatively short

interval and that no other application will attempt to retrieve text from the clipboard before your application is finished appending.

ClipboardGetItem Function

Purpose Returns the handle of the contents of the clipboard of a specified type and the length of a clipboard item.

Declared In `Clipboard.h`

Prototype

```
MemHandle ClipboardGetItem
    (const ClipboardFormatType format,
    size_t *length)
```

Parameters

→ *format*
Text, ink, bitmap, and so on. See `ClipboardFormatType`.

← *length*
The length in bytes of the clipboard item is returned here.

Returns Handle of the clipboard item.

Comments The handle returned is a handle to the actual clipboard memory. It is not suitable for passing to any API that modifies memory (such as `FldSetTextHandle()`). Consider this to be read-only access to the data. Copy the contents of the clipboard to your application's own storage as soon as possible and use that reference instead of the handle returned by this function.

Don't free the handle returned by this function; it is freed when a new item is added to the clipboard.

Text retrieved from the clipboard does not have a null terminator. You must use the *length* parameter to determine the length in bytes of the string you've retrieved.

16

Control Reference

This chapter describes the API for user interface controls. It discusses the following topics:

The header file `Control.h` declares the API that this chapter describes. For more information on using controls see Chapter 3, "Working with Controls," on page 41.

Control Structures and Types

ControlAttrType Struct

Purpose Private structure that specifies the control's visible characteristics.

Declared In `Control.h`

Prototype

```
typedef struct ControlAttrTag ControlAttrType
```

Fields None.

ControlType Struct

Purpose Private structure that defines the type and characteristics of a control. The `ControlPtr` is a pointer to a to a `ControlType` structure.

Declared In `Control.h`

Prototype

```
typedef struct ControlType ControlType;
typedef ControlType *ControlPtr
```

Fields None.

GraphicControlType Struct

Purpose Private structure that defines a graphical control. A graphical control is like any other control except that it displays a bitmap in place of the text label.

Declared In `Control.h`

Prototype
```
typedef struct GraphicControlType
GraphicControlType
```

Fields None.

See Also `CtlGetGraphics()`, `CtlIsGraphicControl()`, `CtlNewGraphicControl()`, `CtlSetGraphics()`

SliderControlType Struct

Purpose Private structure that defines a slider control or a feedback slider control.

Declared In `Control.h`

Prototype
```
typedef struct SliderControlType SliderControlType
```

Fields None.

See Also `CtlGetSliderValues()`, `CtlSetSliderValues()`, `CtlNewSliderControl()`

Control Constants

ButtonFrameType Typedef

Purpose Specifies the border style for a button.

Declared In `Control.h`

Prototype
```
typedef Enum8 ButtonFrameType
```

Constants
`noButtonFrame`
The button has no border.

`standardButtonFrame`
: Standard command button rectangular border with rounded corners.

`boldButtonFrame`
: Bold rectangular border with rounded corners.

`rectangleButtonFrame`
: Rectangular border with square corners.

See Also `CtlSetFrameStyle()`

ControlStyleType Typedef

Purpose Specifies the type of the control defined by a `ControlType` structure.

Declared In `Control.h`

Prototype

```
typedef Enum8 ControlStyleType
```

Constants

`buttonCtl`
: Command button. Command buttons display a text label in a box. The `ButtonFrameType` specifies the type of box.

`pushButtonCtl`
: Push button. Selecting a push button inverts its display so that it appears highlighted.

`checkboxCtl`
: Check box. Check boxes display a setting of either on (checked) or off (unchecked)

`popupTriggerCtl`
: Pop-up trigger. Pop-up triggers display a graphic element followed by a text label. They are used to display pop-up lists.

`selectorTriggerCtl`
: Selector trigger. Selector triggers display a text label surrounded by a gray rectangular frame. The control expands or contracts to the width of the new label.

`repeatingButtonCtl`
: Repeating button. Repeating buttons look like buttons; however, a repeating button is repeatedly selected if the user holds the pen on it.

`sliderCtl`
: Slider. Sliders display two bitmaps: one representing the current value (the thumb), and another representing the scale of available values. The user can slide the thumb to the left or the right to change the value.

`feedbackSliderCtl`
: Feedback slider. A feedback slider looks like a slider; however, a feedback slider sends events each time the thumb moves while the pen is still down. A regular slider sends an event only when the user releases the pen.

See Also `CtlGetControlStyle()`

Control Events

ctlEnterEvent

Purpose Sent when `CtlHandleEvent()` receives a `penDownEvent` within the bounds of a control.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct ctlEnter {
   struct ControlType *pControl;
   uint16_t controlID;
   _BA32_PADDING_16(1)
} ctlEnter;
```

Fields

`pControl`
: Pointer to a control structure.

`controlID`
: Resource ID of the control.

Comments In most cases, you should wait for a `ctlSelectEvent` before taking the intended action of the control. When the application receives the `ctlEnterEvent`, it is not guaranteed that the control will become selected.

ctlExitEvent

Purpose Sent by `CtlHandleEvent()` when the stylus had previously entered the bounds of the control, but is lifted outside of the bounds of the control.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct ctlExit {
   struct ControlType *pControl;
   uint16_t controlID;
   _BA32_PADDING_16(1)
} ctlExit;
```

Fields

`pControl`
Pointer to a control structure.

`controlID`
Resource ID of the control.

The following fields in the `EventType` structure are set for this event:

`penDown`
Always `false`.

`screenX`
Draw window-relative position of the pen in standard coordinates (number of coordinates from the left bound of the window).

`screenY`
Draw window-relative position of the pen in coordinates (number of coordinates from the top of the window).

ctlRepeatEvent

Purpose Sent by `CtlHandleEvent()` periodically while the stylus is down in the bounds of a repeating control (such as a repeating button or a feedback slider).

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct ctlRepeat {
   struct ControlType *pControl;
   uint16_t controlID;
   uint16_t value;
   uint32_t time;
} ctlRepeat;
```

Fields

`pControl`
Pointer to a control structure.

`controlID`
Resource ID of the control.

`value`
Current value if the control is a feedback slider.

`time`
System-ticks count when the event is added to the queue.

Comments When `CtlHandleEvent()` receives a `ctlEnterEvent` in a repeating button or a feedback slider control, it sends a `ctlRepeatEvent`. When `CtlHandleEvent()` receives a `ctlRepeatEvent` in a repeating button, it sends another `ctlRepeatEvent` if the pen remains down within the bounds of the control for 1/2 second beyond the last `ctlRepeatEvent`.

When `CtlHandleEvent()` receives a `ctlRepeatEvent` in a feedback slider control, it sends a `ctlRepeatEvent` each time the slider's thumb moves by at least one pixel. Feedback sliders do not send `ctlRepeatEvents` at regular intervals like repeating buttons do.

If you return `true` in response to a `ctlRepeatEvent`, it stops the `ctlRepeatEvent` loop. No further `ctlRepeatEvents` are sent.

ctlSelectEvent

Purpose Sent by `CtlHandleEvent()` when the stylus had previously entered the bounds of the control and is now lifted within the bounds of the control.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct ctlSelect {
   struct ControlType *pControl;
   uint16_t controlID;
   uint16_t value;
   Boolean on;
   uint8_t reserved1;
   _BA32_PADDING_16(1)
} ctlSelect;
```

Fields

`pControl`
Pointer to a control structure.

`controlID`
Resource ID of the control.

`value`
Current value if the control is a slider.

`on`
`true` when the control has a pressed appearance; otherwise, `false`.

`reserved1`
Unused.

Comments Applications should respond to this event to perform whatever action is necessary for this control.

Palm OS® only responds to this event if the control is a pop-up trigger. In that case, it displays the list associated with the pop-up trigger. If you must respond to this event for a pop-up trigger, return `false` to allow the system to perform this default behavior. If you simply want to capture the user selection from the pop-up list, do not respond to `ctlSelectEvent`; respond to `popSelectEvent` instead.

For all other types of controls, it doesn't matter whether you return `true` or `false` from your event handler since the operating system doesn't handle this event.

frmControlPrvRefreshEvent

Purpose Used to update the display of a slider control.

Declared In `Event.h`

Prototype

```
struct ctlRepeat {
   struct ControlType *pControl;
   uint16_t controlID;
   uint16_t value;
   uint32_t time;
} ctlRepeat;
```

Fields

`pControl`
Pointer to a control structure.

`controlID`
Resource ID of the control.

`value`
Current value.

`time`
System-ticks count when the event is added to the queue.

Comments Applications do not respond to this event. It is used by the system to refresh the display.

Control Functions and Macros

CtlDrawCheckboxControl Function

Purpose Draws a check box.

Declared In `Control.h`

Prototype

```
void CtlDrawCheckboxControl (fcoord_t left,
   fcoord_t top, Boolean selected)
```

Parameters

→ *`left`*
The x coordinate where the left side of the check box should be drawn.

→ *`top`*
The y coordinate where the top of the check box should be drawn.

→ *selected*
If `true`, the check should appear in the checkbox. If `false`, just the box should be drawn.

Returns Nothing.

Comments Don't call this function directly. Instead, add a check box to your form using a resource editor and the system draws the check box for you. If you want to dynamically add a check box to your user interface, use `CtlNewControl()`.

CtlDrawControl Function

Purpose Draws a control (and the text or graphic in it) on screen.

Declared In `Control.h`

Prototype
```
void CtlDrawControl (ControlType *controlP)
```

Parameters → *controlP*
Pointer to the control to draw.

Returns Nothing.

Comments The control is drawn only if it is usable. This function sets the control to be visible if necessary.

See Also `CtlSetUsable()`, `CtlShowControl()`, `FrmDrawForm()`

CtlEnabled Function

Purpose Returns `true` if the control responds to the pen.

Declared In `Control.h`

Prototype
```
Boolean CtlEnabled (const ControlType *controlP)
```

Parameters → *controlP*
Pointer to a control.

Returns `true` if the control responds to the pen; `false` if not.

Comments This function provides no indication of whether the control is visible on the screen. A control that doesn't respond to the pen may be visible, and if so, its appearance is no different from controls that

do respond to the pen. You might use such a control to display some state of your application that cannot be modified.

See Also CtlSetEnabled()

CtlEraseControl Function

Purpose Erases a usable and visible control and its frame from the screen.

Declared In `Control.h`

Prototype
```
void CtlEraseControl (ControlType *controlP)
```

Parameters → *controlP*
Pointer to the control to erase.

Returns Nothing.

Comments Don't call this function directly; instead, use `FrmHideObject()`, which calls this function.

CtlGetControlStyle Function

Purpose Returns the type of the control, such as button, slider, and so on.

Declared In `Control.h`

Prototype
```
ControlStyleType CtlGetControlStyle
   (const ControlType *ctlP)
```

Parameters → *ctlP*
Pointer to the control.

Returns One of the `ControlStyleType` constants.

CtlGetFont Function

Purpose Gets the font used when drawing a specified control's label.

Declared In `Control.h`

Prototype
```
FontID CtlGetFont (const ControlType *ctlP)
```

Parameters → *ctlP*
Pointer to the control.

Returns The FontID of the bitmapped font used to draw the control's label.

See Also CtlSetFont()

CtlGetGraphics Function

Purpose Gets the bitmaps displayed in place of a specified control's label.

Declared In Control.h

Prototype

```
void CtlGetGraphics (const ControlType *ctlP,
    DmResourceID *bitmapID,
    DmResourceID *selectedBitmapID)
```

Parameters

→ *ctlP*
Pointer to the control.

← *bitmapID*
Resource ID of the bitmap to display when the graphical control is not selected.

← *selectedBitmapID*
Resource ID of the bitmap to display when the graphical control is selected, if, when selected, the graphical control should show a different bitmap.

Returns Nothing.

Comments If the specified control is not a graphical control—one that displays a bitmap in place of the text label—**bitmapID* and **selectedBitmapID* are set to zero.

This function works with any graphical control, including sliders.

See Also CtlGetFont(), CtlSetGraphics()

CtlGetLabel Function

Purpose Returns a control's text label.

Declared In Control.h

Prototype

```
const char *CtlGetLabel
    (const ControlType *controlP)
```

Parameters

→ *controlP*
Pointer to the control.

Returns A pointer to a null-terminated string containing the control's label.

Comments Make sure that *controlP* is not a graphical control or a slider control. The graphical control and slider control structures do not contain a text label field.

TIP: Label resources are not controls. To retrieve the text of a label resource, use `FrmGetLabel()`.

See Also `CtlSetLabel()`

CtlGetSliderValues Function

Purpose Returns current values used by a slider control.

Declared In `Control.h`

Prototype

```
void CtlGetSliderValues (const ControlType *ctlP,
   uint16_t *minValueP, uint16_t *maxValueP,
   uint16_t *pageSizeP, uint16_t *valueP)
```

Parameters

→ *ctlP*
Pointer to a slider.

← *minValueP*
The value of the slider when the thumb is all the way to the left. Pass `NULL` if you don't want to retrieve this value.

← *maxValueP*
The value of the slider when the thumb is all the way to the right. Pass `NULL` if you don't want to retrieve this value.

← *pageSizeP*
The amount by which to increase or decrease the slider value when the user taps to the right or left of the thumb. Pass `NULL` if you don't want to retrieve this value.

← *valueP*
The current value represented by the slider. Pass `NULL` if you don't want to retrieve this value.

Returns Nothing.

Comments If *ctlP* is not a slider or a feedback slider, this function immediately returns.

See Also CtlSetSliderValues()

CtlGetValue Function

Purpose Returns the current value of the specified control.

Declared In Control.h

Prototype
```
int16_t CtlGetValue (const ControlType *controlP)
```

Parameters

→ *controlP*
Pointer to a control.

Returns The current value of the control. For most controls the return value is either 0 (off) or 1 (on). For sliders, this function returns the current value represented by the slider.

Don't use this function on selector triggers or pop-up triggers.

See Also CtlSetValue(), FrmGetControlGroupSelection(), FrmSetControlGroupSelection(), FrmGetControlValue(), FrmSetControlValue()

CtlHandleEvent Function

Purpose Handles events in the specified control.

Declared In Control.h

Prototype
```
Boolean CtlHandleEvent (ControlType *controlP,
    EventType *pEvent)
```

Parameters

→ *controlP*
Pointer to a control.

→ *pEvent*
Pointer to an EventType structure.

Returns true if an event is handled by this function or false otherwise.

Comments The control must be usable, visible, and respond to the pen for this function to handle the event.

Table 16.1 lists the events that this function responds to. For update-based windows, the controls do not redraw immediately. Instead, the region corresponding to the control is invalidated and redrawn on the next drawing update cycle.

Table 16.1 CtlHandleEvent() actions

When CtlHandleEvent() receives...	CtlHandleEvent() performs these actions...
`ctlEnterEvent`	Begins tracking the pen. Redraws the control as necessary as either selected or deselected, depending on its previous state. For repeating controls, adds a `ctlRepeatEvent` to the queue.
`ctlRepeatEvent`	Tracks the pen and continues to send `ctlRepeatEvents` at intervals.
`frmControlPrvRefreshEvent`	Tracks the pen and updates the display.
`penDownEvent`	Checks if the pen position is within the bounds of the control. If it is, a `ctlEnterEvent` is added to the event queue and the function exits.
`penMoveEvent`	Redraws the control as necessary as either selected or deselected, depending on its previous state and if the pen is still within the bounds of the control. For repeating controls, adds a `ctlRepeatEvent` to the queue.
`penUpEvent`	If the pen is within the bounds of the control, adds a `ctlSelectEvent` to the event queue for all controls but repeating buttons. If not, adds `ctlExitEvent`. Redraws the control as selected or deselected as necessary.

CtlHideControl Function

Purpose Sets a control to not usable and erases the control from the screen.

Declared In `Control.h`

Prototype
```
void CtlHideControl (ControlType *controlP)
```

Parameters → *controlP*
Pointer to the control to hide.

Returns Nothing.

Comments Don't call this function directly; instead, use `FrmHideObject()`, which performs the same function and works for all user interface elements.

A control that is not usable doesn't draw and doesn't respond to the pen.

This function is the same as `CtlEraseControl()` except that it also makes the control both unusable and not visible. `CtlEraseControl()` only makes the control not visible.

TIP: In Palm OS Cobalt, hiding an element means filling its region with the color set for `UIFormFill`. This is true regardless of the type of window used for the form (legacy, transitional, or update-based).

See Also `CtlShowControl()`

CtlHitControl Function

Purpose Simulates tapping a control. This function adds a `ctlSelectEvent` to the event queue.

Declared In `Control.h`

Prototype
```
void CtlHitControl (const ControlType *controlP)
```

Parameters → *controlP*
Pointer to a control.

Returns Nothing.

Comments Useful for testing.

CtlIsGraphicControl Function

Purpose Returns whether or not a control is a graphical control.

Declared In `Control.h`

Prototype

```
Boolean CtlIsGraphicControl (ControlType *ctlP)
```

Parameters

→ *ctlP*
Pointer to the control.

Returns `true` if the control indicated by *ctlP* is a graphical control, a slider, or a feedback slider. Otherwise, it returns `false`.

CtlNewControl Function

Purpose Creates a new control dynamically and installs it in the specified form.

Declared In `Control.h`

Prototype

```
ControlType *CtlNewControl (void **formPP,
   uint16_t ID, ControlStyleType style,
   const char *textP, Coord x, Coord y,
   Coord width, Coord height, FontID font,
   uint8_t group, Boolean leftAnchor)
```

Parameters

↔ *formPP*
Pointer to the pointer to the form in which the new control is installed. This value is not a handle; that is, the *formPP* value may change if the structure moves in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *ID*
Symbolic ID of the control.

→ *style*
A `ControlStyleType` value specifying the kind of control to create: button, push button, repeating button, check box, pop-up trigger, or selector trigger. To create a graphical control or slider control dynamically, use `CtlNewGraphicControl()` or `CtlNewSliderControl()`, respectively.

→ *textP*
Pointer to the control's label text. If *textP* is `NULL`, the control has no label. Because the contents of this pointer are copied into their own buffer, you can free the *textP* pointer any time after the `CtlNewControl()` function returns. The buffer into which this string is copied is freed automatically when you remove the control from the form or delete the form.

→ *x*
Horizontal coordinate of the upper-left corner of the control's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the control's boundaries, relative to the window in which it appears.

→ *width*
Width of the control, expressed in standard coordinates. If the value of either of the *width* or *height* parameters is 0, the control is sized automatically as necessary to display the text passed as the value of the *textP* parameter.

→ *height*
Height of the control, expressed in standard coordinates. If the value of either of the *width* or *height* parameters is 0, the control is sized automatically as necessary to display the text passed as the value of the *textP* parameter.

→ *font*
Bitmapped font used to draw the control's label. See `FontID`.

→ *group*
Group ID of a push button or a check box that is part of an exclusive group.

→ *leftAnchor*
`true` specifies that the left bound of this control is fixed. This attribute is used by controls that resize dynamically in response to label text changes. It does not affect how the control is resized when the form that contains it is resized.

Returns A pointer to the new control.

See Also CtlValidatePointer(), FrmRemoveObject()

CtlNewGraphicControl Function

Purpose Creates a new graphical control dynamically and installs it in the specified form.

Declared In `Control.h`

Prototype

```
GraphicControlType *CtlNewGraphicControl
    (void **formPP, uint16_t ID,
    ControlStyleType style, DmOpenRef database,
    DmResourceID bitmapID,
    DmResourceID selectedBitmapID, Coord x,
    Coord y, Coord width, Coord height,
    uint8_t group, Boolean leftAnchor)
```

Parameters

↔ *formPP*
Pointer to the pointer to the form in which the new control is installed. This value is not a handle; that is, the *formPP* value may change if the structure moves in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *ID*
Symbolic ID of the control.

→ *style*
A ControlStyleType value specifying the kind of control to create: button, push button, pop-up trigger, repeating button, or selector trigger. Graphical controls cannot be check boxes.

→ *database*
Open database containing the *bitmapID* and *selectedBitmapID* resources.

→ *bitmapID*
Resource ID of the bitmap to display on the control.

→ *selectedBitmapID*
Resource ID of the bitmap to display when the control is selected, if different from *bitmapID*.

→ *x*
Horizontal coordinate of the upper-left corner of the control's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the control's boundaries, relative to the window in which it appears.

→ *width*
Width of the control, expressed in standard coordinates.

→ *height*
Height of the control, expressed in standard coordinates.

→ *group*
Group ID of a push button that is part of an exclusive group.

→ *leftAnchor*
`true` specifies that the left bound of this control is fixed.

Returns A pointer to the new graphical control.

See Also CtlNewSliderControl(), CtlNewControl(), CtlValidatePointer(), FrmRemoveObject()

CtlNewSliderControl Function

Purpose Creates a new slider or feedback slider dynamically and installs it in the specified form.

Declared In `Control.h`

Prototype

```
SliderControlType *CtlNewSliderControl
   (void **formPP, uint16_t ID,
   ControlStyleType style, DmOpenRef database,
   DmResourceID thumbID,DmResourceID backgroundID,
   Coord x, Coord y, Coord width, Coord height,
   uint16_t minValue, uint16_t maxValue,
   uint16_t pageSize, uint16_t value)
```

Parameters

↔ *formPP*
Pointer to the pointer to the form in which the new control is installed. This value is not a handle; that is, the *formPP* value may change if the structure moves in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *ID*
Symbolic ID of the slider.

→ *style*
Either `sliderCtl` or `feedbackSliderCtl`. See `ControlStyleType`.

→ *database*
Open database containing the *thumbID* and *backgroundID* resources. Pass `NULL` for this parameter if *thumbID* and *backgroundID* are both `NULL`.

→ *thumbID*
Resource ID of the bitmap to display as the slider thumb. The slider thumb is the knob that the user can drag to change the slider's value. To use the default thumb bitmap, pass `NULL` for this parameter.

→ *backgroundID*
Resource ID of the bitmap to display as the slider background. To use the default background bitmap, pass `NULL` for this parameter.

→ *x*
Horizontal coordinate of the upper-left corner of the slider's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the slider's boundaries, relative to the window in which it appears.

→ *width*
Width of the slider, expressed in standard coordinates.

→ *height*
Height of the slider, expressed in standard coordinates.

→ *minValue*
Value of the slider when its thumb is all the way to the left.

→ *maxValue*
Value of the slider when its thumb is all the way to the right.

→ *pageSize*
Amount by which to increase or decrease the slider's value when the user clicks to the right or left of the thumb.

→ *value*
The initial value to display in the slider.

Returns A pointer to the new slider control.

See Also CtlNewGraphicControl(), CtlNewControl(), CtlValidatePointer(), FrmRemoveObject()

CtlSetEnabled Function

Purpose Sets a control as enabled or disabled. Disabled controls do not respond to the pen.

Declared In `Control.h`

Prototype

```
void CtlSetEnabled (ControlType *controlP,
   Boolean usable)
```

Parameters

→ *controlP*
Pointer to a control.

→ *usable*
`true` to enable the control; `false` to disable the control.

Returns Nothing.

Comments If you disable a visible control, the control is still displayed, and its appearance is no different from controls that do respond to the pen. You might use such a control to inform your users of some state of your application that cannot be modified.

See Also `CtlEnabled()`

CtlSetFont Function

Purpose Sets the font to use when drawing the control's label.

Declared In `Control.h`

Prototype

```
void CtlSetFont (ControlType *ctlP, FontID fontID)
```

Parameters

→ *ctlP*
Pointer to the control.

→ *fontID*
The `FontID` of the bitmapped font to use when drawing the control's label.

Returns Nothing.

See Also CtlGetFont()

CtlSetFrameStyle Function

Purpose Sets the frame style for a button control.

Declared In Control.h

Prototype
```
void CtlSetFrameStyle (ControlType *ctlP,
    ButtonFrameType frameStyle)
```

Parameters

→ *controlP*
Pointer to a button control.

→ *frameStyle*
Frame style to be applied to the button. Supply one of the values from the ButtonFrameType enum.

Returns Nothing.

CtlSetGraphics Function

Purpose Sets the bitmaps for a graphical control and redraws the control if it is visible.

Declared In Control.h

Prototype
```
void CtlSetGraphics (ControlType *ctlP,
    DmOpenRef database, DmResourceID newBitmapID,
    DmResourceID newSelectedBitmapID)
```

Parameters

→ *ctlP*
Pointer to a graphical control.

→ *database*
Open resource database containing the *newBitmapID* and *newSelectedBitmapID* resources.

→ *newBitmapID*
Resource ID of a new bitmap to display on the control, or NULL to use the current bitmap.

→ *newSelectedBitmapID*
Resource ID of a new bitmap to display when the control is selected, or `NULL` to use the current selected bitmap.

Returns Nothing.

Comments If *ctlP* is not a graphical control, this function immediately returns.

See Also `CtlGetGraphics()`

CtlSetLabel Function

Purpose Sets the current label for the specified control and redraws the control if it is visible.

Declared In `Control.h`

Prototype

```
void CtlSetLabel (ControlType *controlP,
    const char *newLabel)
```

Parameters

→ *controlP*
Pointer to a control.

→ *newLabel*
Pointer to the new text label. Must be a null-terminated string.

Returns Nothing.

Comments This function resizes the width of the control to the size of the new label.

This function stores the *newLabel* pointer in the control's data structure. It doesn't make a copy of the string that is passed in. Therefore, if you use `CtlSetLabel()`, you must manage the string yourself. You must ensure that it persists for as long as it is being displayed (that is, for as long as the control is displayed or until you call `CtlSetLabel()` with a new string), and you must free the string after it is no longer in use (typically after the form containing the control is freed).

If you never use `CtlSetLabel()`, you do not need to worry about freeing a control's label.

Make sure that *controlP* is not a graphical control or a slider control. The graphical controls and slider control structures do not

contain a text label field, so attempting to set one will crash your application.

TIP: Label resources are not controls. To change the text of a label resource, use `FrmCopyLabel()`.

See Also CtlGetLabel()

CtlSetLeftAnchor Function

Purpose Causes a control's left bound to be fixed or to float.

Declared In `Control.h`

Prototype

```
void CtlSetLeftAnchor (ControlType *ctlP,
    Boolean leftAnchor)
```

Parameters

→ *ctlP*
Pointer to the control.

→ *leftAnchor*
A value of `true` causes the left bound of the control to be fixed.

Returns Nothing.

Comments Used by controls that expand and shrink their width when the label is changed.

CtlSetSliderValues Function

Purpose Changes a slider control's values and redraws the slider if it is visible.

Declared In `Control.h`

Prototype

```
void CtlSetSliderValues (ControlType *ctlP,
    const uint16_t *minValueP,
    const uint16_t *maxValueP,
    const uint16_t *pageSizeP,
    const uint16_t *valueP)
```

Parameters

→ *ctlP*
Pointer to an inactive slider or feedback slider control.

→ *minValueP*
Pointer to a new value to use for the slider's minimum or `NULL` if you don't want to change this value. The minimum is the value of the slider when the thumb is all the way to the left.

→ *maxValueP*
Pointer to a new value to use for the slider's maximum, or `NULL` if you don't want to change this value. The maximum is the value of the slider when the thumb is all the way to the right.

→ *pageSizeP*
Pointer to a new value to use for the slider's page size, or `NULL` if you don't want to change this value. The page size is the amount by which to increase or decrease the slider value when the user taps to the right or left of the thumb.

→ *valueP*
Pointer to a new value to use for the current value, or `NULL` if you don't want to change this value.

Returns Nothing.

Comments The control's style must be `sliderCtl` or `feedbackSliderCtl`, and it must not be currently tracking the pen. If the slider is currently tracking the pen, use `CtlSetValue()` to set the `value` field.

See Also `CtlGetSliderValues()`

CtlSetUsable Function

Purpose Sets a control to usable or not usable.

Declared In `Control.h`

Prototype
```
void CtlSetUsable (ControlType *controlP,
    Boolean usable)
```

Parameters → *controlP*
Pointer to a control.

→ *usable*
`true` to have the control be usable; `false` to have the control be not usable.

Returns Nothing.

Comments A control that is not usable doesn't draw and doesn't respond to the pen.

This function doesn't usually update the control.

See Also `CtlEraseControl()`,`CtlHideControl()`,`CtlShowControl()`

CtlSetValue Function

Purpose Sets the current value of the specified control. If the control is visible, it's redrawn.

Declared In `Control.h`

Prototype

```
void CtlSetValue (ControlType *controlP,
    int16_t newValue)
```

Parameters

→ *controlP*
Pointer to a control.

→ *newValue*
New value to set for the control. For sliders, specify a value between the slider's minimum and maximum. For graphical controls, push buttons, or check boxes, specify 0 for off, nonzero for on.

Returns Nothing.

Comments Don't use this function on pop-up triggers or selector triggers.

See Also `CtlGetValue()`, `FrmGetControlGroupSelection()`, `FrmSetControlGroupSelection()`,`FrmGetControlValue()`, `FrmSetControlValue()`

CtlShowControl Function

Purpose Sets a control's `usable` attribute to `true` and draws the control on the screen.

Declared In `Control.h`

Prototype

```
void CtlShowControl (ControlType *controlP)
```

Parameters

→ *controlP*
Pointer to a control.

Returns Nothing.

Comments Don't use this function directly; instead use `FrmShowObject()`, which does the same thing.

See Also `CtlHideControl()`

CtlValidatePointer Function

Purpose Returns `true` if the specified pointer references a valid control.

Declared In `Control.h`

Prototype

```
Boolean CtlValidatePointer
    (const ControlType *controlP)
```

Parameters

→ *controlP*
Pointer to a control.

Returns `true` when passed a valid pointer to a control; otherwise, `false`.

Comments For debugging purposes; do not include this function in commercial products. In debug builds, this function displays a dialog and waits for the debugger when an error occurs.

See Also `FrmValidatePtr()`, `WinValidateHandle()`

17

Date and Time Selection Reference

The Palm OS® UI Library provides two system resources for accepting date and time input values. These resources are dialogs that contain gadgets for entering dates and times. The Palm OS UI Library also provides routines to manage the interaction with these resources. This chapter describes those functions.

The API described in this chapter is declared in the header files `Day.h`, `SelDay.h`, `SelTime.h`, and `SelTimeZone.h`.

Date and Time Selection Structures and Types

DaySelectorType Struct

Purpose Private structure that represents a day selector gadget. The `DaySelectorPtr` defines a pointer to the `DaySelectorType` structure.

Declared In `Day.h`

Prototype

```
typedef struct DaySelectorType DaySelectorType;
typedef DaySelectorType *DaySelectorPtr
```

Fields None.

HMSTime Struct

Purpose Represents a time of day.

Declared In `SelTime.h`

Prototype

```
typedef struct {
   uint8_t hours;
   uint8_t minutes;
   uint8_t seconds;
   uint8_t reserved;
} HMSTime
```

Fields

`hours`
The number of hours.

`minutes`
The number of minutes.

`seconds`
The number of seconds.

`reserved`
Not used.

Date and Time Selection Constants

Miscellaneous Constants

Purpose Constants declared in `SelDay.h`.

Declared In `SelDay.h`

Constants

`#define daySelectorMaxYear lastYear`
The furthest year in the future that the day selector gadget can display.

`#define daySelectorMinYear firstYear`
The furthest year in the past that the day selector gadget can display.

SelectDayType Typedef

Purpose Specifies how a user can select a date.

Declared In `Day.h`

Prototype

```
typedef Enum8 SelectDayType
```

Constants

`selectDayByDay`
The user can select individual days.

`selectDayByWeek`
The user can only select a week at a time.

`selectDayByMonth`
The user can only select a month at a time.

See Also `SelectDay()`

SelectTimeZoneDisplayType Typedef

Purpose Controls what is displayed in the dialog displayed by `SelectTimeZone()`.

Declared In `SelTimeZone.h`

Prototype

```
typedef Enum8 SelectTimeZoneDisplayType
```

Constants

`showNoTime = 0`
The dialog does not show the time of day.

`showCurrentAndNewTimeZone`
The dialog shows the time of day in the current time zone as well as the newly selected time zone.

`showDeviceTimeZone`
The dialog shows the device's time of day in the device's default time zone. This setting is used by the Setup application.

Date and Time Selection Events

daySelectEvent

Purpose Sent when the pen touches and is lifted from a day number in the day selector gadget.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct daySelect {
   struct DaySelectorType *pSelector;
   int16_t selection;
   Boolean useThisDate;
   uint8_t reserved1;
} daySelect;
```

Fields

`pSelector`
Pointer to a day selector structure.

`selection`
Not used.

`useThisDate`
Set to `true` to automatically use the selected date.

`reserved1`
Unused.

Date and Time Selection Functions and Macros

DayDrawDays Function

Purpose Draw only the days-of-the-month portion of a day selector gadget.

Declared In `Day.h`

Prototype `void DayDrawDays (const DaySelectorPtr` *`selectorP`*`)`

Parameters → *`selectorP`*
Pointer to the day selector gadget to draw.

Returns Nothing.

Comments This function is used when the year or month changes. Only drawing the portion of the control that presents the days of the month avoids the flicker that would occur if the week titles were redrawn.

Use this function only if you want to use the day selector gadget in a dialog that is different than the one used in the Date Book application.

See Also DayDrawDaySelector(), SelectDay()

DayDrawDaySelector Function

Purpose Draws a day selector gadget on screen.

Declared In Day.h

Prototype

```
void DayDrawDaySelector
    (const DaySelectorPtr selectorP)
```

Parameters

→ *selectorP*
Pointer to the day selector gadget to draw.

Returns Nothing.

Comments The gadget is drawn only if it is usable.

Use this function only if you want to use the day selector gadget in a dialog that is different than the one used in the Date Book application.

See Also DayDrawDays(), SelectDay()

DayHandleEvent Function

Purpose Handles an event in the specified day selector gadget.

Declared In `Day.h`

Prototype

```
Boolean DayHandleEvent
    (const DaySelectorPtr selectorP,
    const EventType *pEvent)
```

Parameters

→ *selectorP*
Pointer to day selector gadget.

→ *pEvent*
Pointer to an `EventType` structure.

Returns `true` if the event was handled or `false` if it was not.

Comments This function handles events in a similar fashion as `CtlHandleEvent()`. It tracks the pen while the pen is down within the bounds of the day selector gadget. If the pen is released within the gadget's bounds, it adds a `daySelectEvent` with information on whether to use the date. If the pen is released outside of the gadget's bounds, no event is posted.

SelectDay Function

Purpose Displays a dialog showing a date; allows user to select a different date.

Declared In `SelDay.h`

Prototype

```
Boolean SelectDay(const SelectDayType selectDayBy,
    int16_t *month, int16_t *day, int16_t *year,
    const char *title)
```

Parameters

→ *selectDayBy*
The method by which the user should choose the day. See `SelectDayType`.

↔ *month, day, year*
Month, day, and year selected.

→ *title*
String title for the dialog.

Returns `true` if the OK button was pressed. If `true`, *month*, *day*, and *year* contain the new date.

Comments The dialog displayed is the same one used by the Date Book application when the Go To button is tapped.

See Also DayDrawDays(), DayDrawDaySelector()

SelectOneTime Function

Purpose Displays a dialog showing the time and allows the user to select a different time.

Declared In SelTime.h

Prototype

```
Boolean SelectOneTime (int16_t *hour,
   int16_t *minute, const char *titleP)
```

Parameters

↔ *hour*
The hour selected in the form.

↔ *minute*
The minute selected in the form.

→ *titleP*
A pointer to a string to display as the title. Doesn't change as the function executes.

Returns `true` if the user selects OK and `false` otherwise. If `true` is returned, the values in `hour` and `minute` have probably been changed.

Comments Use this function instead of SelectTime() if you want to display a dialog that specifies a single point in time, not a range of time from start to end.

SelectTime Function

Purpose Displays a dialog showing a start and end time. Allows the user to select a different time.

Declared In `SelTime.h`

Prototype

```
Boolean SelectTime (TimeType *startTimeP,
    TimeType *EndTimeP, const Boolean untimed,
    const char *titleP, int16_t startOfDay,
    int16_t endOfDay, int16_t startOfDisplay)
```

Parameters

↔ *startTimeP, EndTimeP*
: Pointers to values of type `TimeType`. Pass initial values to display in these two parameters. If the user makes a selection and taps the OK button, the selected values are returned here.

→ *untimed*
: Pass in `true` to indicate that no time is selected. If the user later sets the time to no time then *startTimeP* and *EndTimeP* are both set to the constant `noTime`.

→ *titleP*
: A pointer to a string to display as the title.

→ *startOfDay*
: The hour that the hour list displays at its top. To see earlier hours, the user can scroll the list up. The value must be between 0 to 12, inclusive.

→ *endOfDay*
: The hour used when the "All Day" button is selected.

→ *startOfDisplay*
: First hour initially visible if the user scrolls the list up to the top.

Returns `true` if the user taps OK or `false` otherwise.

Comments The dialog displayed by this function is intended for scheduling appointments. To select an individual time, use `SelectOneTime()` instead.

See Also `SelectDay()`

SelectTimeZone Function

Purpose Displays a dialog that allows the user to select a time zone.

Declared In `SelTimeZone.h`

Prototype

```
Boolean SelectTimeZone (char *ioZoneIDP,
    const char *titleP,
    SelectTimeZoneDisplayType displayOption)
```

Parameters

↔ *ioZoneIDP*
A pointer to the initially selected time zone. This string should be `TZNAME_MAX` bytes long. Upon return, contains the user's selection.

→ *titleP*
A string to use as the title for the dialog. Pass `NULL` to use the default title, which is "Set Time Zone" in the US English locale.

→ *displayOption*
See SelectTimeZoneDisplayType.

Returns `true` if the user tapped the OK button in the dialog to change the time zone, or `false` if the user tapped the Cancel button.

Comments This function displays the time zone dialog used by the Date and Time panel in the Preferences application.

Example The following example shows how to use the current time zone as the initial value for *ioZoneIDP*:

```
char zoneID[TZNAME_MAX];
Boolean change;

//get the current time zone
gettimmezone(zoneID, TZNAME_MAX);

change = SelectTimeZone(zoneID, NULL,
   showCurrentAndNewTimeZone);

//set the device time zone to the newly selected time zone.
if (change) {
   settimezone(timeZone);
}
```

SelectTimeZoneV50 Function

Purpose Displays a dialog that allows the user to select a different time zone.

Declared In `SelTimeZone.h`

Prototype

```
Boolean SelectTimeZoneV50 (int16_t *ioTimeZoneP,
   LmLocaleType *ioLocaleInTimeZoneP,
   const char *titleP, Boolean showTimes,
   Boolean anyLocale)
```

Parameters

↔ *ioTimeZoneP*
: A pointer to the time zone, given as minutes east of Greenwich Mean Time (GMT). The initial value is used as the initial selection in the form. Upon return, this parameter contains a pointer to the new time zone that the user selected.

↔ *ioLocaleInTimeZoneP*
: A pointer to a locale (see `LmLocaleType`) that identifies the time zone country. This parameter is used for countries that share a time zone, such as Canada and Chile.

  If the time zone specified by *ioTimeZoneP* is specific to a particular country, you do not have to initialize this parameter. Instead, set the *anyLocale* parameter to `true` to have this parameter ignored upon entry.

→ *titleP*
: A string to use as the title for the dialog. Pass `NULL` to use the default title, which is "Set Time Zone" in the US English locale.

→ *showTimes*
: If `true`, the dialog shows the correct times in both the current and newly selected time zones. If `false`, the dialog doesn't show the current time. Using `false` provides a larger area for the list of time zones.

→ *anyLocale*
: If `true`, ignore *ioLocaleInTimeZoneP* upon entry.

Returns `true` if the user tapped the OK button in the dialog to change the time zone, or `false` if the user tapped the Cancel button.

Comments This function displays the time zone dialog used by the Date and Time panel in the Preferences application.

The time zones displayed in the dialog are listed by country. For this reason, if the time zone specified by *ioTimeZoneP* is shared by

several countries, you need to supply a value for *ioLocaleInTimeZoneP* to identify which country should be selected when the list is first displayed. You can use the constant `lmAnyLanguage` as the value for the language field of the structure pointed to by this parameter.

If you don't care which value is initially selected, pass `true` for the *anyLocale* parameter. In this case, the first country that matches the GMT offset given in *ioTimeZoneP* is selected.

Example

The following example shows how to use the time zone preference as the initial value for *ioTimeZoneP*.

```
int16_t timeZone =
  (int16_t)PrefGetPreference(prefTimeZone);
CountryType timeZoneCountry = (CountryType)
  PrefGetPreference(prefTimeZoneCountry);
LmLocaleType timeZoneLocale;
Boolean change;

timeZoneLocale.country = timeZoneCountry;
timeZoneLocale.language = lmAnyLanguage;
change = SelectTimeZoneV50(&timeZone,
  &timeZoneLocale, NULL, true, false);
if (change) {
   PrefSetPreference(prefTimeZone, timeZone);
   PrefSetPreference(prefTimeZoneCountry, timeZoneCountry);
}
```

Compatibility

This function is provided for backward compatibility only. Use `SelectTimeZone()` instead.

18

Field Reference

This chapter provides the following information about text fields:

The header file `Field.h` declares the API that this chapter describes. For more information on fields, see the section Chapter 5, "Displaying Text," on page 79.

Field Structures and Types

FieldAttrType Struct

Purpose Defines the visible characteristics of the field.

Declared In `Field.h`

Prototype

```
typedef struct FieldAttrTag {
   uint16_t usable :1;
   uint16_t visible :1;
   uint16_t editable :1;
   uint16_t singleLine :1;
   uint16_t hasFocus :1;
   uint16_t dynamicSize :1;
   uint16_t insPtVisible :1;
   uint16_t dirty :1;
   uint16_t underlined :2;
   uint16_t justification :2;
```

```
    uint16_t autoShift :1;
    uint16_t hasScrollBar :1;
    uint16_t numeric :1;
    uint16_t unused :1;
} FieldAttrType;
typedef FieldAttrType *FieldAttrPtr
```

Fields

`usable`
: If not set, the field is not considered part of the current interface of the application, and it doesn't appear on screen. The function `FldSetUsable()` sets this value, but it is better to use `FrmShowObject()` and `FrmHideObject()`.

`visible`
: Set or cleared internally when the field is drawn with `FldDrawField()` or `FrmShowObject()`, and erased with `FldEraseField()` or `FrmHideObject()`.

`editable`
: If not set, the field doesn't accept text input or editing commands and the insertion point cannot be positioned with the pen. The text can still be selected and copied.

`singleLine`
: If set, the field is a single line of text high and the text does not wrap when it exceeds the width of the field. If not set, the text wraps to fill multiple lines. You set this value only when creating the field resource in a resource file. You cannot change the field type from single-line to multi-line, or vice versa, by changing the value of this flag.

`hasFocus`
: Set internally when the field has the current focus. The blinking insertion point appears in the field that has the current focus. Use the function `FrmSetFocus()` to set this value. Do *not* use `FldSetAttributes()`.

`dynamicSize`
: If set, a `fldHeightChangedEvent` is generated whenever the number of lines needs to increase or decrease. Your application needs to respond to this event by adjusting the size of the field's bounding box. If not set, the text wraps to fill more (or less) lines as required, but the event is not generated. *Note that this bit does not cause the field to change size*

automatically; your application must respond to the `fldHeightChangedEvent` and resize the field itself.

This attribute should not be set for single-line fields.

`insPtVisible`
If set, the insertion point is scrolled into view. This attribute is set and cleared internally. Do *not* try to set it.

`dirty`
If set, the user has modified the field. The functions `FldDirty()` and `FldSetDirty()` retrieve and set this field's value. Do *not* try to set this value with `FldSetAttributes()`.

`underlined`
If set, each line of the field, including blank lines, is underlined. Possible values are defined by the `UnderlineModeType`.

Editable text fields generally use `grayUnderline` or `thinUnderline` as the value.

`justification`
Specifies the text alignment. See `JustificationType`.

`autoShift`
If set, auto-shift rules are applied.

`hasScrollBar`
If set, the field has a scroll bar. The system sends more frequent `fldChangedEvent`s so the application can adjust the height appropriately.

`numeric`
If set, only the characters 0 through 9 and associated separators are allowed to be entered in the field. The associated separators are the thousands separator and the decimal character. The values of these two characters depend on the settings in the Formats preferences panel.

`unused`
Reserved for future use.

See Also FldGetAttributes(), FldSetAttributes()

FieldType Struct

Purpose An internal structure that represents a field. The `FieldPtr` type defines a pointer to a `FieldType` structure.

Declared In `Field.h`

Prototype

```
typedef struct FieldType FieldType;
typedef FieldType *FieldPtr
```

Fields None.

Field Constants

JustificationType Typedef

Purpose Specifies how the text field justifies the text it contains. See `FieldAttrType`.

Declared In `Field.h`

Prototype

```
typedef Enum8 JustificationType
```

Constants

`leftAlign`
: Text is aligned to the left with a ragged right.

`centerAlign`
: Text is centered. This is not currently used.

`rightAlign`
: Text is aligned along the right with a ragged left. Right justification works with both single and multi-line text fields.

Miscellaneous Constants

Purpose Constants defined in `Field.h`.

Declared In `Field.h`

Constants

`#define maxFieldLines 11`
The maximum number of lines a field can expand to in a standard sized form if the field is using the standard font.

Don't rely on this constant in Palm OS® Cobalt. If a form is dynamically sized, a field may display more or less than 11 lines in the standard font.

`#define maxFieldTextLen SIZE_MAX`
The maximum number of characters any text field can hold. Note that each field can set a maximum number of characters up to this size. See `FldGetMaxChars()` and `FldSetMaxChars()`.

`#define undoBufferSize 100`
The number of bytes that can be held by the undo buffer. See `FldUndo()`.

Field Events

fldChangedEvent

Purpose Sent when the text of a field has or might have been scrolled.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct fldChanged {
   struct FieldType *pField;
   uint16_t fieldID;
   _BA32_PADDING_16(1)
} fldChanged;
```

Fields

`pField`
Pointer to a field structure.

`fieldID`
Resource ID of the field.

Comments Note that this event is not sent every time a field's value changes. It is sent only when the field might have been scrolled. Applications respond to this event to update the associated scroll bar.

fldEnterEvent

Purpose Sent when a `penDownEvent` occurs within the bounds of a field

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct fldEnter {
   struct FieldType *pField;
   uint16_t fieldID;
   _BA32_PADDING_16(1)
} fldEnter;
```

Fields

`pField`
Pointer to a field structure.

`fieldID`
Resource ID of the field.

fldHeightChangedEvent

Purpose Sent when the internal number of lines in the field changes.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct fldHeightChanged {
   struct FieldType *pField;
   uint16_t fieldID;
   Coord newHeight;
   size_t currentPos;
} fldHeightChanged;
```

Fields

`pField`
Pointer to a field structure.

`fieldID`
Resource ID of the field.

`newHeight`
New visible height of the field, in number of lines.

`currentPos`
Current position of the insertion point.

Comments This event is a notification to your application that the height of a field needs to change.

NOTE: This event is only sent for fields that have the dynamic size attribute set. If a field should resize because its form has been resized, the application receives the `winResizedEvent` instead.

If the field is contained in a table, the table's code handles the `fldHeightChangedEvent`. If the field is directly on a form, your application code should handle the `fldHeightChangedEvent` itself. The form code does not handle the event for you. Call `FldSetBounds()` to set the new height of the field, and, if necessary, reposition the other elements on the form to make room. Return `false` to have `FldHandleEvent()` adjust the insertion point for you.

Example This example shows how to change the height of the field.

```
case fldHeightChangedEvent:
   if (eventP->data.fldChanged.fieldID == MyField) {
      RectangleType fldRect;
      FieldType *fldP = eventP->data.fldChanged.pField;

      FldGetBounds(fldP, &fldRect);
      fldRect.extent.y = eventP->data.fldChanged.newHeight;
      FldSetBounds(fldP, fldRect);
      handled = false;
   }
break;
```

insertionPointOffEvent

Purpose Sent when the field has focus and the cursor should blink. `FldHandleEvent()` responds to this event by removing the cursor from the display.

Declared In `EventCodes.h`

Prototype None.

insertionPointOnEvent

Purpose Sent when the field has focus and the cursor should blink. `FldHandleEvent()` responds to this event by drawing the cursor.

Declared In `EventCodes.h`

Prototype None.

Field Functions and Macros

FldCalcFieldHeight Function

Purpose Determines the height of a field for a string.

Declared In `Field.h`

Prototype

```
uint32_t FldCalcFieldHeight (const char *chars,
    Coord maxWidth)
```

Parameters

→ *chars*
Pointer to a null-terminated string.

→ *maxWidth*
Maximum line width in standard coordinates.

Returns The total number of lines needed to draw the string passed.

Example The following code shows how to determine the width of the field and pass that value as the *maxWidth* parameter:

```
FrmGetObjectBounds(frm, FrmGetObjectIndex(frm, fldID),
  &myRect);
fieldWidth = myRect.extent.x;
FldCalcFieldHeight(myString, fieldWidth);
```

See Also FldWordWrap()

FldCompactText Function

Purpose Compacts the memory block that contains the field's text to release any unused space.

Declared In Field.h

Prototype
```
void FldCompactText (FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns Nothing.

Comments As characters are added to the field's text, the block that contains the text is grown. The block is expanded several bytes at a time so that it doesn't have to expand each time a character is added. This expansion may result in some unused space in the text block.

Applications should call this function on fields that edit data records in place within a non-schema database before the field is unlocked, or at any other time when a compact field is desirable; for example, before writing to the storage heap.

Applications should not call this function if the data is being edited in place within a schema database. Doing so has no effect.

This function may display a fatal error message if the memory cannot be compacted.

See Also FldGetTextAllocatedSize(), FldSetTextAllocatedSize()

FldCopy Function

Purpose Copies the current selection to the text clipboard.

Declared In Field.h

Prototype
```
void FldCopy (const FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns Nothing.

Comments This function leaves the current selection highlighted.

This function replaces anything previously in the text clipboard if there is text to copy. If no text is selected, the function beeps and the clipboard remains intact.

See Also `FldCut()`, `FldPaste()`

FldCut Function

Purpose Copies the current selection to the text clipboard, deletes the selection from the field, and redraws the field.

Declared In `Field.h`

Prototype
```
void FldCut (FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns Nothing.

Comments If text is selected, the text is removed from the field, the field's dirty attribute is set, and anything previously in the text clipboard is replaced by the selected text.

If there is no selection or if the field is not editable, this function beeps.

See Also `FldCopy()`, `FldPaste()`, `FldUndo()`

FldDelete Function

Purpose Deletes the specified range of characters from the field and redraws the field.

Declared In `Field.h`

Prototype
```
void FldDelete (FieldType *fldP, size_t start,
    size_t end)
```

Parameters → *fldP*
Pointer to a field.

→ *start*
The beginning of the range of characters to delete given as a valid byte offset into the field's text string.

→ *end*
The end of the range of characters to delete given as a valid byte offset into the field's text string. On systems that support multi-byte characters, this position must be an inter-character boundary. That is, it must not point to a middle byte of a multi-byte character.

If you pass a value that is greater than the number of bytes in the field, all characters in the field are deleted.

Returns Nothing.

Comments This function deletes all characters from the starting offset up to the ending offset and sets the field's dirty attribute. It does not delete the character at the ending offset.

On debug ROMs, this function displays a fatal error message if *start* or *end* point to an intra-character boundary. On release ROMs, if *start* or *end* point to an intra-character boundary, `FldDelete()` attempts to move the offset backward, toward the beginning of the text, until the offset points to an inter-character boundary (that is, the start of a character).

`FldDelete()` posts a `fldChangedEvent` to the event queue. If you call this function repeatedly, you may overflow the event queue with `fldChangedEvents`. An alternative is to remove the text handle from the field, change the text, and then set the field's handle again. See `FldGetTextHandle()` for a code example.

See Also `FldInsert()`, `FldEraseField()`, `TxtCharBounds()`

FldDirty Function

Purpose Returns `true` if the field has been modified since the text value was set.

Declared In `Field.h`

Prototype
```
Boolean FldDirty (const FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns `true` if the field has been modified either by the user or through calls to certain functions such as `FldInsert()` and `FldDelete()`, `false` if the field has not been modified.

See Also `FldSetDirty()`, `FieldAttrType`

FldDrawField Function

Purpose Draws the text of the field.

Declared In `Field.h`

Prototype
```
void FldDrawField (FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns Nothing.

Comments This function draws into a graphics context. Therefore, you must call it from within a `frmUpdateEvent` if the form that contains the field uses an update-based window.

The field must be usable or it won't be drawn.

This function doesn't erase the area behind the field before drawing.

If the field has the focus, the blinking insertion point is displayed in the field.

See Also FldEraseField(), FrmDrawForm()

FldEraseField Function

Purpose Erases the text of a field and turns off the insertion point if it's in the field.

Declared In `Field.h`

Prototype
```
void FldEraseField (FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns Nothing.

Comments You rarely need to call this function directly. Instead, use `FrmHideObject()`, which calls `FldEraseField()` for you.

This function visibly erases the field from the display, but it doesn't modify the contents of the field or free the memory associated with it.

If the field has the focus, the blinking insertion point is turned off.

See Also FldDrawField()

FldFreeMemory Function

Purpose Releases the handle-based memory allocated to the field's text and the associated word-wrapping information.

Declared In `Field.h`

Prototype

```
void FldFreeMemory (FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns Nothing.

Comments This function releases

- The memory allocated to the text of a field if both a text handle and a text string are assigned to the field and are stored in the dynamic heap. If only a text string is assigned to the field but the text handle is `NULL`, the text string is not freed.
- The memory allocated to hold the word-wrapping information.

This function doesn't affect the display of the field. Fields allocate memory for the text string as needed, so it is not an error to call this function while the field is still displayed. That is, if the field's text string is `NULL` and the user starts typing in the field, the field simply allocates memory for text and continues.

FldGetAttributes Function

Purpose Returns the attributes of a field.

Declared In `Field.h`

Prototype
```
void FldGetAttributes (const FieldType *fldP,
    FieldAttrType *attrP)
```

Parameters

→ *fldP*
Pointer to a field.

← *attrP*
Pointer to the `FieldAttrType` structure.

Returns Nothing.

See Also FldSetAttributes()

FldGetBounds Function

Purpose Returns the current bounds of a field.

Declared In `Field.h`

Prototype
```
void FldGetBounds (const FieldType *fldP,
    RectanglePtr rect)
```

Parameters

→ *fldP*
Pointer to a field.

← *rect*
Pointer to a `RectangleType` structure.

Returns Nothing.

See Also `FldSetBounds()`, `FrmGetObjectBounds()`

FldGetFont Function

Purpose Returns the ID of the font used to draw the text of a field.

Declared In `Field.h`

Prototype
```
FontID FldGetFont (const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns The FontID of the bitmapped font.

See Also FldSetFont()

FldGetInsPtPosition Function

Purpose Returns the insertion point position within the string.

Declared In Field.h

Prototype
```
size_t FldGetInsPtPosition (const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns The byte offset of the insertion point.

Comments The insertion point is to the left of the byte offset that this function returns. That is, if this function returns 0, the insertion point is to the left of the first character in the string. In multiline fields, line feeds are counted as a single character in the string, and the byte offset after the line feed character is the beginning of the next line.

See Also FldSetInsPtPosition()

FldGetLineInfo Function

Purpose Retrieves the word-wrapping information for a visible line within a field.

Declared In Field.h

Prototype
```
Boolean FldGetLineInfo (const FieldType *fldP,
   uint32_t lineNum, size_t *startP,
   size_t *lengthP)
```

Parameters

→ *fldP*
Pointer to a field.

→ *lineNum*
The number of the visible line in the field about which you want to retrieve information. Lines are numbered starting at 0.

← *startP*
The byte offset into the `FieldType`'s text string of the first character displayed by this line. If the line is empty, `*startP` is equal to the length of the field's text string.

← *lengthP*
The length in bytes of the portion of the string displayed on this line. If the line is empty, the length is 0.

Returns `true` if *startP* and *lengthP* contain valid values, or `false` if the field is a single-line field or does not contain a line numbered *lineNum*.

FldGetMaxChars Function

Purpose Returns the maximum number of bytes the field accepts.

Declared In `Field.h`

Prototype
```
size_t FldGetMaxChars (const FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns The maximum length in bytes of characters the user is allowed to enter.

See Also FldSetMaxChars()

FldGetNumberOfBlankLines Function

Purpose Returns the number of blank lines that are displayed at the bottom of a field.

Declared In `Field.h`

Prototype
```
uint32_t FldGetNumberOfBlankLines
    (const FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns The number of blank lines visible.

Comments This routine is useful for updating a scroll bar after characters have been removed from the text in a field. See the `NoteViewScroll()` function in the Address Book sample application for an example.

FldGetScrollPosition Function

Purpose Returns the offset of the first character in the first visible line of a field.

Declared In `Field.h`

Prototype
```
size_t FldGetScrollPosition(const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns The offset from the start of the field's text string of the first visible character.

See Also `FldSetScrollPosition()`

FldGetScrollValues Function

Purpose Returns the values necessary to update a scroll bar.

Declared In `Field.h`

Prototype
```
void FldGetScrollValues (const FieldType *fldP,
   uint32_t *scrollLineP, uint32_t *textHeightP,
   uint32_t *fieldHeightP)
```

Parameters

→ *fldP*
Pointer to a field.

← *scrollLineP*
The line of text that is the topmost visible line. Line numbering starts with 0.

← *textHeightP*
The number of lines needed to display the field's text, given the width of the field.

← *fieldHeightP*
The number of visible lines in the field.

Returns Nothing.

Comments May display a fatal error message if the field is a single-line text field.

Example The following example shows how to use the values returned by this function to update the scroll bar associated with the field:

```
FldGetScrollValues (fldP, &scrollPos, &textHeight,
  &fieldHeight);

if (textHeight > fieldHeight)
  maxValue = textHeight - fieldHeight;
else if (scrollPos)
  maxValue = scrollPos;
else
  maxValue = 0;

SclSetScrollBar (bar, scrollPos, 0, maxValue, fieldHeight-1);
```

See Also FldSetScrollPosition()

FldGetSelection Function

Purpose Returns the current selection of a field.

Declared In Field.h

Prototype

```
void FldGetSelection (const FieldType *fldP,
    size_t *startPosition, size_t *endPosition)
```

Parameters

→ *fldP*
Pointer to a field.

← *startPosition*
Pointer to the start of the selected characters range, given as the byte offset into the field's text.

← *endPosition*
Pointer to end of the selected characters range given as the byte offset into the field's text.

Returns The starting and ending byte offsets in *startPosition* and *endPosition*.

Comments The first character in a field is at offset zero.

If the user has selected the first five characters of a field, *startPosition* will contain the value 0 and *endPosition* the value 5, assuming all characters are a single byte long.

See Also FldSetSelection(), FldGetTextPtr()

FldGetTextAllocatedSize Function

Purpose Returns the number of bytes allocated to hold the field's text string. Don't confuse this number with the actual length of the text string displayed in the field.

Declared In `Field.h`

Prototype

```
size_t FldGetTextAllocatedSize
   (const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns The number of bytes allocated for the field's text.

Comments Editable text fields allocate and resize the text memory block as necessary when the user adds more text.

See Also FldSetTextAllocatedSize()

FldGetTextColumn Function

Purpose Returns the schema database, record, column, and offset that this field is modifying.

Declared In `Field.h`

Prototype

```
void FldGetTextColumn (const FieldType *fldP,
   DmOpenRef *databaseP, uint32_t *cursorP,
   uint32_t *positionP, uint32_t *columnIDP,
   size_t *offsetP)
```

Parameters

→ *fldP*
Pointer to a field.

← *databaseP*
Schema database where the text record is located.

← *cursorP*
Database cursor ID, that is, the ID of the currently selected set of records.

← *positionP*
Position of the database row within the cursor set.

← *columnIDP*
ID of the column that the text field should modify.

← *offsetP*
Offset into the column's data at which the text begins.

Returns Nothing.

Comments If the text field does not display information from a schema database, this function returns zeros in all of the parameters.

See Also FldSetTextColumn(), FldReleaseStorage(), FldReturnStorage()

FldGetTextHandle Function

Purpose Returns a handle to the block that contains the text string of a field.

Declared In Field.h

Prototype
```
MemHandle FldGetTextHandle (const FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns The handle to the text string of a field or NULL if no handle has been allocated for the field pointer.

Comments The handle returned by this function is not necessarily the handle to the start of the string. If you've used FldSetText() to set the field's text to a string that is part of a database record, the text handle points to the start of that record. You'll need to compute the offset from the start of the record to the start of the string. You can either store the offset that you passed to FldSetText(), or you can compute the offset by performing pointer arithmetic on the pointer you get by locking this handle and the pointer returned by FldGetTextPtr().

If you are obtaining the text handle so that you can edit the field's text, you must remove the handle from the field before you do so. If

you change the text while it is being used by a field, the field's internal structures specifying the text length, allocated size, and word wrapping information can become out of sync. To avoid this problem, remove the text handle from the field, change the text, and then set the field's text handle again.

Example The following code shows how to properly edit the field's text:

```
/* Get the handle for the string and unlock it by removing it
from the field. */
textH = FldGetTextHandle(fldP);
FldSetTextHandle (fldP, NULL);

/* Insert code that modifies the string here. The basic steps
are: resize the memory if necessary, lock the memory, write
to it, and then unlock the memory. If the text is in a
database record, use Data Manager calls. */

/* Update the text in the field and mark it invalid so that
it will be redrawn. */
FldSetTextHandle (fldP, textH);
FldGetBounds(fldP, &rect);
WinInvalidateRectangle(FrmGetWindowHandle(frmP), rect);
```

See Also FldSetTextHandle(), FldGetTextPtr()

FldGetTextHeight Function

Purpose Returns the height in standard coordinates of the number of visible lines that are not empty.

Declared In `Field.h`

Prototype

```
Coord FldGetTextHeight (const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns The height in standard coordinates of the number of visible lines that are not empty.

Comments Empty lines are all of the lines in the field following the last byte of text. Note that lines that contain only a line feed are not empty if

they are followed by lines with text. Also note that only lines that are visible are counted.

See Also FldCalcFieldHeight()

FldGetTextLength Function

Purpose Returns the length in bytes of the field's text.

Declared In Field.h

Prototype
```
size_t FldGetTextLength (const FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns The length in bytes of a field's text, not including the terminating null character.

FldGetTextPtr Function

Purpose Returns a locked pointer to the field's text string.

Declared In Field.h

Prototype
```
char *FldGetTextPtr (const FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns A locked pointer to the field's text string or NULL if the field is empty.

Comments The pointer returned by this function can become invalid if the user edits the text after you obtain the pointer.

Do not modify the contents of the pointer yourself. If you change the text while it is being used by a field, the field's internal structures specifying the text length, allocated size, and word wrapping information can become out of sync. To avoid this problem, follow the instructions given under FldGetTextHandle().

WARNING! The pointer returned by this function is "owned" by the field until you specify a different pointer for the field. You should not store this pointer for future use because the field can modify the size of the string, which can cause the pointer to become invalid.

See Also FldSetTextPtr()

FldGetVisibleLines Function

Purpose Returns the number of lines that can be displayed within the visible bounds of the field, regardless of what text is stored in the field.

Declared In Field.h

Prototype
```
uint32_t FldGetVisibleLines(const FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns The number of lines the field displays.

See Also FldGetNumberOfBlankLines(), FldCalcFieldHeight()

FldGrabFocus Function

Purpose Turns the insertion point on (if the specified field is visible) and positions the blinking insertion point in the field.

Declared In Field.h

Prototype
```
void FldGrabFocus (FieldType *fldP)
```

Parameters
→ *fldP*
Pointer to a field.

Returns Nothing.

Comments You rarely need to call this function directly. Instead, use FrmSetFocus(), which calls `FldGrabFocus()` for you.

If you call this function on a field whose form is not in the currently active window, this function silently fails.

See Also FrmSetFocus(), FldReleaseFocus()

FldHandleEvent Function

Purpose Handles events that affect the field, including the following: keyDownEvent, penDownEvent, and fldEnterEvent.

Declared In Field.h

Prototype
```
Boolean FldHandleEvent (FieldType *fldP,
    EventType *eventP)
```

Parameters

→ *fldP*
Pointer to a field.

→ *eventP*
Pointer to an event (EventType data structure).

Returns true if the event was handled.

Comments If there is an active Front-End Processor (FEP), this function passes all events to the FEP code to give it a chance to handle them. A FEP is code that translates characters from one alphabet to another. For example, Japanese devices use a FEP to convert from Romaji characters to Katakana characters.

When a keyDownEvent occurs in an editable text field, the keystroke appears in the field if it's a printable character or manipulates the insertion point if it's a "movement" character. The field is automatically updated.

When a penDownEvent occurs, the field sends a fldEnterEvent to the event queue.

When a fldEnterEvent occurs, the field grabs the focus. If the user has tapped twice in the current location, the word at that location is selected. If the user has tapped three times, the entire line is selected. Otherwise, the insertion point is placed in the specified position.

When a menuCmdBarOpenEvent occurs, the field adds paste, copy, cut, and undo buttons to the command tool bar. These buttons are only added if they make sense in the current context. That is, the cut button is only added if the field is editable, the paste button is only added if there is text on the clipboard and the field is editable, and the undo button is only added if there is an action to undo.

When an insertionPointOffEvent or an insertionPointOnEvent occurs, this function either draws or

erases the insertion point in the field and then posts the opposite event to the queue so that the cursor will blink.

If the event alters the contents of the field, this function visually updates the field.

This function doesn't handle any events if the field is not editable or usable.

FldInsert Function

Purpose Replaces the current selection if any with the specified string and redraws the field.

Declared In `Field.h`

Prototype

```
Boolean FldInsert (FieldType *fldP,
   const char *insertChars, size_t insertLen)
```

Parameters

→ *`fldP`*
Pointer to a field.

→ *`insertChars`*
Text string to be inserted.

→ *`insertLen`*
Length in bytes of the text string to be inserted, not counting the trailing null character.

Returns `true` if string was successfully inserted, or `false` if:

- The *`insertLen`* parameter is 0.
- The field is not editable.
- Adding the text would exceed the field's maximum number of characters.
- More memory must be allocated for the field, and the allocation fails.

Comments If there is no current selection, the string passed is inserted at the position of the insertion point.

This function sets the field's dirty attribute and posts a `fldChangedEvent` to the event queue. If you call this function repeatedly, you may overflow the event queue with `fldChangedEvents`. An alternative is to remove the text handle

from the field, change the text, and then set the field's handle again. See `FldGetTextHandle()` for a code example.

See Also FldPaste(), FldDelete(), FldCut(), FldCopy()

FldMakeFullyVisible Function

Purpose Generates an event to cause a dynamically resizable field to expand its height to make its text fully visible.

Declared In `Field.h`

Prototype

```
Boolean FldMakeFullyVisible (FieldType *fldP)
```

Parameters → *fldP*

Pointer to a field.

Returns `true` if the field is dynamically resizable and was not fully visible; `false` otherwise.

Comments Use this function only on fields that can dynamically resize.

This function does not actually resize the field. Instead, it computes how big the field should be to be fully visible and then posts this information to the event queue in a `fldHeightChangedEvent`.

NOTE: The event does not get generated if the number of lines in the field is equal to or greater than the value of the maximum lines attribute for the field.

If the field is contained in a table, the table's code handles the `fldHeightChangedEvent`. If the field is directly on a form, your application code should handle the `fldHeightChangedEvent` itself. The form code does not handle the event for you.

See Also `TblHandleEvent()`

FldNewField Function

Purpose Creates a new field dynamically and installs it in the specified form.

Declared In `Field.h`

Prototype

```
FieldType *FldNewField (void **formPP,
   uint16_t id, Coord x, Coord y, Coord width,
   Coord height, FontID font, size_t maxChars,
   Boolean editable, Boolean underlined,
   Boolean singleLine, Boolean dynamicSize,
   JustificationType justification,
   Boolean autoShift, Boolean hasScrollBar,
   Boolean numeric)
```

Parameters

↔ *formPP*
Pointer to the pointer to the form in which the new field is installed. This value is not a handle; that is, the old form pointer value is not necessarily valid after this function returns. In subsequent calls, always use the new form pointer value returned by this function.

→ *id*
Symbolic ID of the field, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *x*
Horizontal coordinate of the upper-left corner of the field's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the field's boundaries, relative to the window in which it appears.

→ *width*
Width of the field, expressed in standard coordinates.

→ *height*
Height of the field, expressed in standard coordinates.

→ *font*
Font to use to draw the field's text. See FontID.

→ *maxChars*
Maximum number of bytes held by the field this function creates.

→ *editable*
Pass `true` to create a field in which the user can edit text. Pass `false` to create a field that cannot be edited.

→ *underlined*
`true` if the field should be underlined, `false` otherwise. By convention, editable fields are underlined, and noneditable fields are not underlined.

→ *singleLine*
Pass `true` to create a field that can display only a single line of text.

→ *dynamicSize*
Pass `true` to create a field that resizes dynamically according to the amount of text it displays.

→ *justification*
Pass either of the values `leftAlign` or `rightAlign` to specify left justification or right justification, respectively. The `centerAlign` value is not supported.

→ *autoShift*
Pass `true` to specify the use of auto-shift rules.

→ *hasScrollBar*
Pass `true` if you are going to attach a scroll bar control to the field this function creates.

→ *numeric*
Pass `true` to specify that only characters in the range of 0 through 9 are allowed in the field.

Returns A pointer to the new field or `NULL` if there wasn't enough memory to create the field. Out of memory situations could be caused by memory fragmentation.

See Also FrmValidatePtr(), WinValidateHandle(), CtlValidatePointer(), FrmRemoveObject()

FldPaste Function

Purpose Replaces the current selection in the field, if any, with the contents of the text clipboard.

Declared In `Field.h`

Prototype
```
void FldPaste (FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns Nothing.

Comments The function performs these actions:

- Scrolls the field, if necessary, so the insertion point is visible.
- Inserts the clipboard text at the position of the insertion point if there is no current selection.
- Positions the insertion point after the last character inserted.
- Doesn't delete the current selection if there is no text in the clipboard.

See Also `FldInsert()`,`FldDelete()`,`FldCut()`,`FldCopy()FldUndo()`

FldRecalculateField Function

Purpose Updates the structure that contains the word-wrapping information for each visible line.

Declared In `Field.h`

Prototype
```
void FldRecalculateField (FieldType *fldP,
    Boolean redraw)
```

Parameters

→ *fldP*
Pointer to a field.

→ *redraw*
If `true`, redraws the field.

Returns Nothing.

Comments Word wrapping information is always kept current as the field is updated. This function invalidates the window region containing the field so that it is redrawn.

Call this function when you change the field width or the text length of the field. Do not call this function after changing the font or field height.

Note that many of the field functions, including `FldSetTextHandle()`, `FldInsert()`, and `FldDelete()`, recalculate the word-wrapping information for you.

FldReleaseFocus Function

Purpose Turns the blinking insertion point off, resets the pinlet input mode, and resets the undo state.

Declared In `Field.h`

Prototype
```
void FldReleaseFocus (FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns Nothing.

Comments Usually, you don't need to call this function. If the field is in a form or in a table that doesn't use custom drawing functions, the field code releases the focus for you when the focus changes to some other control. If your field is in any other type of user interface element, such as a gadget or a table that uses custom drawing functions, you must call `FldReleaseFocus()` when the focus moves away from the field.

See Also FldGrabFocus()

FldReleaseStorage Function

Purpose Releases the text field's lock on the schema database row with which it is associated.

Declared In `Field.h`

Prototype
```
status_t FldReleaseStorage (FieldType *fldP)
```

Parameters → *fldP*
Pointer to a field.

Returns `errNone` upon success, one of the Database Manager errors if the database cannot be released or `sysErrParamErr` if the field does not point to a schema database column.

Comments Use this function to release the field's lock on a database row. You need to do so when you need to make a change to a value in a different column than the one with which the field is associated or if you need to make some other change to the row (such as changing its category membership).

WARNING! Call `FldReturnStorage()` to reassociate the field with the row before calling any other Field function. Otherwise the application will crash. Do not change the data in the field's column in between calls to `FldReleaseStorage()` and `FldReturnStorage()`. Doing so also causes an application crash.

See Also `FldGetTextColumn()`, `FldSetTextColumn()`

FldReplaceText Function

Purpose Completely replaces the contents of the field's text pointer with the specified text.

Declared In `Field.h`

Prototype

```
void FldReplaceText (FieldType *fldP,
    const char *textP)
```

Parameters

→ *fldP*
Pointer to a field.

→ *textP*
New text that the field should display. The field makes a copy of this text.

Returns Nothing.

Comments This function replaces the contents of the field with the specified text, and recalculates the word wrapping information.

The keyboard dialog that displays on devices that do not have a dynamic input area uses this function to replace the field with the contents of the field in the dialog.

FldReturnStorage Function

Purpose Reassociates a field with its schema database column after `FldReleaseStorage()` has been called.

Declared In `Field.h`

Prototype
```
status_t FldReturnStorage (FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns `errNone` upon success, one of the Database Manager errors if the Database Manager cannot obtain the value of the text field's column, or `sysErrParamErr` if the text field is not associated with a schema database.

Comments Call this function to reassociate a field with a schema database column after you've called `FldReleaseStorage()` to make a change to some other value in the database row. The field assumes that the value for its column has not changed. If it has changed, the application could crash.

See Also `FldGetTextColumn()`, `FldSetTextColumn()`

FldScrollable Function

Purpose Returns `true` if the field is scrollable in the specified direction.

Declared In `Field.h`

Prototype
```
Boolean FldScrollable (const FieldType *fldP,
    WinDirectionType direction)
```

Parameters

→ *fldP*
Pointer to a field.

→ *direction*
The direction to test. See `WinDirectionType`.

Returns `true` if the field is scrollable in the specified direction; `false` otherwise.

See Also FldScrollField()

FldScrollField Function

Purpose Scrolls a field up or down by the number of lines specified.

Declared In `Field.h`

Prototype

```
void FldScrollField (FieldType *fldP,
    uint32_t linesToScroll,
    WinDirectionType direction)
```

Parameters

→ *fldP*
Pointer to a field.

→ *linesToScroll*
Number of lines to scroll.

→ *direction*
The direction to scroll. See WinDirectionType.

Returns Nothing.

Comments This function can't scroll horizontally, that is, right or left.

The field is redrawn if it's scrolled; however, the scroll bar is not updated.

If the field is not scrollable in the direction indicated, this function returns without performing any work. You can use FldScrollable() before calling this function to see if the field can be scrolled.

Example The following example calls this function and then uses SclSetScrollBar() to update the scroll bar:

```
FldScrollField (fldP, linesToScroll, direction);
// Update the scroll bar.
SclGetScrollBar (bar, &value, &min, &max, &pageSize);
if (direction == winUp)
  value -= linesToScroll;
else
  value += linesToScroll;
SclSetScrollBar (bar, value, min, max, pageSize);
```

See Also FldSetScrollPosition()

FldSendChangeNotification Function

Purpose Sends a `fldChangedEvent` to the event queue.

Declared In `Field.h`

Prototype

```
void FldSendChangeNotification
    (const FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns Nothing.

Comments This function is used internally by the field code. You normally never call it in application code.

FldSendHeightChangeNotification Function

Purpose Sends a `fldHeightChangedEvent` to the event queue.

Declared In `Field.h`

Prototype

```
void FldSendHeightChangeNotification
    (const FieldType *fldP, size_t pos,
    int32_t numLines)
```

Parameters

→ *fldP*
Pointer to a field.

→ *pos*
Character position of the insertion point.

→ *numLines*
New number of lines in the field.

Returns Nothing.

Comments This function is used internally by the field code. You normally never call it in application code.

FldSetAttributes Function

Purpose Sets the attributes of a field.

Declared In `Field.h`

Prototype

```
void FldSetAttributes (FieldType *fldP,
    const FieldAttrType *attrP)
```

Parameters

→ *fldP*
Pointer to a field.

→ *attrP*
Pointer to the attributes. See FieldAttrType.

Returns Nothing.

Comments This function does not do anything to make the new attribute values take effect. For example, if you use this function to change the value of the underline attribute, you won't see its effect until some other actions causes the field to be redrawn.

You usually do not have to modify field attributes at runtime, so you rarely need to call this function.

WARNING! You must not call this function to change any attributes that are noted as "for internal use only."

The proper way to use `FldSetAttributes()` is to:

1. Call `FldGetAttributes()` to retrieve the attributes.
2. Set the specific flags that you want to modify.
3. Call `FldSetAttributes()` to make the modifications.

WARNING! You must not call any field routines between calling `FldGetAttributes()` and `FldSetAttributes()`; this can cause the attributes to be out of sync, with unpredictable results.

FldSetBounds Function

Purpose Changes the position or size of a field.

Declared In `Field.h`

Prototype

```
void FldSetBounds (FieldType *fldP,
    const RectangleType *rP)
```

Parameters

→ *fldP*
Pointer to a field.

→ *rP*
Pointer to a `RectangleType` structure that contains the new bounds of the display.

Returns Nothing. May raise a fatal error message if the memory block that contains the word-wrapping information needs to be resized and there is not enough space to do so.

Comments If the field is visible, the field is redrawn within its new bounds.

NOTE: You can change the height or location of the field while it's visible, but do not change the width.

The memory block that contains the word-wrapping information will be resized if the number of visible lines is changed.

Make sure that *rP* is at least as tall as a single line in the current font. (You can determine this value by calling `FntLineHeight()`.) If it's not, results are unpredictable.

See Also `FldGetBounds()`, `FrmSetObjectBounds()`

FldSetDirty Function

Purpose Sets whether the field has been modified.

Declared In `Field.h`

Prototype

```
void FldSetDirty (FieldType *fldP, Boolean dirty)
```

Parameters

→ *fldP*
Pointer to a field.

→ *dirty*
`true` if the text is modified.

Returns Nothing.

Comments You typically call this function when you want to clear the dirty attribute. The dirty attribute is set when the user enters or deletes text in the field. It is also set by certain field functions, such as `FldInsert()` and `FldDelete()`.

See Also FldDirty()

FldSetFont Function

Purpose Sets the font used by the field, updates the word-wrapping information, and draws the field if the field is visible.

Declared In `Field.h`

Prototype
```
void FldSetFont (FieldType *fldP, FontID fontID)
```

Parameters

→ *fldP*
Pointer to a field.

→ *fontID*
ID of new font.

Returns Nothing.

Comments If the form that contains the field uses an update-based window, this function invalidates the window region containing the field. The field is redrawn during the next drawing update cycle.

See Also `FldGetFont()`, `FieldAttrType`

FldSetInsertionPoint Function

Purpose Sets the location of the insertion point based on a specified string position.

Declared In `Field.h`

Prototype
```
void FldSetInsertionPoint (FieldType *fldP,
    size_t pos)
```

Parameters

→ *fldP*
Pointer to a field.

→ *pos*
New location of the insertion point, given as a valid offset in bytes into the field's text. On systems that support multi-byte characters, you must make sure that this specifies an inter-character boundary (does not specify the middle or end bytes of a multi-byte character).

Returns Nothing.

Comments This routine differs from `FldSetInsPtPosition()` in that it doesn't make the character position visible. `FldSetInsertionPoint()` also doesn't make the field the current focus of input if it was not already.

If *pos* indicates a position beyond the end of the text in the field, the insertion point is set to the end of the field's text.

See Also `TxtCharBounds()`

FldSetInsPtPosition Function

Purpose Sets the location of the insertion point for a given string position.

Declared In `Field.h`

Prototype
```
void FldSetInsPtPosition (FieldType *fldP,
    size_t pos)
```

Parameters

→ *fldP*
Pointer to a field.

→ *pos*
New location of the insertion point, given as a valid offset in bytes into the field's text. On systems that support multi-byte characters, you must make sure that this specifies an inter-character boundary (does not specify the middle or end bytes of a multi-byte character).

Returns Nothing.

Comments If the position is beyond the visible text, the field is scrolled until the position is visible.

See Also `FldGetInsPtPosition()`, `TxtCharBounds()`

FldSetMaxChars Function

Purpose Sets the maximum number of bytes the field accepts.

Declared In `Field.h`

Prototype
```
void FldSetMaxChars (FieldType *fldP,
   size_t maxChars)
```

Parameters

→ *fldP*
Pointer to a field.

→ *maxChars*
Maximum size in bytes of the characters the user may enter. You may specify any value up to `maxFieldTextLen`.

Returns Nothing.

Comments Line feed characters are counted when the length of characters is determined.

See Also FldGetMaxChars()

FldSetMaxVisibleLines Function

Purpose Sets the maximum number of visible lines for a dynamically sized field.

Declared In `Field.h`

Prototype
```
void FldSetMaxVisibleLines (FieldType *fldP,
   uint8_t maxLines)
```

Parameters

→ *fldP*
Pointer to a field.

→ *maxLines*
Maximum number of lines to which the field will visually grow.

Returns Nothing.

Comments This function is intended for dynamically sized fields, which are fields for which the application is sent `fldHeightChangedEvent`s so that the application can expand or contract the visible number of lines as the text that the field contains grows and shrinks. Note that this dynamic sizing is independent of the form resizing. A field might grow and shrink as its form grows and shrinks but not be

considered a dynamically sized field. Dynamically sized fields are ones whose size is determined by the amount of text they display.

FldSetScrollPosition Function

Purpose Scrolls the field such that the character at the indicated offset is the first character on the first visible line. Redraws the field if necessary.

Declared In `Field.h`

Prototype

```
void FldSetScrollPosition (FieldType *fldP,
    size_t pos)
```

Parameters

→ *fldP*
Pointer to a field.

→ *pos*
Byte offset into the field's text string of first character to be made visible. On systems that support multi-byte characters, you must make sure that this specifies an inter-character boundary (does not specify the middle or end bytes of a multi-byte character).

Returns Nothing.

Comments This function scrolls the field but does not update the field's scroll bar. You should update the scroll bar after calling this function. To do so, first call `FldGetScrollValues()` to determine the values to use, and then call `SclSetScrollBar()`.

See Also `FldGetScrollPosition()`, `FldScrollField()`, `TxtCharBounds()`

FldSetSelection Function

Purpose Sets the current selection in a field and highlight the selection if the field is visible.

Declared In `Field.h`

Prototype

```
void FldSetSelection (FieldType *fldP,
    size_t startPosition, size_t endPosition)
```

Parameters

→ *fldP*
Pointer to a field.

→ *startPosition*
Starting offset of the character range to highlight, given as a byte offset into the field's text.

→ *endPosition*
Ending offset of the character range to highlight. The ending offset should be greater than or equal to the starting offset. On systems that support multi-byte characters, this position must be an inter-character boundary. That is, it must not point to a middle byte of a multi-byte character.

Returns Nothing.

Comments To cancel a selection, set both *startPosition* and *endPosition* to the same value. If *startPosition* equals *endPosition*, then the current selection is unhighlighted.

On debug ROMs, this function displays a fatal error message if *startPosition* or *endPosition* point to an intra-character boundary. On release ROMs, if either *startPosition* or *endPosition* point to an intra-character boundary, `FldSetSelection()` attempts to move that offset backward, toward the beginning of the string, until the offset points to an inter-character boundary (that is, the start of a character).

See Also `TxtCharBounds()`

FldSetText Function

Purpose Sets the text value of the field without updating the display.

Declared In `Field.h`

Prototype
```
void FldSetText (FieldType *fldP,
   MemHandle textHandle, size_t offset,
   size_t size)
```

Parameters → *fldP*
Pointer to a field.

→ *textHandle*
Unlocked handle of a block containing a null-terminated text string. Pass `NULL` for this parameter to remove the association between the field and the string it is currently displaying so that the string is not freed with the field when the form is deleted.

→ *offset*
Offset from start of block to start of the text string.

→ *size*
The allocated size of the text string. This is not the string length, and should not be set to 0, unless you are setting the text to the empty string.

Returns Nothing.

Comments This function allows applications to perform editing in place in memory. You can use it to point the field to a string in a non-schema database record so that you can edit that string directly using field routines.

As characters are added to the field's text, the block that contains the text is grown. So that the block doesn't have to be expanded for each character, it is expanded several bytes at a time; this expansion may result in some unused space in the text block. As characters are removed from the field's text, the space is not automatically reclaimed. Because adding or removing characters when editing a data record in place may result in unused space at the end of the field's text block, applications should call `FldCompactText()` on before the field is unlocked to release any unused space.

The handle that you pass to this function is assumed to contain a null-terminated string starting at *offset* bytes in the memory block. The string should be between 0 and *size* - 1 bytes in length. The field does not make a copy of the memory block or the string data; instead, it stores the handle to the record in its structure.

WARNING! You cannot use this function to set two fields on a form so that they share a single string value. Thus, for instance, if you have a single string containing a person's name you cannot call `FldSetText()` twice with the same string (but a different offset) to set up a first name field and a last name field.

`FldSetText()` updates the word-wrapping information and places the insertion point after the last visible character, but it does not update the display. To update the display after calling this function, you must call `WinInvalidateRect()` with the window region that displays the field. Do *not* call `FldDrawField()` directly unless your code is handling a `frmUpdateEvent`.

`FldSetText()` increments the lock count for *`textHandle`* and decrements the lock count of its previous text handle (if any).

Because `FldSetText()` (and `FldSetTextHandle()`) may be used to edit database records, they do not free the memory associated with the previous text handle. If the previous text handle points to a string on the dynamic heap and you want to free it, use `FldGetTextHandle()` to obtain the handle before using `FldSetText()` and then free that handle after using `FldSetText()`. See `FldSetTextHandle()` for a code example.

See Also `FldSetTextPtr()`, `FldSetTextHandle()`, `FldSetTextColumn()`

FldSetTextAllocatedSize Function

Purpose Sets the number of bytes allocated to hold the field's text string. Don't confuse this with the actual length of the text string displayed in the field.

Declared In `Field.h`

Prototype

```
void FldSetTextAllocatedSize (FieldType *fldP,
   size_t allocatedSize)
```

Parameters

→ *`fldP`*
Pointer to a field.

→ *`allocatedSize`*
Number of bytes to allocate for the text.

Returns Nothing.

Comments This function generally is not used. It does not resize the field's allocated memory for the text string. It merely sets a value that is computed and maintained internally by the field, so you should not have to call `FldSetTextAllocatedSize()` directly.

See Also `FldGetTextAllocatedSize()`, `FldCompactText()`

FldSetTextColumn Function

Purpose Sets the field to modify a specific column within a schema database record.

Declared In `Field.h`

Prototype

```
void FldSetTextColumn (FieldType *fldP,
    DmOpenRef database, uint32_t cursor,
    uint32_t position, uint32_t columnID,
    size_t offset)
```

Parameters

→ *fldP*
Pointer to a field.

→ *database*
Open schema database where the record is located.

→ *cursor*
Database cursor ID, that is, the ID of the currently selected set of records. Use `DbCursorOpen()` to create a cursor.

→ *position*
Position of the database row within the cursor set.

→ *columnID*
ID of the column that the field should modify.

→ *offset*
Offset into the column's data at which the text begins.

Returns Nothing.

Comments This function allows applications to perform editing in place in memory. You can use it to point the field to a string within a column of a schema database record so that you can edit that string directly using field routines.

The column that you pass to this function is assumed to contain a null-terminated string starting at *offset* bytes in the memory block. The string should be between 0 and *size* - 1 bytes in length. The field does not make a copy of the memory block or the string data; instead, it stores the database information in its structure.

WARNING! You cannot use this function to set two fields on a form so that they share a single string value. Thus, for instance, if you have a single string containing a person's name you cannot call `FldSetTextColumn()` twice with the same string (but a different offset) to set up a first name field and a last name field.

`FldSetTextColumn()` updates the word-wrapping information and places the insertion point after the last visible character, but it does not update the display. To update the display after calling this function, you must call `WinInvalidateRect()` with the window region that displays the field. Do *not* call `FldDrawField()` directly unless your code is handling a `frmUpdateEvent`.

When you use this function, the field holds a lock on the database row that it is currently displaying. If you need to change some other value in the row (such as the value in another column or the row's category membership), you must release the lock before you can do so. Call `FldReleaseStorage()` to release the lock. Then make your change. Then call `FldReturnStorage()` to reassociate the field with its column before the form redraws.

See Also `FldGetTextColumn()`, `FldSetText()`

FldSetTextHandle Function

Purpose Sets the text value of a field to the string associated with the specified handle. Does not update the display.

Declared In `Field.h`

Prototype

```
void FldSetTextHandle (FieldType *fldP,
    MemHandle textHandle)
```

Parameters

→ *fldP*
Pointer to a field.

→ *textHandle*
Unlocked handle of a field's text string. Pass `NULL` for this parameter to remove the association between the field and the string it is currently displaying so that the string is not freed with the field when the form is deleted.

Returns Nothing.

Comments

This function differs from `FldSetText()` in that it uses the entire memory block pointed to by *`textHandle`* for the string. In fact, this function simply calls `FldSetText()` with an offset of 0 and a size equal to the entire length of the memory block. Use it to have the field edit a string in a database record if the entire record consists of that string, or use it to have the field edit a string in the dynamic heap.

As characters are added to the field's text, the block that contains the text is grown. So that the block doesn't have to be expanded for each character, it is expanded several bytes at a time; this expansion may result in some unused space in the text block. As characters are removed from the field's text, the space is not automatically reclaimed. Because adding or removing characters when editing a data record in place may result in unused space at the end of the field's text block, applications should call `FldCompactText()` on before the field is unlocked to release any unused space.

`FldSetTextHandle()` updates the word-wrapping information and places the insertion point after the last visible character, but it does not update the display. You must call `WinInvalidateRect()` with the window region that displays the field. Do *not* call `FldDrawField()` directly unless your code is handling a `frmUpdateEvent`.

`FldSetTextHandle()` increments the lock count for *`textHandle`* and decrements the lock count of its previous text handle (if any).

Because `FldSetTextHandle()` (and `FldSetText()`) may be used to edit database records, they do not free the memory associated with the previous text handle. If the previous text handle points to a string on the dynamic heap and you want to free it, use `FldGetTextHandle()` to obtain the handle before using `FldSetText()` and then free that handle after using `FldSetText()`.

When the form containing this field is closed, all memory associated with the field is freed including the text handle and its associated text string if they are in the dynamic heap. Text handles and text strings that are in the storage heap are not freed. If your text handle is in the dynamic heap and you don't want the memory associated with the text handle freed when the field is freed, use `FldSetTextHandle()` and pass `NULL` for the text handle

immediately before the form is closed. Passing `NULL` removes the association between the field and the text handle that you want retained. That text handle is unlocked as a result of the `FldSetTextHandle()` call, and when the field is freed, there is no text handle to free with it.

Example

The following example shows how to free the old memory associated with the field.

```
/* get the old text handle */
oldTxtH = FldGetTextHandle(fldP);

/* change the text and update the display */
FldSetTextHandle(fldP, txtH);
FldGetBounds(fldP, &rect);
WinInvalidateRectangle(FrmGetWindowHandle(frmP), rect);

/* free the old text handle */
if (oldTxtH != NULL)
  MemHandleFree(oldTxtH);
```

See Also FldSetTextPtr(), FldSetText()

FldSetTextPtr Function

Purpose Sets a noneditable field's text to point to the specified text string.

Declared In `Field.h`

Prototype

```
void FldSetTextPtr (FieldType *fldP, char *textP)
```

Parameters

→ *`fldP`*
Pointer to a field.

→ *`textP`*
Pointer to a null-terminated string.

Returns Nothing.

Comments Do not call `FldSetTextPtr()` with an editable text field. Instead, call `FldSetTextHandle()` for editable text fields. `FldSetTextPtr()` is intended for displaying noneditable text in the user interface.

If the field has more than one line, use `FldRecalculateField()` to recalculate the word wrapping.

This function does *not* visually update the field. To do so, call `WinInvalidateRect()` with the window region that displays the field. Do *not* call `FldDrawField()` directly unless your code is handling a `frmUpdateEvent`.

The field never frees the string that you pass to this function, even when the field itself is freed. You must free the string yourself. Before you free the string, make sure the field is not still displaying it. Set the field's string pointer to some other string or call `FldSetTextPtr(fldP, NULL)` before freeing a string you have passed using this function.

This function may display an error message if passed an editable text field.

See Also `FldSetTextHandle()`, `FldGetTextPtr()`

FldSetUsable Function

Purpose Sets a field to usable or unusable.

Declared In `Field.h`

Prototype
```
void FldSetUsable (FieldType *fldP, Boolean usable)
```

Parameters

→ *fldP*
Pointer to a field.

→ *usable*
`true` to set usable; false to set nonusable.

Returns Nothing.

Comments An unusable field doesn't display or accept input.

Use `FrmHideObject()` and `FrmShowObject()` instead of using this function.

See Also `FldEraseField()`, `FldDrawField()`, `FieldAttrType`

FldUndo Function

Purpose Undoes the last change made to the field, if any. Changes include typing, backspaces, delete, paste, and cut.

Declared In `Field.h`

Prototype

```
void FldUndo (FieldType *fldP)
```

Parameters

→ *fldP*
Pointer to a field.

Returns Nothing.

See Also FldPaste(), FldCut(), FldDelete(), FldInsert()

FldWordWrap Function

Purpose Given a string and a width, returns the number of bytes of characters that can be displayed using the current font.

Declared In `Field.h`

Prototype

```
size_t FldWordWrap (const char *chars,
   Coord maxWidth)
```

Parameters

→ *chars*
Pointer to a null-terminated string.

→ *maxWidth*
Maximum line width in standard coordinates.

Returns The length in bytes of the characters that can be displayed.

See Also FntWordWrap()

19

Fixed Math Reference

This chapter describes the fixed-point computations declared in `FixedMath.h`. It covers:

Fixed Math Structures and Types

Fixed Typedef

Purpose A signed fixed-point value.

Declared In `FixedMath.h`

Prototype

```
typedef int32_t Fixed
```

Fixed32 Typedef

Purpose An unsigned fixed-point 32-bit value.

Declared In `FixedMath.h`

Prototype

```
typedef uint32_t Fixed32
```

Fixed32Intermediate Typedef

Purpose Type used to store intermediate values when working with fixed-point numbers.

Declared In `FixedMath.h`

Prototype

```
typedef unsigned long long Fixed32Intermediate
```

FixedIntermediate Typedef

Purpose Type used to store intermediate values when working with fixed-point numbers.

Declared In `FixedMath.h`

Prototype

```
typedef Fixed32Intermediate FixedIntermediate
```

Fixed Math Constants

Fixed-Point Constants

Purpose Constants declared in `FixedMath.h`.

Declared In `FixedMath.h`

Constants

`#define kFixed32Bias (16)`
Use 16 bits for the integer portion and 16 bits for the fractional portion.

`#define kFixed32FractionMask (0x0000FFFF)`
A mask used to access the fractional portion of a `Fixed32` value.

`#define kFixedBias kFixed32Bias`
Use 16 bits for the integer portion and 16 bits for the fractional portion.

`#define kFixedFractionMask kFixed32FractionMask`
A mask used to access the fractional portion of a `Fixed` value.

`#define kFixedOne 0x00010000`
The integer value 1.

`#define kFixedOneAndOneHalf 0x00018000`
The value 1.5.

`#define kFixedOneHalf 0x00008000`
The value 0.5.

`#define kFixedTwo 0x00020000`
The value 2.

`#define kFixedTwoThirds 0x0000AAAB`
The value 2/3 (0.666666667).

Fixed Math Functions and Macros

DivIntByFixedResultInt Macro

Purpose Divides one value by another and returns an integer.

Declared In `FixedMath.h`

Prototype
```
#define DivIntByFixedResultInt (lhs, rhs)
```

Parameters

→ *lhs*
An integer value.

→ *rhs*
A fixed-point value.

Returns An integer value representing *lhs* divided by *rhs*.

Fixed32Div Macro

Purpose Divides one Fixed32 value by another.

Declared In `FixedMath.h`

Prototype
```
#define Fixed32Div (lhs, rhs)
```

Parameters

→ *lhs*
The dividend.

→ *rhs*
The divisor.

Returns The quotient produced when *lhs* is divided by *rhs*.

Fixed32Fraction Macro

Purpose Accesses the fractional part of a 32-bit fixed point value.

Declared In `FixedMath.h`

Prototype

```
#define Fixed32Fraction (x)
```

Parameters

→ *x*
A 32-bit fixed point value.

Returns A 32-bit fixed point value with only the fractional portion of the value as nonzero.

Fixed32FromInteger Macro

Purpose Converts an integer into a `Fixed32` value.

Declared In `FixedMath.h`

Prototype

```
#define Fixed32FromInteger (n)
```

Parameters

→ *n*
A 16-bit integer.

Returns A `Fixed32` with the integer portion assigned to *n* and the fractional portion of 0.

Fixed32Mul Macro

Purpose Multiplies two fixed-point values.

Declared In `FixedMath.h`

Prototype

```
#define Fixed32Mul (lhs, rhs)
```

Parameters

→ *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The product of *lhs* and *rhs* as a `Fixed32` value.

Fixed32ToInteger Macro

Purpose Converts a `Fixed32` value to an integer type.

Declared In `FixedMath.h`

Prototype
```
#define Fixed32ToInteger (n)
```

Parameters

→ *n*
A `Fixed32` value.

Returns A `uint32_t` value representing only the integer portion of *n*. The fractional portion is discarded.

FixedAdd Macro

Purpose Adds two `Fixed` values.

Declared In `FixedMath.h`

Prototype
```
#define FixedAdd (lhs, rhs)
```

Parameters

→ *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The sum of *lhs* and *rhs*.

FixedDiv Macro

Purpose Divides one value by another.

Declared In `FixedMath.h`

Prototype
```
#define FixedDiv (lhs, rhs)
```

Parameters

→ *lhs*
The divisor.

→ *rhs*
The dividend.

Returns The quotient produced when *lhs* is divided by *rhs*. If both parameters are `Fixed`, the result is `Fixed`. If the values are integers, the result is an integer.

FixedFraction Macro

Purpose Accesses the fractional part of a fixed-point value.

Declared In `FixedMath.h`

Prototype
```
#define FixedFraction (x)
```

Parameters → *x*
A `Fixed` value.

Returns A fixed point value with only the fractional portion of the value as nonzero.

FixedFromInteger Macro

Purpose Converts an integer into a `Fixed32` value.

Declared In `FixedMath.h`

Prototype
```
#define FixedFromInteger (x)
```

Parameters → *x*
A 16-bit integer.

Returns A `Fixed` number with the integer portion assigned to *x* and the fractional portion of 0.

FixedMul Macro

Purpose Multiplies two values.

Declared In `FixedMath.h`

Prototype
```
#define FixedMul (lhs, rhs)
```

Parameters → *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The product of *lhs* and *rhs*. The product is a `Fixed` value, if both parameters are `Fixed`, otherwise, it is an integer value.

FixedMulByFixed Macro

Purpose Multiplies two values and returns the result as a `Fixed` value.

Declared In `FixedMath.h`

Prototype `#define FixedMulByFixed (lhs, rhs)`

Parameters → *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The product of *lhs* and *rhs* as a `Fixed` value.

FixedMulByInt16 Macro

Purpose Multiplies two values and returns the result as a 16-bit integer.

Declared In `FixedMath.h`

Prototype `#define FixedMulByInt16 (lhs, rhs)`

Parameters → *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The product of *lhs* and *rhs* as a `uint16_t` value.

FixedMulByInt32 Macro

Purpose Multiplies two values and returns the result as a 32-bit integer.

Declared In `FixedMath.h`

Prototype `#define FixedMulByInt32 (lhs, rhs)`

Parameters → *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The product of *lhs* and *rhs* as a `uint32_t` value.

FixedPower2Div Macro

Purpose Divides a value by a power of 2.

Declared In `FixedMath.h`

Prototype
```
#define FixedPower2Div (x, power)
```

Parameters

→ *x*
The dividend.

→ *power*
The exponent value for 2.

Returns The value of $x / 2^{power}$.

FixedPower2Mul Macro

Purpose Multiplies a value by a power of 2.

Declared In `FixedMath.h`

Prototype
```
#define FixedPower2Mul (x, power)
```

Parameters

→ *x*
The value to multiply.

→ *power*
The exponent value for 2.

Returns The value of $x * 2^{power}$.

FixedSub Macro

Purpose Subtracts one `Fixed` value from another.

Declared In `FixedMath.h`

Prototype
```
#define FixedSub (lhs, rhs)
```

Parameters

→ *lhs*
The left operand.

→ *rhs*
The right operand.

Returns The difference when *rhs* is subtracted from *lhs*.

FixedToInteger Macro

Purpose Converts a `Fixed` value to an integer.

Declared In `FixedMath.h`

Prototype

```
#define FixedToInteger (x)
```

Parameters → *x*

A `Fixed` value.

Returns A `uint32_t` value representing only the integer portion of *x*. The fractional portion is discarded.

20

Form Reference

This chapter provides the following information about forms:

The header files `Form.h` and `FormLayout.h` declare the API that this chapter describes. For more information on forms, see Chapter 2, "Working with Forms and Dialogs," on page 15.

Form Structures and Types

AlertTemplateType Struct

Purpose Internal structure used to construct alert dialogs.

Declared In `Form.h`

Prototype

```
typedef struct AlertTemplateTag AlertTemplateType
```

Fields None.

See Also `FrmAlert()`, `FrmAlertWithFlags()`, `FrmCustomAlert()`, `FrmCustomAlertWithFlags()`, `FrmCustomResponseAlert()`, `FrmCustomResponseAlertWithFlags()`, `FrmUIAlert()`

FormActiveStateType Struct

Purpose Saves the active state of the current form.

Declared In `Form.h`

Prototype

```
typedef struct FormActiveStateType {
   uint32_t data[5];
   uint16_t dataLast;
   uint16_t paddingNeverUse;
} FormActiveStateType
```

Comments Treat the contents of this structure like a black box. Do not attempt to read to or write from this structure.

See Also `FrmRestoreActiveState()`, `FrmSaveActiveState()`

FormBitmapType Struct

Purpose Defines the visible characteristics of a bitmap on a form.

Declared In `Form.h`

Prototype

```
typedef struct FormBitmapType FormBitmapType
```

Fields None.

FormGadgetAttrType Struct

Purpose Defines a gadget's attributes. See `FormGadgetTypeInCallback`.

Declared In `Form.h`

Prototype

```
typedef struct FormGadgetAttrTag {
   uint16_t usable :1;
   uint16_t extended :1;
   uint16_t visible :1;
   uint16_t reserved :13;
} FormGadgetAttrType
```

Fields

`usable`
Not set if the gadget is not considered part of the current interface of the application, and it doesn't appear on screen. This is set by `FrmShowObject()` and cleared by `FrmHideObject()`.

`extended`
: If set, the gadget is an extended gadget. An extended gadget has the `handler` field defined in its `FormGadgetTypeInCallback` structure. If not set, the gadget is a standard gadget compatible with all releases of Palm OS®.

`visible`
: Set or cleared when the gadget is drawn or erased. `FrmHideObject()` clears this value. You should set it explicitly in the gadget's callback function (if it has one) in response to a draw request.

`reserved`
: Reserved for future use.

FormGadgetType Struct

Purpose A gadget that appears on a form.

Declared In `Form.h`

Prototype
```
typedef struct FormGadgetType FormGadgetType
```

Fields None.

See Also `FrmSetGadgetData()`, `FrmGetGadgetData()`

FormGadgetTypeInCallback Struct

Purpose A gadget element as it is passed to the extended gadget handler (`FormGadgetHandlerType()`).

Declared In `Form.h`

Prototype
```
typedef struct FormGadgetTypeInCallback {
   uint16_t id;
   FormGadgetAttrType attr;
   RectangleType rect;
   const void *data;
   FormGadgetHandlerType *handler;
} FormGadgetTypeInCallback
```

Fields

`id`
: ID of the gadget resource.

`attr`
See `FormGadgetAttrType`.

`rect`
Bounds of the gadget. See `RectangleType`.

`data`
Pointer to any specific data that needs to be stored. You can set and retrieve the value of this field with `FrmGetGadgetData()` and `FrmSetGadgetData()`.

`handler`
Pointer to a callback function that controls the gadget's behavior and responds to events. You can set this field with `FrmSetGadgetHandler()`.

FormLabelType Struct

Purpose A label that appears on the form.

Declared In `Form.h`

Prototype
```
typedef struct FormLabelType FormLabelType
```

Fields None.

FormLayoutType Struct

Purpose Layout information for a control in a form

Declared In `FormLayout.h`

Prototype
```
typedef struct FormLayoutType {
   int32_t structSize;
   uint16_t id;
   uint16_t reserved1;
   uint32_t rule;
} FormLayoutType
```

Fields

`structSize`
Always set to `sizeof(FormLayoutType)`.

`id`
The resource ID of the UI element whose layout rule is specified by this structure.

`reserved1`
Reserved for future use. Always set to 0.

`rule`
One of the Form Layout Constants or a layout rule constructed with `frmLayoutRule()`.

See Also `FrmInitLayout()`

FormType Struct

Purpose An internal structure that describes a form. `FormPtr` defines a pointer to a `FormType` structure.

Declared In `Form.h`

Prototype
```
typedef struct FormType FormType;
typedef FormType *FormPtr
```

Fields None. All fields are private.

FrmGraffitiStateType Struct

Purpose Defines the shift indicator.

Declared In `Form.h`

Prototype
```
typedef struct FormGraffitiStateTag
FrmGraffitiStateType
```

Fields None.

Form Constants

AlertType Enum

Purpose Specifies the type of an alert.

Declared In `Form.h`

Constants `informationAlert`
Informational alert. Use for actions that shouldn't or can't be completed but that don't generate an error or risk data loss.

`confirmationAlert`
: Confirmation alert. Use to confirm an action or suggest options.

`warningAlert`
: Warning alert. Use to confirm a serious or potentially dangerous action.

`errorAlert`
: Error alert. Use for actions that generate an error and cannot be completed.

Custom Response Alert Actions

Purpose Sent to the `FormCheckResponseFuncType()` to specify the action that the callback should perform.

Declared In `Form.h`

Constants

```
#define frmResponseCreate 1974
```
: The function should perform initialization.

```
#define frmResponseQuit ((int16_t) 0xBEEF)
```
: The function should perform cleanup.

Form Basic Layout Constants

Purpose Used to construct a layout rule with `frmLayoutRule()`.

Declared In `FormLayout.h`

Constants

```
frmAttachLeftTop = 0
```
: Element's left and top sides should follow left and top edges of window.

```
frmAttachRightBottom = 1
```
: Element's right and bottom sides should follow right and bottom edges of window.

Form Layout Constants

Purpose Specify the resizing rule for a UI element. This constant is used as the value of the `rule` field in the `FormLayoutType` structure.

Declared In `FormLayout.h`

Constants

`frmFollowNone = 0`
No resizing rule.

`frmFollowAllSides`
The element resizes vertically and horizontally.

`frmFollowLeft`
The element moves so that there is always a fixed amount of space between it and the left side of the form.

`frmFollowRight`
The element moves so that there is always a fixed amount of space between it and the right side of the form.

`frmFollowLeftRight`
The element resizes horizontally if the form is horizontally resized.

`frmFollowTop`
The element moves so that there is always a fixed amount of space between it and the top of the form.

`frmFollowBottom`
The element moves so that there is always a fixed amount of space between it and the bottom of the form.

`frmFollowTopBottom`
The element resizes vertically.

FormObjectKind Typedef

Purpose Types of user interface elements that may appear on a form.

Declared In `Form.h`

Constants

`frmFieldObj`
Text field

`frmControlObj`
Control

`frmListObj`
List

`frmTableObj`
Table

`frmBitmapObj`
Form bitmap

`frmLineObj`
Line

`frmFrameObj`
Frame

`frmRectangleObj`
Rectangle

`frmLabelObj`
Label

`frmTitleObj`
Form title

`frmPopupObj`
Pop-up list

`frmGraffitiStateObj`
Shift indicator

`frmGadgetObj`
Gadget (custom element)

`frmScrollBarObj`
Scroll bar

See Also `FrmGetObjectType()`

Gadget Actions

Purpose Sent to the `FormGadgetHandlerType()` callback to specify the action that the gadget handler should perform.

Declared In `Form.h`

Constants

```
#define formGadgetDrawCmd 0
```

Sent in response to the `frmUpdateEvent` to indicate that the gadget must be drawn or redrawn.

```
#define formGadgetEraseCmd 1
```
Sent by FrmHideObject() to indicate that the gadget is going to be erased. FrmHideObject() clears the visible flag for you. If you return false, it also clears the usable flag and invalidates the window region that displays the gadget so that it is erased on the next update cycle.

```
#define formGadgetHandleEventCmd 2
```
Sent by FrmHandleEvent() to indicate that a gadget event has been received.

```
#define formGadgetDeleteCmd 3
```
Sent by FrmDeleteForm() to indicate that the gadget is being deleted and must clean up any memory it has allocated or perform other cleanup tasks.

Miscellaneous Constants

Purpose

The Form.h file also declares these constants.

Declared In

Form.h

Constants

```
#define customAlertInsertionTag ((char*)"^ErrMsg")
```
Used by ErrAlert(), which is documented in *Exploring Palm OS: System Management*.

```
#define frmInvalidObjectId (0xffff)
```
Returned by some functions if the specified UI element cannot be found.

```
#define frmNoSelectedControl 0xff
```
Returned by FrmGetControlGroupSelection() if no control is selected.

```
#define frmRedrawUpdateCode 0x8000
```
Indicates that the form should be redrawn; flag in a frmUpdateEvent.

```
#define noFocus (frmInvalidObjectId)
```
No element in the form has the focus.

```
#define frmLayoutGraffitiShiftID
  (frmInvalidObjectId - 1)
```
Special constant to use as the resource ID for the shift indicator when setting up the FormLayoutType structure.

Form Events

frmCloseEvent

Purpose Sent when a form is being closed.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmClose {
  uint16_t formID;
} frmClose
```

Fields `formID`
Resource ID of the form.

Comments The functions `FrmGotoForm()` and `FrmCloseAllForms()` send this event. `FrmGotoForm()` sends a `frmCloseEvent` to the currently active form; `FrmCloseAllForms()` sends a `frmCloseEvent` to all forms an application has loaded into memory.

`FrmHandleEvent()` calls `FrmEraseForm()` and `FrmDeleteForm()` in response to the `frmCloseEvent` to remove the form from the screen and deallocate its internal data structure.

Applications should respond to this event if they need to do one of the following before the form is deallocated:

- Save data stored in the form
- Clean up memory used by the form

Return `false` from your event handler to get the default behavior from `FrmHandleEvent()`.

frmGadgetEnterEvent

Purpose Sent when there is a `penDownEvent` within the bounds of an extended gadget.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In Event.h

Prototype
```
struct gadgetEnter {
   struct FormGadgetType *gadgetP;
   uint16_t gadgetID;
   BA32_PADDING_16(1)
} gadgetEnter;
```

Fields

`gadgetID`
Resource ID of the gadget.

`gadgetP`
Pointer to the `FormGadgetType` structure representing this gadget.

Comments The gadget handler function (see `FormGadgetHandlerType()`) should handle this event. It should perform any necessary highlighting. If the gadget needs to track the pen, call `FrmSetPenTracking()` in response to this event. When you do so, the gadget receives all further pen events (`penMoveEvent` and `penUpEvent`).

frmGadgetMiscEvent

Purpose An application may choose to send this event when it needs to pass information to an extended gadget. The `FrmHandleEvent()` function passes `frmGadgetMiscEvents` on to the extended gadget's handler function (see `FormGadgetHandlerType()`).

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In Event.h

Prototype
```
struct gadgetMisc {
   struct FormGadgetType *gadgetP;
   uint16_t gadgetID;
   uint16_t selector;
   void *dataP;
} gadgetMisc;
```

Fields

`gadgetP`
Pointer to the `FormGadgetType` structure representing this gadget.

`gadgetID`
Resource ID of the gadget.

`selector`
Any necessary integer value to pass to the gadget handler function.

`dataP`
A pointer to any necessary data to pass to the gadget handler function.

frmGotoEvent

Purpose An application may choose to send itself this event when it receives a `sysAppLaunchCmdGoTo` launch code. `sysAppLaunchCmdGoto` is generated when the user selects a record in the global find facility.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmGoto {
   uint16_t formID;
   _BA32_PADDING_16(1)
   uint32_t recordNum;
   uint32_t recordID;
   size_t matchPos;
   size_t  matchLen;
   uint32_t matchFieldNum;
   uint32_t matchCustom;
} frmGoto;
```

Fields

`formID`
Resource ID of the form.

`recordNum`
Index of record containing the match string.

`matchPos`
Position of the match.

`matchLen`
Length of the matched string.

`matchFieldNum`
: Number of the field the matched string was found in.

`matchCustom`
: Application-specific information. You might use this if you need to provide extra information to locate the matching string within the record.

Comments The application is responsible for handling this event.

Like `frmOpenEvent`, `frmGotoEvent` is a request that the application initialize a form, but this event provides extra information so that the application may display and highlight the matching string in the form.

frmLoadEvent

Purpose Sent when a form should be loaded from the resource file.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmLoad {
  uint16_t formID;
  uint16_t reserved;
  DmOpenRef formDatabase;
} frmLoad
```

Fields

`formID`
: Resource ID of the form.

`reserved`
: Reserved for future use.

`formDatabase`
: Open database containing the form.

Comments The functions `FrmGotoForm()` and `FrmPopupForm()` send this event. It's a request that the application load a form into memory.

The application is responsible for handling this event. In response to this event, applications typically initialize the form, make it active, set the event handler, and return `true`.

If you want your form automatically resized, also call `FrmInitLayout()` in response to this event.

Example The following example shows the functions that are typically called in response to this event:

```
FormType *frm;
case frmLoadEvent:
{
  frm = FrmInitForm (event->data.frmLoad.formDatabase,
        event->data.frmLoad.formID);
  FrmSetActiveForm (frm);
  FrmInitLayout(frm, MainFormLayout);
  FrmSetEventHandler(frm, MainFormHandleEvent);
  handled = true;
  break;
}
```

frmOpenEvent

Purpose Sent when a form should be initialized.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmOpen {
  uint16_t formID;
} frmOpen
```

Fields

`formID`
Resource ID of the form.

Comments The functions `FrmGotoForm()` and `FrmPopupForm()` send this event. It is a request that the application initialize a form.

The application is responsible for handling this event. Perform any initializations that should take place before the form is drawn and return `true` to indicate that you've handled the event.

Compatibility Applications should no longer call `FrmDrawForm()` in response to this event. Do so only in response to `frmUpdateEvent`.

frmSaveEvent

Purpose Sent when `FrmSaveAllForms()` is called.

Declared In `EventCodes.h`

Prototype No data is passed with this event.

Comments Applications should respond to this event by saving any data stored in an open form.

frmStopDialogEvent

Purpose Causes `FrmDoDialog()` to exit with a simulated default button tap.

Declared In `EventCodes.h`

Prototype No data is passed with this event.

frmTitleEnterEvent

Purpose Sent when a pen down occurs within the form's title. Note that only the written title, not the whole title bar, is active.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmTitleEnter {
   uint16_t formID;
} frmTitleEnter;
```

Fields

`formID`
Resource ID of the form.

Comments Applications do not normally respond to this event. When `FrmHandleEvent()` receives this event, it tracks the pen until it is lifted. If the pen is lifted within the bounds of the same title bar region, a `frmTitleSelectEvent` is added to the event queue.

frmTitleSelectEvent

Purpose Sent when the user taps the form's title.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmTitleSelect {
   uint16_t formID;
} frmTitleSelect;
```

Fields

`formID`
Resource ID of the form.

Comments Applications do not normally respond to this event. `FrmHandleEvent()` responds by displaying the form's menu.

frmUpdateEvent

Purpose Sent when all or a portion of a form needs to be redrawn. This includes when the form is first opened, when it is resized, and as a result of the functions `WinInvalidateWindow()` or `WinInvalidateRect()`.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct frmUpdate {
  uint16_t formID;
  uint16_t updateCode;
  RectangleType dirtyRect;
} frmUpdate;
```

Fields

`formID`
Resource ID of the form.

`updateCode`
The reason for the update request. Most often, this is `frmRedrawUpdateCode`, which indicates that the form needs to be redrawn.

`dirtyRect`
: The portion of the form that needs to be redrawn. (See `RectangleType`.)

Comments

Applications that do not perform any special drawing need not respond to this event. The default handler in `FrmHandleEvent()` calls `FrmDrawForm()`. If your application performs custom drawing, respond to this event. Call `FrmDrawForm()` first and then draw.

All drawing to forms with update-based windows must be done *only* in response to this event.

When a window obscures a portion of your form, your form still receives `frmUpdateEvents`. All drawing into the obscured region is clipped. When the obscuring window is closed, your form receives an update event again with the previously obscured region as the region that needs to be redrawn.

IMPORTANT: Never post a `frmUpdateEvent` or `winUpdateEvent` explicitly. Instead, use one of the functions `WinInvalidateRect()`, `WinInvalidateRectFunc()`, or `WinInvalidateWindow()`.

Compatibility

Forms that use legacy windows, as a rule, do not receive `frmUpdateEvent`. For this style of form, the region that another form obscures is saved and restored by the UI Library without application intervention. However, in cases where the system is running low on memory, the UI library does not save obscured regions itself. In that case, the legacy form receives a `frmUpdateEvent` and redraws the region using the `updateCode` value.

Form Functions and Macros

ECFrmValidatePtr Macro

Purpose Evaluates to `true` if the specified pointer references a valid form.

Declared In `Form.h`

Prototype
```
#define ECFrmValidatePtr (formP)
```

Parameters

→ *formP*
Pointer to be tested.

Returns `true` if the specified pointer is a non-`NULL` pointer to a valid form structure.

Comments This macro is intended for debugging purposes only. Do not include it in released code.

See Also `FrmValidatePtr()`

FrmAlert Function

Purpose Creates a modal dialog from an alert resource and displays it until the user taps a button.

Declared In `Form.h`

Prototype
```
uint16_t FrmAlert (DmOpenRef database,
    uint16_t alertId)
```

Parameters

→ *database*
Open database containing the alert resource.

→ *alertId*
ID of the alert resource.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Alert resources cannot be associated with window constraints resources, and usually one is not necessary. The UI Library makes sure that the alert is always the same size and snaps to the bottom of the application area. If you need to set window creation attributes that you would normally set in a constraints resource, use `FrmAlertWithFlags()`.

See Also `FrmDoDialog()`, `FrmCustomAlert()`, `FrmCustomResponseAlert()`, `FrmAlertWithFlags()`, `FrmUIAlert()`

FrmAlertWithFlags Function

Purpose Creates a modal dialog using the specified window attributes and alert resource. Displays the dialog until the user taps a button.

Declared In `Form.h`

Prototype

```
uint16_t FrmAlertWithFlags (DmOpenRef database,
    uint16_t alertId, WinFlagsType windowFlags)
```

Parameters

→ *database*
Open database containing the alert resource.

→ *alertId*
ID of the alert resource.

→ *windowFlags*
Window attributes to use when creating the dialog. See `WinFlagsType`. The attribute `winFlagModal` is set for you.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Use this function in one of two circumstances:

- You need the window created for the alert to be a transitional window so that you can draw to it outside of `frmUpdateEvent`s. In this case set the `winFlagBackBuffer` bit in *windowFlags*.

- You are displaying an alert outside of the application process and the alert should appear in the same window layer as all of the other windows in that process.

See Also FrmAlert(), FrmCustomResponseAlertWithFlags(), FrmCustomAlertWithFlags()

FrmAmIPenTracking Function

Purpose Returns `true` if the specified element is tracking the pen.

Declared In `Form.h`

Prototype

```
Boolean FrmAmIPenTracking (FormType *formP,
    uint16_t objectID)
```

Parameters

→ *formP*
Pointer to the form.

→ *objectID*
The ID of the element that might be tracking the pen.

Returns `true` if the element is tracking the pen, `false` otherwise.

FrmCloseAllForms Function

Purpose Sends a `frmCloseEvent` to all open forms.

Declared In `Form.h`

Prototype

```
void FrmCloseAllForms (void)
```

Parameters None.

Returns Nothing.

Comments Applications can call this function to ensure that all forms are closed cleanly before exiting `PilotMain()`; that is, before termination.

See Also FrmSaveAllForms()

FrmCopyLabel Function

Purpose Copies the specified string into the data structure of the specified label in the specified form.

Declared In `Form.h`

Prototype

```
void FrmCopyLabel (FormType *formP,
    uint16_t labelID, const char *newLabel)
```

Parameters

→ *formP*
Pointer to the form.

→ *labelID*
Resource ID of the label.

→ *newLabel*
Pointer to a null-terminated string.

Returns Nothing.

Comments The length of the new label **must not** exceed the length of the string defined in the resource. When defining the string in the label resource, specify an initial size at least as big as any of the strings that will be assigned dynamically. This function redraws the label if it is currently displayed on the screen.

This function redraws the label but does not erase the old one first. If the new label is shorter than the old one, the end of the old label will still be visible. To avoid this, you can hide the label using `FrmHideObject()`, then show it using `FrmShowObject()`, after using `FrmCopyLabel()`.

Note that `FrmCopyLabel()` copies the passed string into memory already allocated for the label. Thus, the string doesn't need to remain in existence once `FrmCopyLabel()` returns.

See Also `FrmGetLabel()`

FrmCopyTitle Function

Purpose Copies a new title over the form's current title. If the form is visible, the new title is drawn.

Declared In `Form.h`

Prototype

```
void FrmCopyTitle (FormType *formP,
    const char *newTitle)
```

Parameters

→ *formP*
Pointer to the form.

→ *newTitle*
Pointer to the new title string.

Returns Nothing.

Comments The size of the new title *must not* exceed the title size defined in the resource. When defining the title in the resource, specify an initial size at least as big as any of the strings to be assigned dynamically.

See Also FrmGetTitle(), FrmSetTitle()

FrmCustomAlert Function

Purpose Creates a modal dialog from an alert resource and displays the dialog until the user taps a button in the alert dialog.

Declared In `Form.h`

Prototype

```
uint16_t FrmCustomAlert (DmOpenRef database,
    uint16_t alertId, const char *s1,
    const char *s2, const char *s3)
```

Parameters

→ *database*
Open database containing the alert resource.

→ *alertId*
Resource ID of the alert.

→ *s1, s2, s3*
Strings to replace ^1, ^2, and ^3 (see Comments).

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Up to three strings can be passed to this routine. They are used to replace the variables ^1, ^2 and ^3 that are contained in the message string of the alert resource.

If the variables ^1, ^2, and ^3 occur in the message string, do not pass `NULL` for the arguments *`s1`*, *`s2`*, and *`s3`*. If you want an argument to be ignored, pass the empty string (`""`).

Alert resources cannot be associated with window constraints resources, and usually a constraints resource is not necessary. The UI Library makes sure that the alert is always the same size and snaps to the bottom of the application area. If you need to set window creation attributes that you would normally set in a constraints resource, use `FrmCustomAlertWithFlags()`.

See Also `FrmAlert()`, `FrmDoDialog()`, `FrmCustomResponseAlert()`, `FrmCustomAlertWithFlags()`

FrmCustomAlertWithFlags Function

Purpose Creates a modal dialog with the specified window attributes and alert resource. Displays the dialog until the user taps a button.

Declared In `Form.h`

Prototype

```
uint16_t FrmCustomAlertWithFlags
   (DmOpenRef database, uint16_t alertId,
   WinFlagsType windowFlags, const char *s1,
   const char *s2, const char *s3)
```

Parameters

→ *`database`*
Open database containing the alert resource.

→ *`alertId`*
Resource ID of the alert.

→ *`windowFlags`*
Window attributes to use when creating the dialog. See `WinFlagsType`. The attribute `winFlagModal` is set for you.

→ *s1, s2, s3*
Strings to replace ^1, ^2, and ^3. See the Comments in FrmCustomAlert() for more information.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Use this function in one of two circumstances:

- You need the window created for the alert to be a transitional window so that you can draw to it outside of frmUpdateEvents. In this case set the winFlagBackBuffer bit in *windowFlags*.
- You are displaying an alert outside of the application process and the alert should appear in the same window layer as all of the other windows in that process.

See Also FrmCustomAlert(), FrmAlertWithFlags(), FrmCustomResponseAlertWithFlags()

FrmCustomResponseAlert Function

Purpose Creates a modal dialog with a text field from an alert resource and displays it until the user taps a button in the alert dialog.

Declared In Form.h

Prototype

```
uint16_t FrmCustomResponseAlert
    (DmOpenRef database, uint16_t alertId,
    const char *s1, const char *s2, const char *s3,
    char *entryStringBuf,
    int16_t entryStringBufLength,
    FormCheckResponseFuncPtr callback)
```

Parameters → *database*
Open database containing the alert resource.

→ *alertId*
Resource ID of the alert.

→ *s1, s2, s3*
Strings to replace ^1, ^2, and ^3. See the Comments in `FrmCustomAlert()` for more information.

← *entryStringBuf*
The string the user entered in the text field.

→ *entryStringBufLength*
The maximum length for the string in *entryStringBuf*.

→ *callback*
A callback function that processes the string. See `FormCheckResponseFuncType()`. Pass `NULL` if there is no callback.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments This function differs from `FrmCustomAlert()` in these ways:

- The dialog it displays contains a text field for user entry. The text that the user enters is returned in the *entryStringBuf* parameter.
- When the user taps a button, the *callback* function is called and is passed the button number and *entryStringBuf*. The dialog is only dismissed if the callback returns `true`. This behavior allows you to perform error checking on the string that the user entered and give the user a chance to re-enter the string.

  The callback function is also called with special constants when the alert dialog is being initialized and when it is being deallocated. This allows the callback to perform any necessary initialization and cleanup.

Alert resources cannot be associated with window constraints resources, and usually a constraints resource is not necessary. The UI Library makes sure that the alert is always the same size and snaps to the bottom of the application area. If you need to set window creation attributes that you would normally set in a

constraints resource, use FrmCustomResponseAlertWithFlags().

See Also FrmAlert(), FrmDoDialog(), FrmCustomResponseAlertWithFlags()

FrmCustomResponseAlertWithFlags Function

Purpose Creates a modal dialog with the specified window attributes and a text field from an alert resource. Displays the dialog until the user taps a button.

Declared In Form.h

Prototype

```
uint16_t FrmCustomResponseAlertWithFlags
    (DmOpenRef database, uint16_t alertId,
    WinFlagsType windowFlags, const char *s1,
    const char *s2, const char *s3,
    char *entryStringBuf,
    int16_t entryStringBufLength,
    FormCheckResponseFuncPtr callback)
```

Parameters

→ *database*
Open database containing the alert resource.

→ *alertId*
Resource ID of the alert.

→ *windowFlags*
Window attributes to use when creating the dialog. See WinFlagsType. The attribute winFlagModal is set for you.

→ *s1, s2, s3*
Strings to replace ^1, ^2, and ^3. See the Comments in FrmCustomAlert() for more information.

← *entryStringBuf*
The string the user entered in the text field.

→ *entryStringBufLength*
The maximum length for the string in *entryStringBuf*.

→ *callback*
A callback function that processes the string. See FormCheckResponseFuncType(). Pass NULL if there is no callback.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Use this function in one of two circumstances:

- You need the window created for the alert to be a transitional window so that you can draw to it outside of `frmUpdateEvent`s. In this case set the `winFlagBackBuffer` bit in *`windowFlags`*.
- You are displaying an alert outside of the application process and the alert should appear in the same window layer as all of the other windows in that process.

See Also `FrmAlertWithFlags()`, `FrmCustomAlertWithFlags()`, `FrmCustomResponseAlert()`

FrmDeleteForm Function

Purpose Releases the memory occupied by a form. Any memory allocated to user interface elements in the form is also released.

Declared In `Form.h`

Prototype

```
void FrmDeleteForm (FormType *formP)
```

Parameters → *`formP`*
Pointer to the form.

Returns Nothing.

See Also `FrmInitForm()`, `FrmReturnToForm()`

FrmDispatchEvent Function

Purpose Dispatches an event to the application's handler for the form.

Declared In `Form.h`

Prototype
```
Boolean FrmDispatchEvent (EventType *eventP)
```

Parameters
→ *eventP*
Pointer to an event.

Returns The Boolean value returned by the form's event handler or `FrmHandleEvent()`. (If the form's event handler returns `false`, the event is passed to `FrmHandleEvent()`.) This function also returns `false` if the form specified in the event is invalid.

Comments This function is called from most application event loops. It gives both the application and the UI Library a chance to handle any events sent to a form.

The event is dispatched to the appropriate form's handler. A form's event handler (`FormEventHandlerType()`) is registered by `FrmSetEventHandler()`.

FrmDoDialog Function

Purpose Displays a modal dialog until the user taps a button in the dialog.

Declared In `Form.h`

Prototype
```
uint16_t FrmDoDialog (FormType *formP)
```

Parameters
→ *formP*
Pointer to the form.

Returns The resource ID of the button the user tapped.

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

Comments Before calling `FrmDoDialog()` you must call `FrmInitForm()` to load and initialize the dialog and you must set the event handler, if one is needed. After the call, read any values needed from the dialog's elements and then call `FrmDeleteForm()` to release the memory occupied by the dialog.

Forms displayed using `FrmDoDialog()` do not receive `frmOpenEvent` or `frmCloseEvent`.

Example Use code similar to the following to display a modal dialog:

```
frmP = FrmInitForm(gDbP, MyDialogForm);
FrmSetEventHandler(frmP, MyDialogEventHandler);
/* initialize the dialog's controls here if necessary */

/* open the dialog, and wait until a button is pressed to
close it. */
whichButton = FrmDoDialog(frmP);

if (whichButton == DetailsOKButton) {
  /* get data from controls on the form to save/apply changes
  */
}
FrmDeleteForm(frmP);
```

See Also `FrmCustomAlert()`, `FrmCustomResponseAlert()`

FrmDrawForm Function

Purpose Draws all elements in a form and the frame around the form.

Declared In `Form.h`

Prototype

```
void FrmDrawForm (FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns Nothing.

Comments You do not need to call this function for most forms. `FrmHandleEvent()` does so for you. If the form requires custom drawing, call this in response to the `frmUpdateEvent`. Perform custom drawing after the call to `FrmDrawForm()`.

Do not call this function outside of a `frmUpdateEvent` if the form uses an update-based window.

See Also `FrmEraseForm()`, `FrmInitForm()`

FrmEraseForm Function

Purpose Erases a form from the display.

Declared In `Form.h`

Prototype
```
void FrmEraseForm (FormType *formP)
```

Parameters → *formP*
Pointer to the form.

Returns Nothing.

FrmGetActiveField Function

Purpose Returns the active field for a specified form.

Declared In `Form.h`

Prototype
```
FieldType *FrmGetActiveField
    (const FormType *formP)
```

Parameters → *formP*
Pointer to the form for which the active field should be returned, or `NULL` to obtain the active field on the active form.

Returns A pointer to the active field. `NULL` if the form doesn't have an active field or if there is no active form.

See Also `FrmGetActiveForm()`

FrmGetActiveForm Function

Purpose Returns the currently active form.

Declared In `Form.h`

Prototype
```
FormType *FrmGetActiveForm (void)
```

Parameters None.

Returns A pointer to the active form.

Comments You should not call the `FrmGetActiveForm()` function when a pop-up window is open.

See Also `FrmGetActiveField()`, `FrmGetActiveFormID()`, `FrmSetActiveForm()`

FrmGetActiveFormID Function

Purpose Returns the resource ID of the currently active form.

Declared In `Form.h`

Prototype

```
uint16_t FrmGetActiveFormID (void)
```

Parameters None.

Returns The active form's resource ID number or 0 if there is no active form.

See Also `FrmGetActiveForm()`

FrmGetBitmapHandle Function

Purpose Returns a handle to an internal bitmap structure that stores the bitmap in a format that can be displayed most efficiently. Caches the bitmap structure for future use.

Declared In `Form.h`

Prototype

```
GcBitmapHandle FrmGetBitmapHandle (FormType *form,
    DmOpenRef database, DmResourceType bitmapType,
    DmResourceID bitmapID, uint32_t flags)
```

Parameters

→ *form*
Pointer to the form.

→ *database*
Open database containing the bitmap resource.

→ *bitmapType*
The resource type of the bitmap resource, either `bitmapRsc` for a standard Palm OS bitmap or `'pngf'` for a PNG resource.

→ *bitmapID*
The resource ID of the bitmap.

→ *flags*
One of the Bitmap Loading constants.

Returns A `GcBitmapHandle` for the bitmap or `NULL` if a handle cannot be created for this bitmap.

Comments This function returns a cached handle to the specified bitmap. The returned handle is suitable to pass to `GcDrawBitmapAt()` when you want to draw the bitmap to the screen. If the bitmap has not been previously cached, this function adds it to the cache, creates a bitmap handle for it, and returns the bitmap handle.

Do not use this function if you're performing animation, doing a slide show, or otherwise drawing a lot of bitmaps to the form. If you do, you may quickly fill the bitmap cache. Instead, use `GcLoadBitmap()` directly and release each bitmap as you finish with it.

See Also `FrmRemoveBitmapHandle()`

FrmGetControlGroupSelection Function

Purpose Returns the item number of the control selected in a group of controls.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetControlGroupSelection
    (const FormType *formP, uint8_t groupNum)
```

Parameters → *formP*
Pointer to the form.

→ *groupNum*
Group ID for this group of controls.

Returns The item number of the selected control; returns `frmNoSelectedControl` if no item is selected.

Comments When a group ID is assigned to a group of controls, those controls are mutually exclusive. Only one of the controls can be selected at a time. This function is the way you obtain the user selection for that group of controls.

Currently, only check boxes and push buttons can be grouped. It is more acceptable to group push buttons than check boxes.

NOTE: `FrmSetControlGroupSelection()` sets the selection in a control group based on a resource ID, *not* its index, which `FrmGetControlGroupSelection()` returns.

See Also FrmGetObjectId(), FrmGetObjectPtr()

FrmGetControlValue Function

Purpose Returns the current value of a control.

Declared In `Form.h`

Prototype
```
int16_t FrmGetControlValue (const FormType *formP,
    uint16_t objIndex)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of the control element in the form's data structure. You can obtain this by using `FrmGetObjectIndex()`.

Returns The current value of the control. For most controls the return value is either 0 (off) or 1 (on). For sliders, this function returns the value of the `value` field.

See Also FrmSetControlValue(), CtlGetValue()

FrmGetDefaultButtonID Function

Purpose Gets the resource ID of the form's default button.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetDefaultButtonID
    (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns The resource ID of the button defined as the default, or 0 if the form has no default button.

Comments Modal dialogs have a default button assigned. If the user decides to exit the application rather than respond to the modal dialog, the

system enqueues a `ctlSelectEvent` for the default button, and the application should behave as if this button were tapped. Other types of forms do not have the default button assigned.

See Also `FrmSetDefaultButtonID()`, `FrmDoDialog()`

FrmGetEventHandler Function

Purpose Returns the form's event handler.

Declared In `Form.h`

Prototype

```
FormEventHandlerType *FrmGetEventHandler
    (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns A pointer to the form's event handler. See `FormEventHandlerType()`.

See Also `FrmSetEventHandler()`

FrmGetFirstForm Function

Purpose Returns the first form in the form list.

Declared In `Form.h`

Prototype

```
FormType *FrmGetFirstForm (void)
```

Parameters None.

Returns A pointer to a form, or `NULL` if there are no forms.

Comments The form list is a LIFO stack. The last form created is the first form in the form list.

FrmGetFocus Function

Purpose Returns the item (index) number of the element that has the focus.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetFocus (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns The index of the UI element that has the focus, or `noFocus` if no element has the focus. To convert the element's index to an ID, use `FrmGetObjectId()`.

See Also `FrmGetObjectPtr()`, `FrmSetFocus()`

FrmGetFormBounds Function

Purpose Returns the current bounds of the form.

Declared In `Form.h`

Prototype
```
void FrmGetFormBounds (const FormType *formP,
   RectangleType *rP)
```

Parameters

→ *formP*
Pointer to the form.

← *rP*
Pointer to a `RectangleType` structure where the bounds are returned.

Returns Nothing.

Compatibility For forms with update-based or transitional windows, this function returns incorrect results until the `winResizedEvent` is received. It is acceptable to call this function in response to a `frmUpdateEvent` because `winResizedEvent` is sent before `frmUpdateEvent`.

See Also `FrmGetFormInitialBounds()`

FrmGetFormId Function

Purpose Returns the resource ID of a form.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetFormId (const FormType *formP)
```

Parameters → *formP*
Pointer to the form.

Returns The form's resource ID.

See Also `FrmGetFormPtr()`

FrmGetFormInitialBounds Function

Purpose Returns the bounds of the form as specified in the form resource.

Declared In `FormLayout.h`

Prototype
```
void FrmGetFormInitialBounds
    (const FormType *formP, RectangleType *rP)
```

Parameters → *formP*
Pointer to the form.

← *rP*
Pointer to a `RectangleType` structure where the bounds are returned.

Returns Nothing.

Comments The form may be larger or smaller than the bounds specified by `rP`; it may have been resized due to the user opening or closing the dynamic input area or displaying a slip window.

You may find this function useful when resizing your form in response to a `winResizedEvent`. If you always compare the bounds specified in the resized event to the form's original bounds as returned by this function, you'll avoid the slight rounding errors that may occur if you always compare the form's current size to the new size.

See Also `FrmGetFormBounds()`, `FrmGetObjectInitialBounds()`

FrmGetFormPtr Function

Purpose Returns a pointer to the form that has the specified ID.

Declared In `Form.h`

Prototype
```
FormType *FrmGetFormPtr (uint16_t formID)
```

Parameters

→ *formID*
Resource ID number of the form.

Returns A pointer to the form, or `NULL` if the form is not in memory.

See Also `FrmGetFormId()`

FrmGetFormWithWindow Function

Purpose Returns the form that uses the specified window.

Declared In `Form.h`

Prototype
```
FormType *FrmGetFormWithWindow (WinHandle winH)
```

Parameters

→ *winH*
The handle to a window.

Returns A pointer to the form with the specified window or `NULL` if the form is not in memory.

See Also `FrmGetWindowHandle()`

FrmGetGadgetData Function

Purpose Returns the value stored in the `data` field of the gadget.

Declared In `Form.h`

Prototype
```
void *FrmGetGadgetData (const FormType *formP,
    uint16_t objIndex)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of the gadget in the form's data structure. You can obtain this by using `FrmGetObjectIndex()`.

Returns A pointer to the custom gadget's data.

Comments Gadgets provide a way for an application to attach custom gadgetry to a form. In general, the `data` field of a gadget contains a pointer to the custom element's data structure.

See Also `FrmSetGadgetData()`, `FrmSetGadgetHandler()`

FrmGetHelpID Function

Purpose Gets the resource ID of the form's help resource.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetHelpID (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns The resource ID of the form's help resource. The help resource is a String resource.

See Also `FrmSetHelpID()`

FrmGetLabel Function

Purpose Returns a pointer to the text of the specified label in the specified form.

Declared In `Form.h`

Prototype
```
const char *FrmGetLabel (const FormType *formP,
    uint16_t labelID)
```

Parameters

→ *formP*
Pointer to the form.

→ *labelID*
Resource ID of the label.

Returns A pointer to the label string.

Comments This function returns a pointer to the string itself, not a copy of it.

labelID must be a label. If not, a fatal error message is displayed.

See Also `FrmCopyLabel()`

FrmGetLabelFont Function

Purpose Gets the font used for a particular label that appears on a form.

Declared In `Form.h`

Prototype
```
FontID FrmGetLabelFont (const FormType *formP,
    uint16_t labelID)
```

Parameters

→ *formP*
Pointer to the form.

→ *labelID*
Resource ID of a label in the form (the element's type must be `frmLabelObj`).

Returns The `FontID` of the font used for the label. If *labelID* is invalid or does not specify a label, this function returns 0.

See Also `FrmGetObjectType()`, `FrmSetLabelFont()`

FrmGetMenuBarID Function

Purpose Gets the resource ID of the form's menu bar.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetMenuBarID (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns The resource ID of the form's menu bar, or 0 if the form doesn't have a menu bar.

See Also `FrmSetMenu()`, `MenuGetActiveMenu()`

FrmGetNumberOfObjects Function

Purpose Returns the number of elements in a form.

Declared In `Form.h`

Prototype
```
uint16_t FrmGetNumberOfObjects
    (const FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns The number of elements in the specified form.

See Also FrmGetObjectPtr(), FrmGetObjectId()

FrmGetObjectBounds Function

Purpose Retrieves the bounds of an element given its form and index.

Declared In Form.h

Prototype

```
void FrmGetObjectBounds (const FormType *formP,
    uint16_t objIndex, RectangleType *rP)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

← *rP*
Pointer to a RectangleType structure where the element bounds are returned. The bounds are in window-relative coordinates.

Returns Nothing.

See Also FrmGetObjectPosition(), FrmSetObjectPosition(), FrmGetObjectInitialBounds(), FrmSetObjectBounds()

FrmGetObjectId Function

Purpose Returns the resource ID of the specified UI element.

Declared In Form.h

Prototype

```
uint16_t FrmGetObjectId (const FormType *formP,
    uint16_t objIndex)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

Returns The resource ID of the element or `frmInvalidObjectId` if the *`objIndex`* parameter is invalid.

See Also `FrmGetObjectPtr()`

FrmGetObjectIdFromObjectPtr Function

Purpose Returns the ID of a UI element given a pointer to that element.

Declared In `Form.h`

Prototype

```
uint16_t FrmGetObjectIdFromObjectPtr
    (void *formObjP, FormObjectKind objKind)
```

Parameters

→ *`formObjP`*
Pointer to a user interface element appearing on a form.

→ *`objKind`*
The form's type. Supply one of the values declared in the `FormObjectKind` enum.

Returns The form's resource ID, or `frmInvalidObjectID` if *`objKind`* is not a valid value.

See Also `FrmGetObjectId()`, `FrmGetObjectPtr()`, `FrmGetObjectType()`

FrmGetObjectIndex Function

Purpose Returns the index of a UI element in the form's list of elements given the element's resource ID.

Declared In `Form.h`

Prototype

```
uint16_t FrmGetObjectIndex (const FormType *formP,
    uint16_t objID)
```

Parameters

→ *`formP`*
Pointer to the form.

→ *`objID`*
Resource ID of an element in the form.

Returns The index of the specified element (the index of the first element is 0), or `frmInvalidObjectId` if the supplied element ID is invalid.

Comments Bitmaps use a different mechanism for IDs than the rest of the forms. When finding a bitmap with `FrmGetObjectIndex()`, you need to pass the resource ID of the bitmap itself, not the resource ID of the form bitmap element that contains the bitmap. (Passing the ID of the form bitmap element may or may not give you the right element back, depending on how you created the elements.)

This means that if you've got the same bitmap in two different form bitmap elements on the same form, you won't be able to use `FrmGetObjectIndex()` to obtain the second one; it'll always return the first.

See Also `FrmGetObjectPtr()`, `FrmGetObjectId()`

FrmGetObjectIndexFromPtr Function

Purpose Returns the index of a UI element in the form's list of elements given a pointer to the element.

Declared In `Form.h`

Prototype

```
uint16_t FrmGetObjectIndexFromPtr
    (const FormType *formP, const void *objP)
```

Parameters

→ *formP*
Pointer to the form.

→ *objP*
Pointer to a UI element.

Returns The index of the specified element (the index of the first element is 0), or `frmInvalidObjectId` if the supplied element pointer is invalid.

Comments Normally, you use `FrmGetObjectIndex()` to obtain the index given a resource ID. Use `FrmGetObjectIndexFromPtr`() in cases where you do not know the resource ID of the element.

FrmGetObjectInitialBounds Function

Purpose Returns the bounds specified when creating the UI element.

Declared In `FormLayout.h`

Prototype

```
void FrmGetObjectInitialBounds
   (const FormType *formP, uint16_t objIndex,
   RectangleType *rP)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using `FrmGetObjectIndex()`.

← *rP*
Pointer to a `RectangleType` structure where the bounds are returned. The bounds are in window-relative coordinates.

Returns Nothing.

Comments You must call `FrmInitLayout()` before calling this function.

You may find this function useful when resizing your form in response to a `winResizedEvent`. If you always use the initial bounds when computing offsets from the form's window, you'll avoid the slight rounding errors that can occur if you always compare the new bounds to the current bounds.

See Also `FrmGetFormInitialBounds()`, `FrmGetObjectPosition()`, `FrmGetObjectBounds()`

FrmGetObjectPosition Function

Purpose Returns the coordinates of the specified UI element relative to the form.

Declared In `Form.h`

Prototype

```
void FrmGetObjectPosition (const FormType *formP,
   uint16_t objIndex, Coord *x, Coord *y)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of a UI element in the form. You can obtain this by using FrmGetObjectIndex().

← *x*, *y*
Pointers where the window-relative x and y positions of the UI element are returned. These locate the top-left corner of the element.

Returns Nothing.

See Also FrmGetObjectBounds(), FrmSetObjectPosition()

FrmGetObjectPtr Function

Purpose Returns a pointer to the data structure of an element in a form.

Declared In Form.h

Prototype
```
void *FrmGetObjectPtr (const FormType *formP,
    uint16_t objIndex)
```

Parameters → *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

Returns A pointer to an element in the form.

See Also FrmGetObjectId()

FrmGetObjectType Function

Purpose Returns the type of a UI element.

Declared In Form.h

Prototype
```
FormObjectKind FrmGetObjectType
    (const FormType *formP, uint16_t objIndex)
```

Parameters → *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

Returns The FormObjectKind of the item specified.

FrmGetObjectUsable Function

Purpose Returns whether the UI element is usable.

Declared In Form.h

Prototype
```
Boolean FrmGetObjectUsable (const FormType *formP,
    uint16_t objIndex)
```

Parameters → *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

Returns `true` if the element is usable, meaning that it is considered part of the user interface, or `false` if the element is not usable. Elements that are not usable never appear on the screen. The function FrmHideObject() clears an element's usable bit to hide that the element.

FrmGetTitle Function

Purpose Returns a pointer to the title string of a form.

Declared In Form.h

Prototype
```
const char *FrmGetTitle (const FormType *formP)
```

Parameters → *formP*
Pointer to the form.

Returns A pointer to title string. `NULL` if there is no title string or there is an error finding it.

Comments This is a pointer to the internal structure itself, *not* to a copy.

See Also FrmCopyTitle(), FrmSetTitle()

FrmGetWindowHandle Function

Purpose Obtains the form's window handle.

Declared In `Form.h`

Prototype
```
WinHandle FrmGetWindowHandle
    (const FormType *formP)
```

Parameters → *formP*
Pointer to the form.

Returns The handle to the form's window.

FrmGotoForm Function

Purpose Sends a `frmCloseEvent` to the current form; sends a `frmLoadEvent` and a `frmOpenEvent` to the specified form.

Declared In `Form.h`

Prototype
```
void FrmGotoForm (DmOpenRef database,
    uint16_t formID)
```

Parameters → *database*
Open database containing the form resource.

→ *formID*
Resource ID of the form to display.

Returns Nothing.

Comments The default form event handler (`FrmHandleEvent()`) erases and disposes of a form when it receives a `frmCloseEvent`.

See Also `FrmPopupForm()`

FrmHandleEvent Function

Purpose Handles the event that has occurred in the form.

Declared In `Form.h`

Prototype
```
Boolean FrmHandleEvent (FormType *formP,
    EventType *eventP)
```

Parameters → *formP*
Pointer to the form.

→ *eventP*
Pointer to the event data structure (see EventType).

Returns `true` if the event was handled, or `false` if it was not.

Comments Never call this function directly. Call `FrmDispatchEvent()` instead. `FrmDispatchEvent()` passes events to a form's custom event handler and then, if the event was not handled, to this function.

WARNING! You should never call this function directly. You should call the `FrmDispatchEvent()` function instead.

Table 20.1 provides an overview of how `FrmHandleEvent()` handles different events.

Table 20.1 FrmHandleEvent() Actions

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
`ctlEnterEvent`	Passes the event to `CtlHandleEvent()` for the control specified in the event.
`ctlRepeatEvent`	Passes the event to `CtlHandleEvent()` for the control specified in the event.
`ctlSelectEvent`	Checks if the control is a pop-up trigger. If it is, the list associated with the pop-up trigger is displayed until the user makes a selection or touches the pen outside the bounds of the list. If a selection is made, a `popSelectEvent` is added to the event queue.
`fldEnterEvent` or `fldHeightChangedEvent`	Checks if a field or a table has the focus and passes the event to the appropriate handler (`FldHandleEvent()` or `TblHandleEvent()`). The table is also a container that may contain a field. If `TblHandleEvent()` receives a field event, it passes the event to the field contained within it.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
frmCloseEvent	Erases the form using FrmEraseForm() and releases any memory allocated for it using FrmDeleteForm().
frmControlPrvRefreshEvent	Passes the event to CtlHandleEvent() for the control specified in the event.
frmGadgetEnterEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmGadgetMiscEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmScrollPrvRefreshEvent	Passes the event and a pointer to the element the event occurred in to SclHandleEvent().
frmTitleEnterEvent	Tracks the pen, without blocking, until it is lifted. If it is lifted within the bounds of the form title, adds a frmTitleSelectEvent event to the event queue. FrmHandleEvent() does not block until the pen is lifted; it merely ensures that all future pen events are within the bounds of the title until a penUpEvent is received. Because of the multiprocessing nature of Palm OS Cobalt, a frmTitleSelectEvent is not guaranteed to be immediately preceded by an frmTitleEnterEvent.
frmTitleSelectEvent	Adds a keyDownEvent with the vchrMenu character to the event queue.
frmUpdateEvent	Redraws the dirty region of the form. If the entire form needs to be redrawn, calls FrmDrawForm().
insertionPointOffEvent	Passes the event on to the field that has the focus.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
insertionPointOnEvent	Passes the event on to the field that has the focus.
keyDownEvent	Passes the event to the handler for the element that has the focus. If no element has the focus and the event is a result of the user pressing one of the hardware buttons on the device, simulates a default button push if the form has a default button defined.
lstEnterEvent	Passes the event and a pointer to the element the event occurred in to LstHandleEvent().
menuCmdBarOpenEvent	Checks if a field or a table has the focus and passes the event to the appropriate handler (FldHandleEvent() or TblHandleEvent()), broadcasts the notification sysNotifyMenuCmdBarOpenEvent, and then displays the command tool bar.
menuEvent	Checks if the menu command is one of the system edit menu commands. The system provides a standard edit menu that contains the commands Undo, Cut, Copy, Paste, Select All, and Keyboard. FrmHandleEvent() responds to these commands.
penDownEvent; pen position in the bounds of the form	If this form doesn't currently have the focus, calls WinRequestFocus(), which requests the focus. If the form receives focus, a winFocusGainedEvent is sent to the form. Checks the list of UI elements contained by the form to determine if the pen is within the bounds of one. If it is, the appropriate handler is called to handle the event. For example, if the pen is in a control, CtlHandleEvent() is called.
penMoveEvent	Forwards to the event handler for the UI element that is tracking the pen.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
frmCloseEvent	Erases the form using FrmEraseForm() and releases any memory allocated for it using FrmDeleteForm().
frmControlPrvRefreshEvent	Passes the event to CtlHandleEvent() for the control specified in the event.
frmGadgetEnterEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmGadgetMiscEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmScrollPrvRefreshEvent	Passes the event and a pointer to the element the event occurred in to SclHandleEvent().
frmTitleEnterEvent	Tracks the pen, without blocking, until it is lifted. If it is lifted within the bounds of the form title, adds a frmTitleSelectEvent event to the event queue. FrmHandleEvent() does not block until the pen is lifted; it merely ensures that all future pen events are within the bounds of the title until a penUpEvent is received. Because of the multiprocessing nature of Palm OS Cobalt, a frmTitleSelectEvent is not guaranteed to be immediately preceded by an frmTitleEnterEvent.
frmTitleSelectEvent	Adds a keyDownEvent with the vchrMenu character to the event queue.
frmUpdateEvent	Redraws the dirty region of the form. If the entire form needs to be redrawn, calls FrmDrawForm().
insertionPointOffEvent	Passes the event on to the field that has the focus.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
frmCloseEvent	Erases the form using FrmEraseForm() and releases any memory allocated for it using FrmDeleteForm().
frmControlPrvRefreshEvent	Passes the event to CtlHandleEvent() for the control specified in the event.
frmGadgetEnterEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmGadgetMiscEvent	Passes the event to the gadget's callback function if the gadget has one. See FormGadgetHandlerType().
frmScrollPrvRefreshEvent	Passes the event and a pointer to the element the event occurred in to SclHandleEvent().
frmTitleEnterEvent	Tracks the pen, without blocking, until it is lifted. If it is lifted within the bounds of the form title, adds a frmTitleSelectEvent event to the event queue. FrmHandleEvent() does not block until the pen is lifted; it merely ensures that all future pen events are within the bounds of the title until a penUpEvent is received. Because of the multiprocessing nature of Palm OS Cobalt, a frmTitleSelectEvent is not guaranteed to be immediately preceded by an frmTitleEnterEvent.
frmTitleSelectEvent	Adds a keyDownEvent with the vchrMenu character to the event queue.
frmUpdateEvent	Redraws the dirty region of the form. If the entire form needs to be redrawn, calls FrmDrawForm().
insertionPointOffEvent	Passes the event on to the field that has the focus.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
insertionPointOnEvent	Passes the event on to the field that has the focus.
keyDownEvent	Passes the event to the handler for the element that has the focus. If no element has the focus and the event is a result of the user pressing one of the hardware buttons on the device, simulates a default button push if the form has a default button defined.
lstEnterEvent	Passes the event and a pointer to the element the event occurred in to LstHandleEvent().
menuCmdBarOpenEvent	Checks if a field or a table has the focus and passes the event to the appropriate handler (FldHandleEvent() or TblHandleEvent()), broadcasts the notification sysNotifyMenuCmdBarOpenEvent, and then displays the command tool bar.
menuEvent	Checks if the menu command is one of the system edit menu commands. The system provides a standard edit menu that contains the commands Undo, Cut, Copy, Paste, Select All, and Keyboard. FrmHandleEvent() responds to these commands.
penDownEvent; pen position in the bounds of the form	If this form doesn't currently have the focus, calls WinRequestFocus(), which requests the focus. If the form receives focus, a winFocusGainedEvent is sent to the form. Checks the list of UI elements contained by the form to determine if the pen is within the bounds of one. If it is, the appropriate handler is called to handle the event. For example, if the pen is in a control, CtlHandleEvent() is called.
penMoveEvent	Forwards to the event handler for the UI element that is tracking the pen.

Table 20.1 FrmHandleEvent() Actions *(continued)*

When FrmHandleEvent() receives...	FrmHandleEvent() performs these actions...
penUpEvent	Forwards to the event handler for the UI element that is tracking the pen.
popSelectEvent	Sets the label of the pop-up trigger to the current selection of the pop-up list.
sclEnterEvent or sclRepeatEvent	Passes the event and a pointer to the element the event occurred in to SclHandleEvent().
tblEnterEvent	Passes the event and a pointer to the element the event occurred in to TblHandleEvent(). The element's pointer is obtained from the event data.
tblExitEvent	Releases the focus.
tsmFepButtonEvent	Passes the event on to the field with the focus (if any).
tsmFepChangeEvent	Passes the event on to the field with the focus (if any).
tsmFepModeEvent	Passes the event on to the field with the focus (if any).
tsmFepSelectOptionEvent	Passes the event on to the field with the focus (if any).
winFocusGainedEvent	If the pen was tapped within a text field, gives focus to that field.
winFocusLostEvent	If a text field previously had focus, the focus is released from the field.
winResizedEvent	If FrmInitLayout() had been called on this form, calls FrmPerformLayout() to automatically rearrange and resize the elements on the form.
winUpdateEvent	Sends a frmUpdateEvent to the form's event handler.

FrmHelp Function

Purpose Displays the specified help message until the user taps the Done button in the help dialog.

Declared In `Form.h`

Prototype

```
void FrmHelp (DmOpenRef database,
    uint16_t helpMsgId)
```

Parameters

→ *database*
Open database containing the help string resource.

→ *helpMsgId*
Resource ID of help message string.

Returns Nothing.

Comments The help message is displayed in a modal dialog that supports scrolling the text if necessary.

FrmHelpWithFlags Function

Purpose Displays the specified help message until the user taps the Done button in the help dialog. Uses the attributes in *windowFlags* to create the help dialog.

Declared In `Form.h`

Prototype

```
void FrmHelpWithFlags (DmOpenRef database,
    uint16_t helpMsgId, WinFlagsType windowFlags)
```

Parameters

→ *database*
Open database containing the help string resource.

→ *helpMsgId*
Resource ID of help message string.

→ *windowFlags*
Window attributes to use when creating the dialog. See `WinFlagsType`. You must set `winFlagModal` to make the help dialog modal.

Returns Nothing.

Comments `FrmHandleEvent()` uses this function to create a help dialog if the dialog that contains the help trigger ("i" button) is in a window

layer other than the normal layer. You should not need to use this function directly.

See Also FrmHelp()

FrmHideObject Function

Purpose Erases the specified UI element and sets its attribute data so that it does not redraw or respond to the pen.

Declared In Form.h

Prototype
```
void FrmHideObject (FormType *formP,
    uint16_t objIndex)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of a UI element in the form. You can obtain this by using FrmGetObjectIndex().

Returns Nothing.

Comments For forms using update-based windows, the element may not be hidden immediately. This function invalidates the region of the form that displayed the element. The element will be hidden during the next drawing update cycle. In either case, the element is hidden by filling its region with the color set for UIFormFill.

See Also FrmShowObject()

FrmInitForm Function

Purpose Loads and initializes a form resource.

Declared In Form.h

Prototype
```
FormType *FrmInitForm (DmOpenRef database,
    uint16_t rscID)
```

Parameters

→ *database*
Open database containing the form resource.

→ *rscID*
Resource ID of the form.

Returns A pointer to the form data structure.

Comments This function does not draw the form (use `FrmDrawForm()` to draw the form) nor does it make the form active (use `FrmSetActiveForm()` to make it active).

For each initialized form, you must call `FrmDeleteForm()` to release the form memory when you are done with the form. `FrmHandleEvent()` does this for you in response to `frmCloseEvent`. Alternatively, you can free the active form by calling `FrmReturnToForm()`.

On debug ROMs, `FrmInitForm()` displays an error message if the form has already been initialized.

See Also `FrmInitForm()`

FrmInitFormWithFlags Function

Purpose Loads and initializes a form resource.

Declared In `Form.h`

Prototype

```
FormType *FrmInitFormWithFlags(DmOpenRef database,
    uint16_t rscID, WinFlagsType windowFlags)
```

Parameters

→ *database*
Open database containing the form resource.

→ *rscID*
Resource ID of the form.

→ *windowFlags*
Window attributes to use when creating the form. See `WinFlagsType`.

Returns A pointer to the form data structure.

Comments You typically specify the window attributes in the `WINDOW_CONSTRAINTS_RESOURCE` associated with your form and call `FrmInitForm()` instead of this function. It is used by other functions (such as `FrmAlertWithFlags()`) that create forms with window attribute flags.

On debug ROMs, `FrmInitFormWithFlags()` displays an error message if the form has already been initialized.

See Also `FrmDoDialog()`, `FrmDeleteForm()`, `FrmReturnToForm()`

FrmInitLayout Function

Purpose Prepares the form for automatic resizing by `FrmPerformLayout()`.

Declared In `FormLayout.h`

Prototype

```
status_t FrmInitLayout (FormType *formP,
   const FormLayoutType *formLayoutP)
```

Parameters

→ *formP*
Pointer to the form.

→ *formLayoutP*
An array of `FormLayoutType` structures that specify the resizing rule for each element in the form. Use 0 as the last element in the array.

NOTE: This array must exist for the entire life of the form.

Returns `errNone` upon success or `sysErrNoFreeRAM` if there is not enough memory available to initialize the layout structures.

Comments Call this function if you want your form automatically resized in response to the `winResizedEvent`. Call it as soon after `FrmInitForm()` as possible, which means you typically call this in response to `frmLoadEvent`. It's important to call it before the first `winResizedEvent` is received.

If you need to rearrange UI elements on the form, do so before you call `FrmInitLayout()`. `FrmInitLayout()` stores the position of each element in the form. If you move an element after calling `FrmInitLayout()`, it will snap back to its initial position and size upon the next `winResizedEvent`.

On debug ROMs, this function displays a fatal error message if called more than once.

frmLayoutRule Macro

Purpose Builds a rule that specifies how a UI element should be resized when its form is resized.

Declared In `FormLayout.h`

Prototype
```
#define frmLayoutRule (left, top, right, bottom)
```

Parameters → *left, top, right, bottom*
One of the "Form Basic Layout Constants" on page 382. See the Comments for more information.

Returns A constant that should be assigned to the rule field of a `FormLayoutType` structure.

Comments This macro is used to construct the Form Layout Constants. The provided constants should be sufficient.

The *top* and *bottom* parameters work together to specify the vertical resizing rule. If both are `frmAttachLeftTop`, the element stays in a fixed location. If both are `frmAttachRightBottom`, the element moves but remains the same size. If *top* is `frmAttachLeftTop` and *bottom* is `frmAttachRightBottom`, the element resizes at the same rate as the form resizes. If *top* is `frmAttachRightBottom` and *bottom* is `frmAttachLeftTop`, the element lengthens when the form shortens and shortens when the form lengthens.

The *left* and *right* parameters work together to specify the horizontal resizing rule in a similar manner.

See Also `FrmInitLayout()`, `FrmPerformLayout()`

FrmNewBitmap Function

Purpose Creates a new form bitmap element dynamically.

Declared In `Form.h`

Prototype
```
FormBitmapType *FrmNewBitmap (FormType **formPP,
    DmOpenRef database, uint16_t rscID, Coord x,
    Coord y)
```

Parameters ↔ *formPP*
Pointer to a pointer to the form in which the new bitmap is installed. This value is not a handle; that is, the old *formPP*

value is not necessarily valid after this function returns because the form may be moved in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *database*
Open database containing the bitmap that is to be displayed in the form bitmap element.

→ *rscID*
Resource ID of the bitmap that should be displayed in this form bitmap. This value must be unique within the application scope.

→ *x*
Horizontal coordinate of the upper-left corner of the bitmap's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the bitmap's boundaries, relative to the window in which it appears.

Returns A pointer to the new bitmap, or 0 if the call did not succeed. The most common cause of failure is lack of memory.

See Also FrmRemoveObject()

FrmNewForm Function

Purpose Creates a new form dynamically.

Declared In `Form.h`

Prototype

```
FormType *FrmNewForm (uint16_t formID,
   const char *titleStrP, Coord x, Coord y,
   Coord width, Coord height, Boolean modal,
   uint16_t defaultButton, DmOpenRef helpDatabase,
   uint16_t helpRscID, DmOpenRef menuDatabase,
   uint16_t menuRscID)
```

Parameters → *formID*
Symbolic ID of the form, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *titleStrP*
Pointer to a string that is the title of the form.

→ *x*
Horizontal coordinate of the upper-left corner of the form's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the form's boundaries, relative to the window in which it appears.

→ *width*
Width of the form, expressed in standard coordinates.

→ *height*
Height of the form, expressed in standard coordinates.

→ *modal*
`true` specifies that the form is a modal dialog.

→ *defaultButton*
Resource ID of the button that provides the form's default action.

→ *helpDatabase*
Open database containing the help string resource.

→ *helpRscID*
Resource ID of a help string resource that provides help for using this form. Only modal dialogs can have help resources.

→ *menuDatabase*
Open database containing the menu resource.

→ *menuRscID*
Resource ID of the form's menu.

Returns A pointer to the new form, or 0 if the call did not succeed. The most common cause of failure is lack of memory.

Comments This function creates a form with a legacy window that does not resize as the user opens and closes a dynamic input area. To create a form that dynamically resizes, use `FrmNewFormWithConstraints()`.

See Also `FrmValidatePtr()`, `WinValidateHandle()`, `FrmRemoveObject()`

FrmNewFormWithConstraints Function

Purpose Creates a new form dynamically using the specified window attributes.

Declared In `Form.h`

Prototype

```
FormType *FrmNewFormWithConstraints
   (uint16_t formID, const char *titleStrP,
   uint32_t win_flags,
   const WinConstraintsType *constraints,
   uint16_t defaultButton, DmOpenRef helpDatabase,
   uint16_t helpRscID, DmOpenRef menuDatabase,
   uint16_t menuRscID)
```

Parameters

→ *formID*
Symbolic ID of the form, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *titleStrP*
Pointer to a string that is the title of the form.

→ *windowFlags*
Window attributes to use when creating the form. See WinFlagsType.

→ *constraints*
Size constraints to use when creating the form. See WinConstraintsType.

→ *x*
Horizontal coordinate of the upper-left corner of the form's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the form's boundaries, relative to the window in which it appears.

→ *width*
Width of the form, expressed in standard coordinates.

→ *height*
Height of the form, expressed in standard coordinates.

→ *modal*
`true` specifies that the form is a modal dialog.

→ *defaultButton*
Resource ID of the button that provides the form's default action.

→ *helpDatabase*
Open database containing the help string resource.

→ *helpRscID*
Resource ID of a help string resource that provides help for using this form. Only modal dialogs can have help resources.

→ *menuDatabase*
Open database containing the menu resource.

→ *menuRscID*
Resource ID of the form's menu.

Returns A pointer to the new form, or 0 if the call did not succeed. The most common cause of failure is lack of memory.

See Also FrmNewForm(), FrmValidatePtr(), WinValidateHandle(), FrmRemoveObject()

FrmNewGadget Function

Purpose Creates a new gadget dynamically and installs it in the specified form.

Declared In Form.h

Prototype
```
FormGadgetType *FrmNewGadget (FormType **formPP,
    uint16_t id, Coord x, Coord y, Coord width,
    Coord height)
```

Parameters

↔ *formPP*
Pointer to a pointer to the form in which the new gadget is installed. This value is not a handle; that is, the old *formPP* value is not necessarily valid after this function returns because the form may be moved in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *id*
Symbolic ID of the gadget, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *x*
Horizontal coordinate of the upper-left corner of the gadget's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the gadget's boundaries, relative to the window in which it appears.

→ *width*
Width of the gadget, expressed in standard coordinates. Valid values are 1 - 160.

→ *height*
Height of the gadget, expressed in standard coordinates.Valid values are 1 - 160.

Returns A pointer to the new gadget or 0 if the call did not succeed. The most common cause of failure is lack of memory.

See Also FrmRemoveObject()

FrmNewGsi Function

Purpose Creates a new shift indicator dynamically and installs it in the specified form.

Declared In Form.h

Prototype
```
FrmGraffitiStateType *FrmNewGsi(FormType **formPP,
    Coord x, Coord y)
```

Parameters

↔ *formPP*
Pointer to a pointer to the form in which the new shift indicator is installed. This value is not a handle; that is, the old *formPP* value is not necessarily valid after this function returns because the form may be moved in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *x*
Horizontal coordinate of the upper-left corner of the shift indicator's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the shift indicator's boundaries, relative to the window in which it appears.

Returns A pointer to the new shift indicator or 0 if the call did not succeed. The most common cause of failure is lack of memory.

Comments If the device has a dynamic input area, the currently active pinlet displays the shift state (called the pinlet input mode). Forms with text fields should still include a shift indicator if they might be run on devices that do not have a dynamic input area, for example, if the device has a keyboard instead of an input area.

FrmNewLabel Function

Purpose Creates a new label dynamically and installs it in the specified form.

Declared In `Form.h`

Prototype

```
FormLabelType *FrmNewLabel (FormType **formPP,
   uint16_t ID, const char *textP, Coord x,
   Coord y, FontID font)
```

Parameters

↔ *formPP*
Pointer to a pointer to the form in which the new label is installed. This value is not a handle; that is, the old *formPP* value is not necessarily valid after this function returns because the form may be moved in memory. In subsequent calls, always use the new *formPP* value returned by this function.

→ *ID*
Symbolic ID of the label, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *textP*
Pointer to a string that provides the label text. This string is copied into the label structure.

→ *x*
Horizontal coordinate of the upper-left corner of the label's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the label's boundaries, relative to the window in which it appears.

→ *font*
FontID of the bitmapped font with which to draw the label text.

Returns A pointer to the new label or 0 if the call did not succeed. The most common cause of failure is lack of memory.

See Also CtlValidatePointer(), FrmRemoveObject()

FrmPerformLayout Function

Purpose Resizes or rearranges the elements in the form according to the layout rules specified for them in FrmInitLayout().

Declared In FormLayout.h

Prototype
```
void FrmPerformLayout (FormType *formP,
   const RectangleType *newBounds)
```

Parameters

→ *formP*
Pointer to a form.

→ *newBounds*
The bounds to which the form is being resized. See RectangleType.

Returns Nothing.

Comments FrmHandleEvent() calls this function in response to the winResizedEvent if FrmInitLayout() was called when the form was loaded.

See Also frmLayoutRule(), FormLayoutType

FrmPointInTitle Function

Purpose Checks if a coordinate is within the bounds of the form's title.

Declared In `Form.h`

Prototype
```
Boolean FrmPointInTitle (const FormType *formP,
    Coord x, Coord y)
```

Parameters

→ *formP*
Pointer to the form.

→ *x, y*
Window-relative x and y coordinates.

Returns `true` if the specified coordinate is in the form's title.

FrmPopupForm Function

Purpose Enqueues a `frmLoadEvent` and a `frmOpenEvent` for the specified form.

Declared In `Form.h`

Prototype
```
void FrmPopupForm (DmOpenRef database,
    uint16_t formID)
```

Parameters

→ *database*
Open database containing the form resource.

→ *formID*
Resource ID of form to open.

Returns Nothing.

Comments This function is intended to display modal dialogs only. It differs from `FrmGotoForm()` in that the current form is not closed before the new one is opened. You can call `FrmReturnToForm()` to close a form opened by `FrmPopupForm()`.

FrmRemoveBitmapHandle Function

Purpose Removes a bitmap handle from the form's bitmap cache.

Declared In `Form.h`

Prototype

```
void FrmRemoveBitmapHandle (FormType *form,
    DmOpenRef database, DmResourceType bitmapType,
    DmResourceID bitmapID, uint32_t flags)
```

Parameters

→ *form*
A pointer to the form.

→ *database*
Open database containing the bitmap resource.

→ *bitmapType*
The resource type of the bitmap resource.

→ *bitmapID*
The resource ID of the bitmap.

→ *flags*
One of the Bitmap Loading constants. These must be the same flags you passed to `FrmGetBitmapHandle()`.

Returns Nothing.

Comments Call this function to remove a previously cached bitmap from the form's cache after you no longer need it. As with all other form storage, the bitmap cache is freed when the form is freed, so making a separate call to this function is usually not necessary.

See Also `FrmGetBitmapHandle()`

FrmRemoveObject Function

Purpose Removes the specified UI element from the specified form.

Declared In `Form.h`

Prototype

```
status_t FrmRemoveObject (FormType **formPP,
    uint16_t objIndex)
```

Parameters

↔ *formPP*
Pointer to a pointer to the form from which this function removes an element. This value is not a handle; that is, the old *formPP* value is not necessarily valid after this function

returns. In subsequent calls, always use the new *formPP* value returned by this function.

→ *objIndex*
The index of the element to remove. Use the FrmGetObjectIndex() function to obtain this value.

Returns errNone if no error.

Comments You can use this function to remove any dynamically created element (a bitmap, control, list, and so on) and free the memory allocated to it within the form data structure. The data structures for most elements are embedded within the form data structure memory chunk. This function frees that memory and moves the other element, if necessary, to close up the memory "hole" and decrease the size of the form chunk.

Note that this function does not free memory outside the form data structure that may be allocated to an element, such as the memory allocated to the string in an editable field element.

See Also FrmNewBitmap(), FrmNewForm(), FrmNewGadget(), FrmNewLabel(), CtlNewControl(), FldNewField(), LstNewList()

FrmRestoreActiveState Function

Purpose Restores the active window and form state.

Declared In Form.h

Prototype
```
status_t FrmRestoreActiveState
    (FormActiveStateType *stateP)
```

Parameters → *stateP*
A pointer to the FormActiveStateType structure that you passed to FrmSaveActiveState() when you saved the state.

Returns errNone on success.

Comments Use this function to restore the state of displayed forms to the state that existed before you dynamically showed a new modal form. You must have previously called FrmSaveActiveState() to save the state.

FrmReturnToForm Function

Purpose Erases and deletes the currently active form and make the specified form the active form.

Declared In `Form.h`

Prototype
```
void FrmReturnToForm (uint16_t formID)
```

Parameters

→ *formID*
Resource ID of the form to return to.

Returns Nothing.

Comments Use this function to close a form that was displayed by `FrmPopupForm()`. It is assumed that the form being returned to is already loaded into memory and initialized. That is, unlike `FrmGotoForm()`, `FrmReturnToForm()` does *not* generate `frmLoadEvent` or `frmOpenEvent` for *formID*.

Passing a form ID of 0 returns to the first form in the window list, which is the last form to be loaded.

`FrmReturnToForm()` does not generate a `frmCloseEvent` when called from a modal form's event handler. It assumes that you have already handled cleaning up your form's variables since you are explicitly calling `FrmReturnToForm()`.

FrmSaveActiveState Function

Purpose Saves the active window and form state.

Declared In `Form.h`

Prototype
```
status_t FrmSaveActiveState
    (FormActiveStateType *stateP)
```

Parameters

↔ *stateP*
A pointer to a `FormActiveStateType` structure that is used to save the state. Pass the same pointer to `FrmRestoreActiveState()` to restore the state. Treat the structure like a black box; that is, don't attempt to read it or write to it.

Returns `errNone` on success.

Comments Use this function to save the state of displayed forms before dynamically showing a new modal form. Call `FrmRestoreActiveState()` to restore the state after you remove the modal form.

FrmSaveAllForms Function

Purpose Sends a `frmSaveEvent` to all open forms.

Declared In `Form.h`

Prototype
```
void FrmSaveAllForms (void)
```

Parameters None.

Returns Nothing.

See Also `FrmCloseAllForms()`

FrmSetActiveForm Function

Purpose Sets the active form. All input (key and pen) is directed to the active form, and all drawing occurs there until the next `WinSetDrawWindow()` call.

Declared In `Form.h`

Prototype
```
void FrmSetActiveForm (FormType *formP)
```

Parameters

→ *formP*
Pointer to the form.

Returns Nothing.

Comments This function calls `WinSetActiveWindow()`, which enqueues `winExitEvent` and `winEnterEvent`s and requests that *formP* be granted text input focus. Input focus is granted asynchronously and might not be granted if another process requests the input focus for a window in a higher layer at the same time. The form receives the `winFocusGainedEvent` when it is granted the input focus. Before this occurs, a `winFocusLostEvent` is sent to the window that had the text input focus before this function was called. Keep in mind that this window might have been part of another process.

See Also FrmGetActiveForm()

FrmSetCategoryLabel Function

Purpose Sets the category label displayed on the title line of a form. If the form's `visible` attribute is set, redraw the label.

Declared In `Form.h`

Prototype

```
void FrmSetCategoryLabel (const FormType *formP,
    uint16_t objIndex, char *newLabel)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using `FrmGetObjectIndex()`.

→ *newLabel*
Pointer to the name of the new category.

Returns Nothing.

Comments The pointer to the new label (*newLabel*) is saved in the element.

FrmSetControlGroupSelection Function

Purpose Sets the selected control in a group of controls.

Declared In `Form.h`

Prototype

```
void FrmSetControlGroupSelection
    (const FormType *formP, uint8_t groupNum,
    uint16_t controlID)
```

Parameters

→ *formP*
Pointer to the form.

→ *groupNum*
Control group number.

→ *controlID*
Resource ID of control to set.

Returns Nothing.

Comments This function unsets all the other controls in the group.

This function updates the display. For forms using update-based windows, the region displaying the control is invalidated so that it will be redrawn in the next update cycle.

NOTE: `FrmGetControlGroupSelection()` returns the selection in a control group as an element index, *not* as a resource ID, which `FrmSetControlGroupSelection()` uses to set the selection.

FrmSetControlValue Function

Purpose Sets the current value of a control. If the control is visible, it's invalidated for later redrawing.

Declared In `Form.h`

Prototype

```
void FrmSetControlValue (const FormType *formP,
    uint16_t objIndex, int16_t newValue)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of the control in the form. You can obtain this by using `FrmGetObjectIndex()`.

→ *newValue*
New value to set for the control. For sliders, specify a value between the slider's minimum and maximum. For graphical controls, push buttons, or check boxes, specify 0 for off, nonzero for on.

Returns Nothing.

Comments This function works only with graphical controls, sliders, push buttons, and check boxes. If you set the value of any other type of control, the behavior is undefined.

See Also FrmGetControlValue(), CtlSetValue()

FrmSetDefaultButtonID Function

Purpose Sets the default button for a modal dialog.

Declared In `Form.h`

Prototype

```
void FrmSetDefaultButtonID (FormType *formP,
    uint16_t defaultButton)
```

Parameters

→ *formP*
Pointer to the form.

→ *defaultButton*
Resource ID of the button that should be the default.

Returns Nothing.

Comments Modal dialogs have a default button assigned. If the user decides to exit the application rather than respond to the modal dialog, the system enqueues a `ctlSelectEvent` for the default button, and the application should behave as if this button were tapped. Other types of forms do not have the default button assigned.

You normally set the form's default button in the resource file. Use this function if you need to change the value at runtime.

See Also `FrmGetDefaultButtonID()`, `FrmDoDialog()`

FrmSetEventHandler Function

Purpose Registers the event handler callback routine for the specified form.

Declared In `Form.h`

Prototype

```
void FrmSetEventHandler (FormType *formP,
    FormEventHandlerType *handler)
```

Parameters

→ *formP*
Pointer to the form.

→ *handler*
Address of the form event handler function, `FormEventHandlerType()`.

Returns Nothing.

Comments `FrmDispatchEvent()` calls this handler whenever it receives an event for a specific form.

`FrmSetEventHandler()` must be called during the `frmLoadEvent`. The callback routine it registers is the mechanism for dispatching events to an application.

FrmSetFocus Function

Purpose Sets the focus of a form to the specified UI element.

Declared In `Form.h`

Prototype

```
void FrmSetFocus (FormType *formP,
    uint16_t fieldIndex)
```

Parameters

→ *formP*
Pointer to the form.

→ *fieldIndex*
Index of the element to get the focus in the form or `noFocus` to specify that no element has the focus. Obtain an element's index by using `FrmGetObjectIndex()`.

Returns Nothing.

Comments You can set the focus to a field or table. If the focus is set to a field, this function turns on the insertion point in the field.

See Also `FrmGetFocus()`

FrmSetGadgetData Function

Purpose Stores a data value in the `data` field of the gadget.

Declared In `Form.h`

Prototype

```
void FrmSetGadgetData (FormType *formP,
    uint16_t objIndex, const void *data)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of the gadget. You can obtain this by using `FrmGetObjectIndex()`.

→ *data*
Application-defined value. This value is stored into the `data` field of the gadget data structure (`FormGadgetType`).

Returns Nothing.

Comments Gadgets provide a way for an application to attach custom gadgetry to a form. Typically, the `data` field of a gadget contains a pointer to the custom element's data structure.

See Also `FrmGetGadgetData()`, `FrmSetGadgetHandler()`

FrmSetGadgetHandler Function

Purpose Registers the gadget event handler callback routine for the specified gadget on the specified form.

Declared In `Form.h`

Prototype

```
void FrmSetGadgetHandler (FormType *formP,
   uint16_t objIndex,
   FormGadgetHandlerType *attrP)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of the gadget. You can obtain this by using `FrmGetObjectIndex()`.

→ *attrP*
Address of the callback function. See `FormGadgetHandlerType()`.

Returns Nothing.

Comments This function sets the application-defined function that controls the specified gadget's behavior. This function is called when the gadget needs to be drawn, erased, deleted, or needs to handle an event.

See Also `FrmGetGadgetData()`, `FrmSetGadgetData()`

FrmSetHelpID Function

Purpose Designates the string resource that is to act as the form's help resource.

Declared In `Form.h`

Prototype

```
void FrmSetHelpID (FormType *formP,
    DmOpenRef database, uint16_t helpRscID)
```

Parameters

→ *formP*
Pointer to the form.

→ *database*
Open database containing the help string resource.

→ *helpRscID*
The resource ID of the string resource that is to be the form's help resource.

Returns Nothing.

Comments You normally set the form's help resource in the resource file. Use this function if you need to change the value at runtime.

See Also FrmGetHelpID()

FrmSetLabelFont Function

Purpose Sets the font used for a particular label that appears on a form.

Declared In `Form.h`

Prototype

```
void FrmSetLabelFont (FormType *formP,
    uint16_t labelID, FontID fontID)
```

Parameters

→ *formP*
Pointer to the form.

→ *labelID*
Resource ID of a label in the form (the element's type must be `frmLabelObj`).

→ *fontID*
FontID of the bitmapped font to be used for the label.

Returns Nothing.

Comments This function does nothing if either *labelID* is invalid or if the element indicated by *labelID* has a type other than `frmLabelObj`.

See Also `FrmGetObjectType()`, `FrmGetLabelFont()`

FrmSetMenu Function

Purpose Changes a form's menu bar and makes the new menu active.

Declared In `Form.h`

Prototype

```
void FrmSetMenu (FormType *formP,
   DmOpenRef database, uint16_t menuRscID)
```

Parameters

→ *formP*
Pointer to the form.

→ *database*
Open database containing the menu resource.

→ *menuRscID*
Resource ID of the menu.

Returns Nothing.

See Also `MenuGetActiveMenu()`

FrmSetObjectBounds Function

Purpose Sets the bounds or position of a UI element.

Declared In `Form.h`

Prototype

```
void FrmSetObjectBounds (FormType *formP,
   uint16_t objIndex,
   const RectangleType *bounds)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using `FrmGetObjectIndex()`.

→ *bounds*
Window-relative bounds (see RectangleType). For the following elements, this sets only the position of the top-left corner: label, bitmap, and shift indicator.

Returns Nothing.

Comments This function doesn't update or invalidate the form.

If you use this function, do not use FrmInitLayout() and FrmPerformLayout() to do automatic resizing in response to the winResizedEvent. FrmInitLayout() stores the initial position of a UI element. If you call it and then call FrmSetObjectBounds(), the new location or size is ignored during the next winResizedEvent.

See Also FrmGetObjectPosition(), FrmGetObjectBounds()

FrmSetObjectPosition Function

Purpose Sets the position of a UI element.

Declared In Form.h

Prototype
```
void FrmSetObjectPosition (FormType *formP,
    uint16_t objIndex, Coord x, Coord y)
```

Parameters

→ *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

→ *x*
Window-relative horizontal coordinate.

→ *y*
Window-relative vertical coordinate.

Returns Nothing.

See Also FrmGetObjectPosition(), FrmGetObjectBounds()

FrmSetPenTracking Function

Purpose Sets whether an element in the form is tracking the pen.

Declared In `Form.h`

Prototype

```
void FrmSetPenTracking (FormType *formP,
   Boolean track, const void *frmObj)
```

Parameters

→ *formP*
Pointer to the form.

→ *track*
`true` to track the pen or `false` otherwise.

→ *frmObj*
A pointer to the UI element that is tracking the pen.

Returns Nothing.

Comments This function is used internally to keep track of the UI element that is tracking the pen. Applications should only call it for a gadget that needs to track the pen. Make the call in the gadget's event handler in response to `frmGadgetEnterEvent`.

This function is used in `FrmHandleEvent()` to set the form's internal pen tracking structures. If you call it outside of `FrmHandleEvent()`, you must pass an element that is part of the form's internal list of elements. If you don't, the function attempts to set the form's internal pen tracking index to specify the correct user interface element. This correction works most of the time; however, it is best to avoid calling this function except for the circumstances described above.

See Also `FrmAmIPenTracking()`

FrmSetTitle Function

Purpose Sets the title of a form. If the form is visible, draw the new title.

Declared In `Form.h`

Prototype

```
void FrmSetTitle (FormType *formP,
   char *newTitle)
```

Parameters

→ *formP*
Pointer to the form.

→ *newTitle*
Pointer to the new title string.

Returns Nothing.

Comments This function invalidates the region that displays the title so that it is redrawn in the next update cycle.

This function saves the pointer passed in *newTitle*; it does *not* make a copy. The value of *newTitle* must not be a pointer to a stack-based string.

See Also FrmGetTitle(), FrmCopyTitle(), FrmCopyLabel()

FrmShowObject Function

Purpose Sets a form element as usable. If the form is visible, invalidates the region that displays the element so that it is redrawn.

Declared In Form.h

Prototype
```
void FrmShowObject (FormType *formP,
    uint16_t objIndex)
```

Parameters → *formP*
Pointer to the form.

→ *objIndex*
Index of an element in the form. You can obtain this by using FrmGetObjectIndex().

Returns Nothing.

Comments For forms using update-based windows, the UI element may not be displayed immediately. It will be displayed during the next drawing update cycle.

See Also FrmHideObject()

FrmUIAlert Function

Purpose Displays an alert that is stored in the UI library's resource database.

Declared In `Form.h`

Prototype
```
uint16_t FrmUIAlert (uint16_t alertId)
```

Parameters

→ *alertId*
Resource ID of the alert. The header file `UIResources.h` contains a list of alerts that can be displayed by this function.

Returns The item number of the button the user selected. A button's item number is determined by its order in the alert dialog; the leftmost button has the item number 0 (zero).

NOTE: A default button press is simulated if the user switches to a different application while a modal dialog is active.

See Also `FrmAlert()`

FrmUpdateForm Function

Purpose Sends a `frmUpdateEvent` to the specified form.

Declared In `Form.h`

Prototype
```
void FrmUpdateForm (uint16_t formID,
    uint16_t updateCode)
```

Parameters

→ *formID*
Resource ID of form to update.

→ *updateCode*
An application-defined code that can be used to indicate what needs to be updated. Specify the code `frmRedrawUpdateCode` to indicate that the whole form should be redrawn.

Returns Nothing.

Comments

WARNING! This function is for compatibility purposes only. Do not use it for forms with update-based windows. It does not produce the graphics context needed to draw to an update-based window. Use `WinInvalidateRect()`, `WinInvalidateRectFunc()`, or `WinInvalidateWindow()` instead.

If the `frmUpdateEvent` posted by this function is handled by the default form event handler, `FrmHandleEvent()`, the *updateCode* parameter is ignored. `FrmHandleEvent()` always redraws the form.

If you handle the `frmUpdateEvent` in a custom event handler, you can use the *updateCode* parameter any way you want. For example, you might use it to indicate that only a certain part of the form needs to be redrawn. If you do handle the `frmUpdateEvent`, be sure to return `true` from your event handler so that the default form handler does not also redraw the whole form.

If you do handle the `frmUpdateEvent` in a custom event handler, be sure to handle the case where *updateCode* is set to `frmRedrawUpdateCode`, and redraw the whole form. This event (and code) is sent by the system when the whole form needs to be redrawn because the display needs to be refreshed.

FrmUpdateScrollers Function

Purpose

Visually updates (show or hide) the field scroll arrow buttons.

Declared In

`Form.h`

Prototype

```
void FrmUpdateScrollers (FormType *formP,
    uint16_t upIndex, uint16_t downIndex,
    Boolean scrollableUp, Boolean scrollableDown)
```

Parameters

→ *formP*

Pointer to the form.

→ *upIndex*

Index of the scroll up button. You can obtain this by using `FrmGetObjectIndex()`.

→ *downIndex*
Index of the scroll down button. You can obtain this by using `FrmGetObjectIndex()`.

→ *scrollableUp*
Set to `true` to make the up scroll arrow active (shown), or `false` to hide it.

→ *scrollableDown*
Set to `true` to make the down scroll arrow active (shown), or `false` to hide it.

Returns Nothing.

FrmValidatePtr Function

Purpose Returns `true` if the specified pointer references a valid form.

Declared In `Form.h`

Prototype
```
Boolean FrmValidatePtr (const FormType *formP)
```

Parameters → *formP*
Pointer to be tested.

Returns `true` if the specified pointer is a non-`NULL` pointer to an element having a valid form structure.

Comments This function is intended for debugging purposes only. Do not include it in released code.

FrmVisible Function

Purpose Returns `true` if the form is visible (is drawn).

Declared In `Form.h`

Prototype
```
Boolean FrmVisible (const FormType *formP)
```

Parameters → *formP*
Pointer to the form.

Returns `true` if the form is visible; `false` if it is not visible.

See Also `FrmDrawForm()`, `FrmEraseForm()`

Application-Defined Functions

FormCheckResponseFuncType Function

Purpose Callback function for `FrmCustomResponseAlert()`.

Declared In `Form.h`

Prototype

```
Boolean FormCheckResponseFuncType
    (int16_t button, char *attempt)
```

Parameters

→ *button*
The ID of the button that the user tapped or a constant specifying an action that the callback should take. See the Comments section for details.

→ *attempt*
The string that the user entered in the alert dialog.

Returns `true` if the dialog should be dismissed. Returns `false` if the dialog should not be dismissed.

Comments This function is called at these times during the `FrmCustomResponseAlert()` routine:

- At the beginning of `FrmCustomResponseAlert()`, this function is called with a button ID of `frmResponseCreate`. This constant indicates that the dialog is about to be displayed, and your function should perform any necessary initialization. For example, on a Japanese system, a password dialog might need to disable the Japanese FEP. So it would call `TsmSetFepMode(NULL, tsmFepModeOff)` in this function.
- When the user has tapped a button on the dialog. The function should process the *attempt* string. If the string is valid input, the function should return `true`. If not, it should return `false` to give the user a chance to re-enter the string.
- At the end of `FrmCustomResponseAlert()`, this function is called with a button ID of `frmResponseQuit`. This gives the callback a change to perform any cleanup, such as re-enabling the Japanese FEP.

FormEventHandlerType Function

Purpose The event handler callback routine for a form.

Declared In `Form.h`

Prototype
```
Boolean FormEventHandlerType (EventType *eventP)
```

Parameters

→ *eventP*
Pointer to the event.

Returns `true` if this routine handled the event, otherwise `false`.

Comments This function is where you respond to all of the events that the form receives other than `frmLoadEvent`.

`FrmDispatchEvent()` calls this handler whenever it receives an event for the form.

This callback routine is the mechanism for dispatching events to particular forms in an application. The callback is registered by the routine `FrmSetEventHandler()`.

FormGadgetHandlerType Function

Purpose The event handler callback for an extended gadget.

Declared In `Form.h`

Prototype
```
Boolean (FormGadgetHandlerType)
    (struct FormGadgetTypeInCallback *gadgetP,
    uint16_t cmd, void *paramP)
```

Parameters

→ *gadgetP*
Pointer to the gadget structure. See `FormGadgetTypeInCallback`.

→ *cmd*
A constant that specifies what action the handler should take. This can be one of the following:

`formGadgetDeleteCmd`
Sent by `FrmDeleteForm()` to indicate that the gadget is being deleted and must clean up any memory it has allocated or perform other cleanup tasks.

`formGadgetDrawCmd`
Sent in response to the `frmUpdateEvent` to indicate that the gadget must be drawn or redrawn.

`formGadgetEraseCmd`
Sent by `FrmHideObject()` to indicate that the gadget is going to be erased. `FrmHideObject()` clears the `visible` flag for you. If you return `false`, it also clears the `usable` flag and invalidates the window region that displays the gadget so that it is erased on the next update cycle.

`formGadgetHandleEventCmd`
Sent by `FrmHandleEvent()` to indicate that a gadget event has been received. The *paramP* parameter contains the pointer to the `EventType` structure.

→ *paramP*
`NULL` except if *cmd* is `formGadgetHandleEventCmd`. In that case, this parameter holds the pointer to the `EventType` structure containing the event.

Returns `true` if the event was handed successfully; `false` otherwise.

Comments If this function performs any drawing in response to the `formGadgetDrawCmd`, it should set the gadget's `visible` attribute flag. (`gadgetP->attr.visible = true`). This flag indicates that the gadget appears on the screen. If you don't set the `visible` flag, the gadget won't be erased when `FrmHideObject()` is called. (`FrmHideObject()` immediately returns if the element's `visible` flag is `false`.)

Note that if the function receives the `formGadgetEraseCmd`, it may simply choose to perform any necessary cleanup and return `false`. If the function returns `false`, `FrmHideObject()` erases the gadget's bounding rectangle. If the function returns `true`, it must erase the gadget area itself.

If this function receives a `formGadgetHandleEventCmd`, *paramP* points one of the following events:

- `frmGadgetEnterEvent` (if a pen down occurs within the gadget's bounds)
- `frmGadgetMiscEvent` (if the application specifically sends it)

- penMoveEvent (if the gadget called FrmSetPenTracking() in response to frmGadgetEnterEvent)
- penUpEvent (if the gadget called FrmSetPenTracking() in response to frmGadgetEnterEvent)
- frmObjectFocusTakeEvent (if the user navigates to the gadget)
- frmObjectFocusLostEvent (if the user navigates away from the gadget)
- keyDownEvent containing vchrRocker*xxx* characters (if the gadget has the navigation focus)

See Also FrmSetGadgetHandler()

21

Graphics Context Reference

This chapter describes the graphics context API declared in the header file `GcRender.h`. It discusses the following topics:

For more information on how to work with the graphics context, see Chapter 8, "Drawing," on page 123 and Chapter 9, "Working with Bitmaps," on page 147.

Graphics Context Data Structures and Types

GcBitmapHandle Struct

Purpose The handle to an internal structure that represents the bitmap.

Declared In `GcRender.h`

Prototype

```
typedef struct GcBitmapType *GcBitmapHandle
```

Fields None.

See Also `GcDrawBitmapAt()`, `GcDrawRawBitmapAt()`, `GcPaintBitmap()`, `GcLoadBitmap()`, `GcReleaseBitmap()`, `FrmGetBitmapHandle()`

GcBitmapType Struct

Purpose Internal structure that stores bitmap data in a format that is most efficient for the rendering system to display on screen.

Declared In `GcRender.h`

Prototype

```
struct GcBitmapType
```

Fields None.

See Also `GcLoadBitmap()`, `GcBitmapHandle`

GcColorType Struct

Purpose Represents a color.

Declared In `GcRender.h`

Prototype

```
typedef struct {
   uint8_t red;
   uint8_t green;
   uint8_t blue;
   uint8_t alpha;
} GcColorType
```

Fields

`red`
The red value.

`green`
The green value.

`blue`
The blue value.

`alpha`
The level of transparency, with 0 being transparent and 255 being fully opaque.

See Also `GcInitGradient()`, `RGBColorType`

GcContextType Struct

Purpose Internal structure that keeps track of the current rendering state.

Declared In `GcRender.h`

Prototype
```
struct GcContextType
```

Fields None.

See Also `GcHandle`

GcHandle Typedef

Purpose The handle of a graphics context.

Declared In `GcRender.h`

Prototype
```
typedef struct GcContextType *GcHandle
```

Fields None.

Comments The graphics context keeps track of a current rendering state. You obtain a `GcHandle` to the graphics context using `GcGetCurrentContext()`, perform drawing, and then release it using `GcReleaseContext()` when you are finished.

See Also `GcCreateBitmapContext()`

GcGradientType Struct

Purpose Represents a color gradient.

Declared In `GcRender.h`

Prototype
```
typedef struct {
   float Rdx;
   float Gdx;
   float Bdx;
   float Adx;
   float Rdy;
   float Gdy;
   float Bdy;
   float Ady;
   float R;
```

```
   float G;
   float B;
   float A;
   uint32_t _reserved;
} GcGradientType
```

Fields

`Rdx`
: Change in red per unit step in the horizontal direction.

`Gdx`
: Change in green per unit step in the horizontal direction.

`Bdx`
: Change in blue per unit step in the horizontal direction.

`Adx`
: Change in the alpha transparency value per unit step in the horizontal direction.

`Rdy`
: Change in red per unit step in the vertical direction.

`Gdy`
: Change in green per unit step in the vertical direction.

`Bdy`
: Change in blue per unit step in the vertical direction.

`Ady`
: Change in the alpha transparency value per unit step in the vertical direction.

`R`
: Initial red value.

`G`
: Initial green value.

`B`
: Initial blue value.

`A`
: Initial alpha transparency value.

`_reserved`
: Reserved for future use. Always set to 0.

See Also `GcSetGradient()`, `GcInitGradient()`

Graphics Context Constants

Bitmap Loading Enum

Purpose Flags to pass to `GcLoadBitmap()`, `GcDrawRawBitmapAt()`, and `FrmGetBitmapHandle()` to specify how the bitmap should be converted when it is loaded.

Declared In `GcRender.h`

Constants

`kLoadBitmapUnscaled = 0x00000001`
Do not scale the bitmap if its density does not match the screen density.

Note that this is different from using `WinSetScalingMode()` with `kBitmapScalingOff`. `kBitmapScalingOff` means use the low-density member of a bitmap family and do not scale it. `kLoadBitmapUnscaled` simply means do not scale the bitmap but does not affect which bitmap is selected from a bitmap family.

`kLoadBitmapMask = 0x00000001`
Mask of available flags for loading bitmaps.

GcCapTag Enum

Purpose Specifies what a line ending looks like.

Declared In `GcRender.h`

Constants

`kButtCap = 0`
There is no special line ending. This is the default.

`kRoundCap = 1`
The line end is rounded.

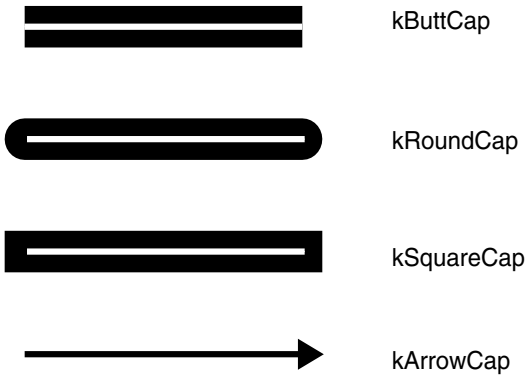
`kSquareCap = 2`
The line end is squared off.

`kArrowCap = 3`
The line end has an arrow head.

Comments Figure 21.1 shows the possible line cap endings. To help you better understand the differences between the line endings, the endings are shown for a very thick pen width with the actual line traced in

white. As you can see, `kButtCap` draws the line with no special ending, `kRoundCap` adds a cap with rounded edges, and `kSquareCap` adds a cap with squared edges. Caps are drawn outside the bounds of the line.

Figure 21.1 Line cap styles



GcJoinTag Enum

Purpose Specify what the corner looks like when two lines in a path connect (see Figure 21.2).

Declared In `GcRender.h`

Constants

`kMiterJoin = 0`
The two lines are joined with a square corner as shown on the left side of Figure 21.2. This type of join requires more processing than `kBevelJoin` and thus should be avoided if unnecessary.

`kRoundJoin = 1`
The corner is rounded as shown in the center of Figure 21.2. Do not use this if you are drawing curved lines (for example, if you are using `GcArc()` or `GcBezierTo()`).

`kBevelJoin = 2`
The corner is angled as shown on the right side of Figure 21.2. This is the default.

Comments These constants are part of the rendering state, but they only affect the `GcPaint()` operation if `GcStroke()` has been called first.

Figure 21.2 Line joins

See Also `GcSetJoin()`

Transform Indexes Enum

Purpose Index values to use for the matrix array that you pass as a parameter to `GcTransform()`.

Declared In `GcRender.h`

Constants

`kGcXX = 0`
The top left coordinate in the matrix.

`kGcYX = 1`
The leftmost coordinate in the middle row of the matrix.

`kGcXY = 2`
The middle coordinate in the top row of the matrix.

`kGcYY = 3`
The middle coordinate in the second row of the matrix.

`kGcXT = 4`
The rightmost coordinate in the top row of the matrix.

`kGcYT = 5`
The rightmost coordinate in the middle row of the matrix.

Comments These constants define coordinates in the general transformation matrix shown in Figure 21.3.

Figure 21.3 Transformation matrix

$$\begin{bmatrix} kGcXX & kGcXY & kGcXT \\ kGcYX & kGcYY & kGcYT \\ 0 & 0 & 1 \end{bmatrix}$$

The coordinates in this matrix are applied to each point in the current path according to the following equations:

newX = (kGcXX * *currentX*) + (kGcXY * *currentY*) + kGcXT

newY = (kGcYX * *currentX*) + (kGcYY * *currentY*) + kGcYT

where (*currentX, currentY*) represents a point on the path and (*newX, newY*) represents that point after the transformation has been applied.

Graphics Context Functions

GcArc Function

Purpose Adds an arc to the current path.

Declared In GcRender.h

Prototype

```
void GcArc (GcHandle ctxt, fcoord_t centerX,
    fcoord_t centerY, fcoord_t radx, fcoord_t rady,
    float startAngle, float arcLen,
    Boolean connected)
```

Parameters

→ *ctxt*
The graphics context.

→ *centerX*
The x coordinate of the arc's center.

→ *centerY*
The y coordinate of the arc's center.

→ *radx*
The horizontal radius.

→ *rady*
The vertical radius.

→ *startAngle*
The angle (in radians) from the horizontal radius to the beginning of the arc.

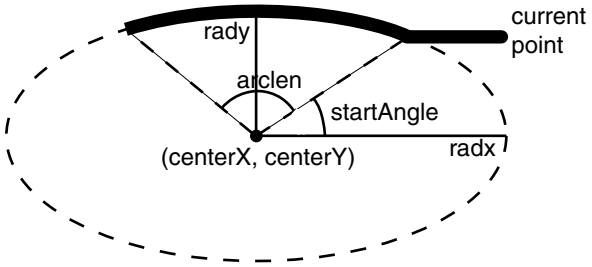
→ *arcLen*
The angle (in radians) of the arc itself. This angle must be a positive value.

→ *connected*
If `true`, a line is drawn from the center point to the arc. If `false`, the arc is drawn without connecting to the center point.

Returns Nothing.

Comments Figure 21.4 illustrates how the rendering system computes where to draw the arc. It constructs a template ellipse whose horizontal radius is *radx* and vertical radius is *rady*. The starting point of the arc is determine by the *startAngle*, and the ending point is determined by the *arcLen* angle. The endpoint of the arc becomes the current point.

Figure 21.4 GcArc example



If *connected* is `true`, a straight line is drawn from the current point to the center point and then another straight line is drawn from the center point to the beginning of the arc. If *connected* is `false`, a straight line is drawn from the current point to the beginning of the arc unless the current point is the same as the center point. If you don't want the arc connected to any previous drawing, call `GcMoveTo()` with *centerX* and *centerY* before calling this function.

The arc is actually constructed as a series of connected Bezier curves. If you are drawing a stroked path (that is, you want to draw the outline of a shape instead of filling it), do not use `kRoundJoin` as the line join. Leave the line join setting at the default.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

Compatibility Palm OS® Cobalt version 6.0 always draws arcs clockwise (as you are looking at the screen). There is no way to draw a counterclockwise arc; however, one may be added in the future.

See Also `GcArcTo()`

GcArcTo Function

Purpose Computes an arc using a radius and two tangent lines and adds the arc to the current path.

Declared In `GcRender.h`

Prototype

```
void GcArcTo (GcHandle ctxt, fcoord_t p1x,
    fcoord_t p1y, fcoord_t p2x, fcoord_t p2y,
    float radius)
```

Parameters

→ *ctxt*
The graphics context.

→ *p1x, p1y*
The point where the two tangent lines intersect. The first tangent is from the path's current point to this point.

→ *p2x, p2y*
The end point of the second tangent line. The second tangent line goes from (*p1x*, *p1y*) to (*p2x*, *p2y*).

→ *radius*
The length in coordinates of the arc's radius. Use –1 for this value if you want the arc to be drawn from the current point to (*p2x*, *p2y*). Note that in this case, the (*p1x*, *p1y*) coordinate must be equidistant from the current point and (*p2x*, *p2y*).

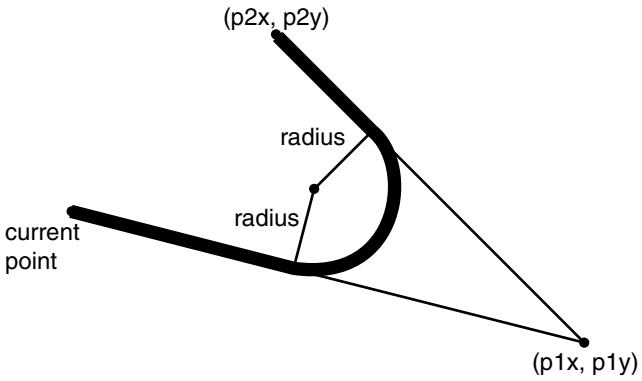
Returns Nothing.

Comments This function is generally useful for drawing boxes with rounded corners; however, the convenience function `GcRoundRect()` is more efficient.

As shown in Figure 21.5, this function constructs an arc using two tangent lines and the specified radius. The first tangent line goes

from the current point to (*p1x*, *p1y*). The second line goes from (*p1x*, *p1y*) to (*p2y*, *p2y*). Next, the center of the arc is computed by finding the point that is *radius* distance from the two tangents at a perpendicular angle. Finally, this function adds a path that is a straight line from the current point to the arc's beginning, then the curved arc, and then a straight line from the end of the arc to (*p2x*, *p2y*). (*p2x*, *p2y*) becomes the current point.

Figure 21.5 GcArcTo() example



Notice that if the radius is very large, the arc may be drawn off screen.

The arc is actually constructed as a series of connected Bezier curves. If you are drawing a stroked path (that is, you want to draw the outline of a shape instead of filling it), do not use `kRoundJoin` as the line join. Leave the line join setting at the default.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

Compatibility Palm OS Cobalt version 6.0 always draws arcs clockwise (as you are looking at the screen). There is no way to draw a counterclockwise arc; however, one may be added in the future.

See Also `GcArc()`, `GcLineTo()`

GcBeginClip Function

Purpose Begins a block of code that specifies a clipping region.

Declared In `GcRender.h`

Prototype

```
void GcBeginClip (GcHandle ctxt, Boolean inverse)
```

Parameters

→ *ctxt*
The graphics context.

→ *inverse*
If `true`, the region that you define will be the inverse of the clipping region. If `false`, the region you define will be the clipping region.

Returns Nothing.

Comments To specify a clipping region, do the following:

1. Call `GcPushState()`.
2. Call `GcBeginClip()`.
3. Call one or more of the path definition functions to specify the clipping region.
4. Call `GcPaint()`, which fills the specified region with white.
5. Call `GcEndClip()`.

After the `GcEndClip()` call, all drawing is constrained to the region that you defined using this procedure. (If you passed `true` for the *inverse* parameter, all drawing is constrained to the region that lies outside of the region you've defined.) The antialiasing value is ignored when the clipping region is specified.

To clear the clipping region, call `GcPopState()`. Nested clipping regions are not allowed; that is, you cannot call `GcBeginClip()` twice without calling `GcEndClip()`. You can, however, define two separate clipping regions in the current rendering state. If you do, the second clipping region is constrained by the first so that the resulting region becomes the intersection of the two regions.

GcBezierTo Function

Purpose Add a Bezier curve to the path.

Declared In `GcRender.h`

Prototype

```
void GcBezierTo (GcHandle ctxt, fcoord_t p1x,
    fcoord_t p1y, fcoord_t p2x, fcoord_t p2y,
    fcoord_t p3x, fcoord_t p3y)
```

Parameters

→ *ctxt*
The current graphics context.

→ *p1x, p1y*
The point that controls the angle and direction of the first curve.

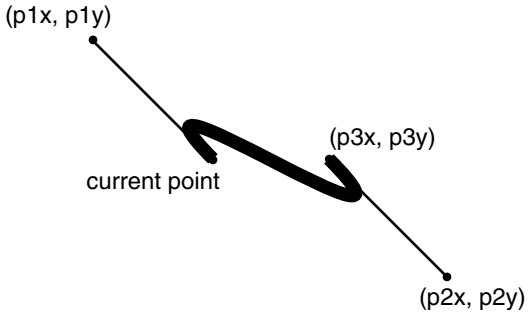
→ *p2x, p2y*
The point that controls the angle and direction of the second curve.

→ *p3x, p3y*
The end point of the line.

Returns Nothing.

Comments As shown in Figure 21.6, this function constructs a template consisting of straight lines from the current point to (*p1x*, *p1y*) and from (*p2x*, *p2y*) to (*p3x*, *p3y*). The path it constructs begins at the current point, is tangent to the line that starts at the point, and goes in the direction of (*p1x*, *p1y*). The end of the path is at (*p3x*, *p3y*). The ending part of the path is tangent to the line from (*p2x*, *p2y*) to (*p3x*, *p3y*) and heads in the direction of (*p2x*, *p2y*). (*p3x*, *p3y*) becomes the current point.

Figure 21.6 Bezier curve



If you are drawing a stroked path (that is, you want to draw the outline of the shape instead of filling it), do not use `kRoundJoin` as the line join. Leave the line join setting at the default.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

See Also `GcArc()`, `GcArcTo()`

GcClosePath Function

Purpose Closes the current subpath using a straight line.

Declared In `GcRender.h`

Prototype
```
void GcClosePath (GcHandle ctxt)
```

Parameters

→ *ctxt*
The rendering context containing the path to close.

Returns Nothing.

Comments This function draws a straight line from the current point to the point specified by the most recent `GcMoveTo()` call in the current path. This starting point becomes the current point.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

GcCommit Function

Purpose Blocks until all pending drawing operations are completed.

Declared In `GcRender.h`

Prototype
```
void GcCommit (GcHandle ctxt)
```

Parameters

→ *ctxt*
The rendering context to commit.

Returns Nothing.

Comments Use this function in the rare circumstances where you need to wait for the rendering system to finish before your application continues.

The rendering system runs in a different process than the application. To avoid IPC overhead, drawing operations are buffered and sent to the rendering system in batches. Because of this, rendering is likely to have not finished, or even started, when a drawing function returns. `GcCommit()` causes all buffered operations to be sent immediately and then waits for drawing to be completed before it returns.

This function is a performance drain and should be used rarely, only when absolutely necessary.

Note that you do not need to use `GcCommit()` when drawing to a local bitmap. All drawing to local bitmaps is performed immediately.

See Also `GcFlush()`

GcCreateBitmapContext Function

Purpose Create a rendering context in a given bitmap.

Declared In `GcRender.h`

Prototype

```
GcHandle GcCreateBitmapContext
    (const BitmapType *bitmapP)
```

Parameters

→ *bitmapP*
The bitmap to use when creating the context.

Returns A handle to the newly created graphics context or `NULL` if one cannot be created. See `GcHandle`.

Comments Use this function if you want to modify a bitmap.

You pass the `GcHandle` returned by this function to the drawing operations such as `GcBezierTo()` and `GcLineTo()`. When you are finished, you call `GcPaint()` and then `GcReleaseContext()`.

It is strongly recommended that you never define a color table for bitmaps. Drawing to a bitmap with a color table is much slower than drawing to a bitmap that uses the default color table. If you must define a color table for your bitmap, when you are ready to draw it to the screen, use `WinPalette()` to change the screen's color table to match your bitmap's color table before you call point

the bitmap to the screen. Drawing a bitmap whose color table does not match the color table of the destination is much slower.

Note that the `BmpCreate()` function creates a single-density bitmap, but the default coordinate system for the context created by `GcCreateBitmapContext()` is native screen density.

Example The following example creates a small high-density bitmap containing a red triangle.

```
BitmapTypeV3 *bmpP;
GcHandle bitmapContext;

bmpP = BmpCreate(12, 12, 16, NULL, &error);
bmpP = BmpCreateBitmapV3(bmpP, kDensityDouble,
       BmpGetBits(bmpP), NULL);
if (bmpP) {
   bitmapContext = GcCreateBitmapContext(bmpP);
   if (bitmapContext) {
      GcSetColor(bitmapContext, 255, 0, 0, 255);
      GcMoveTo(bitmapContext, 2, 2);
      GcLineTo(bitmapContext, 2, 10);
      GcLineTo(bitmapContext, 10, 10);
      GcClosePath(bitmapContext);
      GcPaint(bitmapContext);
      GcReleaseContext(bitmapContext);
   }
}
```

See Also `GcReleaseContext()`

GcDrawBitmapAt Function

Purpose Paints the bitmap at the specified coordinates.

Declared In `GcRender.h`

Prototype

```
void GcDrawBitmapAt (GcHandle ctxt,
    GcBitmapHandle bitmapHandle,
    const FAbsRectType *srcRect, fcoord_t x,
    fcoord_t y)
```

Parameters

→ *ctxt*
The graphics context.

→ *bitmapHandle*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

→ *srcRect*
Rectangle (see `FAbsRectType`) that specifies the portion of the bitmap to be drawn to the screen. Pass `NULL` to draw the entire bitmap. The rectangle must be less than or equal to the bitmap's dimensions and must be expressed in terms of the coordinate system used by the bitmap (that is, the bitmap's density). If you pass a larger rectangle, the bitmap is not tiled.

→ *x, y*
Top-left corner of where the bitmap should be drawn.

Returns Nothing.

Comments This function defines the path into where the bitmap should be drawn, paints the bitmap, and then clears the current path but no other aspect of the rendering state. Most of the rendering state does not affect this call.

See Also `GcDrawRawBitmapAt()`

GcDrawRawBitmapAt Function

Purpose Paints a bitmap at the specified coordinates.

Declared In `GcRender.h`

Prototype

```
void GcDrawRawBitmapAt (GcHandle ctxt,
   const BitmapType *bitmapP,
   const FAbsRectType *srcRect, fcoord_t x,
   fcoord_t y, uint32_t load_flags)
```

Parameters

→ *ctxt*
The graphics context.

→ *bitmapP*
The bitmap to draw. See `BitmapType`.

→ *srcRect*
Rectangle (see `FAbsRectType`) that specifies the portion of the bitmap to be drawn to the screen. Pass `NULL` to draw the entire bitmap. The rectangle must be less than or equal to the bitmap's dimensions and must be expressed in terms of the

coordinate system used by the bitmap (that is, the bitmap's density). If you pass a larger rectangle, the bitmap is not tiled.

→ *x, y*
Top-left corner of where the bitmap should be drawn.

→ *load_flags*
One of the Bitmap Loading constants, which specify whether the bitmap should be scaled.

Returns Nothing.

Comments This function creates a `GcBitmapHandle` for *bitmapP*, draws it to the screen, and then releases the bitmap handle. You are responsible for loading *bitmapP* from the database and releasing it when you are done.

See Also `GcLoadBitmap()`, `GcDrawBitmapAt()`

GcDrawTextAt Function

Purpose Draws text to the screen using the font specified by `GcSetFont()`.

Declared In `GcRender.h`

Prototype

```
void GcDrawTextAt (GcHandle ctxt, fcoord_t x,
    fcoord_t y, const char *text, int32_t length)
```

Parameters

→ *ctxt*
The graphics context.

→ *x*
The x coordinate where the text should begin.

→ *y*
The y coordinate where the baseline of the text should be. (Letters with descenders will draw below this line.)

→ *text*
The text to draw to the screen.

→ *length*
The size in bytes of the *text* string. You can use –1 for this value if *text* is a null-terminated string.

Returns Nothing.

Comments This function generates a path from the supplied text, fills that path, and then clears it. Because this function clears the current path, you must either call `GcPaint()` to draw any path that you have defined before you add text, or you can call `GcPushState()` to preserve the drawing and continue after you add the text.

See Also `GcSetFont()`

GcEndClip Function

Purpose Ends the specification of a clipping region.

Declared In `GcRender.h`

Prototype
```
void GcEndClip (GcHandle ctxt)
```

Parameters

→ *ctxt*
The graphics context.

Returns Nothing.

Comments After this call, all drawing is constrained to the region that you defined using this procedure. (If you passed `true` for `GcBeginClip()`'s *inverse* parameter, all drawing is constrained to the region that lies outside of the region you've defined.)

To clear the clipping region, call `GcPopState()`. Nested clipping regions are not allowed; that is, you cannot call `GcBeginClip()` twice without calling `GcEndClip()`. You can, however, define two separate clipping regions in the current rendering state. If you do, the second clipping region is constrained by the first so that the resulting region becomes the intersection of the two regions.

GcFlush Function

Purpose Blocks until all pending drawing operations have been sent.

Declared In `GcRender.h`

Prototype
```
void GcFlush (GcHandle ctxt)
```

Parameters

→ *ctxt*
The rendering context to flush.

Returns Nothing.

Comments Use this function in the rare circumstances where you need to wait for the drawing buffer to be empty before your application continues.

The rendering system runs in a different process than the application. To avoid IPC overhead, drawing operations are buffered and sent to the rendering system in batches. Because of this, rendering is likely to have not finished, or even started, when a drawing function returns. `GcFlush()` causes all buffered operations to be sent immediately and then it returns.

Drawing may not have completed when `GcFlush()` returns; it simply makes sure all drawing commands have been sent. If you need to make sure that drawing completes before you continue, use `GcCommit()` instead of `GcFlush()`.

This function is a performance drain and should be used rarely, only when absolutely necessary.

Note that you do not need to use `GcFlush()` when drawing to a local bitmap. All drawing to local bitmaps is performed immediately.

GcGetBitmapDensity Function

Purpose Returns the density of a bitmap.

Declared In `GcRender.h`

Prototype

```
uint16_t GcGetBitmapDensity
    (GcBitmapHandle bitmapHandle)
```

Parameters → *bitmapHandle*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns One of the `DensityType` constants, specifying the density of the bitmap.

Comments Because the bitmap is scaled when it is loaded into a `GcBitmapHandle`, the density returned by this function may not be the density set in the bitmap resource. For example, if you have a low-density bitmap and you are drawing to a high-density display, the bitmap is scaled to high density. `GcGetBitmapDensity()` returns `kDensityDouble`.

GcGetBitmapDepth Function

Purpose Retrieves the depth of a bitmap.

Declared In `GcRender.h`

Prototype
```
uint32_t GcGetBitmapDepth
    (GcBitmapHandle bitmapHandle)
```

Parameters

→ *bitmapHandle*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns The bit depth of a bitmap.

GcGetBitmapHeight Function

Purpose Returns the height of a bitmap.

Declared In `GcRender.h`

Prototype
```
uint32_t GcGetBitmapHeight
    (GcBitmapHandle bitmapHandle)
```

Parameters

→ *bitmapHandle*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns Returns the bitmap's height in the bitmap's native coordinates.

Comments You can use `GcGetBitmapDensity()` to determine the bitmap's native density and then convert the value returned by `GcGetBitmapHeight()` to whatever coordinate system you want with the `WinConvertCoord()` function.

Note that the bitmap is scaled when it is loaded into a `GcBitmapHandle`, so the density returned by `GcGetBitmapDensity()` may not be the density set in the bitmap resource. For example, if you have a low-density bitmap of 16 X 16 pixels and you are drawing to a high-density display, the bitmap is scaled to high density. `GcGetBitmapDensity()` returns `kDensityDouble`, and `GcGetBitmapHeight()` returns 32.

GcGetBitmapWidth Function

Purpose Returns the width of the bitmap.

Declared In `GcRender.h`

Prototype

```
uint32_t GcGetBitmapWidth
    (GcBitmapHandle bitmapHandle)
```

Parameters

→ *`bitmapHandle`*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns Returns the bitmap's width in the bitmap's native coordinates.

Comments You can use `GcGetBitmapDensity()` to determine the bitmap's native density and then convert the value returned by `GcGetBitmapWidth()` to whatever coordinate system you want with the `WinConvertCoord()` function.

Note that the bitmap is scaled when it is loaded into a `GcBitmapHandle`, so the density returned by `GcGetBitmapDensity()` may not be the density set in the bitmap resource. For example, if you have a low-density bitmap of 16 X 16 pixels and you are drawing to a high-density display, the bitmap is scaled to high density. `GcGetBitmapDensity()` returns `kDensityDouble`, and `GcGetBitmapWidth()` returns 32.

GcGetCurrentContext Function

Purpose Creates the graphics context for the current draw window.

Declared In `GcRender.h`

Prototype

```
GcHandle GcGetCurrentContext (void)
```

Parameters None.

Returns A handle to the current context (see `GcHandle`) or `NULL` if the graphics context could not be created.

Comments Use this function to obtain the `GcHandle` that you pass to all other drawing functions.

If you are drawing to an on-screen window, call this function in response to a `frmUpdateEvent`. If the draw window is an update-based window, this function returns `NULL` unless called in response

to an update event. If the draw window is a transitional window, you can obtain a graphics handle at any time after the first `winResizedEvent` is posted; however, it is still a good idea to only draw in response to an update event.

WARNING! Check the return value from this function before passing it to any other drawing functions. Drawing into a `NULL` graphics context causes the application to crash.

To draw to an off-screen window, create the window first and then call `WinSetDrawWindow()` before calling `GcGetCurrentContext()`.

To draw into a bitmap, use `GcCreateBitmapContext()` instead of this function.

You must call `GcReleaseContext()` when you are finished drawing.

See Also `WinGetActiveWindow()`

GcInitGradient Function

Purpose Creates a gradient for later use with the `GcSetGradient()` function.

Declared In `GcRender.h`

Prototype

```
status_t GcInitGradient (GcGradientType *gradient,
    const GcPointType *points,
    const GcColorType *colors, uint32_t num)
```

Parameters

← *gradient*
Contains the gradient upon return. See `GcGradientType`.

→ *points*
An array of points (see `GcPointType`). The number of points is specified in *num*.

→ *colors*
An array of colors (see `GcColorType`). The number of colors is specified in *num*.

→ *num*
The number of points and colors. Must be either 2 or 3.

Returns Returns `errNone` upon success or –1 if the *num* parameter is invalid.

Comments This function creates a gradient using the values specified. If you've specified two points and two colors, this function creates a gradient that begins at the first point with the first color and ends at the second point with the color gradually becoming the ending color.

If three values are specified, the gradient works in a triangular fashion with each color starting at its corresponding point and gradually blending into the next color as it approaches that point.

You generally specify a gradient that represents the entire screen and then use a clipping region to constrain the drawing of the gradient to a particular area.

See Also `GcSetGradient()`, `GcBeginClip()`

GcIsBitmapAlphaOnly Function

Purpose Returns whether the bitmap contains only an alpha channel.

Declared In `GcRender.h`

Prototype

```
Boolean GcIsBitmapAlphaOnly
    (GcBitmapHandle bitmapHandle)
```

Parameters

→ *bitmapHandle*
The bitmap. Use `FrmGetBitmapHandle()` or `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns `true` if the bitmap contains only an alpha channel, or `false` otherwise.

Comments Bitmaps containing only an alpha channel are drawn using the current color (set by `GcSetColor()`) and blended with the destination per the alpha channel.

GcLineTo Function

Purpose Adds a straight line to the path. The line goes from the current point to the specified point.

Declared In `GcRender.h`

Prototype

```
void GcLineTo (GcHandle ctxt, fcoord_t x,
    fcoord_t y)
```

Parameters

→ *ctxt*
The graphics context.

→ *x, y*
Ending point of the line to draw. This point becomes the current point.

Returns Nothing.

Comments This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

See Also `GcClosePath()`

GcLoadBitmap Function

Purpose Converts a bitmap into a format that is most efficient for the rendering system to display on screen.

Declared In `GcRender.h`

Prototype

```
GcBitmapHandle GcLoadBitmap (WinHandle win,
    const void *bitmapFileP, size_t size,
    uint32_t flags)
```

Parameters

→ *win*
A handle to the form's window.

→ *bitmapFileP*
A pointer to a `BitmapType` structure or a PNG as loaded from a resource file.

→ *size*
The size of the *bitmapFileP* parameter.

→ *flags*
One of the Bitmap Loading constants, which specifies how the bitmap is scaled when it is drawn.

Returns A `GcBitmapHandle` in the closest match to the screen density and bit depth.

Comments You generally don't use this function directly. Instead, use `FrmGetBitmapHandle()`.

For backward compatibility, the `BitmapType` stores a bitmap's data in big-endian format. This function converts a bitmap into the little-endian format that is used in Palm OS® Cobalt and performs any other conversions that might make it more efficient to draw this bitmap to the screen.

This function makes a copy of the data, so the original bitmap can be deallocated any time after this function is called.

If the bitmap has multiple depths (is a bitmap family), the closest match less than or equal to the current draw window depth is selected from that family and converted. If such a bitmap does not exist, the bitmap with the closest match greater than the draw window depth is used.

Call `GcReleaseBitmap()` when you are finished with the bitmap.

See Also `GcDrawBitmapAt()`, `GcDrawRawBitmapAt()`

GcMoveTo Function

Purpose Moves the current point to the coordinates specified.

Declared In `GcRender.h`

Prototype
```
void GcMoveTo (GcHandle ctxt, fcoord_t x,
    fcoord_t y)
```

Parameters → *ctxt*
The graphics context.

→ *x, y*
The point that should become the current point.

Returns Nothing.

Comments This function starts a new subpath by moving the current point to the location specified without adding anything to the current path other than this point.

You can think of this function as lifting your pencil and moving it to the specified coordinates. All future drawing starts from this point.

GcPaint Function

Purpose Paints the current path onto the screen.

Declared In `GcRender.h`

Prototype
```
void GcPaint (GcHandle ctxt)
```

Parameters

→ *ctxt*
The graphics context.

Returns Nothing.

Comments This function draws the path as specified by the current rendering state and constrained by the current clipping region. Then it clears the current path. It does not clear any other aspect of the rendering state.

`GcPaint()` fills the current path with the current color. If you've defined a non-closed path and you call `GcPaint()`, the result is undefined. To paint just the outline of the path, call `GcStroke()` before calling `GcPaint()`. When you call `GcStroke()`, `GcPaint()` paints the outline of the current path using the current pen size, cap styles, and line join styles.

`GcPaint()` is also used to specify the clipping region. If `GcPaint()` is called in between calls to `GcBeginClip()` and `GcEndClip()`, it fills the specified region with white and all subsequent calls to `GcPaint()` up to the next `GcPopState()` are constrained to the points within that region.

GcPaintBitmap Function

Purpose Fills the current path with the specified bitmap.

Declared In `GcRender.h`

Prototype

```
void GcPaintBitmap (GcHandle ctxt,
    GcBitmapHandle bitmapHandle)
```

Parameters

→ *ctxt*
The graphics context.

→ *bitmapHandle*
The bitmap to draw to the screen. Use `GcLoadBitmap()` to obtain a `GcBitmapHandle` to a bitmap.

Returns Nothing.

Comments Think of this function as painting with a bitmap instead of with the current color. It fills the current path with the bitmap, tiling the bitmap if necessary. This function clears the current path but no other aspect of the rendering state. Most of the rendering state does not affect this call.

See Also `GcCreateBitmapContext()`, `GcDrawBitmapAt()`, `GcDrawRawBitmapAt()`, `GcLoadBitmap()`

GcPopState Function

Purpose Restores the previous rendering state.

Declared In `GcRender.h`

Prototype

```
void GcPopState (GcHandle ctxt)
```

Parameters

→ *ctxt*
The graphics context.

Returns Nothing.

See Also `GcPushState()`

GcPushState Function

Purpose Saves the current rendering state.

Declared In `GcRender.h`

Prototype
```
void GcPushState (GcHandle ctxt)
```

Parameters

→ *ctxt*
The graphics context.

Returns Nothing.

Comments The rendering state is specified by calls to most of the other functions in `GcRender.h`.

The state includes the clipping region and the current path. `GcPushState()` is often used to preserve the current path before making some other change.

See Also `GcPopState()`

GcRect Function

Purpose Adds a rectangle to the path.

Declared In `GcRender.h`

Prototype
```
void GcRect (GcHandle ctxt, fcoord_t left,
    fcoord_t top, fcoord_t right, fcoord_t bottom)
```

Parameters

→ *ctxt*
The graphics context.

→ *left*
x coordinate of the left side of the rectangle.

→ *top*
y coordinate of the top of the rectangle.

→ *right*
x coordinate of the right side of the rectangle.

→ *bottom*
y coordinate of the bottom of the rectangle.

Returns Nothing.

Comments The current point is moved to the top-left corner before the rectangle is added to the path. The rectangle is drawn clockwise

from the top left corner, resulting in a current point that is the top left corner of the rectangle.

`GcRect()` is optimized for speed; always use it when you want to draw a square or rectangular region.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

See Also `GcRoundRect()`, `GcRectI()`

GcRectI Function

Purpose Adds a rectangle to the path using integer coordinates to define the rectangle.

Declared In `GcRender.h`

Prototype

```
void GcRectI (GcHandle ctxt, int32_t left,
    int32_t top, int32_t right, int32_t bottom)
```

Parameters

→ *ctxt*
The graphics context.

→ *left*
y coordinate of the top of the rectangle.

→ *top*
y coordinate of the top of the rectangle.

→ *right*
x coordinate of the right side of the rectangle.

→ *bottom*
y coordinate of the bottom of the rectangle.

Returns Nothing.

Comments This function differs from `GcRect()` in that it takes integer coordinates rather than floating-point coordinates.

GcReflect Function

Purpose Flips the coordinate system at the specified radius.

Declared In `GcRender.h`

Prototype
```
void GcReflect (GcHandle ctxt, fcoord_t rad)
```

Parameters

→ *ctxt*
The graphics context.

→ *rad*
The angle of the axis of symmetry.

Returns Nothing.

Comments A two-dimensional transformation affects only the drawing functions called after it. If you call this function in the middle of a path definition, it applies only to those points in the path defined after this function is called. If you are creating a stroked path, call this and all other two-dimensional transformations before calling `GcStroke()`.

The axis of symmetry runs through the origin of the screen, which may cause some of the path to be drawn off screen. If you want to reflect the path along an axis that runs through the center of the screen, use `GcTranslate()` to translate the origin of the screen to the center before calling this function. See `GcRotate()` for sample code.

See Also `GcScale()`, `GcTransform()`

GcReleaseBitmap Function

Purpose Releases the memory associated with the bitmap handle.

Declared In `GcRender.h`

Prototype
```
status_t GcReleaseBitmap
    (GcBitmapHandle bitmapHandle)
```

Parameters

→ *bitmapHandle*
The bitmap to release.

Returns Always returns `errNone`.

Comments Memory is not actually deallocated until all pending drawing operations are complete.

This function does *not* release a bitmap context. If you've used `GcCreateBitmapContext()` to create a bitmap that receives drawing operations, use `GcReleaseContext()` to release it.

See Also `GcLoadBitmap()`

GcReleaseContext Function

Purpose Releases a rendering context.

Declared In `GcRender.h`

Prototype
```
void GcReleaseContext (GcHandle ctxt)
```

Parameters

→ *ctxt*
The rendering context to release.

Returns Nothing.

Comments Call this function when you are finished drawing. It pops the current rendering state and releases the memory associated with the graphics context.

If you do not release the graphics context, the user interface will eventually freeze.

See Also `GcCreateBitmapContext()`, `GcGetCurrentContext()`

GcRotate Function

Purpose Rotates the coordinate system the number of radians.

Declared In `GcRender.h`

Prototype
```
void GcRotate (GcHandle ctxt, fcoord_t rad)
```

Parameters

→ *ctxt*
The graphics context.

→ *rad*
The number of radians by which to rotate the path.

Returns Nothing.

Comments A two-dimensional transformation affects only the drawing functions called after it. If you call this function in the middle of a path definition, it applies only to those points in the path defined

after this function is called. If you are creating a stroked path, call this and all other two-dimensional transformations before calling GcStroke().

This function rotates the path around the origin (top-left corner) of the screen. If you want to rotate around some other point, use GcTranslate() to move the origin before you call this function.

Example The following code draws a square in the center of the screen and rotates it 45 degrees around the center of the screen.

```
#define DegreesToRadians(x) (fcoord_t)(x * 3.14159/180)

void RotateExample(FormType *pForm)
{
   GcHandle gc = GcGetCurrentContext();
   fcoord_t centerX, centerY, radius;
   fcoord_t extentX = kExtent, extentY = kExtent;
   fcoord_t startX, startY;
   RectangleType winBounds;

   if (!gc) return;

   // Compute bounds of window to decide how to center the
   // square.
   WinGetWindowBounds(&winBounds);
   if (winBounds.extent.x) {
      centerX = ((fcoord_t)(winBounds.extent.x -
         winBounds.topLeft.x)) / 2;
      centerY = ((fcoord_t)(winBounds.extent.y -
         winBounds.topLeft.y)) / 2;
      startX = centerX - extentX/2;
      startY = centerY - extentY/2;
   } else {
      return;
   }

   // Translate the origin of the screen to the center point
   // so that the rest of the transformation is applied from
   // the center rather than the screen's origin.
   GcTranslate(gc, centerX, centerY);
   // Now you need to adjust the starting coordinates
   // accordingly.
   startX -= centerX;
   startY -= centerY;
   // Set up transformation.
   GcRotate(gc, DegreesToRadians(45));
```

```
   //Draw the shape and paint it.
   GcRect(gc, startX, startY, startX + extentX, startY +
      extentY);
   GcPaint(gc);

   GcReleaseContext(gc);
   return;
}
```

See Also GcReflect(), GcScale(), GcTransform()

GcRoundRect Function

Purpose Creates a rectangle path with rounded corners.

Declared In GcRender.h

Prototype

```
void GcRoundRect (GcHandle ctxt, fcoord_t left,
   fcoord_t top, fcoord_t right, fcoord_t bottom,
   fcoord_t radx, fcoord_t rady)
```

Parameters

→ *ctxt*
The graphics context.

→ *left*
x coordinate of the left side of the rectangle.

→ *top*
y coordinate of the top of the rectangle.

→ *right*
x coordinate of the right side of the rectangle.

→ *bottom*
y coordinate of the bottom of the rectangle.

→ *radx*
The horizontal radius of each corner.

→ *rady*
The vertical radius of each corner.

Returns Nothing.

Comments The current point is moved to the top-left corner before the rectangle is added to the path. The rectangle is drawn clockwise

from the top left corner, resulting in a current point that is the top left corner of the rectangle.

The corners are constructed using the same algorithm that `GcArcTo()` uses.

This function adds to the current path in the rendering state; it does not draw to the screen. To draw the current path to the screen, call `GcPaint()`.

See Also `GcRect()`, `GcRectI()`

GcScale Function

Purpose Scales the coordinate system.

Declared In `GcRender.h`

Prototype

```
void GcScale (GcHandle ctxt, fcoord_t sx,
    fcoord_t sy)
```

Parameters

→ *ctxt*
The graphics context.

→ *sx*
Horizontal scaling factor. This must be a positive value.

→ *sy*
Vertical scaling factor. This must be a positive value.

Returns Nothing.

Comments A two-dimensional transformation affects only the drawing functions called after it. If you call this function in the middle of a path definition, it applies only to those points in the path defined after this function is called. If you are creating a stroked path, call this and all other two-dimensional transformations before calling `GcStroke()`.

Scaling occurs starting from the origin of the screen. If you want to scale a path with respect to its center, use `GcTranslate()` to translate the origin to the path's center before scaling. See `GcRotate()` for sample code.

See Also `GcReflect()`, `GcShear()`, `GcTransform()`

GcSetAntialiasing Function

Purpose Specifies whether the rendering system uses antialiasing when drawing.

Declared In `GcRender.h`

Prototype
```
void GcSetAntialiasing (GcHandle ctxt,
    uint32_t value)
```

Parameters

→ *ctxt*
The graphics context.

→ *value*
One of the GcAliasingTag constants.

Returns Nothing.

Comments You rarely need to use this function. Do so only if you are creating a drawing application and want more control over the quality of the rendered image. The default value for antialiasing suffices in most cases.

If the system does not support antialiasing, this setting is ignored.

See Also GcSetFontAntialiasing()

GcSetCaps Function

Purpose Specifies what the line end points look like in a stroked path.

Declared In `GcRender.h`

Prototype
```
void GcSetCaps (GcHandle ctxt, int32_t start_cap,
    int32_t end_cap)
```

Parameters

→ *ctxt*
The graphics context.

→ *start_cap*
A GcCapTag value that specifies how the start of each subpath will look.

→ *end_cap*
A GcCapTag value that specifies how the end of each subpath will look.

Returns Nothing.

Comments This function only affects paths where GcStroke() is called before GcPaint().

GcSetColor Function

Purpose Sets the color used with GcPaint().

Declared In GcRender.h

Prototype
```
void GcSetColor (GcHandle ctxt, uint8_t r,
   uint8_t g, uint8_t b, uint8_t a)
```

Parameters

→ *ctxt*
The graphics context.

→ *r*
The red value.

→ *g*
The green value.

→ *b*
The blue value.

→ *a*
The level of opacity where 255 is fully opaque.

Returns Nothing.

Comments Palm OS supports up to 16-bit direct color displays, which typically uses 5 bits for red, 6 bits for green, and 5 bits for blue. If you specify larger red, green, or blue values, the rendering system converts them before they are drawn to the screen.

See Also GcSetGradient()

GcSetCoordinateSystem Function

Purpose Sets the coordinate system used by subsequent drawing operations.

Declared In GcRender.h

Prototype
```
void GcSetCoordinateSystem (GcHandle ctxt,
   uint32_t coordinateSystem)
```

Parameters

→ *ctxt*
The graphics context.

→ *coordinateSystem*
One of the Coordinate System Constants. The default coordinate system for all graphics context drawing is the native screen format.

Returns Nothing.

Comments It is best to set the coordinate system before any drawing is done. Suppose the current coordinate system is native, and you do something like that shown below on a double-density device:

```
GcHandle ctxt = GcGetCurrentContext();

//Draw a line
GcMoveTo(ctxt, 0, 10);
GcDrawLine(ctxt, 0, 160);

//Draw another line
GcMoveTo(ctxt, 0, 20);
GcDrawLine(ctxt, 0, 160);
GcMoveTo(ctxt, 0, 40);

//BAD Code! Applies only to the next line.
GcSetCoordinateSystem(ctxt, kCoordinatesStandard);
GcLineTo(ctxt, 0, 160);

GcStroke(ctxt);
GcPaint(ctxt);
GcReleaseContext(ctxt);
```

The last line will be twice as long as the other two because the coordinate system switched to standard before its bounds were computed.

The default coordinate system for the graphics context is the coordinate system used by the current draw window.

Be careful not to confuse `WinSetCoordinateSystem()` and this function. If you call `GcSetCoordinateSystem()`, it changes only the coordinate system used by the graphics context, not by the Window Manager. It is a good idea to always make both calls so that you can be certain both coordinate systems are the same.

GcSetFont Function

Purpose Sets the font used by GcDrawTextAt().

Declared In GcRender.h

Prototype
```
void GcSetFont (GcHandle ctxt, GcFontHandle font)
```

Parameters

→ *ctxt*
The graphics context.

→ *font*
The scalable font to use.

Returns Nothing.

See Also GcCreateFont(), GcCreateFontFromFamily(), GcCreateFontFromID()

GcSetGradient Function

Purpose Sets the gradient used with GcPaint().

Declared In GcRender.h

Prototype
```
void GcSetGradient (GcHandle ctxt,
   const GcGradientType *gradient)
```

Parameters

→ *ctxt*
The graphics context.

→ *gradient*
A GcGradientType structure defining the gradations to use.

Returns Nothing.

Comments Any two-dimensional transformations that are set before this call are applied to the gradient when drawing takes place. If you don't want the transformation applied to the gradient, call GcSetGradient() first, and then set the transformations.

See Also GcSetColor(), GcInitGradient()

GcSetJoin Function

Purpose Specifies what the intersection of two lines looks like in a stroked (outlined) path.

Declared In `GcRender.h`

Prototype
```
void GcSetJoin (GcHandle ctxt, int32_t join,
    float miter_limit)
```

Parameters

→ *ctxt*
The graphics context.

→ *join*
One of the GcJoinTag values.

→ *miter_limit*
Controls when the rendering system switches from a miter join to a bevel join. When the length of the point reaches *miter_limit* times the stroke weight, the rendering system switches from a miter join to a bevel join. This only applies to the miter join, not to any of the other join styles.

Returns Nothing.

Comments This function only has affect for paths where GcStroke() is called before GcPaint().

GcSetPenSize Function

Purpose Sets the border width used to draw stroked paths.

Declared In `GcRender.h`

Prototype
```
void GcSetPenSize (GcHandle ctxt,
    fcoord_t penWidth)
```

Parameters

→ *ctxt*
The graphics context.

→ *penWidth*
The pen width given in coordinates.

Returns Nothing.

Comments This function only has affect for paths where GcStroke() is called before GcPaint().

When the path is drawn, rendering system ensures that the path is in the middle of the pen. A path is always assumed to have no width. For example, if you give a path that consists of one vertical line drawn from (1, 0) to (1, 160) and a pen width of 2, `GcPaint()` colors all pixels in the 0 column and all pixels in the 1 column.

GcShear Function

Purpose Applies the shear transformation to the coordinate system.

Declared In `GcRender.h`

Prototype

```
void GcShear (GcHandle ctxt, fcoord_t x,
   fcoord_t y)
```

Parameters

→ *ctxt*
The graphics context.

→ *x*
The amount of shear in the horizontal direction.

→ *y*
The amount of shear in the vertical direction.

Returns Nothing.

Comments The shear transformation angles and stretches a shape. It is most commonly used to convert a font to italics or to convert a rectangle into a parallelogram.

A two-dimensional transformation affects only the drawing functions called after it. If you call this function in the middle of a path definition, it applies only to those points in the path defined after this function is called. If you are creating a stroked path, call this and all other two-dimensional transformations before calling `GcStroke()`.

The shear transformation is applied starting from the origin of the screen. If you want to angle and stretch a shape but have it appear in the same general location on the screen, use `GcTranslate()` to translate the origin to the center of the screen before calling this function. See `GcRotate()` for sample code.

See Also `GcReflect()`, `GcScale()`, `GcTranslate()`, `GcTransform()`

GcStroke Function

Purpose Changes the current path to the outline of an object rather than a filled object.

Declared In `GcRender.h`

Prototype
```
void GcStroke (GcHandle ctxt)
```

Parameters

→ *ctxt*
The graphics context.

Returns Nothing.

Comments Use this function if you want `GcPaint()` to trace the outline of the path rather than filling the path.

The pen size, join, and cap style portions of the rendering state apply if you call this function. Pen size, join, and cap style do not affect fill operations.

The current two-dimensional transformation also affects the stroke. Do not change the two-dimensional transformation in between calls to this function and `GcPaint()`.

See Also `GcReflect()`, `GcRotate()`, `GcScale()`, `GcSetCaps()`, `GcSetJoin()`, `GcSetPenSize()`, `GcTranslate()`

GcTransform Function

Purpose Applies a general two-dimensional transformation matrix.

Declared In `GcRender.h`

Prototype
```
void GcTransform (GcHandle ctxt,
    const fcoord_t *matrix)
```

Parameters

→ *ctxt*
The graphics context.

→ *matrix*
The transformation matrix to be applied.

Returns Nothing.

Comments This function applies a general two-dimensional transformation like that in the Postscript language. The transformation is described by the matrix shown in Figure 21.7.

Figure 21.7 Transformation matrix

$$\begin{bmatrix} matrix[0] & matrix[2] & matrix[4] \\ matrix[1] & matrix[3] & matrix[5] \\ 0 & 0 & 1 \end{bmatrix}$$

The coordinates in this matrix are applied to each point in the current path according to the following equations:

newX `= (matrix[0] *` *currentX*`) + (matrix[2] *` *currentY*`) + matrix[4]`

newY `= (matrix[1] *` *currentX*`) + (matrix[3] *` *currentY*`) + matrix[5]`

where (*currentX*, *currentY*) represents a point on the path and (*newX*, *newY*) represents that point after the transformation has been applied.

See Also GcTranslate(), GcScale(), GcRotate(), GcReflect(), GcSetFontTransform()

GcTranslate Function

Purpose Translates the coordinate system by moving the origin to the specified coordinates.

Declared In `GcRender.h`

Prototype

```
void GcTranslate (GcHandle ctxt, fcoord_t x,
    fcoord_t y)
```

Parameters

→ *ctxt*
The graphics context.

→ *x*
The horizontal amount to move.

→ *y*
The vertical amount to move.

Returns Nothing.

Comments If you are creating a stroked path, call this and all other two-dimensional transformations before calling GcStroke(). Typically, you specify the transformation before you do any drawing.

This function is often used in conjunction with other transformations to ensure that the transformed path still appears on the screen. See `GcRotate()` for sample code.

See Also `GcReflect()`, `GcShear()`, `GcTransform()`

22

List Reference

This chapter provides information about lists by discussing these topics:

The header file `List.h` declares the API that this chapter describes. For more information on lists, see the section "Lists" on page 112.

List Structures and Types

ListType Struct

Purpose Internal structure that defines the list. A `ListPtr` is a pointer to a `ListType` structure.

Declared In `List.h`

Prototype

```
typedef struct ListType ListType;
typedef ListType *ListPtr
```

Fields None.

List Constants

List Constants

Purpose Constants defined in `List.h`.

Declared In `List.h`

Constants

```
#define noListSelection ((int16_t)(-1))
```

Used to denote that no item is selected.

List Events

lstEnterEvent

Purpose Sent when `penDownEvent` occurs within the bounds of a list.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct lstEnter {
   struct ListType *pList;
   uint16_t listID;
   int16_t selection;
} lstEnter
```

Fields

`pList`
Pointer to a list structure.

`listID`
Resource ID of the list.

`selection`
Unused.

Comments In most cases, you should wait for a `lstSelectEvent` or `popSelectEvent` before taking the intended action of the list. When the application receives the `lstEnterEvent`, it is not guaranteed that a list item will become selected.

lstExitEvent

Purpose Sent when the stylus had previously entered the bounds of the list but is lifted outside of the bounds of the list.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct lstExit {
   struct ListType *pList;
   uint16_t listID;
   _BA32_PADDING_16(1)
} lstExit
```

Fields

`pList`
Pointer to a list structure.

`listID`
Resource ID of the list.

lstSelectEvent

Purpose Sent when the stylus had previously entered the bounds of the list and is now lifted within the bounds of the list.

TIP: Pop-up lists don't generate a `lstSelectEvent`. Instead, they generate a `popSelectEvent`.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct lstSelect {
   struct ListType *pList;
   uint16_t listID;
   int16_t selection;
} lstSelect
```

Fields

`pList`
Pointer to a list structure.

`listID`
Resource ID of the list.

`selection`
Item number (zero-based) of the new selection.

Comments Applications should respond to this event if they need to respond to the list selection in some way. Most applications simply use `LstGetSelection()` or `LstGetSelectionText()` to obtain the list selection when it is needed.

popSelectEvent

Purpose Sent when the user selects an item in a pop-up list.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct popSelect {
   uint16_t controlID;
   uint16_t listID;
   struct ControlType *controlP;
   struct ListType *listP;
   int16_t selection;
   int16_t priorSelection;
} popSelect
```

Fields

`controlID`
Resource ID of the pop-up trigger.

`listID`
Resource ID of the list.

`controlP`
Pointer to the pop-up trigger `ControlType` structure.

`listP`
Pointer to the list's structure.

`selection`
Item number (zero-based) of the new list selection.

`priorSelection`
Item number (zero-based) of the prior list selection.

Comments Applications should respond to this event to perform whatever action is necessary for this list. Return `false` from the event handler to have the form set the pop-up trigger's label to the selected list item's text.

Palm OS® can only set the pop-up trigger label if the list items are statically defined. If you are custom drawing the list items, you must respond to this event and set the selected text in the pop-up trigger yourself.

List Functions and Macros

LstDrawList Function

Purpose Makes the list visible and draws the list.

Declared In `List.h`

Prototype
```
void LstDrawList (ListType *listP)
```

Parameters → *listP*
Pointer to a list.

Returns Nothing.

Comments This function does not ensure the current selection is visible; if you need to do this, call the `LstMakeItemVisible()` function.

If the list is disabled, it's drawn grayed-out (strongly discouraged). If it's empty, nothing is drawn. If it's not usable, nothing is drawn.

See Also FrmGetObjectPtr(), LstPopupList(), LstEraseList()

LstEraseList Function

Purpose Erases a list.

Declared In `List.h`

Prototype
```
void LstEraseList (ListType *listP)
```

Parameters → *listP*
Pointer to a list.

Returns Nothing.

See Also FrmGetObjectPtr(), LstDrawList()

LstGetFont Function

Purpose Gets the font used to draw a list's text strings.

Declared In List.h

Prototype
```
FontID LstGetFont (const ListType *listP)
```

Parameters
→ *listP*
Pointer to the list.

Returns The ID of the font used to draw all list text strings.

See Also LstSetFont()

LstGetItemsText Function

Purpose Gets the text strings that represent the items in a list.

Declared In List.h

Prototype
```
char **LstGetItemsText (const ListType *listP)
```

Parameters
→ *listP*
Pointer to the list.

Returns A pointer to an array of pointers to the text of the list choices.

See Also LstGetSelectionText(), LstSetListChoices()

LstGetNumberOfItems Function

Purpose Returns the number of items in a list.

Declared In List.h

Prototype
```
int16_t LstGetNumberOfItems(const ListType *listP)
```

Parameters
→ *listP*
Pointer to a list.

Returns The number of items in a list.

See Also FrmGetObjectPtr(), LstSetListChoices()

LstGetSelection Function

Purpose Returns the currently selected choice in the list.

Declared In List.h

Prototype
```
int16_t LstGetSelection (const ListType *listP)
```

Parameters

→ *listP*
Pointer to a list.

Returns The item number of the current list choice. The list choices are numbered sequentially, starting with 0; Returns noListSelection if none of the items are selected.

See Also FrmGetObjectPtr(), LstSetListChoices(), LstSetSelection(), LstGetSelectionText()

LstGetSelectionText Function

Purpose Returns a pointer to the text of the specified item in the list, or NULL if no such item exists.

Declared In List.h

Prototype
```
char *LstGetSelectionText (const ListType *listP,
    const int16_t itemNum)
```

Parameters

→ *listP*
Pointer to a list.

→ *itemNum*
Item to select (0 = first item in list).

Returns A pointer to the text of the current selection, or NULL if no such item exists.

Comments The returned string is not a copy of the currently selected list item; it is a pointer to a field within the ListType structure. This function is only usable if you supplied an array of strings and a count to LstSetListChoices() or if you statically defined the list in the

resource database; if your application uses a callback function that dynamically generates the list text, this function returns `NULL`.

See Also `FrmGetObjectPtr()`

LstGetTopItem Function

Purpose Returns the topmost visible item.

Declared In `List.h`

Prototype
```
int16_t LstGetTopItem (const ListType *listP)
```

Parameters → *listP*
Pointer to a list.

Returns The item number of the top item that is visible.

See Also `LstGetSelection()`, `LstSetTopItem()`

LstGetVisibleItems Function

Purpose Returns the number of visible items.

Declared In `List.h`

Prototype
```
int16_t LstGetVisibleItems (const ListType *listP)
```

Parameters → *listP*
Pointer to a list.

Returns The number of items visible.

LstHandleEvent Function

Purpose Handles event in the specified list.

Declared In `List.h`

Prototype
```
Boolean LstHandleEvent (ListType *listP,
    const EventType *eventP)
```

Parameters → *listP*
Pointer to a list.

→ *eventP*
Pointer to an EventType structure.

Returns `true` if the event was handled.

Comments The following cases will result in a return value of `true`:

- A penDownEvent within the bounds of the list. The list begins tracking the pen, but it does not block. Adds the `lstEnterEvent` to the event queue.
- A penMoveEvent. This function tracks the pen, updates the current selection. If the pen moves outside of the bounds of the list but is not lifted, this function scrolls the list in the direction indicated. That is, if the pen is dragged to the bottom of the list, the list scrolls down. If the pen is dragged to the top of the list, the list scrolls up.
- A penUpEvent. If the pen is up within the bounds of the list, it enqueues a lstSelectEvent. Otherwise, it enqueues a lstExitEvent.
- A lstEnterEvent with a list ID value that matches the list ID in the list data structure. If the list is a pop-up list, this function displays the pop-up lists window.

LstMakeItemVisible Function

Purpose Makes an item visible, preferably at the top. If the item is already visible, make no changes.

Declared In `List.h`

Prototype
```
void LstMakeItemVisible (ListType *listP,
    const int16_t itemNum)
```

Parameters → *listP*
Pointer to a list.

→ *itemNum*
Item to select. Items are numbered from 0 to LstGetNumberOfItems()–1.

Returns Nothing.

Comments Does *not* visually update the list.

See Also FrmGetObjectPtr(), LstSetSelection(), LstSetTopItem(), LstDrawList()

LstNewList Function

Purpose Creates a new list dynamically and installs it in the specified form. This function can be used to create a new pop-up trigger and its associated list.

Declared In `List.h`

Prototype

```
status_t LstNewList (void **formPP, uint16_t id,
   Coord x, Coord y, Coord width, Coord height,
   FontID font, int16_t visibleItems,
   int16_t triggerId)
```

Parameters

↔ *formPP*
Pointer to the pointer to the form in which the new list is installed. This value is not a handle; that is, the old *formPP* value is not necessarily valid after this function returns. In subsequent calls, always use the new *formPP* value returned by this function.

→ *id*
Symbolic ID of the list, specified by the developer. By convention, this ID should match the resource ID (not mandatory).

→ *x*
Horizontal coordinate of the upper-left corner of the list's boundaries, relative to the window in which it appears.

→ *y*
Vertical coordinate of the upper-left corner of the list's boundaries, relative to the window in which it appears.

→ *width*
Width of the list, expressed in standard coordinates. Valid values are 1 – 160.

→ *height*
Height of the list, expressed in standard coordinates.Valid values are 1 – 160.

→ *visibleItems*
Number of list items that can be viewed together.

→ *triggerId*
Symbolic ID of the pop-up trigger associated with the new list (this ID is specified by the developer). A nonzero value for *triggerId* causes this function to create both the list and its associated pop-up trigger. If the list isn't a pop-up list, pass 0 for *triggerId*.

Returns `errNone` if no error.

See Also `LstDrawList()`, `FrmRemoveObject()`

LstPopupList Function

Purpose Displays a modal window that contains the items in the list.

Declared In `List.h`

Prototype
```
int16_t LstPopupList (ListType *listP)
```

Parameters → *listP*
Pointer to a list.

Returns The list item selected, or `noListSelection` if no item was selected.

Comments Saves the previously active window. Creates and deletes the new pop-up window.

Typical applications do not call this function. Palm OS pops up the list for you when the user holds the pen down on the pop-up trigger that is associated with the list.

See Also FrmGetObjectPtr()

LstScrollList Function

Purpose Scrolls the list up or down a number of times.

Declared In `List.h`

Prototype
```
Boolean LstScrollList (ListType *listP,
   WinDirectionType direction, int16_t itemCount)
```

Parameters → *listP*
Pointer to a list.

→ *direction*
Direction to scroll.

→ *itemCount*
Items to scroll in direction.

Returns `true` when the list is actually scrolled, `false` otherwise. May return `false` if a scroll past the end of the list is requested.

Comments List scrolling is fully handled within Palm OS. Applications do not have to handle any aspect of list scrolling. If you suppress the list scroll arrows, however, and attach a separate scroll bar to the list, you might need to call this function.

LstSetDrawFunction Function

Purpose Sets a callback function to draw each item instead of drawing the item's text string.

Declared In `List.h`

Prototype

```
void LstSetDrawFunction (ListType *listP,
   ListDrawDataFuncPtr func)
```

Parameters → *listP*
Pointer to a list.

→ *func*
Pointer to a function that draws items. See `ListDrawDataFuncType()`.

Returns Nothing.

See Also `FrmGetObjectPtr()`, `LstSetListChoices()`, `ListDrawDataFuncType()`

LstSetFont Function

Purpose Specifies the font to be used to draw a list's text strings.

Declared In `List.h`

Prototype

```
void LstSetFont (ListType *listP, FontID fontID)
```

Parameters → *listP*
Pointer to a list.

→ *fontID*
ID of the font to be used to draw all list text strings.

Returns Nothing.

See Also LstGetFont()

LstSetHeight Function

Purpose Sets the number of items visible in a list.

Declared In List.h

Prototype
```
void LstSetHeight (ListType *listP,
    int16_t visibleItems)
```

Parameters

→ *listP*
Pointer to a list.

→ *visibleItems*
Number of choices visible at once.

Returns Nothing.

Comments This function doesn't redraw the list if it's already visible.

See Also FrmGetObjectPtr()

LstSetIncrementalSearch Function

Purpose Enables or disables incremental search for a sorted pop-up list.

Declared In List.h

Prototype
```
void LstSetIncrementalSearch (ListType *listP,
    Boolean incrementalSearch)
```

Parameters

→ *listP*
Pointer to a list.

→ *incrementalSearch*
Set to `true` to enable incremental search, `false` to disable it.

Returns Nothing.

Comments If incremental search is enabled, when the list is displayed the user can navigate the list by entering up to five characters. The list will

scroll to present the first list item that matches the entered characters. This feature only works if the following are true:

- The list is a pop-up list.
- The items in the list are sorted alphabetically.
- The list items are visible to the List Manager (that is, you don't pass `NULL` to `LstSetListChoices()`).

LstSetListChoices Function

Purpose Sets the items of a list to the array of text string pointers passed to this function. This functions overwrites the previous set of list items.

Declared In `List.h`

Prototype

```
void LstSetListChoices (ListType *listP,
    char **itemsText, int16_t numItems)
```

Parameters

→ *`listP`*
Pointer to a list.

→ *`itemsText`*
Pointer to an array of text strings. See `SysFormPointerArrayToStrings()` for one way to create this array of strings.

→ *`numItems`*
Number of choices in the list.

Returns Nothing.

Comments You need to call the `LstDrawList()` function to update the list if it is displayed when you call this function. `LstSetListChoices()` ensures that the selected item is within the range of the items you supply and that the top item is within the range of possible top items.

NOTE: This function does not copy the strings in the *`itemsText`* array, which means that you need to ensure that the array is not moved or deallocated until after you are done with the list.

If you use a callback routine to draw the items in your list, the *`itemsText`* pointer is simply passed to that callback routine and is

not otherwise used by the List Manager code. See the comments under `ListDrawDataFuncType()` for tips on using the *`itemsText`* parameter with a callback routine.

See Also FrmGetObjectPtr(), LstGetItemsText(), LstSetSelection(), LstSetTopItem(), LstDrawList(), LstSetHeight(), LstSetDrawFunction()

LstSetPosition Function

Purpose Sets the position of a list.

Declared In `List.h`

Prototype

```
void LstSetPosition (ListType *listP,
    const Coord x, const Coord y)
```

Parameters

→ *`listP`*
Pointer to a list.

→ *`x`*
Left bound.

→ *`y`*
Top bound.

Returns Nothing.

Comments The list is not redrawn. Don't call this function when the list is visible.

LstSetSelection Function

Purpose Sets the selection for a list.

Declared In `List.h`

Prototype

```
void LstSetSelection (ListType *listP,
    int16_t itemNum)
```

Parameters

→ *`listP`*
Pointer to a list.

→ *itemNum*
Item to select. Items are numbered from 0 to `LstGetNumberOfItems()`–1. Specify `noListSelection` if you want to deselect all list items.

Returns Nothing.

Comments The old selection, if any, is deselected. If the list is visible, the selected item is visually updated. The list is scrolled to the selection, if necessary, as long as the list is both visible and usable.

See Also `FrmGetObjectPtr()`, `LstSetTopItem()`

LstSetScrollArrows Function

Purpose Enables or disables the drawing of the list's scroll arrows.

Declared In `List.h`

Prototype
```
void LstSetScrollArrows (ListType *listP,
    Boolean visible)
```

Parameters → *listP*
Pointer to a list.

→ *visible*
`true` to have the scroll arrows visible if the list needs scrolling. `false` to suppress the drawing of scroll arrows. The default is `true`.

Returns Nothing.

Comments You might use this function to suppress the drawing the list's scroll indicators if you want the list to look and behave like a table does or if you have a very large list and want to use a scroll bar rather than the built-in scroll arrows. If you suppress the list's scroll indicators, you are responsible for handling scrolling the list.

LstSetTopItem Function

Purpose Sets the item to be the top item visible in the list. The item cannot become the top item if it's on the last page.

Declared In `List.h`

Prototype

```
void LstSetTopItem (ListType *listP,
    const int16_t itemNum)
```

Parameters

→ *listP*
Pointer to a list.

→ *itemNum*
Item to select. Items are numbered from 0 to `LstGetNumberOfItems()`–1.

Returns Nothing.

Comments Does *not* update the display.

See Also `FrmGetObjectPtr()`, `LstSetSelection()`, `LstGetTopItem()`, `LstMakeItemVisible()`, `LstDrawList()`, `LstEraseList()`

Application-Defined Functions

If you need to perform special drawing for items in the list, call `LstSetDrawFunction()` to set the list drawing callback function. The `ListDrawDataFuncType()` section documents the prototype for the callback function you provide for drawing list items.

ListDrawDataFuncType Function

Purpose Callback function that you provide for drawing items in a list. This function is called whenever the Palm OS needs to draw an element in the list. Your callback function declaration must match the prototype shown here.

Declared In `List.h`

Prototype

```
void ListDrawDataFuncType (int16_t itemNum,
    RectangleType *bounds, char **itemsText,
    struct ListType *listP)
```

Parameters

→ *itemNum*
The number of the item to draw.

→ *bounds*
The bounds of the item relative to either the list's window (for a pop-up list) or the form's window (for a regular list). See RectangleType.

→ *itemsText*
A pointer to an array of pointers to the text of the list items. This is the pointer that you supplied when calling LstSetListChoices().

→ *listP*
A pointer to the list in which the item is to be drawn.

Returns Nothing.

Comments You can call LstSetDrawFunction() to register your callback function for the list, which means that your function will be called to draw the list items, rather than using the built-in draw functionality, which displays each item's text string.

Your callback function is called whenever an item in the list needs to be drawn. When it is called, the value of the *itemNum* parameter specifies which item the function is to draw. The *itemsText* parameter, which is what was supplied to `LstSetListChoices()`, doesn't actually need to point to a list of strings: you can pass `NULL`, or you can pass a pointer to anything that will be useful to your drawing function. Note, however, that if you pass anything other than a pointer to a list of strings when you call `LstSetListChoices()`, you must ensure that LstGetSelectionText() is never called since it assumes that this pointer indicates an array of text items. In particular, if your list is associated with a pop-up trigger you must handle the popSelectEvent and return true. If you don't, FrmHandleEvent() will call `LstGetSelectionText()` in response to that event.

Note that the list manages which colors are used to draw the items and how to draw selected versus unselected items. In almost all circumstances, your drawing function does not have to be concerned with these details.

See Also LstSetDrawFunction(), UIColorGetTableEntryIndex(), WinSetForeColor()

23

Menu Reference

This chapter describes the menu API as declared in the header file `Menu.h`. It discusses the following topics:

For more information on menus, see the section Chapter 4, "Working with Menus," on page 73.

Menu Structures and Types

MenuBarType Struct

Purpose Internal structure that defines a menu bar. The `MenuBarPtr` function defines a pointer to a menu bar.

Declared In `Menu.h`

Prototype

```
typedef struct MenuBarType MenuBarType;
typedef MenuBarType *MenuBarPtr
```

Fields None.

Menu Constants

Command Button Location Constants

Purpose Specifies where to place a new button within the menu command tool bar.

Declared In `Menu.h`

Constants

`#define menuCmdBarOnLeft 0xff`
Add the button to the left of the other buttons on the command tool bar.

`#define menuCmdBarOnRight 0`
Add the button to the right of the other buttons.

See Also `MenuCmdBarAddButton()`

Menu Activation Constants

Purpose Indicates how a menu has been activated. These constants are sent as part of the `menuOpenEvent`.

Declared In `Menu.h`

Constants

`#define menuButtonCause 0`
The menu is going to be displayed because the user tapped the Menu icon or tapped the form's title bar.

`#define menuCommandCause 1`
The user drew the menu command shortcut character in the input area, so the menu is becoming active without being displayed.

Menu Error Constants

Purpose Errors returned by menu functions.

Declared In `Menu.h`

Constants

`#define menuErrNoMenu (menuErrorClass | 1)`
There is currently no active menu.

`#define menuErrNotFound (menuErrorClass | 2)`
Could not locate the specified item.

`#define menuErrOutOfMemory (menuErrorClass | 5)`
There is not enough memory available to perform the operation.

`#define menuErrSameId (menuErrorClass | 3)`
The menu already contains an item with the specified ID.

`#define menuErrTooManyItems (menuErrorClass | 4)`
The command tool bar already has the maximum number of buttons allowed (currently 8).

Miscellaneous Constants

Purpose Other constants defined in `Menu.h`.

Declared In `Menu.h`

Constants

`#define MenuSeparatorChar '-'`
Special character indicating that the menu item is a bar used to separate groups of related menu items.

`#define noMenuItemSelection -1`
There is currently no item selected within a menu.

`#define noMenuSelection -1`
There is currently no pull-down menu selected.

`#define separatorItemSelection -2`
The currently selected item within a menu is the bar used to separate groups of related menu items.

`#define menuCmdBarMaxTextLength 20`
The maximum length of the text that is briefly displayed as a status message when a button in the menu command tool bar is tapped.

MenuCmdBarResultType Typedef

Purpose Specifies what the result of a user tapping a specific button in the menu command tool bar is.

Declared In `Menu.h`

Prototype
```
typedef Enum8 MenuCmdBarResultType
```

Constants

`menuCmdBarResultNone`
No result.

`menuCmdBarResultChar`
The result is a character to send in a `keyDownEvent`.

`menuCmdBarResultMenuItem`
The result is the ID of the menu item to send in a `menuEvent`.

`menuCmdBarResultNotify`
The result is a notification constant to be broadcast using `SysNotifyBroadcastDeferred()`.

See Also `MenuCmdBarAddButton()`, `MenuCmdBarGetButtonData()`

Menu Events

menuCloseEvent

Purpose This event is not currently used. If you need to perform some cleanup (such as closing a resource) after the menu item you added is no longer needed, do so in response to `frmCloseEvent`.

menuCmdBarOpenEvent

Purpose Sent when the menu command tool bar is about to be displayed.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype
```
struct menuCmdBarOpen {
   Boolean preventFieldButtons;
   uint8_t reserved;
} menuCmdBarOpen;
```

Fields

`preventFieldButtons`
: If `true`, the UI Library does not add the standard cut, copy, paste, and undo buttons when the focus is on a field. If `false`, the UI Library adds the buttons.

`reserved`
: Unused.

Comments

Applications might respond to this event by calling `MenuCmdBarAddButton()` to add custom buttons to the command tool bar. Return `false` to allow the default system behavior to take place.

Shared libraries or other non-application code resources can add buttons to the tool bar by registering to receive the `sysNotifyMenuCmdBarOpenEvent` notification.

To prevent the command tool bar from being displayed, respond to this event and return `true`. Returning `true` prevents the UI Library from displaying the tool bar.

menuCmdBarTimeoutEvent

Purpose

Sent when the menu command tool bar has reached its timeout limit for activity and is about to be erased.

Declared In

`EventCodes.h`

Prototype

No data is passed with this event.

Comments

Applications do not need to send or handle this event. It is used internally by the menu command tool bar.

menuEvent

Purpose

Sent when one of the following occurs:

- When the user selects an item from a pull-down menu
- When the user draws the menu command stroke followed by a character; for example, Command-C for copy
- When the user taps one of the buttons on the command tool bar and the button is set up to generate a `menuEvent`.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct menu {
  uint16_t itemID;
} menu;
```

Fields

`itemID`
: Item ID of the selected menu command.

Comments Applications respond to this event to perform the action that the user requested. Return `true` to indicate that you've handled the event.

menuOpenEvent

Purpose Sent when a new active menu has been initialized.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct menuOpen {
   uint16_t menuRscID;
   int16_t cause;
} menuOpen;
```

Fields

`menuRscID`
: Resource ID of the menu.

`cause`
: Reason for opening the menu. This value is one of the Menu Activation Constants.

Comments A menu becomes active the first time the user taps the Menu icon or taps the form's title bar, and a menu might need to be re-initialized and reactivated several times during the life of an application.

A menu remains active until one of the following happens:

- A `FrmSetMenu()` call changes the active menu on the form.
- A new form, even a modal form or alert panel, becomes active.

Suppose a user selects your application's About item from the Options menu then clicks the OK button to return to the main form. When the About dialog is displayed, it becomes the active form, which causes the main form's menu state to be erased. This menu state is not restored when the main form becomes active again. The next time the user requests the menu, it must be initialized again, so `menuOpenEvent` is sent again.

Applications might respond to this event by adding, hiding, or unhiding menu items using the functions `MenuAddItem()`, `MenuHideItem()`, or `MenuShowItem()`. Return `false` to allow the default system behavior to take place.

Menu Notifications

sysNotifyMenuCmdBarOpenEvent

Purpose Broadcast when the menu command tool bar is about to be displayed at the bottom of the screen.

Declared In `NotifyMgr.h`

Prototype None.

Comments Register for this notification if you are writing a shared library or system resource that needs to add a button to the menu command bar or to suppress the menu command tool bar. To add a button, call `MenuCmdBarAddButton()`. To suppress the command tool bar, set the `handled` field to `true`.

Applications that need to add their own buttons to the menu command tool bar should do so in response to a `menuCmdBarOpenEvent`. They should *not* register for this notification because an application should only add buttons if it is already the active application. The notification is sent after the event has been received, immediately before the command tool bar is displayed.

Menu Functions and Macros

MenuAddItem Function

Purpose Adds an item to the currently active menu.

Declared In `Menu.h`

Prototype

```
status_t MenuAddItem (uint16_t positionId,
   uint16_t id, char cmd, const char *textP)
```

Parameters

→ *positionId*
ID of an existing menu item. The new menu item is added after this menu item.

→ *id*
ID value to use for the new menu item.

→ *cmd*
Menu command shortcut key. If you provide menu shortcuts, make sure that each shortcut is unique among all commands available at that time.

→ *textP*
Pointer to the text to display for this menu item, including the menu shortcut key. To include a shortcut key, begin the string with the item's text, then type a tab character, and then the item's shortcut key.

To create a separator bar, create a one-character string containing the `MenuSeparatorChar` constant.

Returns `errNone` upon success or one of the following if an error occurs:

`menuErrNoMenu`
The *textP* parameter is `NULL`.

`menuErrSameId`
The menu already contains an item with the ID *id*.

`menuErrNotFound`
The menu doesn't contain an item with the ID *positionId*.

Comments You should call this function only in response to a `menuOpenEvent`. This event is generated when the menu is first made active. In general, a form's menu becomes active the first time a `keyDownEvent` with a `vchrMenu` or `vchrCommand` is generated,

and it remains active until a new form (including a modal form or alert panel) is displayed or until `FrmSetMenu()` is called to change the form's menu. Palm OS® user interface guidelines discourage adding or hiding menu items at any time other than when the menu is first made active.

WARNING! The system may need to resize and move the menu's memory block as a result of this function, invalidating any pointer to the menu. Call `MenuGetActiveMenu()` after calling this function to obtain a pointer to the menu's new location.

This function may display a fatal error message if there is no current menu.

MenuCmdBarAddButton Function

Purpose Defines a button to be displayed on the command tool bar.

Declared In `Menu.h`

Prototype

```
status_t MenuCmdBarAddButton (uint8_t where,
    DmOpenRef database, uint16_t bitmapId,
    MenuCmdBarResultType resultType,
    uint32_t result, char *nameP)
```

Parameters

→ *where*

Either `menuCmdBarOnLeft` to add the button to the left of the other buttons on the command tool bar, `menuCmdBarOnRight` to add it to the right of the other buttons, or a number indicating the exact position of the button. Button positions are numbered from right to left, and the rightmost position is number 1.

It is best to add buttons on the left side. If you add buttons to the right, this function moves all existing buttons over one position to the left.

If you specify an exact position, this function adds buttons to the first open slot if you don't specify buttons in order. That is, if you add a button at position 3 and there are no buttons at positions 1 and 2, this function adds the button at position 1.

→ *database*
Open resource database containing *bitmapId*.

→ *bitmapId*
Resource ID of the bitmap to display on the button. The bitmap's dimensions should be 13 standard coordinates high by 16 standard coordinates wide.

→ *resultType*
The type of data contained in the *result* parameter. See MenuCmdBarResultType.

→ *result*
The data to send when the user taps this button. This can be a character, a menu item ID, or a notification constant.

→ *nameP*
Pointer to the text to display in the status message if the user taps the button. If NULL, the text is taken from the menu item that matches the ID or shortcut character contained in *result*, if a match is found.

If you supply a text buffer for this parameter, MenuCmdBarAddButton() makes a copy of the buffer.

Returns errNone upon success, or one of the following error codes:

menuErrOutOfMemory
There is not enough memory available to perform the operation.

menuErrTooManyItems
The command tool bar already has the maximum number of buttons allowed (currently 8).

Comments Call this function in response to a menuCmdBarOpenEvent or to the notification sysNotifyMenuCmdBarOpenEvent. Both of these signal that the user has written the menu command stroke in the input area and the command tool bar is about to open. Your response should be to add buttons to the tool bar and to return false, indicating that you have not completely handled the event.

The sysNotifyMenuCmdBarOpenEvent notification is intended to be used only by shared libraries, system extensions, and other code resources that do not use an event loop. If you're writing an application, always respond to the event instead of the notification; an application should only add buttons to the tool bar if it is the

current application. If you register for the notification, you receive it each time the command tool bar is displayed, whether your application is active or not.

Note that the command tool bar is allocated each time it is opened and is deallocated when it is erased from the screen.

The *result* and *resultType* parameters specify what the result should be if the user taps the button. *result* contains the actual data, and *resultType* contains a constant that specifies the type of data in *result*. Typically, the result is to enqueue a menuEvent. In this case, *resultType* is menuCmdBarResultMenuItem and the *result* is the ID of the menu item that should included in the event.

You may also specify the shortcut character instead of the menu ID; however, doing so is inefficient. When *result* is a shortcut character, the MenuHandleEvent() function enqueues a keyDownEvent with the character in *result*. During the next cycle of the event loop, MenuHandleEvent() enqueues a menuEvent in response to the keyDownEvent. Thus, it is better to have your button enqueue the menuEvent directly.

If you call MenuCmdBarAddButton() outside of an application, you might not know of any menu items in the active menu (unless your code has added one using MenuAddItem()). In this case, specify a notification to be broadcast. The notification is broadcast at the top of the next event loop, and it must contain no custom data. (Applications may also use the notification result type.)

See Also MenuCmdBarDisplay(), MenuCmdBarGetButtonData()

MenuCmdBarDisplay Function

Purpose Displays the command tool bar.

Declared In Menu.h

Prototype

```
void MenuCmdBarDisplay (void)
```

Parameters None.

Returns Nothing.

Comments This function displays the command tool bar when the user enters the command keystroke. You normally do not call this function in your own code. `FrmHandleEvent()` calls it at the end of its handling of `menuCmdBarOpenEvent`.

See Also `MenuCmdBarAddButton()`, `MenuCmdBarGetButtonData()`

MenuCmdBarGetButtonData Function

Purpose Gets the data for a given command button.

Declared In `Menu.h`

Prototype

```
Boolean MenuCmdBarGetButtonData
    (int16_t buttonIndex, DmOpenRef *databaseP,
    uint16_t *bitmapIdP,
    MenuCmdBarResultType *resultTypeP,
    uint32_t *resultP, char *nameP)
```

Parameters

→ *buttonIndex*
Index of the button for which you want to obtain information. Buttons are ordered from right to left, with the rightmost button at index 0.

← *databaseP*
Open resource database that contains *bitmapId*. Pass `NULL` if you don't want to retrieve this value.

← *bitmapIdP*
The resource ID of the bitmap displayed on the button. Pass `NULL` if you don't want to retrieve this value.

← *resultTypeP*
The type of action this button takes. Pass `NULL` if you don't want to retrieve this value.

← *resultP*
The result of tapping the button. Pass `NULL` if you don't want to retrieve this information.

← *nameP*
The text displayed in the status message when this button is tapped. Pass `NULL` if you don't want to retrieve this information. If not `NULL`, *nameP* must point to a string of `menuCmdBarMaxTextLength` size.

Returns `true` if the information was retrieved successfully, `false` if there is no command tool bar or if there is no button at *`buttonIndex`*.

Comments You can use this function to retrieve information about the buttons that are displayed on the command tool bar. If the command tool bar has not yet been initialized, this function returns `false`.

Note that the command tool bar is allocated when the user enters the command keystroke and deallocated when `MenuEraseStatus()` is called. Thus, the only logical place to call `MenuCmdBarGetButtonData()` is in response to a `menuCmdBarOpenEvent` or a `sysNotifyMenuCmdBarOpenEvent` notification.

See Also `MenuCmdBarDisplay()`, `MenuCmdBarAddButton()`

MenuDispose Function

Purpose Releases any memory allocated to the menu and the command status and restore any saved bits to the screen.

Declared In `Menu.h`

Prototype

```
void MenuDispose (MenuBarType *menuP)
```

Parameters

→ *`menuP`*
Pointer to the menu to dispose. If `NULL`, this function returns immediately.

Returns Nothing.

Comments Most applications do not need to call this function directly. `MenuDispose()` is called by the system when the form that contains the menu is no longer the active form, when the form that contains the menu is freed, and when `FrmSetMenu()` is called to change the form's menu bar.

See Also `MenuInit()`, `MenuDrawMenu()`

MenuDrawMenu Function

Purpose Requests that the current menu bar and the last pull-down that was visible be drawn or redrawn.

Declared In `Menu.h`

Prototype

```
void MenuDrawMenu (MenuBarType *menuP)
```

Parameters

→ *menuP*
Pointer to a menu bar.

Returns Nothing.

Comments Invalidates the windows for the menu bar and the pull-down menu that was last visible so that they are later redrawn.

Most applications do not need to call this function directly. `MenuHandleEvent()` calls `MenuDrawMenu()` when the user taps the Menu icon or the form's title bar.

The menu bar and the pull-down menu are drawn in front of all the application windows.

A menu keeps track of the last pull-down menu displayed for as long as the menu is active. A menu loses its active status under these conditions:

- When `FrmSetMenu()` is called to change the active menu on the form.
- When a new form, even a modal form or alert panel, becomes active.

Suppose a user selects your application's About item from the Options menu then clicks the OK button to return to the main form. When the About dialog is displayed, it becomes the active form, which causes the main form's menu state to be erased. This menu state is not restored when the main form becomes active again. The next time `MenuDrawMenu()` is called (that is, the next time the user taps the Menu silk-screen button), the menu bar is displayed as it was before, and the first pull-down menu listed in the menu bar is displayed instead of the Options pull-down menu.

See Also `MenuInit()`, `MenuDispose()`

MenuEraseStatus Function

Purpose Erases the menu command status.

Declared In `Menu.h`

Prototype
```
void MenuEraseStatus (MenuBarType *menuP)
```

Parameters → *menuP*
Pointer to a menu bar, or `NULL` for the current menu.

Returns Nothing.

Comments When the user selects a menu command using the command keystroke, the command tool bar is displayed at the bottom of the screen. `MenuEraseStatus()` erases the tool bar.

You do not need to call this function explicitly. Let the current menu command status remove itself automatically. Otherwise, you may cause text to be erased before the user has a chance to see it. In Palm OS Cobalt, the menu command bar draws in its own window and does not interfere with your application's drawing.

See Also `MenuInit()`

MenuGetActiveMenu Function

Purpose Returns a pointer to the currently active menu.

Declared In `Menu.h`

Prototype
```
MenuBarType *MenuGetActiveMenu (void)
```

Parameters None.

Returns A pointer to the currently active menu, or `NULL` if there is none.

Comments An active menu is not necessarily visible on the screen. A menu might be active but not visible, for example, if a menu command shortcut has been entered. In general, a form's menu becomes active the first time a `keyDownEvent` with a `vchrMenu` or `vchrCommand` is generated, and it remains active until a new form (including a modal form or alert panel) is displayed or until `FrmSetMenu()` is called to change the form's menu.

See Also `MenuHandleEvent()`, `MenuSetActiveMenu()`

MenuHandleEvent Function

Purpose Handles events in the current menu.

Declared In `Menu.h`

Prototype

```
Boolean MenuHandleEvent (MenuBarType *menuP,
    EventType *event, status_t *error)
```

Parameters

→ *menuP*
Pointer to a menu bar.

→ *event*
Pointer to an `EventType` structure.

← *error*
Error (or `errNone` if no error). Currently this function always sets *error* to zero.

Returns `true` if the event is handled; `false` otherwise.

Comments The following cases will result in a return value of `true`:

- A `penDownEvent` or `penMoveEvent`. If the pen is within the bounds of the menu bar, the selected menu title is inverted and the appropriate pull-down menu is drawn. Any previous pull-down menu is erased. If the pen is within the bounds of a pull-down menu, inverts the selected menu item and removes the selection from any previously selected menu item. If a pull-down menu is currently displayed and the pen is down outside the bounds of the menu, the menu is dismissed.
- A `penUpEvent`. If the pen is up within the bounds of the menu bar, erases the current pull-down menu. If the pen is within the bounds of a menu item in a pull-down menu, sends a `menuEvent` containing the resource ID of the selected menu item. If the pen is up outside the bounds of the menu bar and menu resources, they are both erased and no item is selected.
- `keyDownEvent` containing `vChrCommand` or `vchrMenu`. If the menu is being activated and initialized for the first time, this function enqueues a `menuOpenEvent` containing the reason the menu was initialized (`menuButtonCause` for a

vchrMenu or menuCommandCause for a vchrCommand), and then the current event is added after it.

For vchrCommand, this function sends a menuCmdBarOpenEvent.

- winUpdateEvent. Redraws the currently displayed menu bar and pull-down menu.

MenuHideItem Function

Purpose Hides the specified menu item.

Declared In Menu.h

Prototype
```
Boolean MenuHideItem (uint16_t id)
```

Parameters → *id*
ID of the menu item to hide.

Returns true if the item was successfully hidden, false otherwise.

Comments You should call this function only in response to a menuOpenEvent. This event is generated when the menu is first made active. In general, a form's menu becomes active the first time a keyDownEvent with a vchrMenu or vchrCommand is generated, and it remains active until a new form (including a modal form or alert panel) is displayed or until FrmSetMenu() is called to change the form's menu. Palm OS user interface guidelines discourage adding or hiding menu items at any time other than when the menu is first made active.

See Also MenuShowItem()

MenuInit Function

Purpose Loads a menu resource from a resource file.

Declared In Menu.h

Prototype
```
MenuBarType *MenuInit (DmOpenRef database,
    uint16_t resourceId)
```

Parameters → *database*
Open resource database containing the menu.

→ *resourceId*
ID that identifies the menu resource.

Returns A pointer to a `MenuBarType` structure.

Comments The menu is not usable until `MenuSetActiveMenu()` is called.

Typically, you do not need to call this function directly. A form stores the resource ID of the menu associated with it and initializes that menu as necessary. If you want to change the form's menu, call `FrmSetMenu()`, which handles disposing of the form's current menu, associating the new menu with the form, and initializing when needed.

See Also `MenuSetActiveMenu()`, `MenuDispose()`

MenuSetActiveMenu Function

Purpose Sets the current menu.

Declared In `Menu.h`

Prototype
```
MenuBarType *MenuSetActiveMenu(MenuBarType *menuP)
```

Parameters → *menuP*
Pointer to the structure that contains the new menu, or `NULL` for none.

Returns A pointer to the menu that was active before the new menu was set, or `NULL` if no menu was active.

Comments This function sets the active menu but does not associate it with a form. It's recommended that you call `FrmSetMenu()` instead of `MenuSetActiveMenu()`. `FrmSetMenu()` sets the active menu, frees the old menu, and associates the newly active menu with the form, which means the menu will be freed when the form is freed.

See Also `MenuGetActiveMenu()`

MenuSetActiveMenuRscID Function

Purpose Sets the current menu by resource ID.

Declared In `Menu.h`

Prototype
```
void MenuSetActiveMenuRscID (DmOpenRef database,
    uint16_t resourceId)
```

Parameters

→ *database*
Open resource database containing the menu.

→ *resourceId*
Resource ID of the menu to be made active.

Returns Nothing.

Comments This function is similar to `MenuSetActiveMenu()` except that you pass the menu's resource ID instead of a pointer to a menu structure. It is used as an optimization; with `MenuSetActiveMenu()`, you must initialize the menu before making it active. Potentially, the application may exit before the menu is needed, making this memory allocation unnecessary. `MenuSetActiveMenuRscID()` simply stores the resource ID. The next time a menu is requested, the menu is initialized from this resource.

It's recommended that you call `FrmSetMenu()` instead of calling this function for the reasons given in `MenuSetActiveMenu()`.

MenuShowItem Function

Purpose Makes the specified menu item visible.

Declared In `Menu.h`

Prototype
```
Boolean MenuShowItem (uint16_t id)
```

Parameters

→ *id*
ID of the menu item to display.

Returns `true` if the `hidden` attribute of the specified item was successfully disabled, `false` otherwise.

Comments You should call this function only in response to a `menuOpenEvent`. This event is generated when the menu is first made active. In general, a form's menu becomes active the first time a

keyDownEvent with a vchrMenu or vchrCommand is generated, and it remains active until a new form (including a modal form or alert panel) is displayed or until FrmSetMenu() is called to change the form's menu. Palm OS user interface guidelines discourage adding or hiding menu items at any time other than when the menu is first made active.

See Also MenuHideItem()

24

Phone Lookup Reference

This chapter provides descriptions of the phone lookup functions. It covers the following topics:

The header file `PhoneLookup.h` declares the API that this chapter describes.

Phone Lookup Structures and Types

AddrLookupParamsType Struct

Purpose Structure to pass to `PhoneNumberLookupCustom()` to precisely control the phone number lookup dialog and paste process. The `AddrLookupParamsPtr` type defines a pointer to an `AddrLookupParamsType` structure.

Declared In `PhoneLookup.h`

Prototype

```
typedef struct {
   char *title;
   char *pasteButtonText;
   char lookupString[addrLookupStringLength];
   AddressLookupFields field1;
   AddressLookupFields field2;
   Boolean field2Optional;
```

```
   Boolean userShouldInteract;
   char *formatStringP;
   MemHandle resultStringH;
   uint32_t uniqueID;
} AddrLookupParamsType;
typedef AddrLookupParamsType *AddrLookupParamsPtr
```

Fields

`title`
: Title to appear in the title bar. Supply `NULL` to use the default title.

`pasteButtonText`
: Text to appear in paste button. Supply `NULL` to use the default, "paste".

`lookupString`
: Buffer containing the string to look up. If the string matches only one record, that record is used without presenting the lookup dialog to the user. `PhoneNumberLookup()` and `PhoneNumberLookupCustom()` both set this field based upon the current selection or cursor position.

`field1`
: Field to search by. This field appears on the left side of the lookup dialog. If the field is the sort field, the search is performed using a binary search. If the field isn't the sort field, searching is performed by a linear search, which can be slow. Supply one of the values in the `AddressLookupFields` enum to specify the field to search by.

`field2`
: Field to display on the right. Often displays some information about the person. If it is a phone field and a record has multiple instances of the phone type then the person appears once per instance of the phone type. Either `field1` or `field2` may be a phone field, but not both. Supply one of the values in the `AddressLookupFields` enum to specify the field to display.

`field2Optional`
: A value of `true` means that the record need not have `field2` in order to be listed. A value of `false` indicates that `field2` is required in order for the record to be listed.

`userShouldInteract`
: A value of `true` forces the user to resolve non-unique lookups. A `false` value means a non-unique and complete lookup causes `resultStringH` to be set to `NULL` and `uniqueID` to be set to 0.

`formatStringP`
: Controls the format of the paste string. All characters in the format string are literal unless they identify a field (signified by a caret (^) followed by the field name). For example, the format string "^first - ^home" might result in "Roger - 123-4567". Allowable field names are:

- name
- first
- company
- address
- city
- state
- zipcode
- country
- title
- custom1
- custom2
- custom3
- custom4
- work
- home
- fax
- other
- email
- main
- pager
- mobile
- listname

`resultStringH`
: If there is a format string, a result string is allocated on the dynamic heap and its handle is returned here.

`uniqueID`
: The unique ID of the found record, or 0 if none was found, is returned here.

Phone Lookup Constants

AddressLookupFields Typedef

Purpose Specifies the fields you can search by and the corresponding fields to return using the `field1` and `field2` elements of the `AddrLookupParamsType` structure. For both `field1` and `field2`, pass one of the values up to but not including `addrLookupFieldCount`.

Declared In `PhoneLookup.h`

Prototype

```
typedef Enum8 AddressLookupFields
```

Constants

`addrLookupName`

`addrLookupFirstName`

`addrLookupCompany`

`addrLookupAddress`

`addrLookupCity`

`addrLookupState`

`addrLookupZipCode`

`addrLookupCountry`

`addrLookupTitle`

`addrLookupCustom1`

`addrLookupCustom2`

`addrLookupCustom3`

`addrLookupCustom4`

`addrLookupNote`

`addrLookupWork`

`addrLookupHome`

`addrLookupFax`

`addrLookupOther`

`addrLookupEmail`

```
addrLookupMain
addrLookupPager
addrLookupMobile
addrLookupSortField
addrLookupListPhone
addrLookupFieldCount
addrLookupNoField = 0xff
```

Lookup String Length Constant

Purpose Maximum size of the lookup string.

Declared In `PhoneLookup.h`

Constants `#define addrLookupStringLength 20`
Size of the string contained in the `lookupString` field of `AddrLookupParamsType`.

Phone Lookup Functions and Macros

PhoneNumberLookup Function

Purpose Calls the Address Book application to look up a phone number.

Declared In `PhoneLookup.h`

Prototype `void PhoneNumberLookup (FieldType *fldP)`

Parameters → *`fldP`*
Field in which the text to match is found.

Returns Nothing.

Comments This function displays the user's phone list and inserts the chosen name and number (or company name, name, and number, if that's how the user's Address Book preferences indicate that the phone list should be sorted) into the specified field. When displaying the phone list, `PhoneNumberLookup()` scrolls the list to the entry that best matches the supplied field. The match compares the field

contents against the name or company name (depending on the user's preferences) as follows:

- If the field contains selected text, `PhoneNumberLookup()` tries to match against the selected text. The selected text is then replaced with the text of the chosen address list entry.
- If there is no selected text in the field, `PhoneNumberLookup()` matches against the word in which the cursor lies (the match will take place if the cursor is at the beginning, the end, or within a word). The matched word is replaced with the text of the chosen address list entry.
- If the cursor does not lie within or adjoin a word, `PhoneNumberLookup()` displays the address list starting at the first entry, and the text of the chosen entry is inserted at the current position within the text field.

If the user chooses Cancel when the address list is displayed, the field contents are left unaltered. The paste operation takes place through the clipboard so that Undo can be used to restore the field to its previous state.

See Also `PhoneNumberLookupCustom()`

PhoneNumberLookupCustom Function

Purpose Calls the Address Book application to look up a phone number.

Declared In `PhoneLookup.h`

Prototype

```
void PhoneNumberLookupCustom (FieldType *fldP,
    AddrLookupParamsType *params,
    Boolean useClipboard)
```

Parameters

→ *fldP*
Field in which the text to match is found.

↔ *params*
A structure that allows full control over the find dialog and the format of the resulting paste string. See `AddrLookupParamsType` for a description of the fields in this structure.

→ *useClipboard*
If true, `PhoneNumberLookupCustom()` pastes the result into the field through the clipboard, thereby enabling undo.

Returns Nothing.

Comments This function displays two fields from each record in the user's address list and inserts a formatted string based upon fields in the chosen record into the specified field. When displaying the address list, `PhoneNumberLookupCustom()` scrolls the list to that entry that best matches the supplied field. The match compares the field contents against the field specified in the `params` structure's `field1` element as follows:

- If the field contains selected text, `PhoneNumberLookupCustom()` tries to match against the selected text. The selected text is then replaced with the text of the chosen address list entry.
- If there is no selected text in the field, `PhoneNumberLookupCustom()` matches against the word in which the cursor lies (the match will take place if the cursor is at the beginning, the end, or within a word). The matched word is replaced with the text of the chosen address list entry.
- If the cursor does not lie within or adjoin a word, `PhoneNumberLookupCustom()` displays the address list starting at the first entry, and the text of the chosen entry is inserted at the current position within the text field.

`PhoneNumberLookupCustom()` copies the portion of the field used for the search—the selected text or the word in which the cursor lies—into the `lookupString` field in the `params` structure prior to replacing that part of the field with the user-selected entry.

If the user chooses Cancel when the address list is displayed, the field contents are left unaltered. Depending on the value of the `useClipboard` parameter, the paste operation can take place through the clipboard so that Undo can be used to restore the field to its previous state.

25

Private Records Reference

This chapter describes the private records API as declared in `PrivateRecords.h`. It discusses the following topics:

Private Records Constants

privateRecordViewEnum Typedef

Purpose Provides the available choices for displaying private records.

Declared In `PrivateRecords.h`

Prototype

```
typedef Enum8 privateRecordViewEnum
```

Constants

`showPrivateRecords = 0x00`
Display private records in the user interface.

`maskPrivateRecords`
Show a shaded rectangle in place of a private record.

`hidePrivateRecords`
Hide private records and provide no indication in the user interface that they exist.

Private Records Functions and Macros

SecSelectViewStatus Function

Purpose Displays a form that allows the user to select whether to hide, show, or mask private records.

Declared In `PrivateRecords.h`

Prototype

```
privateRecordViewEnum SecSelectViewStatus (void)
```

Parameters None.

Returns A `privateRecordViewEnum` constant that indicates which option the user selected.

Comments This function displays a dialog that allows users to change the preference `prefShowPrivateRecords`, which controls how private records are displayed.

When the user taps the OK button in this dialog, `SecVerifyPW()` is called to see if the user changed the preference setting and, if so, to prompt the user to enter the appropriate password.

After calling this function, your code should check the return value or the value of `prefShowPrivateRecords` and mask, display, or hide the private records accordingly. See the description of `TblSetRowMasked()` for a partial example.

SecVerifyPW Function

Purpose Displays a password dialog, verify the password, and changes the private records preference.

Declared In `PrivateRecords.h`

Prototype

```
Boolean SecVerifyPW
    (privateRecordViewEnum newSecLevel)
```

Parameters → *`newSecLevel`*
The security level (display, hide, or mask) selected on the private records dialog. See privateRecordViewEnum.

Returns `true` if the `prefShowPrivateRecords` preference was successfully changed, `false` if not.

Comments This function checks `newSecLevel` against the current value for the preference. If the two values differ and `newSecLevel` indicates a decrease in security, a dialog is displayed prompting the user to enter a password. (Hidden is considered the most secure, followed by masked. Showing private records is considered the least secure.) If the password is entered successfully, the preference is changed.

This function also displays an alert message if the security level has changed to either hidden or masked.

26

Progress Manager Reference

This chapter provides reference material for the Progress Manager.

The header file `Progress.h` declares the API that this chapter describes. For more information on the Progress Manager, see the section "Progress Dialogs" on page 31.

Progress Manager Structures and Types

PrgCallbackData Struct

Purpose Structure used to share information between the Progress Manager and the `PrgCallbackFunc()` callback function.

Declared In `Progress.h`

Prototype

```
typedef struct {
   uint16_t stage;
   uint16_t bitmapId;
   DmOpenRef bitmapDatabase;
   uint32_t textLen;
   char *textP;
   char *message;
   status_t error;
   uint16_t canceled:1;
   uint16_t showDetails:1;
```

```
    uint16_t textChanged:1;
    uint16_t timedOut:1;
    uint16_t displaySkipBtn:1;
    uint16_t skipped:1;
    uint16_t spareBits1:10;
    uint16_t padding1;
    uint32_t timeout;
    uint32_t padding2;
    uint32_t barMaxValue;
    uint32_t barCurValue;
    char *barMessage;
    uint16_t barFlags;
    uint16_t delay:1;
    uint16_t spareBits2:15;
    void *userDataP;
} PrgCallbackData, *PrgCallbackDataPtr
```

Fields

→ `stage`
: Current stage passed from `PrgUpdateDialog()`.

↔ `bitmapId`
: Resource ID of the bitmap to display in the progress dialog, if any.

↔ `bitmapDatabase`
: Open database containing `bitmapId`.

→ `textLen`
: Size in bytes of the text buffer `textP`. Note that this value is defined by the system, not by your callback function. Be careful not to exceed this length in `textP`.

↔ `textP`
: Buffer to hold the text to display in the updated dialog. You might want to look up a message in a resource file, based on the value in the `stage` field. Also, you should append the additional text in the `message` field, to form the full string to display. Be sure to include a null terminator at the end of the string you return, and don't exceed the length in `textLen`.

→ `message`
: Additional text to display in the dialog (from the *`messageP`* parameter to `PrgUpdateDialog()`). This string is no longer than `progressMaxMessage`.

↔ `error`
: Current error (passed from the *`err`* parameter to `PrgUpdateDialog()`).

↔ `canceled`
: `true` if user has pressed the Cancel button.

→ `showDetails`
: `true` if user pressed the down arrow button on the device for more details. (Because this is a non-standard user interface technique, you shouldn't use this feature to display details that users need under normal conditions. It's more for debugging purposes.)

← `textChanged`
: If `true`, then the system should update text (defaults to `true`). You can set this to `false` to avoid an update to the text.

→ `timedOut`
: `true` if and only if the update was caused by a timeout. If an application calls `PrgUpdateDialog()` and a timeout occurs before the callback is called, this field is `false`.

← `displaySkipBtn`
: If `true`, the progress dialog displays a skip button, which allows the user to skip the current stage.

↔ `skipped`
: If `true`, the user tapped the skip button to skip the current stage.

`spareBits1`
: Not used.

`padding1`
: Not used.

↔ `timeout`
: Timeout in ticks to force next update. After this number of ticks, an update is automatically triggered (which sets the `timedOut` flag). You can use this feature to do a simple animation effect. Note that you must set the timeout for `EvtGetEvent()` to a value that is equal to or less than this value, otherwise you won't get update events as frequently as you expect.

`padding2`
Not used.

↔ `barMaxValue`
The maximum value for the progress bar.

↔ `barCurValue`
The current value that the progress bar should display.

`barMessage`
Not used.

`barFlags`
Not used.

↔ `delay`
If `true`, delay for one second after updating the dialog. Use this value when you are displaying the final progress message so that users have a chance to see the message before the dialog closes.

`spareBits2`
Not used.

→ `userDataP`
A pointer to any application-defined data that the function needs to access. You specify this pointer as a parameter to `PrgStartDialog()` if the callback function needs to access some application data but does not have access to the application's global variables.

ProgressType Struct

Purpose Internal structure that identifies the current progress. This structure is created with `PrgStartDialog()` and passed to all other Progress Manager functions. A `ProgressPtr` defines a pointer to a `ProgressType` structure.

Declared In `Progress.h`

Prototype
```
typedef struct ProgressType ProgressType,
*ProgressPtr
```

Fields None.

Progress Manager Constants

Progress Manager String Length Constants

Purpose Specify the maximum sizes of the strings that are displayed in the progress dialog.

Declared In `Progress.h`

Constants

`#define progressMaxButtonText 7`
Maximum length of the text in the OK or Cancel button.

`#define progressMaxMessage 128`
Maximum length of the `message` field in `PrgCallbackData`.

`#define progressMaxTitle 31`
Maximum length in bytes for a title to a progress dialog.

Progress Manager Events

prgUpdateEvent

Purpose Sent when another thread has updated the progress.

Declared In `EventCodes.h`

Prototype None.

Progress Manager Functions and Macros

PrgGetError Function

Purpose Returns the current error, if any.

Declared In `Progress.h`

Prototype
```
status_t PrgGetError (const ProgressType *prgP)
```

Parameters

→ *prgP*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

Returns The value of the `error` field in `PrgCallbackData`.

PrgHandleEvent Function

Purpose Handles events related to the active progress dialog.

Declared In `Progress.h`

Prototype
```
Boolean PrgHandleEvent (ProgressPtr prgGP,
    EventType *eventP)
```

Parameters

→ *prgGP*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

→ *eventP*
Pointer to an event. You can pass a `NULL` event to cause this function to immediately call your `PrgCallbackFunc()` function and then update the dialog (for example, after you call `PrgUpdateDialog()`).

Returns `true` if the system handled the event. If it returns `false`, you should check if the user cancelled the dialog by calling `PrgUserCancel()`.

Comments Use this function instead of `SysHandleEvent()` when you have a progress dialog. `PrgHandleEvent()` internally calls `SysHandleEvent()` as needed.

If an update to the dialog is pending (from a call to `PrgUpdateDialog()`, for example) this function handles that

event and causes the dialog to be updated. As part of this process, the `PrgCallbackFunc()` function you specified in your call to `PrgStartDialog()` is called. Your `PrgCallbackFunc()` function should set the `textP` buffer in the `PrgCallbackData` structure with the new message to be displayed in the progress dialog. Optionally, you can also set the `bitmapId` and `bitmapDatabase` fields to include an icon in the update dialog.

See Also `PrgStartDialog()`, `PrgStopDialog()`, `PrgUpdateDialog()`, `PrgUserCancel()`

PrgStartDialog Function

Purpose Displays a progress dialog that can be updated.

Declared In `Progress.h`

Prototype

```
ProgressPtr PrgStartDialog (const char *title,
   PrgCallbackFunc textCallback, void *userDataP)
```

Parameters

→ *title*
Pointer to a title for the progress dialog. Must be a null-terminated string that is no longer than `progressMaxTitle`.

→ *textCallback*
Pointer to a callback function that supplies the text and icons for the current progress state. See `PrgCallbackFunc()`.

→ *userDataP*
A pointer to data that you need to access in the callback function. This address gets passed directly to your function.

Returns A pointer to a progress structure. This pointer must be passed to other Progress Manager functions and *must* be released by calling `PrgStopDialog()`. `NULL` is returned if the system is unable to allocate the progress structure.

Comments The dialog created by this function can be updated by another thread using the `PrgUpdateDialog()` function. The dialog can contain a Cancel or OK button. The initial dialog defaults to stage 0 and calls the *textCallback* function to get the initial text and icon data for the progress dialog.

Progress dialogs cannot be associated with window constraints resources, and usually one is not necessary. The UI Library makes sure that the dialog is always the same size and snaps to the bottom of the application area. If you need to set window creation attributes that you would normally set in a constraints resource, use `PrgStartDialogWithFlags()`.

See Also `PrgHandleEvent()`, `PrgStopDialog()`, `PrgUpdateDialog()`, `PrgUserCancel()`

PrgStartDialogWithFlags Function

Purpose Displays a progress dialog that can be updated. Uses the specified window flags to create the dialog.

Declared In `Progress.h`

Prototype

```
ProgressPtr PrgStartDialogWithFlags
    (const char *title,
    PrgCallbackFunc textCallback, void *userDataP,
    WinFlagsType flags)
```

Parameters

→ *title*
Pointer to a title for the progress dialog. Must be a null-terminated string that is no longer than `progressMaxTitle`.

→ *textCallback*
Pointer to a callback function that supplies the text and icons for the current progress state. See `PrgCallbackFunc()`.

→ *userDataP*
A pointer to data that you need to access in the callback function. This address gets passed directly to your function.

→ *flags*
Window attributes to use when creating the dialog. See `WinFlagsType`. The attribute `winFlagModal` is set for you.

Returns A pointer to a progress structure. This pointer must be passed to other Progress Manager functions and *must* be released by calling `PrgStopDialog()`. `NULL` is returned if the system is unable to allocate the progress structure.

Comments Use this function instead of `PrgStartDialog()` in one of two circumstances:

- You need the window created for the progress dialog to be a transitional window so that you can draw to it outside of `frmUpdateEvent`s. In this case set the `winFlagBackBuffer` bit in *`flags`*.
- You are displaying a progress dialog outside of the application process and it should appear in the same window layer as all of the other windows in that process.

See Also `PrgHandleEvent()`, `PrgStopDialog()`, `PrgUpdateDialog()`, `PrgUserCancel()`

PrgStopDialog Function

Purpose Releases memory used by the progress dialog and restores the screen to its initial state.

Declared In `Progress.h`

Prototype

```
void PrgStopDialog (ProgressPtr prgP,
    Boolean force)
```

Parameters

→ *`prgP`*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

→ *`force`*
`true` removes the progress dialog immediately, `false` causes the system to wait until the user confirms an error, if one is displayed.

Returns Nothing.

Comments If the progress dialog is in a state where it is displaying an error message to the user, this function normally waits for the user to confirm the dialog before it removes the dialog. If you specify `true` for the *`force`* parameter, this causes the system to remove the dialog immediately.

See Also `PrgHandleEvent()`, `PrgStartDialog()`, `PrgUpdateDialog()`, `PrgUserCancel()`

PrgUpdateDialog Function

Purpose Updates the status of the current progress dialog.

Declared In `Progress.h`

Prototype

```
void PrgUpdateDialog (ProgressPtr prgGP,
   status_t err, uint16_t stage,
   const char *messageP, Boolean updateNow)
```

Parameters

→ *prgGP*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

→ *err*
If nonzero, causes the dialog to go into error mode, to display an error message with only an OK button.

→ *stage*
Current stage of progress. The callback function can use this to determine the correct string to display in the updated dialog.

→ *messageP*
Extra text that may be useful in displaying the progress for this stage. Used by the callback function, which can append it to the base message that is based on the stage.

→ *updateNow*
If `true`, the dialog is immediately updated. If `false`, sends a `prgUpdateEvent`, which causes the dialog to be updated on the next call to `PrgHandleEvent()`.

Returns Nothing.

Comments This function can be called from a background thread to update the dialog in the application thread. The *updateNow* parameter must be `false` in this situation.

For more information about how the parameters are used and the callback function, see the section "Application-Defined Functions."

See Also `PrgHandleEvent()`, `PrgStartDialog()`, `PrgStopDialog()`, `PrgUserCancel()`

PrgUserCancel Function

Purpose Returns `true` if the user cancelled the process using the progress dialog.

Declared In `Progress.h`

Prototype
```
Boolean PrgUserCancel (const ProgressType *prgP)
```

Parameters → *prgP*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

Returns `true` if the user tapped the Cancel button; `false` otherwise.

Comments This is a function you can use to check if the user cancelled the process. If the user did cancel, you can change the progress dialog text to something like "Cancelling," or "Disconnecting," or whatever is appropriate for your application. Then you should cancel the process, end the communication session, or do whatever processing is necessary.

See Also `PrgHandleEvent()`, `PrgStartDialog()`, `PrgStopDialog()`, `PrgUpdateDialog()`

PrgUserSkip Function

Purpose Returns `true` if the user skipped the current stage of the process using the progress dialog.

Declared In `Progress.h`

Prototype
```
Boolean PrgUserSkip (const ProgressType *prgP)
```

Parameters → *prgP*
Pointer to a progress structure created by `PrgStartDialog()` or `PrgStartDialogWithFlags()`.

Returns `true` if the user tapped the Skip button; `false` otherwise.

Comments This is a function you can use to check if the user skipped the current stage of the process. If the user did skip, you can change the progress dialog text to something like "Skipping," or whatever is appropriate for your application. Then you should move on to the next stage of the process, or do whatever processing is necessary.

The progress dialog only has a skip button if you specifically enable it by setting `displaySkipBtn` to `true` in the `PrgCallbackData` structure.

Application-Defined Functions

PrgCallbackFunc Function

Purpose Supplies the text and icons for the current progress state.

Declared In `Progress.h`

Prototype

```
Boolean (*PrgCallbackFunc) (PrgCallbackDataPtr cbP)
```

Parameters

↔ *cbP*

A pointer to a `PrgCallbackData` structure.

Returns `true` if the progress dialog should be updated using the values you specified in the `PrgCallbackData` structure. If you return `false`, the dialog is still updated, but with default status messages. (Returning `false` is not recommended.)

Comments This is a callback function that you specify when you call `PrgStartDialog()`. The callback function is called by `PrgHandleEvent()` when it needs current progress information to display in the progress dialog.

In this function, you should set the value of the `textP` buffer to the string you want to display in the progress dialog when it is updated. You can use the value in the `stage` field to look up a message in a string resource. You also might want to append the text in the `message` field to your base string. Typically, the `message` field would contain more dynamic information that depends on a user selection, such as a phone number, device name, or network identifier, etc.

For example, the `PrgUpdateDialog()` function might have been called with a stage of 1 and a message parameter value of a phone number string, "555-1212". Based on the stage, you might find the string "Dialing" in a string resource, and append the phone number, to form the final text "Dialing 555-1212" that you place in the text buffer `textP`.

Keeping the static strings corresponding to various stages in a resource makes it easier to localize your application. More dynamic information can be passed in by calling `PrgUpdateDialog()` with a *`messageP`* parameter.

NOTE: This function is called only if the parameters passed to `PrgUpdateDialog()` have changed from the last time it was called. If `PrgUpdateDialog()` is called twice with exactly the same parameters, the callback function is called only once.

27

Rectangle Reference

This chapter provides reference material for the rectangles API, declared in the header files `Rect.h` and `CmnRectTypes.h`. It is divided into the following sections:

Rectangle Structures and Types

AbsRectType Struct

Purpose Defines a rectangle using four points rather than a top-left coordinate and an extent.

Declared In `CmnRectTypes.h`

Prototype

```
typedef struct AbsRectType {
   Coord left;
   Coord top;
   Coord right;
   Coord bottom;
} AbsRectType
```

Fields

`left`
x coordinate of the left side of the rectangle.

`top`
y coordinate of the top of the rectangle.

`right`
x coordinate of the right side of the rectangle.

`bottom`
y coordinate of the bottom of the rectangle.

Comments The new drawing model defines rectangles by defining left, top, right, and bottom points as this structure does. The Window

Manager drawing model defines rectangles using the RectangleType structure.

See Also AbsToRect(), RectToAbs(), FAbsRectType

PointType Struct

Purpose Defines a point within a window or on the screen.

Declared In CmnRectTypes.h

Prototype

```
typedef struct PointType {
   Coord x;
   Coord y;
} PointType
```

Fields

x
: Horizontal coordinate.

y
: Vertical coordinate.

RectangleType Struct

Purpose Defines a rectangular portion of a window or of the screen. A RectanglePtr defines a pointer to a RectangleType structure.

Declared In CmnRectTypes.h

Prototype

```
typedef struct RectangleType {
   PointType topLeft;
   PointType extent;
} RectangleType;
typedef RectangleType *RectanglePtr
```

Fields

topLeft
: Coordinates of the upper-left corner of the rectangle relative to the window or screen in which the rectangle resides.

extent
: Width (extent.x) and height (extent.y) of the rectangle.

Comments The Window Manager uses rectangles of this type to define regions of the screen. The new drawing model defines rectangles by

defining left, top, right, and bottom point like the AbsRectType structure.

See Also AbsToRect(), RectToAbs()

Rectangle Functions and Macros

AbsToRect Macro

Purpose Converts an AbsRectType to a RectangleType structure.

Declared In CmnRectTypes.h

Prototype
```
#define AbsToRect (a, r)
```

Parameters

→ *a*
Pointer to an AbsRectType structure.

← *r*
Pointer to a RectangleType structure.

Returns Nothing.

See Also RectToAbs()

RctCopyRectangle Macro

Purpose Copies the source rectangle to the destination rectangle.

Declared In Rect.h

Prototype
```
#define RctCopyRectangle (s, d)
```

Parameters

→ *s*
A pointer to the RectangleType to be copied.

→ *d*
A pointer to the destination RectangleType.

Returns Nothing.

See Also RctSetRectangle()

RctGetIntersection Function

Purpose Determines the intersection of two rectangles.

Declared In `Rect.h`

Prototype

```
void RctGetIntersection (const RectangleType *r1P,
    const RectangleType *r2P, RectangleType *r3P)
```

Parameters

→ *r1P*
A pointer to a source `RectangleType`.

→ *r2P*
A pointer to the other source rectangle.

← *r3P*
Upon return, points to a rectangle representing the intersection of *r1P* and *r2P*.

Returns Nothing.

Comments If the rectangles *r1P* and *r2P* do not intersect, *r3P* contains a rectangle whose top-left coordinate is the maximum of *r1P* and *r2P*'s top-left coordinates and whose extent is 0.

RctInsetRectangle Function

Purpose Moves all of the boundaries of a rectangle by a specified offset.

Declared In `Rect.h`

Prototype

```
void RctInsetRectangle (RectangleType *rP,
    Coord insetAmt)
```

Parameters

↔ *rP*
A pointer to a `RectangleType`.

→ *insetAmt*
Number of coordinates to move the boundaries. This can be a negative number.

A positive *insetAmt* creates a smaller rectangle that is contained inside the old rectangle's boundaries. A negative *insetAmt* creates a larger rectangle that surrounds the old rectangle.

Returns Nothing.

Comments This function adds *insetAmt* to the x and y values of the top-left coordinate and then adjusts the width and the height accordingly so that all of the sides of the rectangle are contracted or expanded by the same amount.

See Also RctOffsetRectangle()

RctOffsetRectangle Macro

Purpose Moves the top and left boundaries of a rectangle by the specified values.

Declared In Rect.h

Prototype

```
#define RctOffsetRectangle (rP, dx, dy)
```

Parameters

↔ *rP*
A pointer to a RectangleType.

→ *dx*
Number of coordinates to move the left boundary. This can be a negative number.

→ *dy*
Number of coordinates to move the top boundary. This can be a negative number.

Returns Nothing.

Comments This macro adds *dx* to the x value of the top-left coordinate of *rP* and *dy* to the y value. The width and height are unchanged. Thus, this function shifts the position of the rectangle by the *dx* and *dy* amounts.

See Also RctInsetRectangle()

RctPtInRectangle Function

Purpose Determines if a point lies within a rectangle's boundaries.

Declared In `Rect.h`

Prototype

```
Boolean RctPtInRectangle (Coord x, Coord y,
   const RectangleType *rP)
```

Parameters

→ *x*
The x coordinate of the point.

→ *y*
The y coordinate of the point.

→ *rP*
The rectangle.

Returns `true` if the point (`x`, `y`) lies within the boundaries of rectangle `r`, `false` otherwise.

RctSetRectangle Macro

Purpose Sets a rectangle's values.

Declared In `Rect.h`

Prototype

```
#define RctSetRectangle (rP, l, t, w, h)
```

Parameters

← *rP*
A pointer to a `RectangleType` to be set.

→ *l*
The x value for the top-left coordinate of the rectangle.

→ *t*
The y value for the top-left coordinate of the rectangle.

→ *w*
The rectangle's width.

→ *h*
The rectangle's height.

Returns Nothing.

See Also `RctCopyRectangle()`

RectToAbs Macro

Purpose Converts a `RectangleType` to an `AbsRectType` structure.

Declared In `CmnRectTypes.h`

Prototype

```
#define RectToAbs (r, a)
```

Parameters

→ *r*
Pointer to a `RectangleType` structure.

→ *a*
Pointer to an `AbsRectType` structure.

Returns Nothing.

See Also `AbsToRect()`

28

Resource Loading Reference

This chapter provides reference material for the resource loading API defined in the header file `UIResources.h`.

Resource Loading Functions and Macros

ResLoadConstant Function

Purpose Loads a constant from a `'tint'` resource and returns its value.

Declared In `UIResources.h`

Prototype

```
uint32_t ResLoadConstant (DmOpenRef database,
    DmResourceID rscID)
```

Parameters

→ *database*
Open database containing the resource.

→ *rscID*
The ID of the `'tint'` resource (symbolically named `constantRscType`) to load.

Returns The four-byte value of the constant in the resource, or 0 if the resource could not be found. The return value may be cast as necessary.

Comments Use this function to load "soft constants"—constant values that are stored as `'tint'` resources. (All open resource databases are searched for the resource ID you specify.) You should store a constant value as a resource when its value changes depending on the locale.

As an example, consider the maximum length of the Alarm Sound trigger label in the Datebook application's preferences panel. The list displayed by this trigger uses the localized name for each sound

stored in the system. Because localized names are used, the maximum length that the Datebook application allows for the label differs depending on the current locale. The maximum length is stored as a resource constant so that each overlay database can specify a different value for the constant.

See Also DmGetResource()

ResLoadConstantV50 Function

Purpose Loads a constant from a `'tint'` resource and return its value.

Declared In `UIResources.h`

Prototype

```
uint32_t ResLoadConstantV50 (DmResourceID rscID)
```

Parameters

→ *rscID*
The ID of the `'tint'` resource (symbolically named `constantRscType`) to load.

Returns The four-byte value of the constant in the resource, or 0 if the resource could not be found. The return value may be cast as necessary.

Compatibility This function is provided for backward compatibility only. Use ResLoadConstant() instead.

ResLoadForm Function

Purpose Copies and initializes a form resource. The structures are complete except pointer updating. Pointers are stored as offsets from the beginning of the form.

Declared In `UIResources.h`

Prototype

```
FormType *ResLoadForm (DmOpenRef database,
    DmResourceID rscID)
```

Parameters

→ *database*
Open database containing the resource.

→ *rscID*
The resource ID of the form.

Returns A pointer to a `FormType` structure representing the form.

ResLoadFormV50 Function

Purpose Copies and initializes a form resource. The structures are complete except pointer updating. Pointers are stored as offsets from the beginning of the form.

Declared In `UIResources.h`

Prototype

```
void *ResLoadFormV50 (DmResourceID rscID)
```

Parameters

→ *rscID*
The resource ID of the form.

Returns The handle of the memory block that the form is in.

Compatibility This function is provided for backward compatibility only. Use `ResLoadForm()` instead.

ResLoadFormWithFlags Function

Purpose Copies and initializes a form resource. The structures are complete except pointer updating. Pointers are stored as offsets from the beginning of the form.

Declared In `UIResources.h`

Prototype

```
FormType *ResLoadFormWithFlags(DmOpenRef database,
    DmResourceID rscID, WinFlagsType windowFlags)
```

Parameters

→ *database*
Open database containing the resource.

→ *rscID*
The resource ID of the form.

→ *windowFlags*
Window attributes to use when creating the dialog. See `WinFlagsType`. The attribute `winFlagModal` is set for you.

Returns A pointer to a `FormType` structure representing the form.

ResLoadMenu Function

Purpose Copies and initializes a menu resource. The structures are complete except pointer updating. Pointers are stored as offsets from the beginning of the menu.

Declared In `UIResources.h`

Prototype

```
MenuBarType *ResLoadMenu (DmOpenRef database,
    DmResourceID rscID)
```

Parameters

→ *database*
Open database containing the resource.

→ *rscID*
The resource ID of the menu.

Returns A pointer to a `MenuBarType` structure containing the menus.

ResLoadMenuV50 Function

Purpose Copies and initializes a menu resource. The structures are complete except pointer updating. Pointers are stored as offsets from the beginning of the menu.

Declared In `UIResources.h`

Prototype

```
void *ResLoadMenuV50 (DmResourceID rscID)
```

Parameters

→ *rscID*
The resource ID of the menu.

Returns The handle of the memory block that the menu is in.

Compatibility This function is provided for backward compatibility only. Use `ResLoadMenu()` instead.

ResLoadString Function

Purpose Loads a constant from a string resource and returns its value.

Declared In `UIResources.h`

Prototype

```
char *ResLoadString (DmOpenRef database,
   DmResourceID rscID, char *strP, size_t maxLen)
```

Parameters

→ *database*
Open database containing the resource.

→ *rscID*
The resource ID of the string.

← *strP*
Upon return, the string parameter you pass in contains the contents of the string resource.

→ *maxLen*
The size in bytes of the *strP* parameter.

Returns The *strP* parameter.

29

Scalable Font Reference

This chapter provides the following information regarding scalable font support:

The header file `GcFont.h` declares the API that this chapter describes. For more information on using fonts, see Chapter 5, "Displaying Text," on page 79. For reference information on bitmapped fonts, see Chapter 13, "Bitmapped Font Reference," on page 205.

Scalable Font Structures and Types

FAbsRectType Struct

Purpose A rectangle describing the dimensions of a character or string of characters within a font.

Declared In `GcFont.h`

Prototype

```
typedef struct FAbsRectType {
   fcoord_t left;
   fcoord_t top;
   fcoord_t right;
   fcoord_t bottom;
} FAbsRectType
```

Fields

`left`
: x coordinate of the left side of the rectangle.

`top`
: y coordinate of the top of the rectangle.

`right`
: x coordinate of the right side of the rectangle.

`bottom`
: y coordinate of the bottom of the rectangle.

Comments The rectangle described by this type includes all coordinates from the top-left coordinate up to but not including the bottom-right coordinates.

See Also `GcGetFontBoundingBox()`, `GcFontStringBounds()`, `AbsRectType`, `RectangleType`

FontFamily Typedef

Purpose A string describing the font family.

Declared In `GcFont.h`

Prototype

```
typedef char FontFamily[kFontFamilyLength + 1]
```

FontHeightType Struct

Purpose Describes the vertical spacing of a font.

Declared In `GcFont.h`

Prototype

```
typedef struct FontHeightType {
   fcoord_t ascent;
   fcoord_t descent;
   fcoord_t leading;
} FontHeightType
```

Fields

`ascent`
: The distance in native coordinates from the top of the font rectangle to its baseline.

`descent`
: The distance in native coordinates from the baseline to the bottom of the font rectangle.

`leading`
The vertical space between lines of text, in native coordinates.

See Also `GcGetFontHeight()`

FontStyle Typedef

Purpose A string describing the font's style.

Declared In `GcFont.h`

Prototype

```
typedef char FontStyle[kFontStyleLength + 1]
```

GcFontHandle Struct

Purpose The handle to a font. You pass this handle to all of the functions described in this chapter.

Declared In `GcFont.h`

Prototype

```
typedef struct GcFontType *GcFontHandle
```

Fields None.

See Also `GcCreateFont()`, `GcCreateFontFromFamily()`, `GcCreateFontFromID()`, `GcReleaseFont()`

GcFontType Struct

Purpose Private structure representing the font.

Declared In `GcFont.h`

Prototype

```
struct GcFontType
```

Fields None.

GcPointType Struct

Purpose Represents a point.

Declared In `GcFont.h`

Prototype

```
typedef struct {
   fcoord_t x;
   fcoord_t y;
} GcPointType
```

Fields

`x`
: The x coordinate.

`y`
: The y coordinate.

See Also `GcInitGradient()`

Scalable Font Constants

GcAliasingTag Enum

Purpose Specifies whether to use antialiasing.

Declared In `GcFont.h`

Constants

`kDisableAntialiasing = 0`
: Disable antialiasing. A pixel is only drawn if its center point lies within the current path.

`kNormalAntialiasing = 1`
: Enable antialiasing. Pixels get a percentage of the color if any portion of the pixel lies within the current path.

See Also `GcSetAntialiasing()`, `GcGetFontAntialiasing()`, `GcSetFontAntialiasing()`

Font Faces Enum

Purpose Common font faces. A font face is a logical description of a style. You may specify a font face instead of trying to find the name of a desired font style.

Declared In `GcFont.h`

Constants

`kItalicFace = 0x0001`
Italics or oblique.

`kBoldFace = 0x0020`
Bold.

`kRegularFace = 0x0040`
Plain.

See Also `GcSetFontFace()`, `GcGetFontFace()`

Font Height Enum

Purpose Invalid font height constant.

Declared In `GcFont.h`

Constants

`kInvalidFontHeight = -1`
Invalid font height.

Font Name String Length Constants

Purpose The lengths of the strings that store the font family and style.

Declared In `GcFont.h`

Constants

`#define kFontFamilyLength 63`
The maximum length of the `FontFamily` string.

`#define kFontStyleLength 63`
The maximum length of the `FontStyle` string.

Scalable Font Functions and Macros

GcApplyFontSpec Function

Purpose Applies a font specification to a particular font.

Declared In `GcFont.h`

Prototype

```
status_t GcApplyFontSpec (GcFontHandle font,
   const char *fontSpec)
```

Parameters

→ *font*
A handle to the font to which to apply the specification.

→ *fontSpec*
The attributes of a font. See the Comments section in `GcCreateFont()` for the syntax of a font specification string.

Returns `errNone` upon success or one of the following values:

`sysErrParamErr`
fontSpec could not be parsed.

`sysErrNoInit`
The font has not been initialized.

`sysErrNameNotFound`
fontSpec has an invalid family name.

Comments You can use this function to modify an existing font. For example, you could use this function to apply the bold style to the current font family.

See Also `GcCreateFont()`, `GcSetFontFace()`, `GcSetFontSize()`, `GcSetFontStyle()`

GcCheckFont Function

Purpose Verifies that the font handle is valid.

Declared In `GcFont.h`

Prototype
```
status_t GcCheckFont (GcFontHandle font)
```

Parameters

→ *font*
A font handle.

Returns `errNone` if the font is valid, or one of the following errors:

`sysErrNoInit`
The font has not been initialized.

`sysErrNameNotFound`
The font cannot be found.

GcCountFontFamilies Function

Purpose Returns the number of font families installed in the system.

Declared In `GcFont.h`

Prototype
```
int32_t GcCountFontFamilies (void)
```

Parameters None.

Returns The number of font families installed in the system.

See Also `GcGetFontFamily()`

GcCountFontStyles Function

Purpose Returns the number of font styles (such as "Italic" or "Bold") available for the specified family.

Declared In `GcFont.h`

Prototype
```
int32_t GcCountFontStyles (const char *family)
```

Parameters

→ *family*
The name of a font family installed in the system.

Returns The number of different styles defined for the font, or 0 if the font family cannot be found.

See Also `GcSetFontStyle()`, `GcGetFontStyle()`

GcCreateFont Function

Purpose Returns a font handle given a fully specified font.

Declared In `GcFont.h`

Prototype

```
GcFontHandle GcCreateFont (const char *fontSpec)
```

Parameters

→ *fontSpec*
A CSS-style font specification. See the "Comments" section.

Returns The handle to the specified font or `NULL` if the font can't be found.

Comments A font specification is a CSS-style text string describing the font characteristics. This string has the following format:

```
[face] [size[/lineHeight]] family [, [size[/
lineHeight]] alternateFamily ]
```

where:

face
The name of a font face such as "regular" or "italic." You can specify one of the following strings:

`100, 200, ... 900`
Specifies a level of darkness where each value is a degree darker than the previous level or at least as dark as the previous level. 400 specifies the normal face, and 700 specifies the bold face.

`normal`
Applies the regular face of the family to the current specification.

`bold`
Applies the bold face to the current specification.

`bolder`
Makes the current font one degree darker.

`lighter`
Makes the current font a degree lighter.

`italic`
Applies an italic face to the current specification.

`oblique`
Applies an italic face to the current specification.

`small-caps`
No change.

In the current implementation, the varying degrees of darkness are not implemented. 100 through 500 specify the regular font face, and 600 through 900 specify the bold font face.

`size`
A numeric value specifying the size, followed by `px` for native coordinates or `db` for standard coordinates.

IMPORTANT: Do not use the standard CSS “pt” units.

The size can also be a relative value from the current size, expressed as a percentage (`50%`) or one of the following:

`xx-small`

`x-small`

`small`

`medium`

`large`

`x-large`

`xx-large`

The size can also be one of the following relative sizes to make a font larger or smaller than it currently is:

`larger`

`smaller`

`lineHeight`
Not currently used.

`family`
Specifies the font family such as “Albany” or “Times.” If the name contains spaces, it must be contained in single or double quotes (' ' or " ").

`alternateFamily`
Specifies the name of an alternative font family to use if the *`family`* cannot be found.

The *family* and *alternateFamily* can also be one of the following system standard fonts:

- "palmos-plain"
- "palmos-bold"
- "palmos-large-plain"
- "palmos-large-bold"

These four fonts are not equivalent to the similarly named system-defined bitmapped fonts unless the device does not have a scalable font engine installed. Instead, these are TrueType fonts.

If you always specify one of the system-defined fonts as at least an alternate family, then `GcCreateFont()` always returns a non-`NULL` handle.

The system does *not* treat the *fontSpec* as an exact list of requirements. In some cases, it may return a close match if, for example, the font won't scale well to the requested size. The application must deal gracefully with the font returned by this function and not assume that all specifications have been met.

When you are finished with the font, use `GcReleaseFont()` to release the memory associated with it.

Example The following code draws text in the Cumberland font if it exists. If not, it uses the system-defined plain font.

```
GcHandle gcH;
GcFontHandle myFontH;
char *string = "Hi, Mom!";

myFontH = GcCreateFont("bold 9db 'Cumberland AMT', palmos-
plain");
...
gcH = GcGetCurrentContext();
if (gcH) {
   GcSetFont(gcH, myFontH);
   GcSetCoordinateSystem(gcH, kCoordinatesNative);
   GcDrawTextAt(gcH, 160, 100, string, strlen(string));
   GcReleaseContext(gcH);
```

```
}
GcReleaseFont(myFontH);
```

See Also GcCreateFontFromFamily(), GcCreateFontFromID(), GcApplyFontSpec()

GcCreateFontFromFamily Function

Purpose Returns a font handle given a font family.

Declared In GcFont.h

Prototype

```
GcFontHandle GcCreateFontFromFamily
    (const char *family)
```

Parameters

→ *family*
The name of a font family.

Returns A handle to the specified font. Returns NULL if it cannot find a font with the specified family or it cannot create a handle for the font.

Comments The font returned by this function uses the default style for the specified family and a size of 0. For example, if you specify "Helvetica" as the font family, the returned font uses the Helvetica plain style. You'll want to use GcSetFontSize() to specify a size. You can also change the style later.

When you are finished with the font, use GcReleaseFont() to release the memory associated with it.

See Also GcGetFontFamily()

GcCreateFontFromID Function

Purpose Create a font handle for a bitmapped font.

Declared In GcFont.h

Prototype

```
GcFontHandle GcCreateFontFromID (FontID id)
```

Parameters

→ *id*
The FontID of a bitmapped font.

Returns A font handle for the specified font. Returns `NULL` if the specified font cannot be found or there was a problem creating the font handle.

Comments You can use this function to manipulate a bitmapped font as you would a scalable font. The returned font uses the size and style specified in the bitmapped font version; however, you can use the functions described in this chapter to resize or transform the font in any way that you want.

You still must use the returned font only with the `GcDrawTextAt()` function.

When you are finished with the font, use `GcReleaseFont()` to release the memory associated with it.

See Also `FntGetFont()`

GcFontStringBounds Function

Purpose Returns the bounds of a string if displayed in the specified font.

Declared In `GcFont.h`

Prototype

```
status_t GcFontStringBounds (GcFontHandle font,
   const char *string, size_t length,
   FAbsRectType *outBounds)
```

Parameters

→ *font*
A handle to a font.

→ *string*
A string.

→ *length*
The length of the string. This can be less than the actual length of the string to determine the bounds of displaying a substring.

← *outBounds*
An `FAbsRectType` rectangle giving the dimensions of the string in native coordinates if it were displayed in the specified font.

Returns Always returns `errNone`.

See Also `GcFontStringWidth()`

GcFontStringBytesInWidth Function

Purpose Returns the size in bytes of the substring that can be displayed in a specified width.

Declared In `GcFont.h`

Prototype

```
size_t GcFontStringBytesInWidth(GcFontHandle font,
   const char *string, size_t maxWidth)
```

Parameters

→ *font*
A handle to a font.

→ *string*
A string.

→ *maxWidth*
The maximum width given in native coordinates in which to display the string.

Returns The number of bytes of full characters from *string* that can be displayed in *maxWidth*.

See Also GcFontStringBytesInWidth()

GcFontStringCharsInWidth Function

Purpose Returns the number of characters that can be displayed in a specified width.

Declared In `GcFont.h`

Prototype

```
size_t GcFontStringCharsInWidth(GcFontHandle font,
   const char *string, size_t maxWidth)
```

Parameters

→ *font*
A handle to a font.

→ *string*
A string.

→ *maxWidth*
The maximum width given in native coordinates in which to display the string.

Returns The number of glyphs that can be displayed in the width. This return value means that the first *n* characters you can obtain from string can be displayed within the specified width.

IMPORTANT: The return value does not equal the number of bytes in *string* that can be displayed. A character within a string may be either a single byte long or multiple bytes long.

See Also GcFontStringBytesInWidth()

GcFontStringEscapement Function

Purpose Returns the dimensions of the string when displayed in the specified font.

Declared In GcFont.h

Prototype

```
void GcFontStringEscapement (GcFontHandle font,
   const char *string, size_t length,
   GcPointType *outEscapement)
```

Parameters

→ *font*
A handle to a font.

→ *string*
A string.

→ *length*
The length of the string. This can be less than the total number of bytes in the string if you want to find out the width of a prefix of string.

← *outEscapement*
The dimensions of the string, both width and height, in native coordinates.

Returns None.

GcFontStringWidth Function

Purpose Gets the width of the specified character string. The **missing character symbol** (an open rectangle) is substituted for any character that does not exist in the current font.

Declared In `GcFont.h`

Prototype

```
fcoord_t GcFontStringWidth (GcFontHandle font,
    const char *string, size_t length)
```

Parameters

→ *font*
A handle to a font.

→ *string*
A string.

→ *length*
The length of the string. This can be less than the total number of bytes in the string if you want to find out the width of a prefix of *string*.

Returns The width in native coordinates required to display the string in its entirety.

See Also GcFontStringBounds()

GcGetFontAntialiasing Function

Purpose Returns whether antialiasing is used when drawing text.

Declared In `GcFont.h`

Prototype

```
uint32_t GcGetFontAntialiasing (GcFontHandle font)
```

Parameters

→ *font*
A handle to a font.

Returns One of the GcAliasingTag constants.

See Also GcSetAntialiasing(), GcSetFontAntialiasing()

GcGetFontBoundingBox Function

Purpose Returns a rectangle specifying an average character's dimensions in the specified font.

Declared In `GcFont.h`

Prototype
```
status_t GcGetFontBoundingBox (GcFontHandle font,
    FAbsRectType *bbox)
```

Parameters

→ *font*
A handle to a font.

← *bbox*
An `FAbsRectType`, which describes the dimensions of a character in the font.

Returns `errNone` upon success or `sysErrParamErr` if *bbox* is `NULL`.

Comments The returned dimensions give the average character width and the tallest possible characters in the font including any ascenders and descenders. Most characters will be shorter than the dimensions returned here, and they may be narrower or wider.

See Also `GcGetFontHeight()`

GcGetFontFace Function

Purpose Returns the face currently used in the font.

Declared In `GcFont.h`

Prototype
```
uint16_t GcGetFontFace (GcFontHandle font)
```

Parameters

→ *font*
A handle to a font.

Returns One of the Font Faces.

See Also `GcSetFontFace()`

GcGetFontFamily Function

Purpose Returns a font family given its index into the system's font family list.

Declared In `GcFont.h`

Prototype

```
status_t GcGetFontFamily (int32_t index,
    FontFamily *family)
```

Parameters

→ *index*
An index into the font family list. Families are indexed from 0 to `GcCountFontFamilies()`–1.

← *family*
The name of the font family at *index*.

Returns `errNone` if the family was found or `sysErrBadIndex` if the family cannot be found.

Comments Use this function in conjunction with `GcCountFontFamilies()` to iterate through the system's list of font families.

See Also `GcCreateFontFromFamily()`

GcGetFontFamilyAndStyle Function

Purpose Returns the family and style of the specified font.

Declared In `GcFont.h`

Prototype

```
status_t GcGetFontFamilyAndStyle
    (GcFontHandle font, FontFamily *family,
    FontStyle *style)
```

Parameters

→ *font*
A handle to a font.

← *family*
The font's family.

← *style*
The font's style.

Returns `errNone` upon success or one of the following:

`sysErrParamErr`
If *`family`* or *`style`* is `NULL`.

See Also `GcGetFontFamily()`, `GcGetFontStyle()`

GcGetFontHeight Function

Purpose Returns the font's height at its current size.

Declared In `GcFont.h`

Prototype
```
status_t GcGetFontHeight (GcFontHandle font,
   struct FontHeightType *height)
```

Parameters

→ *`font`*
A handle to a font.

← *`height`*
A `FontHeightType` structure.

Returns `errNone` upon success, or one of the following errors:

`sysErrNoInit`
The font has not been initialized.

`sysErrNameNotFound`
The font cannot be found.

GcGetFontHinting Function

Purpose Returns whether font hinting is enabled or disabled.

Declared In `GcFont.h`

Prototype
```
Boolean GcGetFontHinting (GcFontHandle font)
```

Parameters

→ *`font`*
A handle to a font.

Returns `true` if font hinting is enabled, meaning that glyphs are drawn so that their lines are always on exact pixel boundaries. `false` if the path of a glyph may land on sub-pixel boundaries.

Comments Characters are always positioned on exact pixel boundaries. Font hinting controls wether the rest of a glyph's path lies on exact pixel

boundaries. If enabled, characters in a string may be drawn wider than when text hinting is off.

See Also GcSetFontHinting()

GcGetFontSize Function

Purpose Returns the size of the font.

Declared In GcFont.h

Prototype
```
fcoord_t GcGetFontSize (GcFontHandle font)
```

Parameters

→ *font*
A handle to a font.

Returns The font size in native coordinates.

See Also GcSetFontSize()

GcGetFontStyle Function

Purpose Returns the style (such as "Regular" or "Bold") stored at a given index within the font family.

Declared In GcFont.h

Prototype
```
status_t GcGetFontStyle (const char *family,
   int32_t index, FontStyle *style)
```

Parameters

→ *family*
The name of a font family installed on the device.

→ *index*
An index into the font family's list of styles. Styles are indexed from 0 to GcCountFontStyles()–1.

← *style*
Upon return, a string containing the style name.

Returns errNone upon success or one of the following:

sysErrBadIndex
The style cannot be found.

sysErrNameNotFound
The specified *family* does not exist.

Comments Use this function in conjunction with `GcCountFontStyles()` to iterate through the list of available styles for a font family.

See Also `GcGetFontFamily()`

GcGetFontTransform Function

Purpose Returns the font's transformation.

Declared In `GcFont.h`

Prototype
```
status_t GcGetFontTransform (GcFontHandle font,
    fcoord_t *out_matrix)
```

Parameters

→ *font*
A handle to a font.

← *out_matrix*
An array of four elements that specify the transformation matrix. See `GcTransform()` for more information about the transformation matrix.

Returns Always returns `errNone`.

Comments This transformation is applied when text is rendered to the screen with the font.

See Also `GcSetFontTransform()`

GcReleaseFont Function

Purpose Frees the memory associated with the font handle.

Declared In `GcFont.h`

Prototype
```
void GcReleaseFont (GcFontHandle font)
```

Parameters

→ *font*
The font handle to free.

Returns Nothing.

GcSetFontAntialiasing Function

Purpose Specifies whether the rendering system uses antialiasing when drawing text with the specified font.

Declared In `GcFont.h`

Prototype
```
status_t GcSetFontAntialiasing (GcFontHandle font,
    uint32_t value)
```

Parameters

→ *font*
A handle to a font.

→ *value*
One of the GcAliasingTag values.

Returns Always returns `errNone`.

Comments You rarely need to use this function. Do so only if you are creating a drawing application and want more control over the quality of the rendered image. The default value for text antialiasing suffices in most cases.

See Also GcSetAntialiasing()

GcSetFontFace Function

Purpose Sets the font to the closest match to the specified flags.

Declared In `GcFont.h`

Prototype
```
status_t GcSetFontFace (GcFontHandle font,
    uint16_t face)
```

Parameters

→ *font*
A handle to a font.

→ *face*
One of the Font Faces.

Returns `errNone` upon success or one of the following:

`sysErrParamErr`
The font handle is invalid.

`sysErrNameNotFound`
The font family for *font* cannot be found.

Comments If the font does not support the specified face, this function returns `errNone`, and the font is set to use the face that is considered the closest possible match. This may be the regular face.

See Also `GcGetFontFace()`

GcSetFontHinting Function

Purpose Enables or disables font hinting. If hinting is enabled, letters are drawn to exact pixel boundaries.

Declared In `GcFont.h`

Prototype

```
status_t GcSetFontHinting (GcFontHandle font,
    Boolean enabled)
```

Parameters

→ *font*
A handle to a font.

→ *enabled*
If `true`, enables hinting for the font so that the glyph's path is aligned to exact pixel boundaries. If `false`, disable font hinting.

Returns Always returns `errNone`.

Comments Characters are always positioned on exact pixel boundaries. Font hinting controls wether the rest of a glyph's path lies on exact pixel boundaries. If enabled, characters in a string may be drawn wider than when text hinting is off.

GcSetFontSize Function

Purpose Sets the font to a specified size.

Declared In `GcFont.h`

Prototype

```
status_t GcSetFontSize (GcFontHandle font,
    fcoord_t size)
```

Parameters

→ *font*
A handle to a font.

→ *size*
A size given in native coordinates.

Returns Always returns `errNone`.

See Also `GcGetFontSize()`

GcSetFontStyle Function

Purpose Sets the font to use the specified style (such as "Regular" or "Bold").

Declared In `GcFont.h`

Prototype

```
status_t GcSetFontStyle (GcFontHandle font,
   const char *style)
```

Parameters

→ *font*
A handle to a font.

→ *style*
A string containing a font style. This style must be valid for the specified font.

Returns `errNone` upon success or `sysErrParamErr` if the style is not supported by the font.

See Also `GcGetFontStyle()`, `GcSetFontFace()`

GcSetFontTransform Function

Purpose Applies a general transformation matrix to the font.

Declared In `GcFont.h`

Prototype

```
status_t GcSetFontTransform (GcFontHandle font,
   const fcoord_t *matrix)
```

Parameters

→ *font*
A handle to the font.

→ *matrix*
An array of four elements. See `GcTransform()` for more information about the transformation matrix.

Returns Always returns `errNone`.

Comments The matrix that is applied is considered to be a 2 X 2 matrix.

This transformation is applied when text is rendered to the screen with the font.

See Also GcGetFontTransform()

30

Screen Orientation Reference

This chapter provides reference material for the screen orientation API:

The header file `SystemMgr.h` declares the API that this chapter describes.

Screen Orientation Constants

Orientation States

Purpose Specify the display orientation.

Declared In `SystemMgr.h`

Constants

`#define sysOrientationUser 0`
Pass this value to `SysSetOrientation()` to tell the system to activate the last user-selected orientation.

`#define sysOrientationPortrait 1`
The display is in portrait orientation.

`#define sysOrientationLandscape 2`
The display is in landscape orientation.

`#define sysOrientationReversePortrait 3`
The display is in reverse portrait orientation (upside-down from the normal portrait orientation).

`#define sysOrientationReverseLandscape 4`
The display is in reverse landscape orientation (upside-down from the normal landscape orientation).

Orientation Trigger States

Purpose Specify the state of the orientation icon in the status bar (the icon that allows the user to change the display orientation). This icon currently does not exist on Palm OS® Cobalt version 6.0.

Declared In `SystemMgr.h`

Constants

`#define sysOrientationTriggerDisabled 0`
The orientation trigger is disabled, meaning that the user is not allowed to change the display orientation.

`#define sysOrientationTriggerEnabled 1`
The orientation trigger is enabled, meaning that the user is allowed to change the display orientation.

Screen Orientation Functions and Macros

SysGetOrientation Function

Purpose Returns the display orientation.

Prototype
```
uint16_t SysGetOrientation (void)
```

Returns One of the constants listed in "Orientation States" on page 611.

Comments Palm OS Cobalt version 6.0 currently only supports `sysOrientationPortrait`.

SysGetOrientationTriggerState Function

Purpose Returns the display orientation trigger state.

Prototype
```
uint16_t SysGetOrientationTriggerState (void)
```

Returns One of the constants listed in "Orientation Trigger States" on page 612.

Comments Palm OS Cobalt version 6.0 currently always returns `sysOrientationTriggerDisabled`.

SysSetOrientation Function

Purpose Sets the display orientation.

Prototype
```
status_t SysSetOrientation (uint16_t orientation)
```

Parameters

→ *orientation*
The orientation to which the display should be set. See "Orientation States" on page 611 for a list of possible values.

Returns Always returns `sysErrNotAllowed`.

SysSetOrientationTriggerState Function

Purpose Sets the display orientation trigger state.

Prototype
```
status_t SysSetOrientationTriggerState
   (uint16_t triggerState)
```

Parameters

→ *triggerState*
One of the constants listed in "Orientation Trigger States" on page 612.

Returns Always returns `sysErrNotAllowed`.

31
Scroll Bar Reference

This chapter provides reference material for the scroll bar API.

The header file `ScrollBar.h` declares the API that this chapter describes. For more information on scroll bars, see "Scroll Bars" on page 68.

Scroll Bar Structures and Types

ScrollBarType Struct

Purpose Internal structure representing a scroll bar. The `ScrollBarPtr` defines a pointer to a `ScrollBarType` structure.

Declared In `ScrollBar.h`

Prototype

```
typedef struct ScrollBarType ScrollBarType;
typedef ScrollBarType *ScrollBarPtr
```

Fields None.

Scroll Bar Events

frmScrollPrvRefreshEvent

Purpose Used to update the display of a scroll bar.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct sclRepeat {
   struct ScrollBarType *pScrollBar;
   uint16_t scrollBarID;
   uint16_t firstTapTime;
   int32_t value;
   int32_t newValue;
   uint32_t time;
} sclRepeat
```

Fields

`pScrollBar`
Pointer to the scroll bar structure.

`scrollBarID`
Resource ID of the scroll bar.

`firstTapTime`
System-ticks count when the first pen down that started this sequence of refresh events occurred.

`value`
Initial position of the scroll bar.

`newValue`
New position of the scroll bar. Given `value` and `newValue`, you can tell how much you have scrolled.

`time`
System-ticks count when the event is added to the queue. Used to determine when the next event should occur.

The following fields in the `EventType` structure are set for this event:

`penDown`
Always `true`.

`screenX`
Draw window-relative position of the pen in standard coordinates (number of coordinates from the left bound of the window).

`screenY`
Draw window-relative position of the pen in coordinates (number of coordinates from the top of the window).

Comments Applications do not respond to this event. It is used by the system to refresh the display.

sclEnterEvent

Purpose Sent when `penDownEvent` occurs within the bounds of a list.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct sclEnter {
   struct ScrollBarType *pScrollBar;
   uint16_t scrollBarID;
   _BA32_PADDING_16(1)
} sclEnter
```

Fields

`pScrollBar`
Pointer to the scroll bar structure.

`scrollBarID`
Resource ID of the scroll bar.

Comments Applications usually don't have to handle this event. Instead, wait for a `sclExitEvent` or `sclRepeatEvent`.

sclExitEvent

Purpose Sent when the user lifts the pen from the scroll bar.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct sclExit {
   struct ScrollBarType *pScrollBar;
   uint16_t scrollBarID;
   _BA32_PADDING_16(1)
   int32_t value;
   int32_t newValue;
} sclExit
```

Fields

`pScrollBar`
Pointer to the scroll bar structure.

`scrollBarID`
Resource ID of the scroll bar.

`value`
: Initial position of the scroll bar

`newValue`
: New position of the scroll bar. Given `value` and `newValue`, you can tell how much you have scrolled.

Comments

Applications that want to implement non-dynamic scrolling should wait for this event, then scroll the item using the values provided in `value` and `newValue`. Return `true` to indicate that you've handled the event.

Note that this event is sent regardless of previous `sclRepeatEvents`. If, however, the application has implemented dynamic scrolling, it doesn't have to respond to this event.

sclRepeatEvent

Purpose

Sent when the pen is continually held within the bounds of a scroll bar.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In

`Event.h`

Prototype

```
struct sclRepeat {
   struct ScrollBarType *pScrollBar;
   uint16_t scrollBarID;
   uint16_t firstTapTime;
   int32_t value;
   int32_t newValue;
   uint32_t time;
} sclRepeat
```

Fields

`pScrollBar`
: Pointer to the scroll bar structure.

`scrollBarID`
: Resource ID of the scroll bar.

`firstTapTime`
: System-ticks count when the first pen down that started this sequence of repeat events occurred.

`value`
Initial position of the scroll bar.

`newValue`
New position of the scroll bar. Given `value` and `newValue`, you can actually tell how much you have scrolled.

`time`
System-ticks count when the event is added to the queue. Used to determine when the next event should occur.

The following fields in the `EventType` structure are set for this event:

`penDown`
Always `true`.

`screenX`
Draw window-relative position of the pen in standard coordinates (number of coordinates from the left bound of the window).

`screenY`
Draw window-relative position of the pen in coordinates (number of coordinates from the top of the window).

Comments Applications that implement dynamic scrolling should watch for this event. In dynamic scrolling, the display is updated as the user drags the scroll bar (not after the user releases the scroll bar). Return `true` to indicate that you've handled the event.

Scroll Bar Functions and Macros

SclDrawScrollBar Function

Purpose Draws a scroll bar.

Declared In `ScrollBar.h`

Prototype
```
void SclDrawScrollBar (const ScrollBarPtr bar)
```

Parameters → *bar*
Pointer to a scroll bar structure.

Returns Nothing.

Comments This function is called internally by `SclSetScrollBar()` and `FrmDrawForm()`. You rarely need to call it yourself.

SclGetScrollBar Function

Purpose Retrieves a scroll bar's current position, its range, and the size of a page.

Declared In `ScrollBar.h`

Prototype

```
void SclGetScrollBar (const ScrollBarPtr bar,
    int32_t *valueP, int32_t *minP, int32_t *maxP,
    int32_t *pageSizeP)
```

Parameters

→ *bar*
Pointer to a scroll bar structure.

← *valueP*
A value representing the scroll car's current position. (The scroll car is the dark region that indicates the position in the document.)

← *minP*
A value representing the top of the user interface element.

← *maxP*
A value representing the bottom of the user interface element.

← *pageSizeP*
Pointer to size of a page (used when page scrolling).

Returns Nothing.

Comments You might use this function immediately before calling `SclSetScrollBar()` to update the scroll bar. `SclGetScrollBar()` returns the scroll bar's current values, which you can then adjust as necessary and pass to `SclSetScrollBar()`.

SclHandleEvent Function

Purpose Handles events that affect a scroll bar.

Declared In `ScrollBar.h`

Prototype
```
Boolean SclHandleEvent (const ScrollBarPtr bar,
    const EventType *event)
```

Parameters

→ *bar*
Pointer to a scroll bar structure.

→ *event*
Pointer to an event (`EventType`).

Returns `true` if the event was handled.

Comment When a `penDownEvent` occurs, the scroll bar sends an `sclEnterEvent`. While the pen is held down, it sends a `sclRepeatEvent`. Applications that implement dynamic scrolling should catch this event and scroll the item each time one arrives.

When a `penUpEvent` occurs and the pen is within the bounds of the scroll bar, it adds a `sclExitEvent`. Applications that implement non-dynamic scrolling should catch `sclExitEvent` and scroll the item when it arrives. Applications that implement dynamic scrolling can ignore this event.

When a `penMoveEvent` occurs and the pen is within the bounds of the car, this function moves the car in the same direction as the pen.

SclSetScrollBar Function

Purpose Sets the scroll bar's current position, its range, and the size of a page. If the scroll bar is visible and its minimum and maximum values are not equal, it's redrawn.

Declared In `ScrollBar.h`

Prototype
```
void SclSetScrollBar (const ScrollBarPtr bar,
    int32_t value, const int32_t min,
    const int32_t max, const int32_t pageSize)
```

Parameters

→ *bar*
Pointer to a scroll bar structure.

→ *value*
The position the scroll car should move to. (The scroll car is the dark region that indicates the position in the document.)

→ *min*
Minimum value.

→ *max*
Maximum value.

→ *pageSize*
Number of lines of the item that can be displayed on a the screen at one time (used when page scrolling).

Returns Nothing.

Comments Call this function when the user adds or deletes text in a field or when a table row is added or deleted.

For scrolling fields, your application should catch the `fldChangedEvent` and update the scroll bar at that time.

The *max* parameter is computed as:

number of lines of text – page size + overlap

where number of lines of text is the total number of lines or rows needed to display the entire UI element, page size is the number of lines or rows that can be displayed on the screen at one time, and overlap is the number of lines or rows from the bottom of one page to be visible at the top of the next page.

For example, if you have 100 lines of text and 10 lines show on a page, the *max* value would be 90 or 91, depending on the overlap. So if *value* is greater than or equal to 90 or 91, the scroll car is at the very bottom of the scrolling region.

For scrolling text fields, you can use the `FldGetScrollValues()` function to retrieve the values that you pass to `SclSetScrollBar()`. For scrolling tables, there is no equivalent to `FldGetScrollValues()`. Your application must scroll the table itself and keep track of the scroll values. See the `ListViewUpdateScrollers()` function in the Memo example application (`MemoMain.c`) for an example of setting scroll bar values for a table.

This function may display a fatal error message if the *min* parameter is greater than the *max* parameter.

Example The following code shows how to use the function FldGetScrollValues() to compute the values you pass for *value*, *min*, and *max*:

```
FldGetScrollValues (fld, &scrollPos, &textHeight,
   &fieldHeight);

if (textHeight > fieldHeight)
  maxValue = textHeight - fieldHeight;
else if (scrollPos)
  maxValue = scrollPos;
else
  maxValue = 0;

SclSetScrollBar (bar, scrollPos, 0, maxValue,
   fieldHeight-1);
```

In this case, `textHeight` is the number of lines of text and `fieldHeight` is the page size. No lines overlap when you scroll one page. Notice that if the page size is greater than the lines of text, then *max* equals *min*, which means that the scroll bar is not displayed.

See Also SclGetScrollBar()

32

Standard UI Dialogs Reference

This chapter describes the UI controls API as declared in `UIControls.h`.

UI Dialog Constants

UIPickColorStartType Typedef

Purpose Controls how the dialog displayed by `UIPickColor()` presents the available colors to the user.

Declared In `UIControls.h`

Prototype

```
typedef uint16_t UIPickColorStartType
```

Constants

`#define UIPickColorStartPalette 0`
Displays a series of color squares

`#define UIPickColorStartRGB 1`
Displays individual sliders for the red, green, and blue values.

UI Dialog Functions and Macros

UIBrightnessAdjust Function

Purpose Displays the brightness adjust slip window.

Declared In `UIControls.h`

Prototype
```
void UIBrightnessAdjust (void)
```

Parameters None.

Returns Nothing.

Comments On hardware that supports a brightness setting, this function displays a slip window that allows the user to change the brightness level. On hardware that has a backlight, this function toggles the backlight.

NOTE: If the screen icon is not present on the status bar, this function has no effect.

UIContrastAdjust Function

Purpose Displays the contrast adjust slip window.

Declared In `UIControls.h`

Prototype
```
void UIContrastAdjust (void)
```

Parameters None.

Returns Nothing.

NOTE: If the screen icon is not present on the status bar, this function has no effect.

UIPickColor Function

Purpose Displays a dialog that allows the user to choose a color.

Declared In `UIControls.h`

Prototype

```
Boolean UIPickColor (IndexedColorType *indexP,
   RGBColorType *rgbP, UIPickColorStartType start,
   const char *titleP, const char *tipP)
```

Parameters

↔ *indexP*
: Index value of the selected color. (See `IndexedColorType`.) Upon entry, this points to the index value of the color initially selected. Upon return, this points to the index value of the color the user selected. Pass `NULL` to not set or return this value.

↔ *rgbP*
: RGB value of the selected color. (See `RGBColorType`.) Upon entry, this points to the RGB value of the color initially selected when the dialog is displayed. Upon return, this points to the RGB value that the user selected. Pass `NULL` to not set or return this value.

→ *start*
: One of the `UIPickColorStartType` constants. This parameter is only used if both `indexP` and `rgbP` are not `NULL`.

→ *titleP*
: String to display as the title of the dialog. Specify `NULL` to use the default title, which is "Pick Color" in the US English locale.

→ *tipP*
: Not used.

Returns `true` if a new color was selected, `false` otherwise.

Comments Use this function to allow users to choose a color used in your user interface. (The system never calls `UIPickColor()`.)

This function can display two versions of the dialog: palette or RGB. The palette version of the dialog displays a series of squares, each containing a different color. The *indexP* value contains the index of the square that is initially selected.

The RGB version of the dialog displays three sliders that allow the user to select the level of red, green, and blue in the color. The *rgbP* parameter contains the red, green, and blue values initially shown in the dialog. The sliders only allow values that are defined in the current system color table.

If *indexP* is initially NULL, only the RGB dialog is displayed. Similarly, if *rgbP* is NULL, only the palette version is displayed. If both parameters are non-NULL, the system adds a pull-down list that allows the user to switch between the palette dialog and the RGB dialog, and the *start* parameter controls which version of the dialog is initially shown. In this case, both *indexP* and *rgbP* contain the value of the user-selected color upon return.

See Also WinSetBackColor(), WinSetForeColor(), WinSetTextColor(), UIColorSetTableEntry()

33

Status Bar Reference

This chapter provides reference material for the status bar API:

The header file `StatusBar.h` declares the API that this chapter describes.

Status Bar Constants

Error Code Constants

Purpose Errors returned by the status bar functions.

Declared In `StatusBar.h`

Constants

`#define statErrInputWindowOpen (statErrorClass | 3)`
An attempt was made to hide the status bar, but the dynamic input area is open. The status bar can only be hidden if the input area is closed.

`#define statErrInvalidLocation (statErrorClass | 1)`
Not currently used.

`#define statErrInvalidName (statErrorClass | 2)`
Not currently used.

StatAttrType Enum

Purpose Status bar attribute values that you pass as a parameter to `StatGetAttribute()`.

Declared In `StatusBar.h`

Constants

`statAttrBarVisible`
`StatGetAttribute()` returns a Boolean value of `true` to indicate the status bar is currently visible or `false` to indicate the status bar is currently hidden.

`statAttrDimension`
`StatGetAttribute()` returns a pointer to a `RectangleType` structure that gives the dimensions in active coordinates of the status bar.

Status Bar Functions and Macros

StatGetAttribute Function

Purpose Retrieves the value of a status bar attribute.

Declared In `StatusBar.h`

Prototype

```
status_t StatGetAttribute (StatAttrType selector,
   uint32_t *dataP)
```

Parameters

→ *selector*
One of the `StatAttrType` constants.

← *dataP*
The value of the specified status bar attribute.

Returns `errNone` upon success, or one of the following:

`sysErrParamErr`
The *dataP* parameter is `NULL` or the *selector* is an unknown value.

StatHide Function

Purpose Hides the status bar.

Declared In `StatusBar.h`

Prototype

```
status_t StatHide (void)
```

Parameters None.

Returns `errNone` upon success or one of the following:

`statErrInputWindowOpen`
: The device contains a dynamic input area and the dynamic input area is open. The status bar can only be hidden if the dynamic input area is closed.

Comments Applications should only hide the status bar if there is a very good reason to make the application entirely full screen. For example, a game application might need as much of the screen as possible and may choose to hide the status bar as well as the input area.

If you do hide the status bar, you must provide a way for the user to exit your application, either through your user interface or by using a hard key. You must also show the status bar again upon application exit.

Note that devices that do not have a dynamic input area in general show the status bar only when the Launcher application is active.

StatShow Function

Purpose Displays the status bar or ensures that the status bar is being displayed.

Declared In `StatusBar.h`

Prototype

```
status_t StatShow (void)
```

Parameters None.

Returns Always returns `errNone`.

Comments You might use this function to start displaying the status bar again after a previous call to `StatHide()`.

TIP: If you want the status bar to display on devices without a dynamic input area while your application runs, set the minimum size constraint for each form to be less than 160 coordinates high.

34

Table Reference

This chapter describes the table API as declared in the header file `Table.h`. It discusses the following topics:

For more information on tables, see "Tables" on page 109.

Table Structures and Types

TableType Struct

Purpose Internal structure that represents a table. A `TablePtr` defines a pointer to a `TableType` structure.

Declared In `Table.h`

Prototype

```
typedef struct TableType TableType;
typedef TableType *TablePtr
```

Fields None.

Table Constants

Miscellaneous Table Constants

Purpose Constants defined in `Table.h`.

Declared In `Table.h`

Constants

`#define tableDefaultColumnSpacing 1`
Never used.

`#define tableMaxTextItemSize 255`
The maximum length of an editable text field within a table cell.

`#define tableNoteIndicatorHeight 11`
The height in standard coordinates of the note indicator for tables items of type `textWithNoteTableItem`.

`#define tableNoteIndicatorWidth 7`
The width in standard coordinates of the note indicator for tables items of type `textWithNoteTableItem`.

`#define tblUnusableRow 0xffff`
Value returned by `TblGetLastUsableRow()` if none of the table's rows are usable.

TableItemStyleType Typedef

Purpose The type of the item, such as a control, a text label, and so on within a table cell.

Declared In `Table.h`

Prototype

```
typedef Enum8 TableItemStyleType
```

Constants

`checkboxTableItem`
A check box control.

The table stores an integer value for this type of table cell. Retrieve its value with `TblGetItemInt()`. The pointer value is not defined.

`customTableItem`
: A cell drawn using the callback function `TableDrawItemFuncType()`.

  If you want, you can store data for custom cells within the table by using `TblSetItemInt()` or `TblSetItemPtr()`.

`dateTableItem`
: Non-editable date in the form *month*/*day*, or a dash if the date value is –1. The date is followed by an exclamation point if it has passed.

  The table stores an integer value for this type of table cell. Retrieve its value with `TblGetItemInt()`. The returned value should be cast to `DateType`. `DateType` is currently defined as a 16-bit number:

  ```
  dddddmmmmyyyyyyy
  ```

  The first 5 bits are the day, the next 4 bits are the month, and the next 7 bits are the year given as the offset since 1904.

  Dates are always drawn in the current font.

`labelTableItem`
: Non-editable text displayed in the system's default font with a terminating colon character (":"). Use a `labelNoColonTableItem` if you don't want a colon.

  The table stores a string pointer value for this type of table cell. Retrieve its value with `TblGetItemPtr()`. The integer value is not defined.

`numericTableItem`
: Non-editable integer. Integers are displayed in the system's default bold font.

  The table stores an integer value for this type of table cell. Retrieve its value with `TblGetItemInt()`. The pointer value is not defined.

`popupTriggerTableItem`
: A pop-up list displayed in the system's default font with a trailing colon character. Use `popupTriggerNoColonTableItem` if you don't want the trailing colon.

  For this type of table cell, both an integer and a pointer are defined. `TblGetItemInt()` returns the index of the list item

that should be displayed. TblGetItemPtr() returns a pointer to the list (ListType *).

`textTableItem`
Editable text field. For this item type, implement the callback function TableLoadDataFuncType() to load text into the table cell and implement the callback TableSaveDataFuncType() to save data before the field is freed.

You can set the font used in the text field with TblSetItemFont(). TblGetItemPtr() returns a pointer to the field (FieldType *). The integer value is not defined.

`textWithNoteTableItem`
Editable text field and a note icon to the right of the text. For this item type, implement the callback function TableLoadDataFuncType() to load text into the table cell and implement the callback TableSaveDataFuncType() to save data before the field is freed.

You can set the font used in the text field with TblSetItemFont(). TblGetItemPtr() returns a pointer to the field (FieldType *). The integer value is not defined.

`timeTableItem`
Not implemented.

`narrowTextTableItem`
Editable text with space reserved on the right side of the cell. For this item type, implement the callback function TableDrawItemFuncType() to draw in the space reserved on the right side of the cell, the TableLoadDataFuncType() callback function to load text into the table cell, and the callback function TableSaveDataFuncType() to save data before the field is freed.

You can set the font used in the text field with TblSetItemFont(). TblGetItemPtr() returns a pointer to the field (FieldType *). The integer value set by TblSetItemInt() is the number of standard coordinates to reserve on the right side of the cell.

`tallCustomTableItem`
A cell drawn using the callback function TableDrawItemFuncType(). The height of the item is

equal to the height of the row (set by TblSetRowHeight()) in which the item is located. In Palm OS Cobalt, this style is identical to customTableItem.

If you want, you can store data for custom cells within the table by using TblSetItemInt() or TblSetItemPtr().

labelNoColonTableItem
: Non-editable text displayed in the system's default font without the terminating colon character.

  The table stores a string pointer value for this type of table cell. Retrieve its value with TblGetItemPtr(). The integer value is not defined.

popupTriggerNoColonTableItem
: A pop-up list displayed in the system's default font without a trailing colon.

  For this type of table cell, both an integer and a pointer are defined. TblGetItemInt() returns the index of the list item that should be displayed. TblGetItemPtr() returns a pointer to the list (ListType *).

Comments The private TableType structure has fields in which to store an integer, a pointer, and a bitmapped font ID for each table item. In many cases, only one of the three is defined. Which values are defined determines how you retrieve the value of the table cell. See the descriptions above to determine how to work with each item type.

See Also TblSetItemStyle(), TblGetItemStyle()

Table Events

tblEnterEvent

Purpose Sent when a penDownEvent occurs within the bounds of an active item in a table.

For this event, the EventType data field contains the structure shown in the Prototype section, below.

Declared In Event.h

Prototype

```
struct tblEnter {
   struct TableType *pTable;
   uint16_t tableID;
   int16_t row;
   int16_t column;
   _BA32_PADDING_16(1)
} tblEnter
```

Fields

`pTable`
Pointer to a table structure.

`tableID`
Resource ID of the table.

`row`
Row of the item.

`column`
Column of the item.

tblExitEvent

Purpose Sent when the stylus had previously entered the bounds of the table but is lifted outside of the bounds of the table.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In Event.h

Prototype

```
struct tblExit {
   struct TableType *pTable;
   uint16_t tableID;
   int16_t row;
   int16_t column;
   _BA32_PADDING_16(1)
} tblExit
```

Fields

`pTable`
Pointer to a table structure.

`tableID`
Resource ID of the table.

`row`
Row of the item.

`column`
Column of the item.

tblSelectEvent

Purpose Sent when the stylus had previously entered the bounds of the table and is now lifted within the bounds of the table. The pen must be lifted within the bounds of the same table item on which the original `penDownEvent` occurred.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct tblSelect {
   struct TableType *pTable;
   uint16_t tableID;
   int16_t row;
   int16_t column;
   _BA32_PADDING_16(1)
} tblSelect
```

Fields

`pTable`
Pointer to a table structure.

`tableID`
Resource ID of the table.

row
Row of the item.

column
Column of the item.

See Also `TblGetSelection()`

Table Functions and Macros

TblColumnUsable Function

Purpose Returns whether a specified column is usable. Unusable columns do not display.

Declared In `Table.h`

Prototype
```
Boolean TblColumnUsable (const TableType *tableP,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

Returns `true` if the column is usable or `false` otherwise.

See Also `TblSetColumnUsable()`

TblDrawTable Function

Purpose Draws a table.

Declared In `Table.h`

Prototype
```
void TblDrawTable (TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns Nothing.

Comments This function is called as part of `FrmDrawForm()` when the form contains a table.

This function draws into a graphics context. Therefore, you must call it from within a `frmUpdateEvent` if the form that contains the table uses an update-based window.

This function draws the entire table, marking all rows valid before drawing. See the `TableItemStyleType` description for more information about how each type of table cell is drawn.

When drawing cells with editable text fields (`textTableItem`, `textWithNoteTableItem`, or `narrowTextTableItem`), this function uses the `TableLoadDataFuncType()` callback function to load the text into the table cells. The text field does not retain the text handle that your `TableLoadDataFuncType()` returns, meaning that you are responsible for freeing the memory that you load into the table.

When drawing `narrowTextTableItem` cells, `customTableItem` cells or `tallCustomTableItem` cells, this function uses the `TableDrawItemFuncType()` callback function to draw the extra pixels after the text or to draw the entire cell.

On color systems, tables are always drawn using the same color as is used for the field background color.

When the user has set the security setting to mask private records, table cells that contain private database records are drawn as shaded cells to obscure the information they contain. You must explicitly tell the table which cells are masked by making the appropriate calls to `TblSetRowMasked()` and `TblSetColumnMasked()`.

See Also `TblEraseTable()`, `TblRedrawTable()`, `TblSetCustomDrawProcedure()`, `TblInvalidate()`

TblEditing Function

Purpose Checks whether a table is in edit mode.

Declared In `Table.h`

Prototype

```
Boolean TblEditing (const TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns `true` if the table is in edit mode, `false` otherwise.

Comments The table is in edit mode while the user edits a text item. More specifically, the table is in edit mode when a `tblEnterEvent` is received on an editable table cell (`textTableItem`, `textWithNoteTableItem`, or `narrowTextTableItem`), or when `TblGrabFocus()` is called.

The table is taken out of edit mode when the user places the pen on a note in a `textWithNoteTableItem` or when the table releases the focus (`TblReleaseFocus()`).

TblEraseTable Function

Purpose Erases a table.

Declared In `Table.h`

Prototype
```
void TblEraseTable (TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns Nothing.

Comments This function sets the table's visible and selected attributes to `false`. It does not invalidate table rows.

See Also `TblDrawTable()`, `TblSetCustomDrawProcedure()`, `TblRedrawTable()`

TblFindRowData Function

Purpose Returns the number of the row that contains the specified data value.

Declared In `Table.h`

Prototype
```
Boolean TblFindRowData (const TableType *tableP,
    uint32_t data, int16_t *rowP)
```

Parameters

→ *tableP*
Pointer to a table.

→ *data*
Row data to find.

← *rowP*
Pointer to the row number (return value).

Returns `true` if a match was found, `false` otherwise.

Comments This function searches for a row whose attributes contain the specified data. The data is any application-specific data that you have set with `TblSetRowData()`.

See Also `TblGetRowData()`, `TblFindRowID()`

TblFindRowID Function

Purpose Returns the number of the row with the specified ID.

Declared In `Table.h`

Prototype

```
Boolean TblFindRowID (const TableType *tableP,
   uint16_t id, int16_t *rowP)
```

Parameters

→ *tableP*
Pointer to a table.

→ *id*
Row ID to find.

← *rowP*
Pointer to the row number (return value).

Returns `true` if a match was found, `false` otherwise.

See Also `TblSetRowID()`, `TblFindRowData()`

TblGetBounds Function

Purpose Returns the bounds of a table.

Declared In `Table.h`

Prototype

```
void TblGetBounds (const TableType *tableP,
   RectangleType *rP)
```

Parameters

→ *tableP*
Pointer to a table.

← *rP*
A `RectangleType` structure in which the bounds are returned.

Returns Nothing.

See Also `TblGetItemBounds()`

TblGetColumnMasked Function

Purpose Determines whether a particular table column is masked.

Declared In `Table.h`

Prototype
```
Boolean TblGetColumnMasked
    (const TableType *tableP, int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

Returns `true` if the column is masked, `false` otherwise.

Comments If a table cell's column is masked and the cell's row is also masked, the table cell is drawn on the screen but is shaded to obscure the information that it contains.

See Also `TblSetColumnMasked()`

TblGetColumnSpacing Function

Purpose Returns the spacing after the specified column.

Declared In `Table.h`

Prototype
```
Coord TblGetColumnSpacing(const TableType *tableP,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

Returns The spacing after the column (in standard coordinates).

This function may display a fatal error message if the *column* parameter is invalid.

See Also TblGetColumnWidth(), TblSetColumnSpacing(), TblSetColumnUsable()

TblGetColumnWidth Function

Purpose Returns the width of the specified column.

Declared In `Table.h`

Prototype
```
Coord TblGetColumnWidth (const TableType *tableP,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

Returns The width of a column (in standard coordinates). This function may display a fatal error message if the *column* parameter is invalid.

See Also TblGetColumnSpacing(), TblSetColumnWidth(), TblSetColumnUsable()

TblGetCurrentField Function

Purpose Returns a pointer to the `FieldType` in which the user is currently editing a text item.

Declared In `Table.h`

Prototype
```
FieldPtr TblGetCurrentField
    (const TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns A pointer to the currently selected field, or `NULL` if the table is not in edit mode.

See Also TblGetSelection()

TblGetItemBounds Function

Purpose Returns the bounds of an item in a table.

Declared In `Table.h`

Prototype
```
void TblGetItemBounds (const TableType *tableP,
    int16_t row, int16_t column, RectangleType *rP)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

← *rP*
Pointer to a `RectangleType` structure that holds the bounds of the item.

Returns Nothing.

Comments This function may raise a fatal exception if the *row* or *column* parameter specifies a row or column that does not appear on screen.

If the item is a `customTableItem`, the returned height is equal to the height of the font used to draw the item. If the item is a `tallCustomTableItem`, the return height is the full height of the table row.

TblGetItemFont Function

Purpose Returns the font used to display a table item.

Declared In `Table.h`

Prototype
```
FontID TblGetItemFont (const TableType *tableP,
    int16_t row, int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

Returns The ID of the font used for the table item at the row and column indicated.

Comments This function returns the font ID stored in the table structure for this item. Only certain types of table items use the font specified by the table structure when they are displayed. The `TableItemStyleType` description specifies what font is used to display each type of table item.

This function may display a fatal error message if the *`row`* or *`column`* parameter specifies a row or column that is not on the screen.

See Also `TblSetItemFont()`

TblGetItemInt Function

Purpose Returns the integer value stored in a table item.

Declared In `Table.h`

Prototype

```
int16_t TblGetItemInt (const TableType *tableP,
    int16_t row, int16_t column)
```

Parameters

→ *`tableP`*
Pointer to a table.

→ *`row`*
Row number of the item (zero-based).

→ *`column`*
Column number of the item (zero-based).

Returns The integer value.

Comments This function returns the integer value stored internally for this table item. Certain types of table items display the integer value, and other types display the value pointed to by the internal pointer value. See the `TableItemStyleType` description for details. If the integer value was never set for this table item, this function returns 0.

This function may display a fatal message if the *`row`* or *`column`* does not appear on the screen.

See Also `TblSetItemInt()`, `TblGetItemPtr()`

TblGetItemPtr Function

Purpose Returns the pointer value stored in a table item.

Declared In `Table.h`

Prototype
```
void *TblGetItemPtr (const TableType *tableP,
    int16_t row, int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

Returns The item's pointer value or `NULL` if the item does not have a pointer value.

Comments This function returns the pointer value stored internally for this table item. Certain types of table items display the value pointed to by this pointer, and other types display an integer value stored in an integer field. See the `TableItemStyleType` description for details. An application may have set the value of the pointer anyway, even for items that use the integer value. This function always returns that pointer value.

This function may display a fatal message if the *row* or *column* parameter is invalid.

See Also `TblSetItemPtr()`

TblGetItemStyle Function

Purpose Returns a table item's style.

Declared In `Table.h`

Prototype
```
TableItemStyleType TblGetItemStyle
    (const TableType *tableP, int16_t row,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

Returns A `TableItemStyleType` constant that specifies the item's style.

TblGetLastUsableRow Function

Purpose Returns the last row in a table that is usable (visible).

Declared In `Table.h`

Prototype
```
int16_t TblGetLastUsableRow
    (const TableType *tableP)
```

Parameters → *tableP*
Pointer to a table.

Returns The row index (zero-based) or `tblUnusableRow` if there are no usable rows.

See Also `TblGetRowData()`, `TblGetRowID()`

TblGetNumberOfColumns Function

Purpose Returns the number of columns in a table.

Declared In `Table.h`

Prototype
```
int16_t TblGetNumberOfColumns
    (const TableType *tableP)
```

Parameters → *tableP*
Pointer to a table.

Returns The number of columns in a table.

See Also `TblGetTopRow()`, `TblSetSelection()`

TblGetNumberOfRows Function

Purpose Returns the number of rows in a table.

Declared In `Table.h`

Prototype
```
int16_t TblGetNumberOfRows
    (const TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns The maximum number of visible rows in the specified table.

Comments Note that even though you can add and remove rows to and from a table, the value returned by this function does not change. The value returned by this function indicates the maximum number of rows that can be displayed on the screen at one time. It is set when you create the table resource.

TblGetRowData Function

Purpose Returns the data value of the specified row.

Declared In `Table.h`

Prototype
```
uint32_t TblGetRowData (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Number of the row (zero-based).

Returns The application-specific data stored for this row, if any. Returns 0 if there is no application-specific data value.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblFindRowData(), TblSetRowData()

TblGetRowHeight Function

Purpose Returns the height of the specified row.

Declared In `Table.h`

Prototype
```
Coord TblGetRowHeight (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Number of the row (zero-based).

Returns The height in standard coordinates.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblGetItemBounds(), TblSetRowHeight()

TblGetRowID Function

Purpose Returns the ID value of the specified row.

Declared In `Table.h`

Prototype
```
uint16_t TblGetRowID (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Number of the row (zero-based).

Returns The ID value of the row in the table.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblGetRowData(), TblSetRowID(), TblFindRowID()

TblGetSelection Function

Purpose Returns the row and column of the currently selected table item.

Declared In `Table.h`

Prototype
```
Boolean TblGetSelection (const TableType *tableP,
    int16_t *rowP, int16_t *columnP)
```

Parameters

→ *tableP*
Pointer to a table.

← *rowP, columnP*
The row and column indexes (zero-based) of the currently selected item.

Returns `true` if the item is selected, `false` if not.

See Also `TblSetRowSelectable()`, `TblSetSelection()`

TblGetTopRow Function

Purpose Returns the top visible row of a table.

Declared In `Table.h`

Prototype
```
int16_t TblGetTopRow (const TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns The top visible row in a table.

See Also `TblGetNumberOfColumns()`

TblGrabFocus Function

Purpose Puts a table into edit mode.

Declared In `Table.h`

Prototype
```
void TblGrabFocus (TableType *tableP, int16_t row,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Current row to be edited (zero-based).

→ *column*
Current column to be edited (zero-based).

Returns Nothing.

Comments This function may display a fatal error message if the table already has the focus or if the *row* or *column* parameter is invalid.

This function puts the table into edit mode and sets the current item to the one at the row and column passed in. An editable field must exist in the coordinates passed to this function.

You must call `FrmSetFocus()` before calling this function. `FrmSetFocus()` releases the focus from the UI element that previously had it and sets the form's internal structures.

Example The following function from the Address Book application sets the focus in an editable field within a table:

```
static void EditViewRestoreEditState () {
  uint16_t row;
  FormType *frm;
  TableType *table;
  FieldType *fld;

  if (CurrentFieldIndex == noFieldIndex)
    return;

  // Find the row that the current field is in.
  table = GetObjectPtr (EditTable);
  if ( ! TblFindRowID (table, CurrentFieldIndex, &row) )
    return;

  frm = FrmGetActiveForm ();
  FrmSetFocus (frm, FrmGetObjectIndex (frm, EditTable));
  TblGrabFocus (table, row, editDataColumn);

  // Restore the insertion point position.
  fld = TblGetCurrentField (table);
  FldSetInsPtPosition (fld, EditFieldPosition);
}
```

See Also `TblReleaseFocus()`

TblHandleEvent Function

Purpose Handles an event for the table.

Declared In `Table.h`

Prototype

```
Boolean TblHandleEvent (TableType *tableP,
    EventType *event)
```

Parameters

→ *tableP*
Pointer to a table.

→ *event*
The event to be handled.

Returns `true` if the event was handled, `false` if the event was not handled or if the table is not editable.

Comments If the table is editable, this function passes along any `keyDownEvent`, `fldEnterEvent`, `menuCmdBarOpenEvent`, FEP events, or insertion point events to the currently selected field.

When a `fldHeightChangedEvent` occurs, this function changes the height of the specified field as indicated by the event. If the field being resized is going to scroll off the bottom of the screen, then instead the table scrolls the rows above it up off the top. Otherwise, the table is scrolled downward and rows below the current row are scrolled off the bottom as necessary.

Note that the `fldHeightChangedEvent` is only handled for dynamically sized fields. See the descriptions of "Multi-line Text Fields" on page 85 for more information on dynamically sized fields.

When a `penDownEvent` occurs, the table checks to see if the focus is being changed. If it is and the user was previously editing a text field within the table, it saves the data in the table cell using the `TableSaveDataFuncType()` callback function, then it enqueues a `tblEnterEvent` with the new row and column that are selected.

When a `tblEnterEvent` occurs, this function tracks the pen until it is lifted. If the pen is lifted within the bounds of the same item it went down in, a `tblSelectEvent` is added to the event queue; if not, a `tblExitEvent` is added to the event queue.

TblHasScrollBar Function

Purpose Sets whether the table has a scroll bar associated with it.

Declared In `Table.h`

Prototype

```
void TblHasScrollBar (TableType *tableP,
    Boolean hasScrollBar)
```

Parameters

→ *tableP*
Pointer to a table.

→ *hasScrollBar*
`true` to set the attribute, `false` to unset it.

Returns Nothing.

Comments Your application must scroll the table itself and keep track of the scroll values. See the `ListViewUpdateScrollers()` function in the Memo example application (`MemoMain.c`) for an example of setting scroll bar values for a table.

TblInsertRow Function

Purpose Inserts a row into the table before the specified row.

Declared In `Table.h`

Prototype

```
void TblInsertRow (TableType *tableP, int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row to insert (zero-based).

Returns Nothing.

Comments The number of rows in a table is the maximum number of rows displayed on the screen. Unlike a multi-line text field, there is no notion of a table that is bigger than the available screen. For this reason, this function does not increase the number of table rows.

Instead of keeping track of a total number of rows in the table and a number of rows displayed on the screen, tables mark any row that isn't currently displayed as unusable.

The newly inserted row is marked as invalid, unusable, and not masked. If you want to display the newly inserted row, call `TblSetRowUsable()` after making sure that the row displays a value and then call `TblInvalidate()` to trigger a `frmUpdateEvent` that redraws the table.

See Also `TblRemoveRow()`, `TblSetRowUsable()`, `TblSetRowSelectable()`

TblInvalidate Function

Purpose Invalidates the table's region and redraws the table if possible.

Declared In `Table.h`

Prototype

```
void TblInvalidate (TableType *tblP)
```

Parameters

→ *tblP*
Pointer to a table.

Returns Nothing.

Comments This function invalidates the region of the form that contains the table so that a `frmUpdateEvent` is generated. If the form's window is a transitional window, the table is redrawn immediately using a private callback function. If the form is an update-based window, the callback function may be called if this is the only pending drawing update. Otherwise, your application receives and should handle the `frmUpdateEvent`.

See Also `WinInvalidateRect()`, `WinInvalidateWindow()`, `TblDrawTable()`, `TblRedrawTable()`

TblMarkRowInvalid Function

Purpose Marks the row invalid.

Declared In `Table.h`

Prototype

```
void TblMarkRowInvalid (TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

Returns Nothing.

Comments Invalidating a table row does not trigger a `frmUpdateEvent`. If you want to trigger a `frmUpdateEvent`, you must call `TblInvalidate()` after calling this function.

This function may display a fatal error message if the *row* parameter is invalid.

See Also `TblRemoveRow()`, `TblSetRowUsable()`, `TblSetRowSelectable()`, `TblMarkTableInvalid()`, `TblRowInvalid()`

TblMarkTableInvalid Function

Purpose Marks all the rows in a table invalid.

Declared In `Table.h`

Prototype
```
void TblMarkTableInvalid (TableType *tableP)
```

Parameters → *tableP*
Pointer to a table.

Returns Nothing.

Comments Invalidating table rows does not trigger a `frmUpdateEvent`. If you want to trigger a `frmUpdateEvent`, you must call `TblInvalidate()` after calling this function.

See Also `TblEraseTable()`, `TblRedrawTable()`

TblRedrawTable Function

Purpose Redraws the rows of the table that are marked invalid.

Declared In `Table.h`

Prototype
```
void TblRedrawTable (TableType *tableP)
```

Parameters → *tableP*
Pointer to a table.

Returns Nothing.

Comments This function draws into a graphics context. Therefore, you must call it from within a `frmUpdateEvent` if the form that contains the field uses an update-based window.

This function draws the invalid rows in the table. See the `TableItemStyleType` description for more information about how each type of table cell is drawn.

When drawing cells with editable text fields (`textTableItem`, `textWithNoteTableItem`, or `narrowTextTableItem`), this function uses the `TableLoadDataFuncType()` callback function to load the text into the table cells. The text field does not retain the text handle that your `TableLoadDataFuncType()` returns, meaning that you are responsible for freeing the memory that you load into the table.

When drawing `narrowTextTableItem` cells, `customTableItem` cells, or `tallCustomTableItem` cells, this function uses the `TableDrawItemFuncType()` callback function to draw the extra pixels after the text or to draw the entire cell.

On color systems, tables are always drawn using the same color as is used for the field background color.

When the user has set the security setting to mask private records, table cells that contain private database records are drawn as shaded cells to obscure the information they contain. You must explicitly tell the table which cells are masked by making the appropriate calls to `TblSetRowMasked()` and `TblSetColumnMasked()`.

See Also `TblMarkTableInvalid()`, `TblMarkRowInvalid()`, `TblDrawTable()`, `TblInvalidate()`

TblReleaseFocus Function

Purpose Releases the focus.

Declared In `Table.h`

Prototype

```
void TblReleaseFocus (TableType *tableP)
```

Parameters → *tableP*
Pointer to a table.

Returns Nothing.

Comments Applications typically do not call this function. Instead, they call `FrmSetFocus()` with an element index of `noFocus` to notify the form that the table has lost focus. The form code calls `TblReleaseFocus()` for you.

If the current item is a text item, the `TableSaveDataFuncType()` callback function is called to save the text in the currently selected field, the memory allocated for editing is released, and the insertion point is turned off.

Also note that you might have to call `FldReleaseFocus()` if the focus is in an editable text field and that field uses a custom drawing function (`TableDrawItemFuncType()`).

See Also `TblGrabFocus()`

TblRemoveRow Function

Purpose Removes the specified row from the table.

Declared In `Table.h`

Prototype
```
void TblRemoveRow (TableType *tableP, int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row to remove (zero-based).

Returns Nothing.

Comments The number of rows in the table is not decreased; instead, this row is moved from its current spot to the end of the table and is marked unusable so that it won't be displayed when the table is redrawn.

This function does not visually update the display. To do so, you must call `TblInvalidate()`.

This function may raise a fatal error message if an invalid row is specified.

See Also TblInsertRow(), TblSetRowUsable(), TblSetRowSelectable(), TblMarkRowInvalid()

TblRowInvalid Function

Purpose Returns whether a row is invalid.

Declared In `Table.h`

Prototype
```
Boolean TblRowInvalid (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

Returns `true` if the row is invalid, `false` if it's valid.

Comments Invalid rows need to be redrawn. Use `TblRedrawTable()` to do so.

This function may raise a fatal error message if the *row* parameter is invalid.

See Also `TblMarkRowInvalid()`, `TblMarkTableInvalid()`

TblRowMasked Function

Purpose Returns whether a row is masked.

Declared In `Table.h`

Prototype
```
Boolean TblRowMasked (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

Returns `true` if the row is masked, `false` otherwise.

Comments Your code should set a row to masked if it contains a private database record and the user has set the display preference for private records to masked. Masked cells are displayed as shaded.

Note that a table cell is not masked unless both its row and column are masked. This allows non-private information in the row item to

remain visible. For example, the Date Book application shows the time for a private appointment, but it does not show the description.

See Also TblSetRowMasked(), TblSetColumnMasked(), SecSelectViewStatus()

TblRowSelectable Function

Purpose Returns whether the specified row is selectable.

Declared In Table.h

Prototype
```
Boolean TblRowSelectable (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

Returns `true` if the row is selectable, `false` if it's not.

TblRowUsable Function

Purpose Determines whether the specified row is usable.

Declared In Table.h

Prototype
```
Boolean TblRowUsable (const TableType *tableP,
    int16_t row)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

Returns `true` if the row is usable, `false` if it's not.

Comments This function may display a fatal error message if the *row* parameter is invalid.

Rows that are not usable do not display.

See Also TblRowSelectable(), TblGetLastUsableRow(), TblSetRowUsable()

TblSelectItem Function

Purpose Selects the specified item. If there is already a selected item, that item is deselected.

Declared In `Table.h`

Prototype

```
void TblSelectItem(TableType *tableP, int16_t row,
    int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row of the item to select (zero-based).

→ *column*
Column of the item to select (zero-based).

Returns Nothing.

Comments This function may display a fatal error message if the *column* or *row* parameter point to an item that is not on the screen.

If *row* contains a masked private database record, then the item remains unselected.

If the selected table item contains a text field, the text field displays the insertion point, and all of the other items in the same row as the text field that have the edit indicator flag set are highlighted.

If the selected table item does not contain a text field, that item is highlighted and the previously selected table item is unhighlighted. If you want to the entire row to be highlighted, either create a table that has a single column, or write your own selection code.

If the selected item contains a check box or pop-up trigger, it is not highlighted; however, the current selection is still set to this table item.

See Also TblRowSelectable(), TblGetItemBounds(), TblGetItemInt()

TblSetBounds Function

Purpose Sets the bounds of a table.

Declared In `Table.h`

Prototype
```
void TblSetBounds (TableType *tableP,
    const RectangleType *rP)
```

Parameters

→ *tableP*
Pointer to a table.

→ *rP*
Pointer to a `RectangleType` structure that specifies the bounds for the table.

Returns Nothing.

TblSetColumnEditIndicator Function

Purpose Sets the column attribute that controls whether a column highlights when the table is in edit mode.

Declared In `Table.h`

Prototype
```
void TblSetColumnEditIndicator (TableType *tableP,
    int16_t column, Boolean editIndicator)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero based).

→ *editIndicator*
`true` to highlight, `false` to turn off highlight.

Returns Nothing.

Comments The edit indicator controls whether the item in the column is highlighted when a field within the current row is being edited. By default, text field items have the edit indicator value of `false`, while all other table item types have an edit indicator of `true`.

TblSetColumnMasked Function

Purpose Sets whether the column is masked.

Declared In `Table.h`

Prototype

```
void TblSetColumnMasked (TableType *tableP,
    int16_t column, Boolean masked)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *masked*
`true` to have the column be masked, `false` otherwise.

Returns Nothing.

Comments Masked cells are displayed as shaded. You should set a column to masked if its contents should be hidden when it contains information from a private database record and the user has set the display preference for private records to masked.

A table cell is not masked unless both its row and column are masked. This allows non-private information in the row item to remain visible. For example, the Date Book application shows the time for a private appointment, but it does not show the description.

Because the number of columns is static, you only need to call this function once per column when you first set up the table. The masked attribute on the row will determine if the contents of the table cell are actually displayed as shaded.

See Also `TblRowMasked()`, `TblSetRowMasked()`, `SecSelectViewStatus()`

TblSetColumnSpacing Function

Purpose Sets the spacing after the specified *column*.

Declared In `Table.h`

Prototype
```
void TblSetColumnSpacing (TableType *tableP,
    int16_t column, Coord spacing)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *spacing*
Spacing after the column in standard coordinates.

Returns Nothing.

Comments This function may display a fatal error message if the *column* parameter is invalid.

See Also `TblSetColumnUsable()`

TblSetColumnUsable Function

Purpose Sets a column in a table to usable or unusable.

Declared In `Table.h`

Prototype
```
void TblSetColumnUsable (TableType *tableP,
    int16_t column, Boolean usable)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *usable*
`true` for usable or `false` for not usable.

Returns Nothing.

Comments This function may display a fatal error message if the *column* parameter is invalid.

Columns that are not usable do not display.

See Also TblMarkRowInvalid()

TblSetColumnWidth Function

Purpose Sets the width of the specified column.

Declared In `Table.h`

Prototype

```
void TblSetColumnWidth (TableType *tableP,
    int16_t column, Coord width)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *width*
Width of the column (in standard coordinates).

Returns Nothing.

This function may display a fatal error message if the *column* parameter is invalid.

See Also TblGetColumnWidth()

TblSetCustomDrawProcedure Function

Purpose Sets the custom draw callback procedure for the specified column.

Declared In `Table.h`

Prototype

```
void TblSetCustomDrawProcedure (TableType *tableP,
    int16_t column,
    TableDrawItemFuncPtr drawCallback)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number.

→ *drawCallback*
Callback function.

Returns Nothing.

Comments The custom draw callback function is used to draw table items with a `TableItemStyleType` of `customTableItem` or `tallCustomTableItem`.

This function may display a fatal error message if the *`column`* parameter is invalid.

See Also TableDrawItemFuncType(), TblDrawTable()

TblSetItemFont Function

Purpose Sets the font used to display a table item.

Declared In `Table.h`

Prototype
```
void TblSetItemFont (TableType *tableP,
    int16_t row, int16_t column, FontID fontID)
```

Parameters

→ *`tableP`*
Pointer to a table.

→ *`row`*
Row number of the item (zero-based).

→ *`column`*
Column number of the item (zero-based).

→ *`fontID`*
`FontID` of the font to be used.

Returns Nothing.

Comments This function sets the internal font value stored for this table item. Only certain types of table items use this stored font value when they are displayed. The `TableItemStyleType` description specifies what font is used to display each type of table item. It is not an error to set the font for a table item that does not use it.

This function may display a fatal error message if the *`row`* or *`column`* parameter specifies a row or column that is not on the screen.

See Also `TblGetItemFont()`

TblSetItemInt Function

Purpose Sets the integer value of the specified item.

Declared In `Table.h`

Prototype

```
void TblSetItemInt(TableType *tableP, int16_t row,
    int16_t column, int16_t value)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

→ *value*
Any byte value (an integer).

Returns Nothing.

Comments This function may display a fatal error message if the *row* or *column* parameter is invalid.

You typically use this function when setting up and initializing a table for the first time to set the value of each table cell.

This function sets the internal integer value stored for this table item. Certain types of table items display their integer value, and other types display the value pointed to by their internal pointer value. See the `TableItemStyleType` description for details. If you set the integer value of an item that displays its pointer value, it is not an error. An application can store whatever value it wants in the integer value field; however, be aware that this has nothing to do with the value displayed by such a table cell.

See Also `TblGetItemInt()`, `TblSetItemPtr()`

TblSetItemPtr Function

Purpose Sets the item to the specified pointer value.

Declared In `Table.h`

Prototype

```
void TblSetItemPtr(TableType *tableP, int16_t row,
    int16_t column, void *value)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

→ *value*
Pointer to data to display in the table item.

Returns Nothing.

Comments This function may display a fatal error message if the *row* or *column* parameter is invalid.

This function sets the internal pointer value stored for this table item. Certain types of table items display the value pointed to by this pointer, and other types display an integer value. See the `TableItemStyleType` description for details. If you set the pointer value of an item that displays its integer value, it is not an error. An application can store whatever value it wants in the pointer field; however, be aware that this has nothing to do with the value displayed by such a table cell.

See Also `TblGetItemPtr()`, `TblSetItemInt()`

TblSetItemStyle Function

Purpose Sets the type of item to display; for example, text, numbers, dates, and so on.

Declared In `Table.h`

Prototype

```
void TblSetItemStyle (TableType *tableP,
    int16_t row, int16_t column,
    TableItemStyleType type)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item (zero-based).

→ *column*
Column number of the item (zero-based).

→ *type*
The type of item, such as an editable text field or a check box. See `TableItemStyleType` for a list of possible values.

Returns Nothing.

Comments This function may display a fatal error message if the *row* or *column* parameter is invalid.

Applications typically use this function when first setting up and initializing a table; you do not dynamically change item styles.

Follow this function with a call to either `TblSetItemInt()` or `TblSetItemPtr()` to set the value displayed by the table item. You should call one or the other of these functions depending on the type of table item you specified. See the `TableItemStyleType` description for details.

Note also that a table column always contains items of the same type.

Example The following code initializes a table:

```
for (row = 0; row < rowsInTable; row++) {
  TblSetItemStyle (table, row, completedColumn,
    checkboxTableItem);
  TblSetItemStyle (table, row, priorityColumn,
    numericTableItem);
  TblSetItemStyle (table, row, descColumn, textTableItem);
```

```
  TblSetItemStyle (table, row, dueDateColumn,
    customTableItem);
  TblSetItemStyle (table, row, categoryColumn,
    customTableItem);
}
```

See Also TblSetCustomDrawProcedure()

TblSetLoadDataProcedure Function

Purpose Sets the load-data callback procedure for the specified column.

Declared In Table.h

Prototype

```
void TblSetLoadDataProcedure (TableType *tableP,
    int16_t column,
    TableLoadDataFuncPtr loadDataCallback)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *loadDataCallback*
Callback procedure. See TableLoadDataFuncType().

Returns Nothing.

Comments The callback procedure is used to load the data values of a table item. See the TableLoadDataFuncType() for more information on writing the callback function.

You typically use this function when first setting up and initializing a table.

See Also TblSetCustomDrawProcedure()

TblSetRowData Function

Purpose Sets the data value of the specified row. The data value is a placeholder for application-specific values.

Declared In `Table.h`

Prototype

```
void TblSetRowData (TableType *tableP, int16_t row,
    uint32_t data)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *data*
Application-specific data value to store for this row. For example, the Date Book and To Do applications use this field to store the unique ID of the database record displayed by this row.

Returns Nothing.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblGetRowData(), TblFindRowData()

TblSetRowHeight Function

Purpose Sets the height of the specified row.

Declared In `Table.h`

Prototype

```
void TblSetRowHeight (TableType *tableP,
    int16_t row, Coord height)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *height*
New height in standard coordinates.

Returns Nothing.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblGetRowHeight(), TblSetRowStaticHeight()

TblSetRowID Function

Purpose Sets the ID value of the specified row.

Declared In Table.h

Prototype
```
void TblSetRowID (TableType *tableP, int16_t row,
    uint16_t id)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *id*
ID to identify a row.

Returns Nothing.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblGetRowID(), TblFindRowID()

TblSetRowMasked Function

Purpose Sets a row in a table to masked or unmasked.

Declared In Table.h

Prototype
```
void TblSetRowMasked (TableType *tableP,
    int16_t row, Boolean masked)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *masked*
`true` to have the row be masked, `false` otherwise.

Returns Nothing.

Comments Masked cells are displayed as shaded. You should call this function before drawing or redrawing the table. If a table row contains a private database record and the user has set the display preference for private records to masked, you must call this function on that row.

Note that a table cell is not masked unless both its row and column are masked. This allows non-private information in the row item to remain visible. For example, the Date Book application shows the time for a private appointment, but it does not show the description.

Example The following example shows how to mask private records:

```
uint16_t attr;
privateRecordViewEnum privateRecordStatus;
Boolean masked;

privateRecordStatus = (privateRecordViewEnum)
  PrefGetPreference(prefShowPrivateRecords);
....
DmGetRecordAttr (ToDoDB, recordNum, &attr);
masked = ((attr & dmRecAttrSecret) &&
  (privateRecordStatus == maskPrivateRecords));
TblSetRowMasked(tableP, row, masked);
```

See Also TblRowMasked(), TblSetColumnMasked(), SecSelectViewStatus()

TblSetRowSelectable Function

Purpose Sets a row in a table to selectable or unselectable.

Declared In Table.h

Prototype

```
void TblSetRowSelectable (TableType *tableP,
    int16_t row, Boolean selectable)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *selectable*
`true` or `false`.

Returns Nothing.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblRowSelectable(), TblSetRowUsable()

TblSetRowStaticHeight Function

Purpose Sets the static height attribute of a row.

Declared In `Table.h`

Prototype
```
void TblSetRowStaticHeight (TableType *tableP,
    int16_t row, Boolean staticHeight)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *staticHeight*
`true` to set the static height, `false` to unset it.

Returns Nothing.

Comments A row that has its static height attribute set will not expand or contract the height of the row as text is added or removed from a text item.

This function may display a fatal error message if the *row* parameter is invalid.

TblSetRowUsable Function

Purpose Sets a row in a table to usable or unusable. Rows that are not usable do not display.

Declared In `Table.h`

Prototype

```
void TblSetRowUsable (TableType *tableP,
    int16_t row, Boolean usable)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number (zero-based).

→ *usable*
`true` or `false`.

Returns Nothing.

Comments This function may display a fatal error message if the *row* parameter is invalid.

See Also TblRowUsable(), TblSetRowSelectable()

TblSetSaveDataProcedure Function

Purpose Sets the save-data callback function for the specified column.

Declared In `Table.h`

Prototype

```
void TblSetSaveDataProcedure (TableType *tableP,
    int16_t column,
    TableSaveDataFuncPtr saveDataCallback)
```

Parameters

→ *tableP*
Pointer to a table.

→ *column*
Column number (zero-based).

→ *saveDataCallback*
Callback function. See TableSaveDataFuncType().

Returns Nothing.

Comments The callback function is called when the table determines the data of a text field needs to be saved.

This function may display a fatal error message if the *column* parameter is invalid.

See Also TblSetCustomDrawProcedure()

TblSetSelection Function

Purpose Sets a table item.

Declared In Table.h

Prototype
```
void TblSetSelection (TableType *tableP,
    int16_t row, int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Table row.

→ *column*
Table column.

Returns Nothing.

Comments This function sets a table item, determined by the *row* and *column* parameters, as the current selection. This function does not update the display. To do so, you must call TblInvalidate(). Do *not* call TblDrawTable() directly unless your code is handling a frmUpdateEvent.

See Also TblGetNumberOfColumns(), TblGetTopRow()

TblUnhighlightSelection Function

Purpose Unhighlights the currently selected item in a table.

Declared In Table.h

Prototype
```
void TblUnhighlightSelection (TableType *tableP)
```

Parameters

→ *tableP*
Pointer to a table.

Returns Nothing.

Application-Defined Functions

TableDrawItemFuncType Function

Purpose Draws a custom table item.

Declared In `Table.h`

Prototype

```
void TableDrawItemFuncType (void *tableP,
    int16_t row, int16_t column,
    RectangleType *bounds)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the item to be drawn (zero-based).

→ *column*
Column number of the item to be drawn (zero-based).

→ *bounds*
The area of the screen in which the item is to be drawn. See `RectangleType`.

Returns Nothing.

Comments This function is called during `TblDrawTable()` and `TblRedrawTable()`.

You implement a custom drawing function when your table contains items of type `customTableItem` (to draw the entire item) or `narrowTextTableItem` (to draw whatever is required in the space between the text and the right edge of the table cell).

You may implement a custom drawing function to include any style of information in the table. If you don't like the way a predefined item is drawn, you may prefer to use a `customTableItem` instead. For example, if you want to include a date in your table but you want it to show the year as well as the month and day, you should implement a custom drawing function.

Compatibility Previously, this callback function never had to do anything special to draw a selected table item. The operating system would swap the foreground and background colors for selected table items after the callback function returned. In Palm OS Cobalt, this is no longer

possible. The Palm OS Cobalt table drawing code sets the colors that the table item should use before calling this callback function. If the item is not selected, it sets the colors to the normal table colors. If the item is selected, it sets the colors to the highlighted colors when appropriate.

If your callback function sets its own colors, it must check to see if it is drawing an item that should be highlighted and adjust its color scheme accordingly.

- Use `TblGetSelection()` to determine what the currently selected item is. If the row and column it returns match the row and column passed to the callback function, then you should draw the item as highlighted.
- If you have set the edit indicator flag for the column that is passed in, the function `TblEditing()` returns `true`, and the row returned by `TblGetSelection()` matches the row passed into the callback function, the table item should be drawn as highlighted to indicate that the item is in the row currently being edited.

  There is no function to check whether a column has the edit indicator flag set, so your code will need to keep track of which custom drawn columns should have an edit indicator.

  Listing 34.1 shows some code that determines if the cell should draw as selected.

You only need to worry about setting the appropriate color if the custom drawn item uses its own color scheme. If the item uses the colors currently set in the UI color table, then the operating system handles setting the color for you. This is true whether the application uses the system default application colors or it uses `UIColorSetTableEntry()` to set a color scheme before this callback function is called.

Listing 34.1 Determining if an item should draw as selected

```
void MyCustomDrawItemFunction(void *tableP, int16_t row,
int16_t column, RectangleType *bounds) {
   int16_t selectedRow, selectedColumn;
   Boolean drawAsSelected = false;

   if (TblGetSelection((TableType *)tableP, &selectedRow,
         &selectedColumn) {
```

```
      if (selectedRow == row) {
         if (selectedColumn == column)
            drawAsSelected = true;
          else if (TblEditing((TableType *)tableP) &&
               (column == myColumnWithEditIndicator))
            drawAsSelected = true;
      }
   }
   if (drawAsSelected)
      // Set colors to highlighted colors.
   else
      // Set colors to normal colors.
   // Now draw the item.
}
```

See Also TblSetCustomDrawProcedure()

TableLoadDataFuncType Function

Purpose Loads data into a column.

Declared In `Table.h`

Prototype

```
status_t TableLoadDataFuncType (void *tableP,
   int16_t row, int16_t column, Boolean editable,
   MemHandle *dataH, int16_t *dataOffset,
   int16_t *dataSize, FieldPtr fld)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the table item to load.

→ *column*
Column number of the table item to load.

→ *editable*
If `true`, the table is currently being edited. If `false`, the table is being drawn but not necessarily being edited.

← *dataH*
Unlocked handle of a block containing a null-terminated text string.

← *dataOffset*
Offset from start of block to start of the text string.

← *dataSize*
Allocated size of text string, *not* the string length.

→ *fld*
Pointer to the text field in this table cell.

Returns `errNone` upon success or an error if unsuccessful.

Comments This function is called in two cases: when a text field item is being drawn (`TblDrawTable()` or `TblRedrawTable()`) and when a text field item is being selected (part of `TblHandleEvent()`'s handling of `tblEnterEvent`). If this function returns an error (any nonzero value) and the item is being selected, then the item is not selected and the table's editing attribute is set to `false`.

You return the same values for *dataH*, *dataOffset*, and *dataSize* that you would pass to `FldSetText()`. That is, you can use this function to point the table cell's text field to a string in a database record so that you can edit that string directly using text field routines. To do so, return the handle to a database record in *dataH*, the offset from the start of the record to the start of the string in *dataOffset*, and the allocated size of the string in *dataSize*.

The handle that you return from this function is assumed to contain a null-terminated string starting at *dataOffset* bytes in the memory chunk. The string should be between 0 and *dataSize* - 1 bytes in length.

You are responsible for freeing the memory associated with the *dataH* parameter. You can do so in the `TableSaveDataFuncType()` function, but it is only called for a cell that has been edited. For non-editable text cells or text cells that are editable but were never selected, free the memory when you close the form.

The *fld* pointer passed to your function has already been initialized with default values by the table code. If you want to override a field's attributes (for example, if you want to change the underline mode) you can do so in this function.

See Also `TblDrawTable()`, `TblHandleEvent()`, `TableLoadDataFuncType()`

TableSaveDataFuncType Function

Purpose Saves the data associated with a text field.

Declared In `Table.h`

Prototype

```
Boolean TableSaveDataFuncType (void *tableP,
    int16_t row, int16_t column)
```

Parameters

→ *tableP*
Pointer to a table.

→ *row*
Row number of the table item to load.

→ *column*
Column number of the table item to load.

Returns `true` if the table should be redrawn, or `false` if the table does not need to be redrawn.

Comments This is called before the memory associated with the currently selected text field in a table cell is freed. Implement this function if you need to do any special cleanup before this memory is freed.

This function is called only when the currently selected editable text field is releasing the focus. You can use `TblGetCurrentField()` to retrieve a pointer to this field. It is called only on the currently selected field, not on any other fields in the table.

Note that the table manager already disassociates the memory handle from the text field for you so that the memory associated with your data is not freed when the field is freed. The table manager also calls `FldCompactText()` for you.

If the text handle you returned in your `TableLoadDataFuncType()` callback points to a string on the dynamic heap, you should implement this callback function to store or free the handle. You can use `FldGetTextHandle()` to obtain the handle.

If you return `true` from this function, `TblRedrawTable()` is called. You should mark invalid any table rows that you want redrawn before returning.

See Also TblSetSaveDataProcedure()

35

UI Color List Reference

This chapter provides information about the UI color list by discussing the following topics:

The header file `UIColor.h` declares the API that this chapter describes. For more information on the color list, see "Modifying the UI Color List" on page 163.

UI Color Constants

UIColorTableEntries Typedef

Purpose Declares symbolic color constants for the various UI elements.

Declared In `UIColor.h`

Prototype

```
typedef Enum8 UIColorTableEntries
```

Constants

`UIObjectFrame = 0`
: Color for the border of user interface elements (such as command buttons, push buttons, selector triggers, menus, check boxes, and other controls).

`UIObjectFill`
: The background color for a solid or "filled" user interface element.

  Note that UI elements in tables use the `UIField`... colors instead of the `UIObject`... colors.

`UIObjectForeground`
: The color for foreground items (such as labels or graphics) in a user interface element.

`UIObjectSelectedFill`
: The background color of the currently selected user interface element, whether that element is solid or not.

`UIObjectSelectedForeground`
: The color of foreground items in a selected user interface element.

`UIMenuFrame`
: The color of the border around the menu.

`UIMenuFill`
: The background color of a menu item.

`UIMenuForeground`
: The color of the menu's text.

`UIMenuSelectedFill`
: The background color of a selected menu item.

`UIMenuSelectedForeground`
: The color of the text of a selected menu item.

`UIFieldBackground`
: The background color of an editable text field.

`UIFieldText`
: The color of the text in the editable field.

`UIFieldTextLines`
: The color of underlines in an editable field.

`UIFieldCaret`
: The color of the cursor in an editable text field.

`UIFieldTextHighlightBackground`
: The background color for selected text in an editable text field.

`UIFieldTextHighlightForeground`
: The color of the selected text in an editable text field.

`UIFieldFepRawText`
: Color used for unconverted text in the inline conversion area when a FEP is used as a text input method (for example, on Japanese devices).

  If the FEP colors are identical to field colors, unconverted text has a solid underline.

`UIFieldFepRawBackground`
: The background color for unconverted text in the inline conversion area when a FEP is used as a text input method.

  If the FEP colors are identical to field colors, unconverted text has a solid underline.

`UIFieldFepConvertedText`
: Color used for converted text in the inline conversion area when a FEP is used as a text input method (for example, on Japanese devices).

  If the FEP colors are identical to field colors, converted text has a double-thick underline.

`UIFieldFepConvertedBackground`
: The background color used for converted text in the inline conversion area.

  If the FEP colors are identical to field colors, converted text has a double-thick underline.

`UIFieldFepUnderline`
: The color used for underlines beneath unconverted text in the inline conversion area.

`UIFormFrame`
: The color of the border and title bar on a form.

`UIFormFill`
: The background color of a form. White is recommended for this value.

`UIDialogFrame`
: The color of a border and title bar on a modal form.

`UIDialogFill`
: The background color of a modal form.

`UIAlertFrame`
: The color of the border and title bar on an alert panel.

`UIAlertFill`
The background color of an alert panel.

`UIOK`
Not used.

`UICaution`
Not used.

`UIWarning`
Not used.

`UILastColorTableEntry`
Placeholder to indicate end of enum.

Comments Do not confuse the UI color list with the system color table. The **system color table** (or **system palette**) was used in earlier Palm OS® releases to define all available colors for the display or draw window, whether they are in use or not. The **UI color list** defines the colors used to draw the interface elements.

UI Color Functions and Macros

UIColorGetTableEntryIndex Function

Purpose Returns the index value for a UI color for the current system palette.

Declared In `UIColor.h`

Prototype
```
IndexedColorType UIColorGetTableEntryIndex
    (UIColorTableEntries which)
```

Parameters → *`which`*
One of the symbolic color constants. See `UIColorTableEntries`.

Returns The system color table index of the color used for the specified symbolic color.

Comments Because Palm OS Cobalt supports direct color displays rather than indexed color displays, `UIColorGetTableEntryRGB()` is more efficient.

See Also `IndexedColorType`, `WinIndexToRGB()`

UIColorGetTableEntryRGB Function

Purpose Return the RGB value for the UI color.

Declared In `UIColor.h`

Prototype

```
void UIColorGetTableEntryRGB
    (UIColorTableEntries which, RGBColorType *rgbP)
```

Parameters

→ *which*
One of the symbolic color constants. See `UIColorTableEntries`.

← *rgbP*
Pointer to an RGB color value corresponding to the current color used for the symbolic color. (See `RGBColorType`.)

Returns Returns nothing.

Example The following code shows how to loop through the UI color table and store a list of all colors used (excluding duplicates):

```
RGBColorType colorsUsed[UILastColorTableEntry];
RGBColorType currentColor;
uint16_t i, j, numColors = 0;
Boolean isNew ;
...
for (i = 0; i < UILastColorTableEntry; i++) {
  isNew = true;
  currentColor = UIColorGetTableEntryIndex(i);

  for (j = 0; ((j < numColors) && isNew); j++)
    if (colorsUsed[j] == currentColor)
      isNew = false; /* exit loop */
  if (isNew) {
    numColors++;
    colorsUsed[j] = currentColor;
  }
}
```

See Also `UIColorGetTableEntryIndex()`, `WinRGBToIndex()`

UIColorSetTableEntry Function

Purpose Change a value in the UI color list.

Declared In `UIColor.h`

Prototype

```
status_t UIColorSetTableEntry
   (UIColorTableEntries which,
   const RGBColorType *rgbP)
```

Parameters

→ *which*
One of the symbolic color constants. See `UIColorTableEntries`.

→ *rgbP*
The RGB value of the color that should be used for the specified UI element. (See `RGBColorType`.)

Returns Returns 0 upon success.

Comments Sets the value of a UI color entry to the passed RGB value. Updates the index for that UI color entry to the current best fit for that RGB value according to the palette used by the current draw window.

It is best to use this function only if the draw window is currently on-screen. Otherwise, the best-fit algorithm may choose a color that is not available on the current screen.

See Also `WinIndexToRGB()`, `UIColorGetTableEntryIndex()`, `UIColorGetTableEntryRGB()`

36

Window Reference

This chapter provides information about windows by discussing these topics:

The header files `Window.h` and `CmnDrawingTypes.h` declare the API that this chapter describes. Chapter 1, "The Display," Chapter 2, "Working with Forms and Dialogs," and Chapter 8, "Drawing," describe some window-related concepts.

Window Structures and Types

CustomPatternType Typedef

Purpose Defines a pattern.

Declared In `CmnDrawingTypes.h`

Prototype

```
typedef uint8_t CustomPatternType[8];
```

Comments `CustomPatternType` holds an 8-by-8 bit pattern that is one bit deep. Each byte specifies a row of the pattern. When drawing, a pattern is tiled to fill a specified region.

See Also `WinFillLine()`, `WinFillRectangle()`, `PatternType`

FrameBitsType Struct

Purpose Specifies attributes of a window's frame.

Declared In `Window.h`

Prototype

```
typedef union FrameBitsType {
struct oldBitsTag {
      uint16_t cornerDiam : 8;
      uint16_t reserved_3 : 3;
      uint16_t threeD : 1;
      uint16_t shadowWidth : 2;
      uint16_t width : 2;
   } bits;
   uint16_t word;
} FrameBitsType
```

Fields

`cornerDiam`
The corner radius of frame; maximum is 38.

`reserved_3`
Reserved.

`threeD`
Set this bit to draw a 3D button. This feature is not currently supported.

`shadowWidth`
The width of the shadow.

`width`
The frame width.

`word`
Reserved.

See Also `FrameType`

IndexedColorType Typedef

Purpose Specifies a color by its index value; that is, by its location in a color table.

Declared In `CmnDrawingTypes.h`

Prototype

```
typedef uint8_t IndexedColorType;
```

Comments Color tables are defined by the `ColorTableType` structure, which is declared in `CmnBitmapTypes.h`. The `IndexedColorType` can hold a 1, 2, 4, or 8-bit index.

WinConstraintsType Struct

Purpose Specifies a window's position and sizing requirements.

Declared In `Window.h`

Prototype

```
typedef struct WinConstraintsType {
   uint32_t x_flags;
   int16_t x_pos;
   int16_t x_min;
   int16_t x_pref;
   int16_t x_max;
   uint32_t y_flags;
   int16_t y_pos;
   int16_t y_min;
   int16_t y_pref;
   int16_t y_max;
} WinConstraintsType
```

Fields

`x_flags`
Currently unused. Set this to 0.

`x_pos`
The position, given in standard coordinates, of the left side of the window. Use the `winUndefConstraint` constant for this value to have the system place the window.

`x_min`
The minimum possible width of the window in standard coordinates.

`x_pref`
The preferred width of the window in standard coordinates.

`x_max`
The maximum width of the window in standard coordinates.

`y_flags`
Currently unused. Set this to 0.

`y_pos`
: The position of the top of the window in standard coordinates. Use the `winUndefConstraint` constant for this value to have the system place the window.

`y_min`
: The minimum height that the window should ever be made in standard coordinates.

`y_pref`
: The preferred height for this window in standard coordinates.

`y_max`
: The maximum height for the window in standard coordinates.

Comments

You set a window's size constraints when you add a `WINDOW_CONSTRAINTS_RESOURCE` to a form or when you programmatically create a window using `WinCreateWindowWithConstraints()`. The actual size of the window is negotiated at run time taking into account the size requirements for the active pinlet in the input area, the status bar, and any other panel that is being displayed.

These constraints should specify a window's ideal sizes independent of the screen's actual size. It's the Window Manager's job to take the screen size into consideration when actually sizing the window.

For example, consider a window showing a scrolling list of text. This window should report its minimum height as being the smallest size it can be while still being usable. For many applications, the minimum height is 160, which is also the default for legacy-mode windows. The window's preferred size should be the height required to display all of the text it contains. The maximum height could be one of two possibilities. If the text is not editable, it should be the same as the preferred height. If the text is editable, it will probably be the constant `winMaxConstraint`, which means make the window as large as possible because the user can use that extra space for editing.

See Also

`WinSetConstraints()`, "Constraint Constants"

WinHandle Typedef

Purpose Provides the handle to a window.

Declared In `Window.h`

Prototype

```
struct WindowType
typedef struct WindowType *WinHandle;
```

Comments A window is an abstract drawing region. Typically, you don't work with windows directly. You work with forms instead. You might work with windows if your application needs custom drawing.

Compatibility The `WindowType` structure is no longer public and has changed significantly from its definition in earlier releases. Do not attempt to access the `WindowType` structure.

See Also `FrmGetWindowHandle()`, `WinGetActiveWindow()`, `WinGetDrawWindow()`

WinLineType Struct

Purpose Defines a line.

Declared In `Window.h`

Prototype

```
typedef struct WinLineType {
   Coord x1;
   Coord y1;
   Coord x2;
   Coord y2;
} WinLineType
```

Fields

`x1, y1`
The starting point of the line.

`x2, y2`
The endpoint of the line.

See Also `WinPaintLines()`

Window Constants

Constraint Constants

Purpose Special values for the fields in `WinConstraintsType`.

Declared In `Window.h`

Constants

`#define winUndefConstraint SHRT_MIN //—32768`
Specifies an undefined constraint. The system decides the value of this constraint. Undefined constraints are most commonly used for the position values so that the Window Manager positions the window at the preferred location.

`#define winMaxConstraint SHRT_MAX // 32767`
Specifies a value that means "as large as possible."

Coordinate System Constants

Purpose Specifies the coordinate system to be used when drawing with a given window.

Declared In `Window.h`

Constants

`#define kCoordinatesNative 0`
Use the bitmap's or screen's native coordinate system; this enables a 1-to-1 correspondence between coordinates and pixels.

If you use the native coordinate system, keep in mind that Palm Powered™ devices come in a variety of screen densities. Do not assume that native coordinates always means double-density.

`#define kCoordinatesStandard 72`
On a single-density handheld (that contains a 160 X 160 screen), there is one screen pixel per standard coordinate. On a high-density screen, there is more than one screen pixel per standard coordinate.

`#define kCoordinatesOneAndAHalf 108`
One and a half times the standard coordinate system.

`#define kCoordinatesDouble 144`
Twice the standard coordinate system.

`#define kCoordinatesTriple 216`
Three times the standard coordinate system.

`#define kCoordinatesQuadruple 288`
Four times the standard coordinate system.

See Also `WinSetCoordinateSystem()`, `WinGetCoordinateSystem()`, `GcSetCoordinateSystem()`

FrameType Typedef

Purpose Specifies a window frame style.

Declared In `Window.h`

Prototype

```
typedef UInt16 FrameType;
```

Constants

`#define noFrame 0x0000`
No frame.

`#define simpleFrame 0x0001`
A plain rectangular frame.

`#define rectangleFrame simpleFrame`
A plain rectangular frame.

`#define simple3DFrame 0x0012`
3D frame with width of 2. This frame type is not supported.

`#define roundFrame 0x0401`
A round frame with width of 1.

`#define boldRoundFrame 0x0702`
A round frame with width of 2.

`#define popupFrame 0x0205`
Pop-up frame style with slight corner roundness, width of 1 and shadow of 1.

`#define dialogFrame 0x0302`
Dialog frame style with slight corner roundness and width of 2.

`#define menuFrame popupFrame`
Same as `popupFrame`.

Comments The `FrameType` can be set to one of the defined frame types listed here or to a custom frame type as defined by a `FrameBitsType` structure.

See Also `WinGetFrameType()`, `WinSetFrameType()`

PatternType Enum

Purpose Specifies a pattern for drawing.

Declared In `CmnDrawingTypes.h`

Constants

`blackPattern`
All bits on.

`whitePattern`
All bits off.

`grayPattern`
Alternating on and off bits.

`customPattern`
Custom pattern specified by `CustomPatternType`.

`lightGrayPattern`
One out of four bits in each row turned on.

`darkGrayPattern`
One out of four bits in each row turned off.

Comments These patterns all operate with current foreground and background color instead of black and white. In effect, `blackPattern` is only black if the current foreground color is black. `whitePattern` uses the current background color. `grayPattern` and `customPattern` uses a combination of background and foreground colors.

Patterns are expanded to the destination bit depth when drawing patterned lines and filled rectangles.

The three standard gray patterns—`grayPattern`, `lightGrayPattern`, and `darkGrayPattern`—are always drawn using the screen density to improve the appearance of gray fills. Custom patterns, however, are stretched as appropriate based on the destination density.

See Also `WinGetPatternType()`, `WinSetPatternType()`

Other Patterns

Purpose Specifies special patterns.

Declared In `CmnDrawingTypes.h`

Constants

`#define noPattern blackPattern`
The shape is filled with no special pattern. Only the current color.

`#define grayHLinePattern 0xAA`
The system uses this to draw a gray diagonal line.

`#define grayHLinePatternOdd 0x55`
The system uses this to draw a gray diagonal line.

Scaling Mode Constants

Purpose Constants used to construct a scaling mode that is passed to `WinSetScalingMode()`.

Declared In `Window.h`

Constants

`#define kBitmapScalingOff 1`
Turns bitmap scaling off, which causes the rendering system to draw the low-density bitmap family member, unscaled, when the application calls one of the bitmap drawing functions (even if the bitmap family contains a bitmap whose density matches the window's). This causes your bitmaps to be drawn smaller, as if they were being viewed from a distance.

`#define kTextScalingOff 2`
Turns text scaling off, which causes the rendering system to render text using the low-density family member of the appropriate font, unscaled, when the application subsequently draws text (even if the font family contains a font whose density matches the window's). This results in smaller-sized text being drawn on the screen.

`#define kTextPaddingOff 4`
Turns off text padding. When text padding is turned off, the rendering system does not insert pixels between glyphs in order to align glyphs on standard coordinates. (Text padding is on by default.) By setting the `kTextPaddingOff` bit, an application can lay out text using high density coordinates

and preserve the native font's internal kerning. This unpadded mode provides the best appearance of text, but requires the application to use the high-density coordinate system.

Currently, setting `kTextPaddingOff` has no effect.

See Also `WinGetScalingMode()`

UnderlineModeType Enum

Purpose Specifies possible values for the underline mode stored in the drawing state.

Declared In `CmnDrawingTypes.h`

Constants

`noUnderline`
No underline.

`grayUnderline`
A dotted line in the current foreground color.

`solidUnderline`
A solid line in the foreground color.

`colorUnderline`
A solid line in the foreground color.

`thinUnderline`
A thin single-pixel solid line in the foreground color.

WinDirectionType Typedef

Purpose Specifies a scrolling direction.

Declared In `Window.h`

Prototype

```
typedef Enum8 WinDirectionType;
```

Constants

`winUp = 0`
Scroll up.

`winDown`
Scroll down.

`winLeft`
Scroll left.

`winRight`
Scroll right.

See Also `WinScrollRectangle()`, `FldScrollable()`, `FldScrollField()`, `LstScrollList()`

WinDrawOperation Enum

Purpose Specifies the transfer mode for color drawing.

Declared In `CmnDrawingTypes.h`

Constants

`winPaint`
Write color-matched source pixels to the destination. If a bitmap's `bmpFlagsHasTransparency` flag is set, `winPaint` behaves like `winOverlay` instead.

`winErase`
Write using the background color if the source pixel is transparent.

`winMask`
Write using the background color if the source pixel is not transparent.

`winInvert`
Bitwise XOR the color-matched source pixel onto the destination. This mode does not honor the transparent color in any way.

`winOverlay`
Write color-matched source pixel to the destination if the source pixel is not transparent. Transparent pixels are skipped. For a 1-bit display, the "off" bits are considered to be the transparent color.

`winPaintInverse`
Invert the source pixel color and then proceed as with `winPaint`.

`winSwap`
Swap the background color and foreground color destination colors if the source is a pattern (the type of pattern is disregarded). If the source is a bitmap, then the bitmap is transferred using `winPaint` mode instead.

Comments

Note that 2-bit, 4-bit, and 8-bit source bitmaps that don't have a color table inherit the system default color table for their given depth. 1-bit sources (bitmap, text, and patterns) that don't have a color table are given a color table where entry 0 is the background color and entry 1 is the foreground color (text color for text).

`winSwap` is not a color invert operation, although a pair of `winSwap` operations will restore the original graphics data. This mode is used by the OS to select and deselect areas of the screen. It changes destination pixels matching the foreground color to the background color, and changes destination pixels matching the background color to the foreground color. It is a mode available for rectangles, lines, and pixels, but not text or bitmaps. This mode ignores the current pattern.

The Transparent Color

Bitmaps have a `bmpFlagsHasTransparency` flag and may designate a transparent color:

- Bitmaps that don't specify any transparent color (text, patterns, and version 0 bitmaps) are assumed to have a transparent color of index 0 and the `bmpFlagsHasTransparency` bit is assumed to be `false`.
- When the `bmpFlagsHasTransparency` flag is set and the transfer mode is `winPaint`, only the non-transparent pixels are copied to the destination. With text and patterns, Palm OS® assumes that the "off" bits are the ones designated as transparent and acts as if the `bmpFlagsHasTransparency` flag is always `false`. This assumption retains backwards compatibility and unifies the use of transparency across all source data.

Compatibility

`winInvert`, `winErase`, `winMask`, `winPaintInverse`, and `winSwap` are deprecated in Palm OS Cobalt. `winInvert` and `winSwap` do not work when drawing text, drawing unfilled rectangles, or drawing to update-based windows. When you try to use them in these instances, nothing is drawn to the screen.

See Also

`WinCopyRectangle()`, `WinSetDrawMode()`

WinFlagsType Enum

Purpose Specifies how the window looks and behaves. These flags are passed as arguments to `WinCreateWindowWithConstraints()` when creating a window.

Declared In `Window.h`

Constants

`winFlagModal = 0x00000001`
The window is modal. When a modal window is the topmost window, it receives the pen down events and keystrokes for the window that it obscures, even when it is not the same size as that window.

`winFlagNonFocusable = 0x00000002`
The window cannot become the active window.

`winFlagBackBuffer = 0x00000004`
The window is back-buffered. A back-buffered window draws into a memory region. The system does not update the screen until the application returns to its event loop or it calls `WinFlush()`. Legacy windows and transitional windows are back-buffered.

`winFlagVisibilityEvents = 0x00000008`
The window receives the `winVisibilityChangedEvent` when anything obscures the window or when a previously obscured window is revealed.

`winLayerMask = 0x000000f0`
A mask value defining all of the layer constants.

`winLayerNormal = 0x00000000`
Used for application windows.

`winLayerPriority = 0x00000010`
Used for system dialogs.

`winLayerSlip = 0x00000040`
Used for the slip windows created by the status bar.

`winLayerSecurity = 0x00000050`
Used for password dialogs and other security-related dialogs.

`winLayerSystem = 0x00000020`
: Used for the dynamic input area, the status bar, and catastrophic system messages such as the low battery alert dialog

`winLayerMenu = 0x00000030`
: Used for menus.

See Also `WinGetWindowFlags()`

WindowFormatType Enum

Purpose Specifies the window format when creating an off-screen window with the `WinCreateOffscreenWindow()` function.

Declared In `Window.h`

Constants

`screenFormat = 0`
: The window's bitmap is allocated using the hardware screen's depth, but for backward compatibility the bitmap associated with the off-screen window is always low density, and the window always uses a coordinate system that directly maps off-screen pixels to coordinates.

`genericFormat`
: Like `screenFormat`.

`nativeFormat`
: Reflects the actual hardware screen format in all ways, including screen depth, density, and pixel format. Applications must always use the APIs when drawing to a `nativeFormat` off-screen window: directly accessing off-screen pixels will produce undefined results. When using this format, the width and height arguments must be specified using the active coordinate system.

Miscellaneous Constants

Purpose Other Window Manager constants.

Declared In `Window.h`

Constants

`#define DrawStateStackSize 7`
The maximum size of the draw state stack that is grown with `WinPushDrawState()` and shrunk with `WinPopDrawState()`.

`#define invalidWindowHandle 0`
Specifies an invalid window handle.

`#define kWinVersion 10`
The current version of the Window Manager. The feature constant `sysFtrNumWinVersion` is set to this value.

Window Events

sysClearUIEvent

Purpose Sent to all threads outside of the application's main UI thread to indicate that the application is about to be exited. This event allows background threads to close dialogs or remove any other user interface that they display.

Declared In `Event.h`

Prototype None.

Comments `SysHandleEvent()` handles the `sysClearUIEvent` for you by transforming it into an `appStopEvent` in any threads that are displaying a user interface but are not the main UI thread.

Most applications do not need to respond to this event. You only should do so if your application displays a dialog from a background thread and you don't want that dialog erased with the rest of the application's user interface. In that case, you should intercept the `sysClearUIEvent` before the call `SysHandleEvent()` and ignore it.

winEnterEvent

Purpose

Sent when a window becomes the active window. This can happen in two ways: a call to `WinSetActiveWindow()` is issued (`FrmSetActiveForm()` does this), or the user taps within the bounds of a window (such as a menu window) that is visible but not active.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In

`Event.h`

Prototype

```
struct _WinEnterEventType {
   WinHandle enterWindow;
   WinHandle exitWindow;
}
```

Fields

`enterWindow`
: The handle to the window being entered. This may be the window for a form, a menu, or some other type of window.

`exitWindow`
: The handle to the window being exited if there is currently an active window, or 0 if there is no active window.

Comments

`winEnterEvent` and `winExitEvent` are only sent for windows created using the UI Library. If the user taps the status bar, for example, the application receives only the `winFocusLostEvent`, not the `winExitEvent`.

`winEnterEvent` and `winExitEvent` are typically received when the active window within a thread changes. For example, when the user displays the menu, the application receives a `winExitEvent` for the current form's window and a `winEnterEvent` for the menu's window. When the menu is dismissed, the application receives the `winExitEvent` for the menu's window followed by the `winEnterEvent` for the form's window.

winExitEvent

Purpose Sent when a window is deactivated. A window is deactivated when another window becomes the active window.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _WinExitEventType {
   WinHandle enterWindow;
   WinHandle exitWindow;
}
```

Fields

`enterWindow`
The handle to the window being entered. This may be the window for a form, a menu, or some other type of window.

`exitWindow`
The handle to the window being exited.

Comments The `winExitEvent`, like all events, is sent to the active form's event handler. Because `winExitEvent` is often posted to the event queue as a result of a `FrmSetActiveForm()` call, it is sent to the event handler for the form that has just been made active, not to the event handler for the form whose window is being exited.

See Also `winEnterEvent`

winFocusGainedEvent

Purpose Sent when a new window gains the input focus.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _WinFocusGainedEventType {
   WinHandle window;
   uint32_t flags;
}
```

Fields

`window`
The handle to the window that's gained the focus.

`flags`
Currently unused.

Comments The input focus determines which thread is sent `keyDownEvents`. This event is sent to the event handler for the window that will now receive `keyDownEvents`. For example, if the user is currently working in a slip window and then taps in your application's form, the form's window receives the input focus. The form's event handler receives the `winFocusGainedEvent`.

This event is only sent regarding focusable windows. Menus, pop-up lists, and the input area all use nonfocusable windows. Entering and exiting these windows do not generate `winFocusGainedEvent` or `winFocusLostEvent`.

`FrmHandleEvent()` responds to this event by assigning the focus to a field within that form if necessary. That is, if a text field or table previously had the input focus when the form lost focus, it is assigned the input focus when the form regains focus.

Applications typically do not respond to this event. Games might want to do so to resume activity after receiving the input focus.

winFocusLostEvent

Purpose Sent when a new window gains input focus.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _WinFocusLostEventType {
   WinHandle window;
   uint32_t flags;
};
```

Fields

`window`
The handle to the window that's lost the focus.

`flags`
Currently unused.

Comments This event is sent to the event handler for the window that currently has the input focus and is about to lose it. For example, if a form within your application currently has input focus and the user taps

the status bar, your form's event handler receives the `winFocusLostEvent`.

Note the difference between this event and the `winExitEvent`. If your application has the input focus and the user switches to another form in the same application:

- The first form receives `winFocusLostEvent`.
- The second form receives a `winExitEvent` about the first form's window.
- The second form receives a `winEnterEvent` about its own window.
- If the second form is granted input focus, it receives the `winFocusGainedEvent` as well. It is not guaranteed to receive this event because another window in a different thread might request the input focus at the same time.

This event is only sent regarding focusable windows. Menus, pop-up lists, and the input area all use nonfocusable windows. Entering and exiting these windows do not generate `winFocusGainedEvent` or `winFocusLostEvent`.

`FrmHandleEvent()` responds to this event by releasing any focus assigned to a field within the form that's losing focus.

Applications typically do not respond to this event. Games might want to do so to suspend activity upon losing the input focus.

See Also `WinSetActiveWindow()`

winResizedEvent

Purpose Sent when a window's size or position changes.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _WinResizedEventType {
   WinHandle window;
   RectangleType newBounds;
};
```

Fields

`window`
The handle to the window that needs to be resized.

`newBounds`
A rectangle specifying the new bounds for the window. See `RectangleType`.

Comments

Applications should respond to this event by rearranging the controls in the form as necessary. You can set up the form so that this is handled for you. Call `FrmInitLayout()` when the form is loaded and specify what should happen for each element in the form when the form is resized (whether that element should move or it should resize). After you make this call, `FrmHandleEvent()` calls `FrmPerformLayout()` in response to the `winResizedEvent`. `FrmPerformLayout()` uses the resizing rules you specify to rearrange the elements in the form.

If you need more control over the layout of one or more elements on the form, handle `winResizedEvent`. Call `FrmPerformLayout()` and then tweak the positions or sizes of the elements you need to control. Optionally, you can handle all elements in the form yourself.

Do not draw the controls until you receive a `frmUpdateEvent`.

`winResizedEvent` is not sent for legacy windows; legacy windows do not resize.

winUpdateEvent

Purpose

Sent when all or a portion of a window needs to be redrawn. This includes when the window is first created, when it is resized, and as a result of the functions `WinInvalidateRect()` or `WinInvalidateWindow()`.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _WinUpdateEventType {
   WinHandle window;
   RectangleType dirtyRect;
   void *prv;
};
```

Fields

`window`
The handle to the window that needs to be redrawn.

`dirtyRect`
The portion of the window that needs to be redrawn. See `RectangleType`.

`prv`
System use only.

Comments This event is sent for any type of window other than legacy windows. A window may either represent a form, a menu, or a pinlet.

`winUpdateEvent` is a signal that the rendering system has created a graphics context for the window and is ready for the application to draw to the window.

When the window represents a form, no action is required for this event. If it is not handled, the system posts a `frmUpdateEvent`. `FrmHandleEvent()` responds to the `frmUpdateEvent` by calling `FrmDrawForm()` and then applications can perform any custom drawing.

The system handles updating the windows for menus and other user interface items, so no action is typically required from an application if *`window`* represents a menu.

IMPORTANT: Never post a `winUpdateEvent` or a `frmUpdateEvent` explicitly. Instead, use one of the functions `WinInvalidateRect()`, `WinInvalidateRectFunc()`, or `WinInvalidateWindow()`.

---

winVisibilityChangedEvent

Purpose Sent to specially marked windows when another window obscures a portion of it.

For this event, the `EventType` `data` field contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct winVisibilityChanged {
   WinHandle window;
   uint32_t visibility;
} winVisibilityChanged;
```

Fields

`window`
The handle to the window that has been obscured.

`visibility`
One of the following:

`windowFullyHidden`
The window is completely obscured.

`windowPartiallyVisible`
A portion of the window is visible.

`windowFullyVisible`
The window is completely visible.

Comments Only windows that have the `winFlagVisibilityEvents` flag set in the constraints resource receive this event. You should only set the flag on windows that display time-sensitive information that can become meaningless if obscured. For example, games may want to set the flag for their windows so that they can delay the game if the user launches a slip window or if any other window obscures the game.

Most applications should not care if another window obscures the application's windows. When the window is redrawn, drawing is always clipped to the visible region.

Window Manager Notifications

sysNotifyDisplayChangeEvent

Purpose Broadcast whenever the display mode changes. That is, either the color table has been set to use a specific palette using the `WinPalette()` function or the bit depth has changed using the `WinScreenMode()` function.

Declared In `NotifyMgr.h`

Parameters `notifyDetailsP` points to a `SysNotifyDisplayChangeDetailsType` structure

Comments The `notifyDetailsP` parameter indicates how the bit depth changed. If the two values in the struct are equal, it means that the color palette has changed instead of the bit depth.

See Also "Notification Manager" in *Exploring Palm OS: Programming Basics*

Window Management Functions and Macros

This section lists and describes the Window Manager functions that involve the creation, deletion, and manipulation of windows as abstract regions.

ECWinValidateHandle Macro

Purpose Calls `WinValidateHandle()`.

Declared In `Window.h`

Prototype
```
#define ECWinValidateHandle (winHandle)
```

Parameters → *winHandle*
The `WinHandle` to validate.

Returns Nothing.

Comments On debug ROMs, displays a fatal error message if the window handle is invalid. On release ROMs, this macro has no effect.

Do not use this macro in production code. It is for debugging purposes only.

WinConvertCoord Function

Purpose Converts a coordinate from one screen density to another.

Declared In `Window.h`

Prototype

```
Coord WinConvertCoord (uint16_t currentDensity,
    uint16_t newDensity, Coord coord,
    Boolean ceiling)
```

Parameters

→ *currentDensity*
The current density for the specified coordinate.

→ *newDensity*
The density to which to scale the coordinate.

→ *coord*
Coordinate in the *currentDensity*.

→ *ceiling*
`true` to round up or `false` to round down.

Returns Coordinate in *newDensity* that provides the same location as *coord*.

Comments This function converts a coordinate by multiplying it by the coordinate scaling factor, which is computed from the current and new densities. What happens next depends upon the value of *ceiling*:

if *ceiling* == true {

  use `lfloorf()` function when scaleFactor > 1

  use `lceilf()` function when scaleFactor < 1

else

  use `lceilf()` function when scaleFactor > 1

  use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

See Also `WinScaleCoord()`, `WinScaleCoordNativeToActive()`, `WinScreenGetAttribute()`

WinConvertPoint Function

Purpose Converts a point from one screen density to another.

Declared In `Window.h`

Prototype

```
void WinConvertPoint (uint16_t currentDensity,
   uint16_t newDensity, PointType *pointP,
   Boolean ceiling)
```

Parameters

→ *currentDensity*
The current density for the specified point.

→ *newDensity*
The density to which to scale the point.

↔ *pointP*
A point in the *currentDensity*. Upon return, a point in *newDensity* that represents the same location.

→ *ceiling*
`true` to round up or `false` to round down.

Returns Nothing.

Comments This function converts a point by multiplying its x and y coordinates by the coordinate scaling factor., which is computed from the current and new densities What happens next depends upon the value of *ceiling*:

if *ceiling* == true {

use `lfloorf()` function when scaleFactor > 1

use `lceilf()` function when scaleFactor < 1

else

use `lceilf()` function when scaleFactor > 1

use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

See Also `WinScalePoint()`, `WinScreenGetAttribute()`

WinConvertRectangle Function

Purpose Converts a rectangle from one screen density to another.

Declared In `Window.h`

Prototype

```
void WinConvertRectangle
    (uint16_t currentDensity, uint16_t newDensity,
    RectangleType *rectP)
```

Parameters

→ *currentDensity*
The current density for the specified rectangle.

→ *newDensity*
The density to which to scale the rectangle.

↔ *rectP*
A rectangle in the *currentDensity*. Upon return, the same rectangle with the same location and dimensions but given in *newDensity*.

Returns Nothing.

See Also `WinScaleRectangle()`, `WinScreenGetAttribute()`

WinCreateBitmapWindow Function

Purpose Creates a new off-screen window.

Declared In `Window.h`

Prototype

```
WinHandle WinCreateBitmapWindow
    (BitmapType *bitmapP, status_t *error)
```

Parameters

→ *bitmapP*
A pointer to a bitmap to associate with the window. (See `BitmapType`.)

← *error*
A pointer to any error this function encounters.

Returns The handle of the new window upon success, or `NULL` if an error occurs. The *error* parameter contains one of the following:

`errNone`
No error.

`sysErrParamErr`
: The *`bitmapP`* parameter is invalid. The bitmap must be uncompressed and it must have a valid pixel size (1, 2, 4, 8, or 16). It must not be the screen bitmap.

`sysErrNoFreeResource`
: There is not enough memory to allocate a new window structure.

Comments

Use `WinCreateBitmapWindow()` if you want to draw into a previously created bitmap, such as a bitmap created using `BmpCreate()`.

This function generates a window wrapper for the specified bitmap. The newly created window is off-screen and uses the generic format (for device independence). Use `WinSetDrawWindow()` to make it the draw window, and then use the window drawing functions to modify the bitmap.

When you use this function to create a window and then delete the window with `WinDeleteWindow()`, the bitmap is **not** freed when the window is freed.

`WinCreateOffscreenWindow()` uses this function to create its off-screen window. If you call `WinCreateOffscreenWindow()` instead of using this function, the bitmap is freed when `WinDeleteWindow()` is called.

Compatibility

This function exists for compatibility purposes only. Use `GcCreateBitmapContext()` instead.

See Also

`WinCreateWindow()`, `WinCreateOffscreenWindow()`

WinCreateOffscreenWindow Function

Purpose Creates a new off-screen window.

Declared In `Window.h`

Prototype

```
WinHandle WinCreateOffscreenWindow (Coord width,
    Coord height, WindowFormatType format,
    status_t *error)
```

Parameters

→ *width*
The width of the window in pixels. The coordinate system you use for this parameter depends upon the value of *format*.

→ *height*
The height of the window in pixels. The coordinate system you use for this parameter depends upon the value of *format*.

→ *format*
One of the window formats defined by `WindowFormatType`.

← *error*
A pointer to any error this function encounters.

Returns The handle of the new window upon success, or `NULL` if an error occurs. The *error* parameter contains one of the following:

`errNone`
No error.

`sysErrParamErr`
The *width* or *height* parameter is `NULL` or the current color table is invalid.

`sysErrNoFreeResource`
There is not enough memory to complete the function.

Comments Windows created with this routine draw to a memory buffer instead of the display. Use this function for temporary drawing operations such as double-buffering or save-behind operations.

The memory buffer has two formats: screen format and generic format. Screen format is the native format of the video system; windows in this format can be copied to the display faster. The generic format is device-independent.

This function differs from `WinCreateBitmapWindow()` in the following ways:

- `WinCreateOffscreenWindow()` creates a new bitmap in the same depth as the current screen. `WinCreateBitmapWindow()` uses the bitmap you pass in, which may or may not be in the same depth as the current screen.
- `WinCreateOffscreenWindow()` uses the screen format you specify. `WinCreateBitmapWindow()` always uses `genericFormat` for the format argument.
- When you delete the window created with `WinCreateOffscreenWindow()`, its bitmap is freed along with the window. The bitmap used in the `WinCreateBitmapWindow()` is not freed when the window is freed.

Note that if you aren't directly accessing the bits of an off-screen window's bitmap but are just using the APIs, you can always pass `nativeFormat` for the screen format and things will work as expected. If you need direct access to the bits of the off-screen window's bitmap, however, call `BmpCreate()` and then call `WinCreateBitmapWindow()`. Because you created the bitmap, you know its format and thus can safely manipulate its bits. Calling `WinCreateOffscreenWindow()` with a *`format`* argument of `nativeFormat` can result in a bitmap with an unexpected format: the endianness, number of bits per pixel, and so on would match the screen and therefore be fastest to draw, but your application wouldn't be able to manipulate the pixels directly.

The bitmap data will not be blitted properly if the depth of the screen is changed using `WinScreenMode()` and the new window uses a bitmap that does not define the bitmap's color table. See `WinScreenMode()` for information on how to work around this limitation.

See Also `WinCreateWindow()`

WinCreateWindow Function

Purpose Creates a new legacy-mode window.

Declared In `Window.h`

Prototype

```
WinHandle WinCreateWindow
    (const RectangleType *bounds, FrameType frame,
    Boolean modal, Boolean focusable,
    status_t *error)
```

Parameters

→ *bounds*
The display-relative bounds of the window. See RectangleType.

→ *frame*
The type of frame around the window (see FrameType).

→ *modal*
`true` if the window is modal.

→ *focusable*
`true` if the window can be the active window.

← *error*
A pointer to any error encountered by this function.

Returns The handle of the new window upon success, or `NULL` if an error occurs. The *error* parameter contains one of the following:

`errNone`
No error.

`sysErrNoFreeResource`
There is not enough memory to complete the operation.

Comments The window created with this function is a legacy-mode window, meaning that it does not receive events such as winUpdateEvent or winResizedEvent.

You can only call this function within the main UI thread.

See Also WinDeleteWindow(), WinCreateWindowWithConstraints()

WinCreateWindowWithConstraints Function

Purpose

Creates a new update-based or transitional window with the specified size constraints.

Declared In

`Window.h`

Prototype

```
WinHandle WinCreateWindowWithConstraints
    (FrameType frame, WinFlagsType flags,
    const WinConstraintsType *constraints,
    status_t *error)
```

Parameters

→ *frame*
The type of frame around the window (see `FrameType`).

→ *flags*
Window creation flags (see `WinFlagsType`).

→ *constraints*
The size constraints (see `WinConstraintsType`).

← *error*
A pointer to any error encountered by this function.

Returns

The handle of the new window upon success, or `NULL` if an error occurs. The *error* parameter contains one of the following:

`errNone`
No error.

`sysErrNoFreeResource`
There is not enough memory to complete the operation.

Comments

Windows created by this routine draw to the display. See `WinCreateOffscreenWindow()` for information on drawing off screen.

You typically don't call this function directly. Instead, you use `FrmInitForm()` to create form windows from a resource. Forms are much more flexible and have better system support. All forms are windows, but not all windows are forms.

The window is created with the size constraints and frame type that you specify. See `WinConstraintsType` for an explanation of how to set the size constraints.

Newly created windows are disabled and invisible. You must specifically enable the window before the window can accept input. You can do so with `WinSetActiveWindow()`.

In Palm OS Cobalt, there are three types of windows that may be created:

- **Legacy windows**. These windows are created by `WinCreateWindow()` or they are created if you don't specify a `WINDOW_CONSTRAINTS_RESOURCE` for a form. You can draw to a legacy window whenever you want, not just in an update event. Legacy windows do not receive the new events (such as `winUpdateEvent` or `winResizedEvent`).
- **Update-based windows**. These windows are created by calling `WinCreateWindowWithConstraints()` or by adding a `WINDOW_CONSTRAINTS_RESOURCE` to a form. Drawing to an update-based window is asynchronous and should only be performed when the system requests you to do so (by posting a `winUpdateEvent`).
- **Transitional windows**. These windows are created by calling `WinCreateWindowWithConstraints()` and specifying `winFlagBackBuffer` in the *`flags`* parameter or by adding to a form a `WINDOW_CONSTRAINTS_RESOURCE` with the `winFlagBackBuffer` bit set. You must wait for an initial `winResizedEvent` before you draw to a transitional window. After the initial event, you can draw to a transitional window any time you like.

Use this function instead of calling `WinCreateWindow()` followed by `WinSetConstraints()` because it avoids the intermediate state of the window.

WinDeleteWindow Function

Purpose Destroys the record for the window. The `WinHandle` is no longer valid after calling this function.

Declared In `Window.h`

Prototype

```
void WinDeleteWindow (WinHandle winHandle,
    Boolean eraseIt)
```

Parameters

→ *`winHandle`*
The handle of the window to delete.

→ *eraseIt*
If `true`, the window is erased before it is deleted. If `false`, the window is not erased.

Returns Nothing.

Comments This function frees all memory associated with the window that was allocated by the system. Windows created using `WinCreateOffscreenWindow()` have their bitmaps freed; windows created using `WinCreateWindow()` or `WinCreateBitmapWindow()` do not.

The *eraseIt* parameter affects on-screen transitional or legacy windows only; off-screen windows are never erased. As a performance optimization, you might set *eraseIt* to `false` for an on-screen window if you know that you are going to immediately redraw the area anyway.

WinDisplayToWindowPt Function

Purpose Converts a screen-relative coordinate to a window-relative coordinate. The coordinate returned is relative to the display window.

Declared In `Window.h`

Prototype
```
void WinDisplayToWindowPt (Coord *x, Coord *y)
```

Parameters

↔ *x*
A pointer to x coordinate to convert.

↔ *y*
A pointer to y coordinate to convert.

Returns Nothing.

See Also `WinWindowToDisplayPt()`

WinFinishThreadUI Function

Purpose Delete the user interface context for a thread.

Declared In `Window.h`

Prototype
```
status_t WinFinishThreadUI (void)
```

Parameters None.

Returns Always returns `errNone`.

Comments This function decrements the reference count for the UI context in this thread. If the reference count becomes zero, it then deletes the reference context and all windows that have been created in the thread.

See Also `WinStartThreadUI()`

WinFlush Function

Purpose Refreshes the screen for transitional windows.

Declared In `Window.h`

Prototype
```
void WinFlush (void)
```

Parameters None.

Returns Nothing.

WinGetActiveWindow Function

Purpose Returns the window handle of the active window.

Declared In `Window.h`

Prototype
```
WinHandle WinGetActiveWindow (void)
```

Parameters None.

Returns The handle of the active window. All user input is directed to the active window.

See Also `WinSetActiveWindow()`, `WinGetDisplayWindow()`, `WinGetDrawWindow()`

WinGetBitmap Function

Purpose Returns a pointer to a window's bitmap, which holds the window contents.

Declared In `Window.h`

Prototype
```
BitmapType *WinGetBitmap (WinHandle winHandle)
```

Parameters

→ *winHandle*
The handle of window from which to get the bitmap.

Returns A pointer to the bitmap or `NULL` if *winHandle* is invalid.

Comments For on-screen windows, the bitmap returned always represents the whole screen. Thus, the top-left corner of the returned bitmap may not be the top-left corner of the window.

Compatibility For update-based windows, this function works but the returned structure does not contain any bitmap data. If you try to write to or access the bitmap data, your application will crash. It is acceptable to access the other fields of the `BitmapType`.

WinGetBitmapDimensions Function

Purpose Returns a bitmap's dimensions using the draw window's active coordinate system.

Declared In `Window.h`

Prototype
```
void WinGetBitmapDimensions
    (const BitmapType *bmP, Coord *widthP,
    Coord *heightP)
```

Parameters

→ *bmP*
A pointer to the bitmap (see `BitmapType`).

← *widthP*
The width of the bitmap.

← *heightP*
The height of the bitmap.

Returns Nothing.

See Also `BmpGetDimensions()`

WinGetBounds Function

Purpose Returns the bounds of the current draw window in display-relative coordinates.

Declared In `Window.h`

Prototype
```
void WinGetBounds (WinHandle winH,
    RectangleType *rP)
```

Parameters

→ *winH*
The handle to a window.

← *rP*
A pointer to a `RectangleType` structure.

Returns Nothing.

Comments This function returns in *rP* the bounds of the window represented by *winH*. This corresponds to the convention used by `WinSetBounds()`, because it takes a window handle as an argument.

Compatibility For update-based or transitional windows, this function returns incorrect results until the `winResizedEvent` is received. It is acceptable to call this function in response to a `frmUpdateEvent` or `winUpdateEvent` because `winResizedEvent` is sent before `winUpdateEvent`.

See Also `WinGetWindowExtent()`, `WinGetDrawWindowBounds()`

WinGetDisplayExtent Function

Purpose Returns the width and height of the application area of the screen.

Declared In `Window.h`

Prototype
```
void WinGetDisplayExtent (Coord *extentX,
    Coord *extentY)
```

Parameters

← *extentX*
A pointer to the width of the display in pixels.

← *extentY*
A pointer to the height of the display in pixels.

Returns Nothing.

Comments This function returns the dimensions of the portion of the screen that is not used by the dynamic input area or the status bar. To find out the actual screen size, call `WinScreenGetAttribute()`.

WinGetDisplayWindow Function

Purpose Returns the window handle of the display (screen) window.

Declared In `Window.h`

Prototype

```
WinHandle WinGetDisplayWindow (void)
```

Parameters None.

Returns The handle of display window.

Comments The display window is created by the system at start-up; it has the same size as the Palm OS drawable area of the physical display (screen).

See Also WinGetDisplayExtent(), WinGetActiveWindow(), WinGetDrawWindow()

WinGetDrawWindow Function

Purpose Returns the window handle of the current draw window.

Declared In `Window.h`

Prototype

```
WinHandle WinGetDrawWindow (void)
```

Parameters None.

Returns The handle of the draw window.

See Also WinGetDisplayWindow(), WinGetActiveWindow(), WinSetDrawWindow()

WinGetDrawWindowBounds Function

Purpose Returns the bounds of the draw window.

Declared In `Window.h`

Prototype
```
void WinGetDrawWindowBounds (RectangleType *rP)
```

Parameters

← *rP*
A pointer to a `RectangleType` structure specifying the window bounds.

Returns Nothing.

Comments This function is equivalent to `WinGetBounds()`.

Compatibility For update-based or transitional windows, this function returns incorrect results until the `winResizedEvent` is received. It is acceptable to call this function in response to a `frmUpdateEvent` or `winUpdateEvent` because `winResizedEvent` is sent before `winUpdateEvent`.

WinGetFrameType Function

Purpose Gets the frame type for a specified window.

Declared In `Window.h`

Prototype
```
FrameType WinGetFrameType (const WinHandle winH)
```

Parameters

→ *winH*
The window's handle.

Returns A `FrameType` value indicating the window's frame style.

See Also `WinSetFrameType()`

WinGetFramesRectangle Function

Purpose Returns the rectangle that includes a rectangle together with the specified frame around it.

Declared In `Window.h`

Prototype

```
void WinGetFramesRectangle (FrameType frame,
   const RectangleType *rP,
   RectangleType *obscuredRect)
```

Parameters

→ *frame*
The type of rectangle frame (see FrameType).

→ *rP*
A pointer to the rectangle to frame (see RectangleType).

← *obscuredRect*
A pointer to the rectangle that includes both the specified rectangle and its frame.

Returns Nothing.

Comments Frames are always drawn around (outside) a rectangle.

See Also WinGetWindowFrameRect(), WinGetBounds()

WinGetPixel Function

Purpose Returns the color value of a pixel in the current draw window.

Declared In `Window.h`

Prototype

```
IndexedColorType WinGetPixel (Coord x, Coord y)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

Returns The indexed color value of the pixel. See IndexedColorType. A return value of 0 means that the window is on-screen, that the coordinates do not lie in the current draw window, or that they do and the color of that pixel is index 0 (typically white).

Compatibility This function only works if the active window is off-screen, is a legacy window, or is a transitional window.

See Also WinIndexToRGB()

WinGetPixelRGB Function

Purpose Returns the RGB color values of a pixel in the current draw window.

Declared In Window.h

Prototype

```
status_t WinGetPixelRGB (Coord x, Coord y,
   RGBColorType *rgpP)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

← *rgbP*
The RGB color components of the pixel.

Returns One of the following:

`errNone`
Success.

`sysErrParamErr`
When the draw window is on-screen, when the *x* or *y* arguments are < 0, or when they are outside the bounds of the draw window.

Comments This function only works if the draw window is off-screen.

The RGB color values of the pixel are returned as an RGBColorType. This function can be used with both indexed or direct color modes. A return value of `sysErrParamErr` means that the coordinates do not lie within the current draw window.

WinGetSupportedDensity Function

Purpose Enumerates the various display densities supported by the rendering system.

Declared In `Window.h`

Prototype

```
status_t WinGetSupportedDensity
    (uint16_t *densityP)
```

Parameters

↔ *densityP*
: A pointer to a supported density value. Set this value to zero before calling this function for the first time. Subsequent calls cause this value to be set to one of the display densities supported by the handheld.

Returns One of the following:

`errNone`
: Success.

`sysErrParamErr`
: The value you supplied in `*densityP` isn't a supported density and isn't zero.

Comments Initialize `*densityP` to zero before your application calls this function for the first time. Repeated calls to `WinGetSupportedDensity()` will cause the value pointed to by *densityP* to change; these values represent the supported display densities, in order from low to high density. After the last supported density value, this function sets `*densityP` back to zero.

NOTE: The densities reported by this function are those that are supported by the rendering system. These densities are not necessarily supported by the underlying hardware. A device with a low-density screen that is able to scale high-density bitmaps will report that it can handle both high and low density bitmaps. Use `WinScreenGetAttribute()` to determine the density of the handheld's screen.

Density values are defined in `CmnBitmapTypes.h`; see the `DensityType` enum. Only those values supported by a given device will be returned by `WinGetSupportedDensity()`. For example, on a device with a double-density display this function returns `kDensityLow`, followed by `kDensityDouble`, followed

by 0. For each supported density, the inverse scaling factor is supported. In this example, the rendering system supports pixel-doubling low-density data for a double-density destination, and the rendering system supports pixel-halving high-density data for a low-density destination.

The value pointed to by *densityP* should only be zero or one of the density values supported by the handheld. If it has any other value when you call `WinGetSupportedDensity()`, this function will simply return `sysErrParamErr`.

WinGetWindowBounds Macro

Purpose Calls `WinGetDrawWindowBounds()`.

Declared In `Window.h`

Prototype
```
#define WinGetWindowBounds (rP)
```

Parameters

← *rP*
A pointer to a `RectangleType` structure specifying the window bounds.

Returns Nothing.

WinGetWindowExtent Function

Purpose Returns the width and height of the current draw window.

Declared In `Window.h`

Prototype
```
void WinGetWindowExtent (Coord *extentX,
   Coord *extentY)
```

Parameters

← *extentX*
A pointer to the width in pixels of the draw window.

← *extentY*
A pointer to the height in pixels of the draw window.

Returns Nothing.

Compatibility For update-based or transitional windows, this function returns incorrect results until the `winResizedEvent` is received. It is acceptable to call this function in response to a `frmUpdateEvent`

or `winUpdateEvent` because `winResizedEvent` is sent before `winUpdateEvent`.

See Also `WinGetBounds()`, `WinGetWindowFrameRect()`

WinGetWindowFlags Function

Purpose Returns the flags used when the window was created.

Declared In `Window.h`

Prototype
```
WinFlagsType WinGetWindowFlags
    (WinHandle winHandle)
```

Parameters

→ *winHandle*
The handle to the window.

Returns An integer containing the `WinFlagsType` flags that were set when this window was created.

See Also `WinCreateWindowWithConstraints()`

WinGetWindowFrameRect Function

Purpose Returns a rectangle, in display-relative coordinates, that defines the size and location of a window and its frame.

Declared In `Window.h`

Prototype
```
void WinGetWindowFrameRect (WinHandle winHandle,
    RectangleType *r)
```

Parameters

→ *winHandle*
The handle to the window whose coordinates are desired.

← *r*
A pointer to the coordinates of the window (see `RectangleType`).

Returns Nothing.

Compatibility For update-based or transitional windows, this function returns incorrect results until the `winResizedEvent` is received. It is acceptable to call this function in response to a `frmUpdateEvent`

or `winUpdateEvent` because `winResizedEvent` is sent before `winUpdateEvent`.

See Also `WinGetBounds()`

WinIndexToRGB Function

Purpose Converts an index in the currently active color table to an RGB value.

Declared In `Window.h`

Prototype

```
void WinIndexToRGB (IndexedColorType i,
    RGBColorType *rgbP)
```

Parameters

→ *i*
A color index value. See `IndexedColorType`.

← *rgbP*
A pointer to an RGB color value corresponding to the index value *i*. See `RGBColorType`.

Returns Nothing.

See Also `WinRGBToIndex()`

WinInvalidateRect Function

Purpose Requests an update for a portion of a window.

Declared In `Window.h`

Prototype

```
void WinInvalidateRect (WinHandle window,
    const RectangleType *dirtyRect)
```

Parameters

→ *window*
A handle to an on-screen window.

→ *dirtyRect*
The bounds given in active coordinates of the area that needs to be redrawn (see `RectangleType`).

Returns Nothing.

Comments This function generates a `winUpdateEvent`. If the window is update-based, the *dirtyRect* is added to a pending update queue.

All updates in this queue are sent during a single `winUpdateEvent` the next time through the event loop.

If the window is a transitional window, the update event for *`dirtyRect`* is enqueued immediately.

If the window is off-screen, this function has no effect.

See Also `WinInvalidateRectFunc()`, `WinInvalidateWindow()`

WinInvalidateRectFunc Function

Purpose Requests an update for a portion of the window, specifying a function that can handle the update if it is feasible.

Declared In `Window.h`

Prototype

```
void WinInvalidateRectFunc (WinHandle window,
   const RectangleType *dirtyRect,
   winInvalidateFunc func, void *state)
```

Parameters

→ *`window`*
A handle to the window.

→ *`dirtyRect`*
The bounds given in active coordinates of the area that needs to be redrawn (see `RectangleType`).

→ *`func`*
A `winInvalidateFunc()` callback function. The system calls this function to perform the redraw.

→ *`state`*
Any data that needs to be passed to the *`func`*.

Returns Nothing.

Comments This function is an optimization on invalidating a window that saves the application from having to wait for a `frmUpdateEvent` in certain circumstances. The supplied `winInvalidateFunc()` is called instead of performing an update under the following conditions:

- The window is a transitional window. In this case, `winInvalidateFunc()` is called directly from `WinInvalidateRectFunc()`.

- If this is the only update in the pending update queue the next time through the event loop, the Window Manager calls `winInvalidateFunc()` directly instead of generating a `winUpdateEvent`.

If any other portion of the window is dirty for an update-based window, then the `winUpdateEvent` is generated as usual.

See Also `WinInvalidateRect()`, `WinInvalidateWindow()`

WinInvalidateWindow Function

Purpose Requests that an entire window be redrawn.

Declared In `Window.h`

Prototype
```
void WinInvalidateWindow (WinHandle window)
```

Parameters

→ *window*
The window to be redrawn.

Returns Nothing.

Comments This function generates a `winUpdateEvent` for the window. If the window is update-based, it is added to a pending update queue. All updates in this queue are sent during a single `winUpdateEvent` the next time through the event loop.

If the window is a transitional window, the update event is enqueued immediately.

If the window is off-screen, this function has no effect.

See Also `WinInvalidateRect()`, `WinInvalidateRectFunc()`

WinModal Function

Purpose Returns `true` if the specified window is modal.

Declared In `Window.h`

Prototype
```
Boolean WinModal (WinHandle winHandle)
```

Parameters

→ *winHandle*
The handle of a window.

Returns `true` if the window is modal, otherwise `false`.

See Also `FrmAlert()`, `FrmCustomAlert()`, `FrmDoDialog()`

WinPalette Function

Purpose Sets or retrieves the palette for the draw window.

Declared In `Window.h`

Prototype

```
status_t WinPalette (uint8_t operation,
   int16_t startIndex, uint16_t paletteEntries,
   RGBColorType *tableP)
```

Parameters

→ *operation*

One of the following values:

`winPaletteGet`
: Retrieve the palette. Entries are read from the palette beginning at *startIndex* and placed into *tableP* beginning at index 0.

`winPaletteSet`
: Set the palette. Entries from *tableP* (beginning at index 0) are set into the palette beginning at *startIndex* in the palette. Use only for off-screen windows.

`winPaletteSetToDefault`
: Set the palette to the default system palette. Use only for off-screen windows.

→ *startIndex*

Identifies where in the palette to start reading or writing. Specify `WinUseTableIndexes` to indicate that the entries are not to be set or read sequentially; instead, the `index` value in each `RGBColorType` entry in *tableP* determines which slot in the palette is to be set or read. You can use this technique to get or set several discontiguous palette entries with a single function call.

→ *paletteEntries*

The number of palette entries to get or set.

↔ *tableP*
: A pointer to a buffer of RGBColorType entries that is either read from or written to, depending on the *operation* parameter; the table entries from 0 to *paletteEntries* – 1 are affected by this routine.

Returns One of the following values:

`errNone`
: Success.

`winErrPalette`
: The current draw window does not have a color table, a set operation has overflowed the color table, or one of the entries in *tableP* has an invalid index value

`sysErrParamErr`
: The *startIndex* value is invalid.

Comments Here are some examples of how this routine works:

- If *startIndex* is 0 and *paletteEntries* is 10, the first 10 elements of the palette will be set from *tableP* or will be copied into *tableP*.
- If *startIndex* is 10 and *paletteEntries* is 5, then entries 10, 11, 12, 13, and 14 in the palette will be set from or copied to elements 0, 1, 2, 3, and 4 in *tableP*.
- If *startIndex* is `WinUseTableIndexes` and *paletteEntries* is 1, then the index value in the `RGBColorType` of element 0 of *tableP* will be read from or copied to *tableP*; in this case, the *index* field of the `RGBColorType` will not change.

One use for this function is if you need to display a bitmap that uses a color table other than the one in use by the system. You can attach a custom color table to a bitmap, and if you do, the bitmap is drawn using that color table. However, this is a performance drain. As an optimization, you can use `WinPalette()` to change the system color table to match that used by the bitmap, display the bitmap, and use `WinPalette()` to reset the color table when the bitmap is no longer visible.

When the palette is changed, this function broadcasts the `sysNotifyDisplayChangeEvent` to notify any interested observer that the color palette has changed.

Palette changes affect only future drawing. The colors of elements already on the screen do not change. It is strongly recommended that you set the palette before you do any drawing.

Compatibility Earlier releases of Palm OS supported 1, 2, 4, and 8 bit per pixel grayscale and 8 and 16-bit color at the hardware level. Palm OS Cobalt removes hardware support for all but 16-bit color. All previously supported color depths are emulated for compatibility.

TIP: If you previously set your application to run in 8-bit color on a 16-bit device for better performance, you will now experience worse performance because 8-bit color is emulated.

WinRequestFocus Function

Purpose Asks that the input focus be given to the specified window.

Declared In `Window.h`

Prototype

```
status_t WinRequestFocus (WinHandle winHandle,
   uint32_t flags)
```

Parameters

→ *`winHandle`*
The window that wants the focus.

→ *`flags`*
Currently unused.

Returns One of the following values:

`errNone`
Success.

`winErrInvalidWindowHandle`
`winHandle` is not a valid window.

Comments The system satisfies the request asynchronously. At some point in the future, the application receives a `winFocusGainedEvent` to inform you that you have been granted focus. The application might not be granted input focus if a window in a higher layer within another thread requests input focus at the same time.

The input focus can only be set for focusable windows. Menus, pop--up lists, and the input area do not use focusable windows.

See Also WinSetActiveWindow()

WinRGBToIndex Function

Purpose Converts an RGB value to the index of the closest color in the currently active color lookup table (CLUT).

Declared In Window.h

Prototype

```
IndexedColorType WinRGBToIndex
    (const RGBColorType *rgbP)
```

Parameters

→ *rgbP*
A pointer to an RGB color value.

Returns The index of the closest matching color in the CLUT.

Comments The number of colors supported is hardware-dependent. The number of possible RGB colors may exceed the number of supported colors. For this reason, an exact match may not be available for *rgbP*. If there is no exact RGB match, then a luminance best-fit is used if the color lookup table is entirely gray scale (red, green, and blue values for each entry are identical), or a shortest-distance fit in RGB space is used if the palette contains colors. RGB shortest distance may not always produce the actual closest perceptible color, but it's relatively fast and works for the system palette.

`WinRGBToIndex()` uses the draw window's color table to return the appropriate color table index. If the draw window does not have a color table, the default color table of the current screen is used.

If the draw window does not have a color table, and if the depth of the draw window and the depth of the screen are different, this function will return an inappropriate index. If this situation exists, the application should either define a color table for the draw window, or use WinScreenMode() to set the screen depth to the same depth as the draw window before calling `WinRGBToIndex()`.

NOTE: The bitmap data will not be blitted properly if the depth of the screen is changed using `WinScreenMode()` and the new window uses a bitmap that does not define the bitmap's color table. See `WinScreenMode()` for information on how to work around this limitation.

See Also `WinIndexToRGB()`, `WinScreenMode()`

WinScaleCoord Function

Purpose Converts a single coordinate from the standard coordinate system to the active coordinate system.

Declared In `Window.h`

Prototype

```
Coord WinScaleCoord (Coord coord,
    Boolean ceiling)
```

Parameters

→ *coord*
A coordinate in the standard coordinate system.

→ *ceiling*
Pass `true` to round up, `false` to truncate the fractional part when scaling.

Returns The coordinate scaled to the active coordinate system.

Comments This function converts a coordinate by multiplying it by the coordinate scaling factor. What happens next depends upon the value of *ceiling*:

if *ceiling* == true {

use `lfloorf()` function when scaleFactor > 1

use `lceilf()` function when scaleFactor < 1

else

use `lceilf()` function when scaleFactor > 1

use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

If the active coordinate system is `kCoordinatesStandard`, the returned coordinate is equal to the supplied coordinate.

See Also `WinScalePoint()`, `WinScaleRectangle()`, `WinUnscaleCoord()`, `WinScaleCoordNativeToActive()`

WinScaleCoordNativeToActive Function

Purpose Converts a single coordinate from the device's native coordinate system to the active coordinate system.

Declared In `Window.h`

Prototype

```
Coord WinScaleCoordNativeToActive (Coord coord,
    Boolean ceiling)
```

Parameters

→ *coord*
A coordinate in the native coordinate system.

→ *ceiling*
Pass `true` to round up, `false` to truncate the fractional part when scaling.

Returns The coordinate scaled to the active coordinate system.

Comments This function converts a coordinate by multiplying it by the coordinate scaling factor. What happens next depends upon the value of *ceiling*:

if *ceiling* == true {

use `lfloorf()` function when scaleFactor > 1

use `lceilf()` function when scaleFactor < 1

else

use `lceilf()` function when scaleFactor > 1

use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

If the active coordinate system is `kCoordinatesNative`, the returned coordinate is equal to the supplied coordinate.

See Also `WinScaleCoord()`

WinScalePoint Function

Purpose Converts a point from the standard coordinate system to the active coordinate system.

Declared In `Window.h`

Prototype

```
void WinScalePoint (PointType *pointP,
    Boolean ceiling)
```

Parameters

↔ *pointP*
: A pointer to a `PointType` structure that, before the call, should contain a point's standard coordinate system coordinates. After this function is called, the `PointType` structure contains the coordinates of the point scaled to the active coordinate system.

→ *ceiling*
: Pass `true` to round up, `false` to truncate the fractional part when scaling.

Returns Nothing.

Comments This function converts a point by multiplying its x and y coordinates by the coordinate scaling factor. What happens next depends upon the value of *ceiling*:

if *ceiling* == true {

use `lfloorf()` function when scaleFactor > 1

use `lceilf()` function when scaleFactor < 1

else

use `lceilf()` function when scaleFactor > 1

use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

If the active coordinate system is `kCoordinatesStandard`, `pointP` is not changed by this function.

See Also `WinConvertPoint()`, `WinScaleCoord()`, `WinScaleRectangle()`, `WinUnscalePoint()`

WinScaleRectangle Function

Purpose Converts a rectangle from the standard coordinate system to the active coordinate system.

Declared In `Window.h`

Prototype

```
void WinScaleRectangle (RectangleType *rectP)
```

Parameters

↔ *rectP*

A pointer to a `RectangleType` structure that, before the call, should contain a rectangle's standard coordinate system coordinates. After this function is called the `RectangleType` structure contains the coordinates of the rectangle scaled to the active coordinate system.

Returns Nothing. The coordinates of the rectangle indicated by *rectP* are converted to the native coordinate system.

Comments This function scales the rectangle's `topLeft` and `extent` points by multiplying their `x` and `y` coordinates by the coordinate scaling factor. All values are then truncated, but if either `topLeft.x` or `extent.x` had a fractional part, `extent.x` is incremented by 1 (and, similarly, if either `topLeft.y` or `extent.y` had a fractional part, `extent.y` is incremented by 1).

If the active coordinate system is `kCoordinatesStandard`, *rectP* is not changed by this function.

You can use this function when your gadget handler draws using a more precise coordinate system than the Form Manager and needs to convert the form-based bounds of the gadget to the high-density bounds used by the gadget's drawing function.

See Also `WinScaleCoord()`, `WinScalePoint()`, `WinUnscaleRectangle()`

WinScreenGetAttribute Function

Purpose Obtains various attributes of the screen.

Declared In `Window.h`

Prototype

```
status_t WinScreenGetAttribute
    (WinScreenAttrType selector, uint32_t *attrP)
```

Parameters

→ *selector*

One of the following:

`winScreenWidth`
: The width of the screen, in pixels.

`winScreenHeight`
: The height of the screen, in pixels.

`winScreenRowBytes`
: The number of bytes used by each row in the screen buffer.

`winScreenDepth`
: The screen depth.

`winScreenAllDepths`
: All screen depths (in bitmap format).

`winScreenDensity`
: The screen bitmap's density.

`winScreenPixelFormat`
: The `PixelFormatType` appropriate for the screen.

`winScreenResolutionX`
: The number of pixels per inch along the screen's x axis.

`winScreenResolutionY`
: The number of pixels per inch along the screen's y axis.

← *attrP*

A pointer to a `uint32_t` into which the specified attribute value is placed by this function.

Returns One of the following values:

`errNone`
: Success.

`sysErrParamErr`
: *`selector`* doesn't represent a screen attribute.

Comments This function returns many of the attributes that can be obtained with `WinScreenMode()`. Unlike `WinScreenMode()`, however, this function can also return the number of bytes used by each row in the screen buffer as well as the number of pixels per inch on the screen's x and y axes.

Unlike `WinScreenMode()`, you cannot set any attributes with this function. Also, you cannot use this function to obtain the "color enabled" attribute. And unlike `WinScreenMode()`, this function always returns the true screen dimensions; `WinScreenMode()` converts the dimensions to the active coordinate system.

Applications can use the screen resolution information to make intelligent decisions about how to draw primitives on Palm Powered devices with different screen resolutions.

WinScreenLock Function

Purpose "Locks" the current screen by switching the UI concept of the screen base address to an area that is not reflected on the display.

Declared In `Window.h`

Prototype
```
uint8_t *WinScreenLock (WinLockInitType initMode)
```

Parameters → *`initMode`*
Indicates how to initialize the new screen area. Specify one of the following values:

`winLockCopy`
: Copy old screen to new.

`winLockErase`
: Erase new screen to white.

`winLockDontCare`
: Don't do anything

Returns A pointer to the new screen base address, or `NULL` if this routine fails.

Comments This routine can be used to "freeze" the display while doing lengthy drawing operations to avoid a flickering effect. Call

WinScreenUnlock() to unlock the display and cause it to be updated with any changes. The screen must be unlocked as many times as it is locked to actually update the display.

Because this function copies the screen, using it is a relatively expensive operation.

Compatibility You can only call this function on a legacy or transitional window. Calling this function when the active window is update-based has no effect.

WinScreenMode Function

Purpose Sets or returns display parameters, including display geometry, bit depth, and color support.

Declared In `Window.h`

Prototype

```
status_t WinScreenMode
    (WinScreenModeOperation operation,
    uint32_t *widthP, uint32_t *heightP,
    uint32_t *depthP, Boolean *enableColorP)
```

Parameters

→ *operation*

The work this function is to perform, as specified by one of the following selectors:

`winScreenModeGet`
: Return the current settings for the display.

`winScreenModeGetDefaults`
: Return the default settings for the display.

`winScreenModeGetSupportedDepths`
: Return in *depthP* a hexadecimal value indicating the supported screen depths. The binary representation of this value defines a bit field in which the value 1 indicates support for a particular display depth. The position representing a particular bit depth corresponds to the value $2^{(bitDepth-1)}$. See the Example at the end of this function description for more information.

`winScreenModeGetSupportsColor`
Return `true` as the value of the *`enableColorP`* parameter when color mode can be enabled.

↔ *`widthP`*
A pointer to new/old screen width. For backward compatibility, when *`operation`* is `winScreenModeGet` or `winScreenModeGetDefaults`, a single-density width is returned, even if the handheld has a double-density display. Use `WinScreenGetAttribute()` to retrieve the true hardware dimensions of the display.

↔ *`heightP`*
A pointer to new/old screen height. For backward compatibility, when *`operation`* is `winScreenModeGet` or `winScreenModeGetDefaults`, a single-density height is returned, even if the handheld has a double-density display. Use `WinScreenGetAttribute()` to retrieve the true hardware dimensions of the display.

↔ *`depthP`*
A pointer to new/old/available screen depth.

↔ *`enableColorP`*
Pass `true` to enable color drawing mode. The returned value (when using an operation that returns a value through this parameter) simply indicates whether or not the hardware supports color; its value does not change based on the current screen depth.

Returns One of the following values:

`errNone`
Success.

`sysErrParamErr`
An invalid argument was specified.

`memErrNotEnoughSpace`
There is not enough memory to perform the requested operation.

Comments The *`widthP`*, *`heightP`*, *`depthP`*, and *`enableColorP`* parameters are used in different ways for different operations. All "get" operations overwrite these values with a result when the function returns. The `winScreenModeSet` operation changes current display parameters when passed valid argument values that are not

NULL pointers. The winScreenModeSetToDefaults operation ignores values passed for all of these parameters.

Table 36.1 summarizes parameter usage for each operation this function performs.

Table 36.1 Use of parameters to WinScreenMode function

Operation winScreenMode…	widthP	heightP	depthP	enableColorP
…Get	returned	returned	returned	returned
…GetDefaults	returned	returned	returned	returned
…GetSupportedDepths	ignored	ignored	returned	pass in
…GetSupportsColor	ignored	ignored	pass in	returned
…Set	pass in	pass in	pass in	pass in
…SetToDefaults	ignored	ignored	ignored	ignored

This function ignores NULL pointer arguments to the *widthP*, *heightP*, *depthP*, and *enableColorP* parameters; thus, you can pass a NULL pointer for any of these values to leave the current value unchanged. Similarly, when getting values, this function does not return a value for any NULL pointer argument.

If you change the display depth, it is recommended that you restore it to its previous state when your application closes, even though the system sets display parameters back to their default values when launching an application.

Avoid bit depths of 1, 2, or 4 bits per pixel. They are supported, but they cause rendering to be much slower on Palm OS Cobalt.

Note that none of the other operations interprets the depth parameter the same way that winScreenModeGetSupportedDepths does. For example, to set the display depth to 8-bit mode, you use 8 (decimal) for the display depth, not 0x80 (128 decimal).

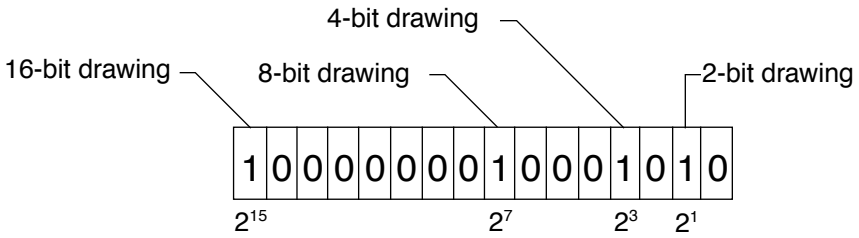
When a window is created, and if the window's associated bitmap does not have its own color table, the window will use the system's default color translation tables when a drawing operation occurs to that window. When the system's bit depth changes, the system's

default color translation tables are recalculated based on the new screen depth. When the blit occurs at the new screen depth to the off-screen window, the color translation tables are out of sync.

To work around this system limitation, developers should change the bit depth first, then create any off-screen windows.

Example Here are some additional examples of return values provided by the `winScreenModeGetSupportedDepths` mode of the `WinScreenMode()` function.

This function indicates support for 4-bit drawing by returning a value of `0x08`, or 2^3, which corresponds to a binary value of `1000`. Support for bit depths of 2 and 1 is indicated by a return value of `0x03`. Support for bit depths of 4, 2, and 1 is indicated by `0x0B`, which is a binary value of `1011`. Support for bit depths of 16, 8, 4 and 2 is indicated by `0x808A`. The figure immediately following depicts this final example graphically.



Compatibility In earlier releases of Palm OS, this function supported two more operations: `winScreenModeSet` and `winScreenModeSetToDefaults`. In Palm OS Cobalt, `WinScreenMode()` does not support these two operations.

See Also `WinScreenGetAttribute()`

WinScreenUnlock Function

Purpose Unlocks the screen and updates the display.

Declared In `Window.h`

Prototype

```
void WinScreenUnlock (void)
```

Parameters None.

Returns Nothing.

Comments The screen must be unlocked as many times as it is locked to actually update the display.

Compatibility You can only call this function on a legacy or transitional window. Calling this function when the active window is update-based has no effect.

See Also `WinScreenLock()`

WinScrollRectangle Function

Purpose Scrolls a rectangle in the draw window.

Declared In `Window.h`

Prototype

```
void WinScrollRectangle (const RectangleType *rP,
   WinDirectionType direction, Coord distance,
   RectangleType *vacatedP)
```

Parameters

→ *rP*
A pointer to the rectangle to scroll.

→ *direction*
The direction to scroll (`winUp`, `winDown`, `winLeft`, or `winRight`).

→ *distance*
The distance to scroll in pixels.

← *vacatedP*
A pointer to the rectangle that needs to be redrawn because it has been vacated as a result of the scroll.

Returns Nothing.

Comments The rectangle scrolls within its own bounds. Any portion of the rectangle that is scrolled outside its bounds is clipped.

Compatibility Use this function only in legacy applications that contain only legacy windows. Palm OS Cobalt native applications should use `WinScrollRectangleAsync()`.

WinScrollRectangleAsync Function

Purpose Scrolls a window by generating an update event.

Declared In `Window.h`

Prototype

```
status_t WinScrollRectangleAsync
   (WinHandle winHandle, const RectangleType *rP,
   Coord dx, Coord dy)
```

Parameters

→ *winHandle*
The window to scroll.

→ *rP*
A pointer to the rectangle to scroll.

→ *dx*
The amount by which to scroll horizontally. A negative value means scroll to the left.

→ *dy*
The amount by which to scroll vertically. A negative amount means scroll to the top.

Returns One of the following values:

`errNone`
Success.

`winErrInvalidWindowHandle`
winHandle is not a valid window.

Comments This function differs from `WinScrollRectangle()` not only by how you specify the direction to scroll, but also by how scrolling occurs. This function marks the area as invalid and then triggers a `winUpdateEvent` with the region that needs to be redrawn.

For off-screen windows, this merely performs a copy.

WinSetActiveWindow Function

Purpose Makes a window the active window.

Declared In `Window.h`

Prototype

```
void WinSetActiveWindow (WinHandle winHandle)
```

Parameters

→ *winHandle*
The handle of a window.

Returns Nothing.

Comments The active window is not actually set in this routine; flags are set to indicate that a window is being exited and another window is being entered, which generates the following sequence of events:

- A `winExitEvent` specifying the current window as the exit window and *`winHandle`* as the new window.
- A `winEnterEvent` specifying the current window as the exit window and *`winHandle`* as the new window.
- The window is enabled.
- If *`winHandle`* should be able to receive `keyDownEvents`, input focus is requested for *`winHandle`*.

  At some point later on, if the system grants input focus to *`winHandle`*, the `winFocusLostEvent` is sent to the event queue of the thread that contains the window that currently has input focus. Then a `winFocusGainedEvent` specifying *`winHandle`* are is sent to the current thread's event queue.

All user input is directed to the active window.

See Also `WinGetActiveWindow()`, `EvtGetEvent()`, `WinSetDrawWindow()`, `WinRequestFocus()`

WinSetBounds Function

Purpose Sets the bounds of the window to display-relative coordinates.

Declared In `Window.h`

Prototype

```
void WinSetBounds (WinHandle winHandle,
   const RectangleType *rP)
```

Parameters

→ *`winHandle`*
A handle for the window for which to set the bounds.

→ *`rP`*
A pointer to a rectangle to use for bounds.

Returns Nothing.

Compatibility Visible windows and transitional or update-based windows cannot have their bounds modified. You can only modify the bounds of a legacy or an off-screen window. For update-based or transitional windows, you specify the size requirements (or constraints) when

you create the window. The system determines what size to make it, and then sends the winResizedEvent.

See Also WinGetBounds()

WinSetConstraints Function

Purpose Sets a window's constraints.

Declared In Window.h

Prototype
```
status_t WinSetConstraints (WinHandle winHandle,
    const WinConstraintsType *constraints)
```

Parameters

→ *winHandle*
A handle to the window.

→ *constraints*
A WinConstraintsType structure specifying the position and minimum, preferred, and maximum sizes for the window.

Returns Always returns errNone.

Comments You can use this to change a window's size constraints. Calling this function causes the Window Manager to re-evaluate the window's size. Eventually, a winResizedEvent and a winUpdateEvent are generated.

See Also WinCreateWindowWithConstraints()

WinSetDrawWindow Function

Purpose Sets the draw window. (All drawing operations are relative to the draw window.)

Declared In Window.h

Prototype
```
WinHandle WinSetDrawWindow (WinHandle winHandle)
```

Parameters

→ *winHandle*
The handle of a window.

Returns The previous draw window.

Comments Do not call this function in response to a `winUpdateEvent`. You cannot change the draw window during an update.

See Also `WinGetDrawWindow()`, `WinSetActiveWindow()`

WinSetWindowBounds Macro

Purpose Calls `WinSetBounds()`.

Declared In `Window.h`

Prototype
```
#define WinSetWindowBounds (winH, rP)
```

Parameters

→ *winHandle*
A `WinHandle` for the window for which to set the bounds.

→ *rP*
A pointer to a `RectangleType` to use for bounds.

Returns Nothing.

WinStartThreadUI Function

Purpose Starts the user interface context for a thread.

Declared In `Window.h`

Prototype
```
status_t WinStartThreadUI (void)
```

Parameters None.

Returns `errNone` upon success, or `sysErrNoFreeRAM` if an error occurs.

Comments Call this function from a thread spawned in your program if you want that thread to display a user interface. All of the windows that you create to display inside of this thread must be updated-based windows.

Nested calls to `WinStartThreadUI()` are allowed.

See Also `WinFinishThreadUI()`

WinUnscaleCoord Function

Purpose Converts a single coordinate from the active coordinate system to the standard coordinate system.

Declared In `Window.h`

Prototype

```
Coord WinUnscaleCoord (Coord coord,
    Boolean ceiling)
```

Parameters

→ *coord*
A coordinate in the active coordinate system.

→ *ceiling*
Pass `true` to round up, `false` to truncate the fractional part when scaling.

Returns The coordinate scaled to the standard coordinate system.

Comments This function converts a coordinate by dividing it by the coordinate scaling factor. What happens next depends upon the value of `ceiling`:

if `ceiling` == true {

use `lfloorf()` function when scaleFactor > 1

use `lceilf()` function when scaleFactor < 1

else

use `lceilf()` function when scaleFactor > 1

use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

If the active coordinate system is `kCoordinatesStandard`, the returned coordinate is equal to the supplied coordinate.

See Also `WinScaleCoord()`, `WinUnscalePoint()`, `WinUnscaleRectangle()`

WinUnscalePoint Function

Purpose Converts a point from the active coordinate system to the standard coordinate system.

Declared In `Window.h`

Prototype

```
void WinUnscalePoint (PointType *pointP,
    Boolean ceiling)
```

Parameters

↔ *pointP*
: A pointer to a `PointType` structure that, before the call, should contain a point's coordinates using the active coordinate system. After this function is called the `PointType` structure contains the coordinates of the point scaled to the standard coordinate system.

→ *ceiling*
: Pass `true` to round up, `false` to truncate the fractional part when scaling.

Returns Nothing. The coordinates of the point indicated by *pointP* are converted to the standard coordinate system.

Comments This function converts a point by dividing its x and y coordinates by the coordinate scaling factor. What happens next depends upon the value of `ceiling`:

if `ceiling` == true {

  use `lfloorf()` function when scaleFactor > 1

  use `lceilf()` function when scaleFactor < 1

else

  use `lceilf()` function when scaleFactor > 1

  use `lfloorf()` function when scaleFactor < 1

}

The objective of this algorithm is to ensure that for any `Coord`, `Point`, or `Rectangle` "x", x = unscaled(scaled(x)).

If the active coordinate system is `kCoordinatesStandard`, *pointP* is not changed by this function.

See Also `WinScalePoint()`, `WinUnscaleCoord()`, `WinUnscaleRectangle()`

WinUnscaleRectangle Function

Purpose Converts a rectangle from the active coordinate system to the standard coordinate system.

Declared In `Window.h`

Prototype
```
void WinUnscaleRectangle (RectangleType *rectP)
```

Parameters

↔ *rectP*

A pointer to a `RectangleType` structure that, before the call, should contain a rectangle's coordinates using the active coordinate system. After this function is called the `RectangleType` structure contains the coordinates of the rectangle scaled to the standard coordinate system.

Returns Nothing. The coordinates of the rectangle indicated by *rectP* are converted to the standard coordinate system.

Comments This function scales the rectangle's `topLeft` and `extent` points by dividing their `x` and `y` coordinates by the coordinate scaling factor. All values are then truncated, but if either `topLeft.x` or `extent.x` had a fractional part, `extent.x` is incremented by 1 (and, similarly, if either `topLeft.y` or `extent.y` had a fractional part, `extent.y` is incremented by 1).

If the active coordinate system is `kCoordinatesStandard`, *rectP* is not changed by this function.

See Also `WinScaleRectangle()`, `WinUnscaleCoord()`, `WinUnscalePoint()`

WinValidateHandle Function

Purpose Validates a window handle.

Declared In `Window.h`

Prototype
```
Boolean WinValidateHandle (WinHandle winHandle)
```

Parameters

→ *winHandle*

The handle to be tested.

Returns `true` if the specified handle references a window that exists, `false` otherwise.

Comments For debugging purposes only. Do not include this function in commercial applications.

See Also `FrmValidatePtr()`, `FrmRemoveObject()`

WinWindowToDisplayPt Function

Purpose Converts a window-relative coordinate to a display-relative coordinate.

Declared In `Window.h`

Prototype
```
void WinWindowToDisplayPt (Coord *x, Coord *y)
```

Parameters

↔ *extentX*
A pointer to x coordinate to convert.

↔ *extentY*
A pointer to y coordinate to convert.

Returns Nothing.

Comments The coordinate passed is assumed to be relative to the draw window.

See Also `WinDisplayToWindowPt()`

Window Drawing Functions and Macros

This section lists and describes drawing-related functions implemented in the Window Manager. All of these functions work as expected in Palm OS Cobalt, but they are considered obsolete and replaced by the "Graphics Context Reference" drawing functions. The two types of drawing functions maintain separate draw states and should not be mixed. Use either the `Win...` drawing functions or the `Gc...` drawing functions.

WinClipRectangle Function

Purpose Truncates the rectangle to make it fit within the clipping region of the current draw window.

Declared In `Window.h`

Prototype
```
void WinClipRectangle (RectangleType *rP)
```

Parameters

↔ *rP*

A pointer to a `RectangleType` structure holding the rectangle to clip. The rectangle returned is the intersection of the rectangle passed and the clipping bounds of the draw window.

Returns Nothing.

Comments This function does not change the clipping rectangle of the window. To modify the window's clipping rectangle, use the `WinSetClip()` and `WinResetClip()` functions.

The draw window is the window to which all drawing functions send their output. It is returned by `WinGetDrawWindow()`.

Compatibility Do not use this function if you are using the graphics context APIs.

See Also `WinCopyRectangle()`, `WinDrawRectangle()`, `WinEraseRectangle()`, `WinGetClip()`

WinCopyRectangle Function

Purpose Copies a rectangular region from one place to another (either between windows or within a single window).

Declared In `Window.h`

Prototype
```
void WinCopyRectangle (WinHandle srcWin,
    WinHandle dstWin,
    const RectangleType *srcRect, Coord destX,
    Coord destY, WinDrawOperation mode)
```

Parameters

→ *srcWin*

The window from which the rectangle is copied. If `NULL`, use the draw window.

→ *dstWin*

The window to which the rectangle is copied. If `NULL`, use the draw window.

→ *srcRect*
The bounds of the region to copy (see RectangleType).

→ *destX*
The left bound of the rectangle in destination window.

→ *destY*
The top bound of the rectangle in destination window.

→ *mode*
Must be set to the `winPaint` transfer mode.

Returns Nothing.

Comments Copies the bits of the window inside the rectangle region.

If the destination bitmap is compressed, the mode parameter must be `winPaint`, and the destination coordinates must be (0,0). If the width of the destination rectangle is less than 16 pixels or if the destination coordinates are not (0,0), then this function turns off compression for the destination bitmap. Normally, you do not copy to a compressed bitmap. Instead, you copy to an uncompressed bitmap and compress it afterwards.

This function does not work if the source window is an update-based window. It is best not to use this function to copy between two on-screen rectangles. It is better to copy your data to an off-screen window to capture data and then copy that onto the screen as needed.

Compatibility In earlier releases of Palm OS, this function used the transfer mode passed as a parameter. In Palm OS Cobalt, you must pass `winPaint` as the transfer mode.

See Also WinDrawBitmap()

WinDrawBitmap Function

Purpose Draws a bitmap at the given coordinates in `winPaint` mode (see `WinDrawOperation` for mode details).

Declared In `Window.h`

Prototype

```
void WinDrawBitmap (BitmapPtr bitmapP, Coord x,
    Coord y)
```

Parameters

→ *bitmapP*
A pointer to a bitmap.

→ *x*
The x coordinate of the top-left corner.

→ *y*
The y coordinate of the top-left corner.

Returns Nothing.

Comments If the bitmap has multiple depths (is a bitmap family), the closest match less than or equal to the current draw window depth is used. If such a bitmap does not exist, the bitmap with the closest match greater than the draw window depth is used.

If the bitmap has its own color table, color conversion to the draw window color table will be applied. This color conversion is slow and not recommended.

Compatibility Do not use this function in conjunction with the graphics context. Use `WinDrawBitmapHandle()`, `GcDrawBitmapAt()`, `GcDrawRawBitmapAt()`, or `GcPaintBitmap()` instead.

See Also `WinEraseRectangle()`

WinDrawBitmapHandle Function

Purpose Draws a bitmap at the coordinates specified, using a GcBitmapHandle.

Declared In GcRender.h

Prototype
```
void WinDrawBitmapHandle
    (GcBitmapHandle bitmapHandle, Coord x,
    Coord y)
```

Parameters

→ *bitmapHandle*
The bitmap Use FrmGetBitmapHandle() or GcLoadBitmap() to obtain a GcBitmapHandle to a bitmap.

→ *x*
The x coordinate of the top-left corner.

→ *y*
The y coordinate of the top-left corner.

Returns Nothing.

Comments A GcBitmapHandle stores a bitmap that has been converted to draw to the screen more efficiently than a BitmapType. Use this function if you want the efficiency of the GcBitmapHandle type but do not want to convert all of your drawing code to use the new drawing model.

This function behaves exactly like WinDrawBitmap() except that it ignores the current scaling mode. Scaling for GcBitmapHandle bitmaps is controlled when the GcBitmapHandle is created.

WinDrawChar Function

Purpose Draws the specified character in the draw window.

Declared In Window.h

Prototype
```
void WinDrawChar (wchar32_t theChar, Coord x,
    Coord y)
```

Parameters

→ *theChar*
The character to draw. This may be either a single-byte character or a multi-byte character.

→ *x*
The x coordinate of the location where the character is to be drawn (left bound).

→ *y*
The y coordinate of the location where the character is to be drawn (top bound).

Returns Nothing.

Comments Before calling this function, call `WinSetUnderlineMode()` and `FntSetFont()` to set the desired underline and font to draw the characters.

This function differs from `WinPaintChar()` in that this function always uses `winPaint` mode (see `WinDrawOperation`). This means the on bits are drawn in the text color, the off bits are in the background color, and underlines are in the foreground color. `WinPaintChar()` uses the current drawing state transfer mode instead of `winPaint`.

Compatibility The `winInvert` drawing mode does not work when you draw text. If you try to use it, nothing is drawn to the screen.

Do not use this function in conjunction with the graphics context APIs. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the font baseline whereas `WinDrawChar()` uses the y value as the top of the character.

See Also `WinDrawChars()`, `WinDrawInvertedChars()`, `WinDrawTruncChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChars()`

WinDrawChars Function

Purpose Draws the specified characters in the draw window.

Declared In `Window.h`

Prototype
```
void WinDrawChars (const char *chars,
    int16_t len, Coord x, Coord y)
```

Parameters → *chars*
A pointer to the characters to draw.

→ *len*
The length in bytes of the characters to draw.

→ *x*
The x coordinate of the first character to draw (left bound).

→ *y*
The y coordinate of the first character to draw (top bound).

Returns Nothing.

Comments This function is useful for printing non-editable status or warning messages on the screen.

Before calling this function, call `WinSetUnderlineMode()` and `FntSetFont()` to set the desired underline and font to draw the characters.

This function differs from `WinPaintChars()` in that this function always uses `winPaint` mode (see `WinDrawOperation`). This means the on bits are drawn in the text color, the off bits are in the background color, and underlines are in the foreground color. `WinPaintChar()` uses the current drawing state transfer mode instead of `winPaint`.

Compatibility The `winInvert` drawing mode does not work when you draw text. If you try to use it, nothing is drawn to the screen.

Do not use this function in conjunction with the graphics context APIs. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the font baseline whereas `WinDrawChars()` uses the y value as the top of the character.

See Also `WinDrawChar()`, `WinDrawInvertedChars()`, `WinDrawTruncChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChar()`

WinDrawGrayLine Function

Purpose Draws a dashed line in the draw window.

Declared In `Window.h`

Prototype
```
void WinDrawGrayLine (Coord x1, Coord y1,
    Coord x2, Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Comments This routine does not draw in the gray color; it draws with alternating foreground and background pixels. That is, it uses the `grayPattern` pattern type.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcLineTo()` instead.

See Also `WinDrawLine()`, `WinEraseLine()`, `WinFillLine()`, `WinInvertLine()`, `WinPaintLine()`, `WinPaintLines()`

WinDrawGrayRectangleFrame Function

Purpose Draws a gray rectangular frame in the draw window.

Declared In `Window.h`

Prototype
```
void WinDrawGrayRectangleFrame (FrameType frame,
    const RectangleType *rP)
```

Parameters

→ *frame*
The type of frame to draw (see `FrameType`).

→ *rP*
A pointer to the rectangle to frame (see `RectangleType`).

Returns Nothing.

Comments This routine does not draw in the gray color; it draws with alternating foreground and background pixels. The standard gray pattern is not used by this routine; rather, the frame is drawn so that the top-left pixel of the frame is always on.

Compatibility The `winInvert` and `winSwap` drawing modes do not work when you draw a rectangle frame or unfilled rectangle. Nothing is drawn to the screen.

Do not use this function in conjunction with the graphics context APIs. Use `GcRect()` instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also `WinDrawRectangleFrame()`, `WinEraseRectangleFrame()`, `WinGetFramesRectangle()`, `WinInvertRectangleFrame()`, `WinPaintRectangleFrame()`

WinDrawInvertedChars Function

Purpose Draws the specified characters inverted (background color) in the draw window.

Declared In `Window.h`

Prototype

```
void WinDrawInvertedChars (const char *chars,
   int16_t len, Coord x, Coord y)
```

Parameters

→ *chars*
A pointer to the characters to draw.

→ *len*
The length in bytes of the characters to draw.

→ *x*
The x coordinate of the first character to draw (left bound).

→ *y*
The y coordinate of the first character to draw (top bound).

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also `WinDrawChar()`, `WinDrawChars()`, `WinDrawTruncChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChar()`, `WinPaintChars()`

WinDrawLine Function

Purpose Draws a line in the draw window using the current foreground color.

Declared In `Window.h`

Prototype
```
void WinDrawLine (Coord x1, Coord y1, Coord x2,
    Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Comments This function differs from `WinPaintLine()` in that it always uses `winPaint` mode (see `WinDrawOperation`). `WinPaintLine()` uses the current drawing state transfer mode instead of `winPaint`.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcLineTo()` instead.

See Also `WinDrawGrayLine()`, `WinEraseLine()`, `WinFillLine()`, `WinInvertLine()`, `WinPaintLine()`, `WinPaintLines()`

WinDrawPixel Function

Purpose Draws a pixel in the draw window using the current foreground color.

Declared In `Window.h`

Prototype
```
void WinDrawPixel (Coord x, Coord y)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

Returns Nothing.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcLineTo()` instead.

See Also `WinErasePixel()`, `WinInvertPixel()`, `WinPaintPixel()`, `WinPaintPixels()`

WinDrawRectangle Function

Purpose Draws a rectangle in the draw window using the current foreground color.

Declared In `Window.h`

Prototype

```
void WinDrawRectangle (const RectangleType *rP,
    uint16_t cornerDiam)
```

Parameters

→ *rP*
A pointer to the rectangle to draw (see `RectangleType`).

→ *cornerDiam*
The radius of rounded corners. Specify zero for square corners.

Returns Nothing.

Comments The *cornerDiam* parameter specifies the radius of four imaginary circles used to form the rounded corners. An imaginary circle is placed within each corner tangent to the rectangle on two sides.

This function differs from `WinPaintRectangle()` in that it always uses `winPaint` mode (see `WinDrawOperation`). `WinPaintRectangle()` uses the current drawing state transfer mode instead of `winPaint`.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcRect()` instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also `WinEraseRectangle()`, `WinFillRectangle()`, `WinInvertRectangle()`

WinDrawRectangleFrame Function

Purpose Draws a rectangular frame in the draw window using the current foreground color.

Declared In `Window.h`

Prototype

```
void WinDrawRectangleFrame (FrameType frame,
    const RectangleType *rP)
```

Parameters

→ *frame*
The type of frame to draw (see `FrameType`).

→ *rP*
A pointer to the rectangle to frame (see `RectangleType`).

Returns Nothing.

Comments The frame is drawn outside the specified rectangle.

This function differs from `WinPaintRectangleFrame()` in that it always uses `winPaint` mode (see `WinDrawOperation`). `WinPaintRectangleFrame()` uses the current drawing state transfer mode instead of `winPaint`.

Compatibility The `winInvert` and `winSwap` drawing modes do not work when you draw a rectangle frame or unfilled rectangle. Nothing is drawn to the screen.

Do not use this function in conjunction with the graphics context APIs. Use `GcRect()` instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also WinDrawGrayRectangleFrame(), WinEraseRectangleFrame(), WinGetFramesRectangle(), WinInvertRectangleFrame()

WinDrawTruncChars Function

Purpose Draws the specified characters in the draw window, truncating the characters to the specified width.

Declared In `Window.h`

Prototype

```
void WinDrawTruncChars (const char *chars,
    int16_t len, Coord x, Coord y, Coord maxWidth)
```

Parameters

→ *chars*
A pointer to the characters to draw.

→ *len*
The length in bytes of the characters to draw.

→ *x*
The x coordinate of the first character to draw (left bound).

→ *y*
The y coordinate of the first character to draw (top bound).

→ *maxWidth*
The maximum width in pixels of the characters that are to be drawn.

Returns Nothing.

Comments Before calling this function, consider calling `WinSetUnderlineMode()` and `FntSetFont()`.

If drawing all of the specified characters requires more space than *maxWidth* allows, `WinDrawTruncChars()` draws one less than the number of characters that can fit in *maxWidth* and then draws an ellipsis (...) in the remaining space. (If the boundary characters are narrower than the ellipsis, more than one character may be dropped to make room.) If *maxWidth* is narrower than the width of an ellipsis, nothing is drawn.

Use this function to truncate text that may contain multi-byte characters.

This function differs from `WinPaintTruncChars()` in that it always uses `winPaint`. `WinPaintTruncChars()` uses the current draw state transfer mode.

Compatibility The `winInvert` drawing mode does not work when you draw text. Nothing is drawn to the screen.

Do not use this function in conjunction with the graphics context APIs. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the font baseline whereas `WinDrawTruncChars()` uses the y value as the top of the character.

See Also `WinDrawChar()`, `WinDrawChars()`, `WinDrawInvertedChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChar()`, `WinPaintChars()`, `WinPaintTruncChars()`

WinEraseChars Function

Purpose Erases the specified characters in the draw window.

Declared In `Window.h`

Prototype

```
void WinEraseChars (const char *chars,
    int16_t len, Coord x, Coord y)
```

Parameters

→ *chars*
A pointer to the characters to erase.

→ *len*
The length in bytes of the characters to erase.

→ *x*
The x coordinate of the first character to erase (left bound).

→ *y*
The y coordinate of the first character to erase (top bound).

Returns Nothing.

Comments The `winMask` transfer mode is used to erase the characters. See `WinDrawOperation` for more information. This has the effect of erasing only the on bits for the characters rather than the entire text rectangle. This function only works if the foreground color is black and the background color is white.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()`

uses the y value as the font baseline whereas `WinEraseChars()` uses the y value as the top of the character.

See Also WinDrawChar(), WinDrawChars(), WinDrawInvertedChars(), WinDrawTruncChars(), WinInvertChars(), WinPaintChar(), WinPaintChars()

WinEraseLine Function

Purpose Draws a line in the draw window using the current background color.

Declared In `Window.h`

Prototype

```
void WinEraseLine (Coord x1, Coord y1, Coord x2,
    Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcLineTo()` instead.

See Also `WinDrawGrayLine()`, `WinDrawLine()`, `WinFillLine()`, `WinInvertLine()`, `WinPaintLine()`, `WinPaintLines()`

WinErasePixel Function

Purpose Draws a pixel in the draw window using the current background color.

Declared In `Window.h`

Prototype
```
void WinErasePixel (Coord x, Coord y)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

Returns Nothing.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcLineTo()` instead.

See Also `WinDrawPixel()`, `WinInvertPixel()`, `WinPaintPixel()`, `WinPaintPixels()`

WinEraseRectangle Function

Purpose Draws a rectangle in the draw window using the current background color.

Declared In `Window.h`

Prototype
```
void WinEraseRectangle (const RectangleType *rP,
    uint16_t cornerDiam)
```

Parameters

→ *rP*
A pointer to the rectangle to erase (see `RectangleType`).

→ *cornerDiam*
The radius of rounded corners. Specify zero for square corners.

Returns Nothing.

Comments The *cornerDiam* parameter specifies the radius of four imaginary circles used to form the rounded corners. An imaginary circle is placed within each corner tangent to the rectangle on two sides.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcRect()` instead. Note that `GcRect()` specifies a

rectangle's bounds differently than a `RectangleType` structure does.

See Also `WinDrawRectangle()`, `WinFillRectangle()`, `WinInvertRectangle()`, `WinPaintRectangle()`

WinEraseRectangleFrame Function

Purpose Draws a rectangular frame in the draw window using the current background color.

Declared In `Window.h`

Prototype

```
void WinEraseRectangleFrame (FrameType frame,
    const RectangleType *rP)
```

Parameters

→ *frame*
The type of frame to draw (see `FrameType`).

→ *rP*
A pointer to the rectangle to frame (see `RectangleType`).

Returns Nothing.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use `GcRect()` instead.

See Also `WinDrawGrayRectangleFrame()`, `WinDrawRectangleFrame()`, `WinGetFramesRectangle()`, `WinInvertRectangleFrame()`, `WinPaintRectangleFrame()`

WinEraseWindow Function

Purpose Erases the contents of the draw window.

Declared In `Window.h`

Prototype

```
void WinEraseWindow (void)
```

Parameters None.

Returns Nothing.

Comments `WinEraseRectangle()` is used to erase the window. This routine doesn't erase the frame around the draw window. See

WinEraseRectangleFrame() and WinGetWindowFrameRect().

Compatibility Do not use this function in conjunction with the graphics context APIs. Use GcPaint() instead.

WinFillLine Function

Purpose Fills a line in the draw window with the current pattern.

Declared In Window.h

Prototype

```
void WinFillLine (Coord x1, Coord y1, Coord x2,
    Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Comments You can set the current pattern with WinSetPattern().

Compatibility Do not use this function in conjunction with the graphics context APIs. Use GcLineTo() instead.

See Also WinDrawGrayLine(), WinDrawLine(), WinEraseLine(), WinInvertLine(), WinPaintLine(), WinPaintLines()

WinFillRectangle Function

Purpose Draws a rectangle in the draw window with current pattern.

Declared In `Window.h`

Prototype
```
void WinFillRectangle (const RectangleType *rP,
    uint16_t cornerDiam)
```

Parameters

→ *rP*
A pointer to the rectangle to draw (see RectangleType).

→ *cornerDiam*
The radius of rounded corners. Specify zero for square corners.

Returns Nothing.

Comments You can set the current pattern with WinSetPattern().

The *cornerDiam* parameter specifies the radius of four imaginary circles used to form the rounded corners. An imaginary circle is placed within each corner tangent to the rectangle on two sides.

Compatibility Do not use this function in conjunction with the graphics context APIs. Use GcRect() instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also WinDrawRectangle(), WinEraseRectangle(), WinInvertRectangle(), WinPaintRectangle()

WinGetClip Function

Purpose Returns the clipping rectangle of the draw window.

Declared In `Window.h`

Prototype
```
void WinGetClip (RectangleType *rP)
```

Parameters

← *rP*
A pointer to a structure to hold the clipping bounds (see RectangleType).

Returns Nothing.

See Also WinSetClip()

WinGetCoordinateSystem Function

Purpose Gets the coordinate system.

Declared In `Window.h`

Prototype
```
uint16_t WinGetCoordinateSystem (void)
```

Parameters None.

Returns A value representing the current coordinate system. See "Coordinate System Constants" on page 694 for the values that this function can return.

Comments Use this function to determine the active window coordinate system. Armed with this information, an application can properly initialize graphic primitive coordinates and dimensions, or can modify the coordinate system with `WinSetCoordinateSystem()`.

Compatibility Do not use this function in conjunction with the graphics context APIs. It returns the coordinate system of the window rather than the coordinate system of the graphics context.

WinGetPattern Function

Purpose Returns the current fill pattern.

Declared In `Window.h`

Prototype
```
void WinGetPattern (CustomPatternType *patternP)
```

Parameters ← *patternP*
The buffer where the current pattern is returned (see `CustomPatternType`).

Returns Nothing.

Comments The fill pattern is used by `WinFillLine()` and `WinFillRectangle()`.

This function returns the pattern data stored in the drawing state. The drawing state only has pattern data if the pattern field itself is set to `customPattern`. Therefore, it's a good idea to use `WinGetPatternType()` instead of this function on systems that support `WinGetPatternType()`.

See Also `WinSetPattern()`

WinGetPatternType Function

Purpose Returns the current pattern type.

Declared In `Window.h`

Prototype
```
PatternType WinGetPatternType (void)
```

Parameters None.

Returns The current draw window pattern type (see `PatternType`). If the return value is `customPattern`, you can retrieve the pattern with `WinGetPattern()`.

Comments The fill pattern is used by `WinFillLine()` and `WinFillRectangle()`.

See Also `WinSetPatternType()`

WinGetScalingMode Function

Purpose Gets the current scaling mode.

Declared In `Window.h`

Prototype
```
uint32_t WinGetScalingMode (void)
```

Parameters None.

Returns One of the Scaling Mode Constants.

See Also `WinSetScalingMode()`

WinInvertChars Function

Purpose Inverts the specified characters in the draw window.

Declared In `Window.h`

Prototype
```
void WinInvertChars (const char *chars,
   int16_t len, Coord x, Coord y)
```

Parameters

→ *chars*
A pointer to the characters to invert.

→ *len*
The length in bytes of the characters to invert.

→ *x*
The x coordinate of the first character to invert (left bound).

→ *y*
The y coordinate of the first character to invert (top bound).

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also WinDrawChar(), WinDrawChars(), WinDrawInvertedChars(), WinDrawTruncChars(), WinEraseChars(), WinPaintChar(), WinPaintChars()

WinInvertLine Function

Purpose Inverts a line in the draw window (using the WinDrawOperation winInvert).

Declared In Window.h

Prototype
```
void WinInvertLine (Coord x1, Coord y1, Coord x2,
    Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also WinDrawGrayLine(), WinDrawLine(), WinEraseLine(), WinFillLine(), WinPaintLine(), WinPaintLines()

WinInvertPixel Function

Purpose Inverts a pixel in the draw window (using the `WinDrawOperation` `winInvert`).

Declared In `Window.h`

Prototype
```
void WinInvertPixel (Coord x, Coord y)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also `WinDrawPixel()`, `WinErasePixel()`, `WinPaintPixel()`, `WinPaintPixels()`

WinInvertRectangle Function

Purpose Inverts a rectangle in the draw window (using the `WinDrawOperation` `winInvert`).

Declared In `Window.h`

Prototype
```
void WinInvertRectangle (const RectangleType *rP,
   uint16_t cornerDiam)
```

Parameters

→ *rP*
A pointer to the rectangle to invert (see `RectangleType`).

→ *cornerDiam*
The radius of rounded corners. Specify zero for square corners.

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also `WinDrawRectangle()`, `WinEraseRectangle()`, `WinFillRectangle()`, `WinPaintRectangle()`

WinInvertRectangleFrame Function

Purpose Inverts a rectangular frame in the draw window (using the WinDrawOperation winInvert).

Declared In Window.h

Prototype
```
void WinInvertRectangleFrame (FrameType frame,
    const RectangleType *rP)
```

Parameters

→ *frame*
The type of frame to draw (see FrameType).

→ *rP*
A pointer to the rectangle to frame (see RectangleType).

Returns Nothing.

Compatibility This function is deprecated in Palm OS Cobalt.

See Also WinDrawGrayRectangleFrame(), WinDrawRectangleFrame(), WinEraseRectangleFrame(), WinGetFramesRectangle(), WinPaintRectangleFrame()

WinPaintBitmap Function

Purpose Draws a bitmap in the current draw window at the specified coordinates using the winPaint transfer mode.

Declared In Window.h

Prototype
```
void WinPaintBitmap (BitmapType *bitmapP,
    Coord x, Coord y)
```

Parameters

→ *bitmapP*
A pointer to a bitmap.

→ *x*
The x coordinate of the top-left corner.

→ *y*
The y coordinate of the top-left corner.

Returns Nothing.

Comments If the bitmap has multiple depths (is a bitmap family), the closest match less than or equal to the current draw window depth is used. If such a bitmap does not exist, the bitmap with the closest match greater than the draw window depth is used.

Using `WinPaintBitmap()` is recommended instead of the rendering bitmaps into an off-screen window and then using `WinCopyRectangle()` to draw them on screen.

If the bitmap has its own color table, color conversion to the draw window color table will be applied. This color conversion is slow and not recommended.

Compatibility Do not use this function with the graphics context. Use `GcPaintBitmap()` instead.

In earlier releases, this function used the transfer mode set in the draw state to paint the bitmap to the screen. Palm OS Cobalt ignores the current transfer mode and uses `winPaint` instead.

See Also `WinDrawBitmap()`, `WinEraseRectangle()`, `WinPaintTiledBitmap()`

WinPaintChar Function

Purpose Draws a character in the draw window using the current drawing state.

Declared In `Window.h`

Prototype

```
void WinPaintChar (wchar32_t theChar, Coord x,
    Coord y)
```

Parameters

→ *theChar*
The character to draw. This may be either a single-byte character or a multi-byte character.

→ *x*
The x coordinate of the location where the character is to be drawn (left bound).

→ *y*
The y coordinate of the location where the character is to be drawn (top bound).

Returns Nothing.

Comments `WinPaintChar()` draws the **on** bits in the text color and the **off** bits in the background color, with underlines (if any) drawn in the foreground color using the current drawing mode.

This function uses the current drawing state.

Compatibility The `winInvert` drawing mode does not work when you draw text. Nothing is drawn to the screen.

Do not use this function with the graphics context. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the baseline for the font whereas `WinPaintChar()` uses the y value as the top of the character.

See Also `WinDrawChar()`, `WinDrawChars()`, `WinDrawInvertedChars()`, `WinDrawTruncChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChars()`

WinPaintChars Function

Purpose Draws the specified characters in the draw window with the current draw state.

Declared In `Window.h`

Prototype

```
void WinPaintChars (const char *chars,
    int16_t len, Coord x, Coord y)
```

Parameters

→ *chars*
A pointer to the characters to draw.

→ *len*
The length in bytes of the characters to draw.

→ *x*
The x coordinate of the first character to draw (left bound).

→ *y*
The y coordinate of the first character to draw (top bound).

Returns Nothing.

Comments `WinPaintChars()` draws the **on** bits in the text color and the **off** bits in the background color, with underlines (if any) drawn in the foreground color using the current drawing mode.

This function uses the current drawing state.

Before calling this function, consider calling `WinSetUnderlineMode()` and `FntSetFont()`.

Compatibility The `winInvert` drawing mode does not work when you draw text. Nothing is drawn to the screen.

Do not use this function with the graphics context. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the baseline for the font whereas `WinPaintChars()` uses the y value as the top of the character.

See Also `WinDrawChar()`, `WinDrawChars()`, `WinDrawInvertedChars()`, `WinDrawTruncChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChar()`

WinPaintLine Function

Purpose Draws a line in the draw window using the current drawing state.

Declared In `Window.h`

Prototype

```
void WinPaintLine (Coord x1, Coord y1, Coord x2,
    Coord y2)
```

Parameters

→ *x1*
The x coordinate of line beginning point.

→ *y1*
The y coordinate of line beginning point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Comments This function uses the current drawing state.

Compatibility Do not use this function with the graphics context. Use `GcLineTo()` instead.

See Also `WinDrawLine()`, `WinDrawGrayLine()`, `WinEraseLine()`, `WinFillLine()`, `WinInvertLine()`, `WinPaintLines()`

WinPaintLines Function

Purpose Draws several lines in the draw window using the current drawing state.

Declared In `Window.h`

Prototype
```
void WinPaintLines (uint16_t numLines,
    WinLineType lines[])
```

Parameters

→ *numLines*
The number of lines to paint.

→ *lines*
An array of lines. See `WinLineType`.

Returns Nothing.

Comments This function uses the current drawing state.

Compatibility Do not use this function with the graphics context. Use `GcLineTo()` instead.

See Also `WinDrawLine()`, `WinDrawGrayLine()`, `WinEraseLine()`, `WinFillLine()`, `WinInvertLine()`, `WinPaintLine()`

WinPaintPixel Function

Purpose Renders a pixel in the draw window using the current drawing state.

Declared In `Window.h`

Prototype
```
void WinPaintPixel (Coord x, Coord y)
```

Parameters

→ *x*
A pointer to the x coordinate of a pixel.

→ *y*
A pointer to the y coordinate of a pixel.

Returns Nothing.

Comments This function uses the current drawing state.

Compatibility Do not use this function with the graphics context. Use `GcLineTo()` instead.

See Also `WinDrawPixel()`, `WinErasePixel()`, `WinInvertPixel()`, `WinPaintPixels()`

WinPaintPixels Function

Purpose Renders several pixels in the draw window using the current drawing state.

Declared In `Window.h`

Prototype
```
void WinPaintPixels (uint16_t numPoints,
    PointType pts[])
```

Parameters

→ *numPoints*
The number of pixels to paint.

→ *pts*
An array of pixels.

Returns Nothing.

Comments This function uses the current drawing state.

Compatibility Do not use this function with the graphics context. Use `GcLineTo()` instead.

See Also `WinDrawPixel()`, `WinErasePixel()`, `WinInvertPixel()`, `WinPaintPixel()`

WinPaintRectangle Function

Purpose Draws a rectangle in the draw window using the current drawing state.

Declared In `Window.h`

Prototype
```
void WinPaintRectangle (const RectangleType *rP,
    uint16_t cornerDiam)
```

Parameters

→ *rP*
The pointer to the rectangle to draw (see `RectangleType`).

→ *cornerDiam*
The radius of rounded corners. Specify zero for square corners.

Returns Nothing.

Comments The *cornerDiam* parameter specifies the radius of four imaginary circles used to form the rounded corners. An imaginary circle is placed within each corner tangent to the rectangle on two sides.

This function uses the current drawing state.

Compatibility The `winInvert` and `winSwap` drawing mode do not work when you draw a rectangle frame or unfilled rectangle. Nothing is drawn to the screen.

Do not use this function with the graphics context. Use `GcRect()` instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also `WinDrawRectangle()`, `WinEraseRectangle()`, `WinFillRectangle()`, `WinInvertRectangle()`

WinPaintRectangleFrame Function

Purpose Draws a rectangular frame in the draw window using the current drawing state.

Declared In `Window.h`

Prototype

```
void WinPaintRectangleFrame (FrameType frame,
    const RectangleType *rP)
```

Parameters

→ *frame*
The type of frame to draw (see `FrameType`).

→ *rP*
A pointer to the rectangle to frame (see `RectangleType`).

Returns Nothing.

Comments The frame is drawn outside the specified rectangle.

This function uses the current drawing state.

Compatibility The `winInvert` and `winSwap` drawing mode do not work when you draw a rectangle frame or unfilled rectangle. Nothing is drawn to the screen.

Do not use this function with the graphics context. Use `GcRect()` instead. Note that `GcRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

See Also `WinDrawGrayRectangleFrame()`, `WinDrawRectangleFrame()`, `WinEraseRectangleFrame()`, `WinGetFramesRectangle()`, `WinInvertRectangleFrame()`, `WinPaintRoundedRectangleFrame()`

WinPaintRoundedRectangleFrame Function

Purpose Draws a rectangular frame with rounded corners in the draw window using the current drawing state.

Declared In `Window.h`

Prototype

```
void WinPaintRoundedRectangleFrame
   (const RectangleType *rP, Coord width,
   Coord cornerRadius, Coord shadowWidth)
```

Parameters

→ *rP*
A pointer to the rectangle to frame (see RectangleType).

→ *width*
The width of the frame, interpreted using the active coordinate system.

→ *cornerRadius*
The radius of the rectangle's rounded corners, interpreted using the active coordinate system.

→ *shadowWidth*
The shadow offset, interpreted using the active coordinate system.

Returns Nothing.

Comments The `winInvert` and `winSwap` drawing mode do not work when you draw a rectangle frame or unfilled rectangle. Nothing is drawn to the screen.

This function allows you to draw a rectangle with a frame width and corner radius specified in the active coordinate system. It is necessary because `WinPaintRectangleFrame()` doesn't allow you to draw rounded rectangles with a frame width greater than 2. Note that because there isn't a function that parallels either `WinDrawRectangleFrame()`, `WinEraseRectangleFrame()`, or `WinInvertRectangleFrame()`, you must set the drawing mode and colors as appropriate and use `WinPaintRoundedRectangleFrame()` to achieve the desired effect.

Compatibility Do not use this function with the graphics context. Use `GcRoundRect()` instead. Note that `GcRoundRect()` specifies a rectangle's bounds differently than a `RectangleType` structure does.

WinPaintTiledBitmap Function

Purpose Fills a rectangle with a pattern defined by a bitmap.

Declared In `Window.h`

Prototype

```
void WinPaintTiledBitmap
    (const BitmapType *bitmapP,
    RectangleType *rectP)
```

Parameters

→ *bitmapP*
A pointer to the bitmap that contains the desired pattern.

→ *rectP*
A pointer to the rectangle that is to be filled (see `RectangleType`).

Returns Nothing. On a debug ROM, if either *bitmapP* or *rectP* are `NULL`, an error is displayed.

Comments This function makes it possible for an application to define a pattern that is larger than the standard 8 by 8 custom pattern, and to define high-density custom patterns.

The pattern is scaled using the density of *bitmapP* and the density of the screen bitmap. *bitmapP* can be a bitmap family; if it is, the Window Manager selects a bitmap using the same algorithm used by `WinPaintBitmap()`. As with other patterns, the tiled pattern is anchored to the window's origin.

Compatibility Do not use this function with the graphics context. Use `GcPaintBitmap()` instead.

WinPaintThinLine Function

Purpose Draws a line that is one pixel thick.

Declared In `Window.h`

Prototype

```
void WinPaintThinLine (Coord x1, Coord y1,
    Coord x2, Coord y2)
```

Parameters

→ *x1*
The x coordinate of line start point.

→ *y1*
The y coordinate of line start point.

→ *x2*
The x coordinate of line endpoint.

→ *y2*
The y coordinate of line endpoint.

Returns Nothing.

Comments The text field uses this function when it needs to extend the text underline past the end of the text.

Compatibility Do not use this function with the graphics context. Use `GcLineTo()` instead.

See Also `WinPaintLine()`

WinPaintTruncChars Function

Purpose Draws the specified characters in the draw window using the current transfer mode, truncating the characters to the specified width.

Declared In `Window.h`

Prototype

```
void WinPaintTruncChars (const char *chars,
   int16_t len, Coord x, Coord y, Coord maxWidth)
```

Parameters

→ *chars*
A pointer to the characters to draw.

→ *len*
The length in bytes of the characters to draw.

→ *x*
The x coordinate of the first character to draw (left bound).

→ *y*
The y coordinate of the first character to draw (top bound).

→ *maxWidth*
The maximum width in pixels of the characters that are to be drawn.

Returns Nothing.

Comments Before calling this function, consider calling `WinSetTextColor()`, `WinSetUnderlineMode()` and `FntSetFont()`.

If drawing all of the specified characters requires more space than *maxWidth* allows, `WinPainTruncChars()` draws one less than the number of characters that can fit in *maxWidth* and then draws an ellipsis (...) in the remaining space. (If the boundary characters are narrower than the ellipsis, more than one character may be dropped to make room.) If *maxWidth* is narrower than the width of an ellipsis, nothing is drawn.

Use this function to truncate text that may contain multi-byte characters.

This function differs from `WinDrawTruncChars()` in that it uses the current drawing state's transfer mode. `WinDrawTruncChars()` always uses `winPaint`.

Compatibility Do not use this function with the graphics context. Use `GcDrawTextAt()` instead. Note that `GcDrawTextAt()` uses the y value as the baseline for the font whereas `WinPaintTruncChars()` uses the y value as the top of the character.

See Also `WinDrawChar()`, `WinDrawChars()`, `WinDrawInvertedChars()`, `WinEraseChars()`, `WinInvertChars()`, `WinPaintChar()`, `WinPaintChars()`, `WinDrawTruncChars()`

WinPopDrawState Function

Purpose Restores the draw state values to the last saved set on the stack.

Declared In `Window.h`

Prototype

```
void WinPopDrawState (void)
```

Parameters None.

Returns Nothing.

Comments Use this routine to restore the draw state saved by the previous call to `WinPushDrawState()`.

Compatibility Do not use this function with the graphics context. Use `GcPopState()` instead.

WinPushDrawState Function

Purpose Saves the current draw state values onto the draw state stack.

Declared In `Window.h`

Prototype
```
void WinPushDrawState (void)
```

Parameters None.

Returns Nothing.

Comments Use this routine to save the current draw state before making changes to it. Call `WinPopDrawState()` to restore the saved settings.

Compatibility Do not use this function with the graphics context. Use `GcPushState()` instead.

WinResetClip Function

Purpose Resets the clipping rectangle of the draw window to the portion of the draw window that is within the bounds of the display.

Declared In `Window.h`

Prototype
```
void WinResetClip (void)
```

Parameters None.

Returns Nothing.

See Also `WinSetClip()`

WinSetBackColor Function

Purpose Sets the background color to use in subsequent draw operations.

Declared In `Window.h`

Prototype
```
IndexedColorType WinSetBackColor
    (IndexedColorType backColor)
```

Parameters → *`backColor`*
The color to set; specify a value of type `IndexedColorType`.

Returns The previous background color index.

Comments This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

To set the foreground color to a predefined UI color default, use `UIColorGetTableEntryIndex()` as an input to this function. For example:

```
curColor = WinSetBackColor
(UIColorGetTableEntryIndex(UIFieldBackground));
```

Using `WinSetBackColorRGB()` is preferred over this function. It is faster.

See Also `WinSetForeColor()`, `WinSetTextColor()`

WinSetBackColorRGB Function

Purpose Sets the background color to use in subsequent draw operations.

Declared In `Window.h`

Prototype

```
void WinSetBackColorRGB
    (const RGBColorType *newRgbP,
    RGBColorType *prevRgbP)
```

Parameters

→ *newRgbP*
The color to set; specify a value of type `RGBColorType`.

← *prevRgbP*
The previous color; specify a value of type `RGBColorType`.

Returns Nothing

Comments This function takes new and previous `RGBColorType` arguments. It is okay to set *newRgbP* or *prevRgbP* to `NULL`. If an application only wants to get the current color, the *newRgbP* argument is set to `NULL`. If the application does not care about the previous color, *prevRgbP* can be set to `NULL`.

This function sets the background color to the value specified by *newRgbP*. It then sets the index field of `backColorRGB` to the 8 bit system palette entry that most closely matches the RGB

components. Finally, it sets the `backColor` index field to this index value.

This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context. Use `GcSetColor()` instead.

See Also `WinSetForeColorRGB()`, `WinSetTextColorRGB()`

WinSetClip Function

Purpose Sets the clipping rectangle of the draw window.

Declared In `Window.h`

Prototype
```
void WinSetClip (const RectangleType *rP)
```

Parameters

→ *rP*
A pointer to a structure holding the clipping bounds (see `RectangleType`).

Returns Nothing.

Compatibility This function sets a rectangular region of the window into which the application may draw. It differs from `GcBeginClip()` in that `GcBeginClip()` works only with a graphics context and is able to define non-rectangular clipping regions.

See Also `WinClipRectangle()`, `WinSetClip()`, `WinGetClip()`

WinSetColors Function

Purpose Sets the colors.

Declared In `Window.h`

Prototype

```
void WinSetColors
    (const RGBColorType *newForeColorP,
    RGBColorType *oldForeColorP,
    const RGBColorType *newBackColorP,
    RGBColorType *oldBackColorP)
```

Parameters

→ *newForeColorP*
The color to use for the foreground and the text; specify a value of type `RGBColorType`.

← *oldForeColorP*
Previous foreground color.

→ *newBackColorP*
The color to use for the background; specify a value of type `RGBColorType`.

← *oldBackColorP*
Previous background color.

Returns Nothing.

See Also `WinSetBackColorRGB()`, `WinSetTextColorRGB()`, `WinSetForeColorRGB()`

WinSetCoordinateSystem Function

Purpose Establishes the coordinate system to be used for subsequent drawing operations.

Declared In `Window.h`

Prototype

```
uint16_t WinSetCoordinateSystem
    (uint16_t coordSys)
```

Parameters

→ *coordSys*
The desired coordinate system. Supply one of the values defined in "Coordinate System Constants" on page 694.

Returns The previous coordinate system value.

Comments This function modifies the draw state. As when making other modifications to a window's draw state, applications should call `WinPushDrawState()` before modifying the coordinate system. To restore the coordinate system, your application can then call `WinPopDrawState()`.

To calculate the amount by which it will scale all coordinates, the Window Manager divides the density of the bitmap associated with the draw window by *coordSys*. If *coordSys* is `kCoordinatesNative`, the Window Manager sets the scale field to 1.0, which to enables 1-to-1 mapping of coordinates to pixels.

If you supply a value of `kCoordinatesStandard` for *coordSys*, subsequent drawing will use the standard coordinate system.

Compatibility If you are making graphics context calls, use `GcSetCoordinateSystem()` to set the coordinate system that the graphics context uses within the window. It is a good idea to always make both calls so that you are certain that the window's coordinate system is the same as the graphics context's coordinate system.

See Also `WinGetCoordinateSystem()`

WinSetDrawMode Function

Purpose Sets the transfer mode to use in subsequent draw operations.

Declared In `Window.h`

Prototype

```
WinDrawOperation WinSetDrawMode
    (WinDrawOperation newMode)
```

Parameters

→ *newMode*
The transfer mode to set; specify one of the `WinDrawOperation` values.

Returns The previous transfer mode.

Comments This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context.

WinSetForeColor Function

Purpose Sets the foreground color to use in subsequent draw operations.

Declared In `Window.h`

Prototype

```
IndexedColorType WinSetForeColor
    (IndexedColorType foreColor)
```

Parameters

→ *foreColor*
The color to set; specify a value of type `IndexedColorType`.

Returns The previous foreground color index.

Comments This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

To set the foreground color to a predefined UI color default, use `UIColorGetTableEntryIndex()` as an input to this function. For example:

```
curColor = WinSetForeColor
   (UIColorGetTableEntryIndex(UIObjectForeground));
```

Using `WinSetForeColorRGB()` is preferred over this function. It is faster.

Compatibility Do not use this function with the graphics context. Use `GcSetColor()` instead.

See Also `WinSetBackColor()`, `WinSetTextColor()`

WinSetForeColorRGB Function

Purpose Sets the foreground color to use in subsequent draw operations.

Declared In `Window.h`

Prototype

```
void WinSetForeColorRGB
    (const RGBColorType *newRgbP,
    RGBColorType *prevRgbP)
```

Parameters

→ *newRgbP*
The color to set; specify a value of type `RGBColorType`.

← *prevRgbP*
Previous color

Returns Nothing.

Comments This function takes new and previous `RGBColorType` arguments. It is okay to set *newRgbP* or *prevRgbP* to `NULL`. If an application only wants to get the current color, the *newRgbP* argument is set to `NULL`. If the application does not care about the previous color, *prevRgbP* can be set to `NULL`.

This function sets the `foreColorRGB` field to the value specified by *newRgbP*. It then sets the index field of `foreColorRGB` to the 8 bit system palette entry that most closely matches the RGB components. Finally, it sets the `foreColor` index field to this index value.

This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context. Use `GcSetColor()` instead.

See Also `WinSetBackColorRGB()`, `WinSetTextColorRGB()`

WinSetFrameType Function

Purpose Sets the type of frame to be used for a specified window.

Declared In `Window.h`

Prototype
```
void WinSetFrameType (const WinHandle winH,
    FrameType frame)
```

Parameters → *winH*
The window's handle.

→ *frame*
The style of frame to be used. See `FrameType`.

Returns Nothing.

See Also `WinGetFrameType()`

WinSetPattern Function

Purpose Sets the current fill pattern.

Declared In `Window.h`

Prototype

```
void WinSetPattern
    (const CustomPatternType *patternP)
```

Parameters

→ *patternP*
The pattern to set (see `CustomPatternType`).

Returns Nothing.

Comments The fill pattern is used by `WinFillLine()` and `WinFillRectangle()`.

This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context.

See Also `WinGetPattern()`

WinSetPatternType Function

Purpose Sets the current pattern type.

Declared In `Window.h`

Prototype

```
void WinSetPatternType (PatternType newPattern)
```

Parameters

→ *newPattern*
The pattern type to set for the draw window (see `PatternType`).

Returns Nothing.

Comments This function sets the drawing state's pattern `newPattern` and sets the custom pattern data to `NULL`. To set a custom pattern use `WinSetPattern()`.

The fill pattern is used by `WinFillLine()` and `WinFillRectangle()`.

This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state

before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context.

See Also `WinGetPatternType()`

WinSetScalingMode Function

Purpose Controls the scaling of bitmaps and bitmapped fonts and the spacing between text glyphs.

Declared In `Window.h`

Prototype

```
uint32_t WinSetScalingMode (uint32_t mode)
```

Parameters

→ *mode*
One or more of the Scaling Mode Constants OR'd together.

Returns The previous scaling mode.

Comments If you need to display a large, low-density bitmap—for instance, a map or a photograph—you can use `WinSetScalingMode()` to draw your bitmap unscaled, allowing the user to see more of the bitmap at one time. If you want to draw more, but smaller, text on a device that has a high-density display, you can use `WinSetScalingMode()` to draw text unscaled.

On a device with a high-density display, if the bitmap being drawn, or the font being used, is part of a family that contains both low- and high-density members, `WinSetScalingMode()` controls which member is used. If `kTextScalingOff` is set, for instance, the low-density font is used, unscaled, when drawing subsequent text.

When an application calls `WinSetCoordinateSystem()` to use native coordinates for a high-density window, the operating system turns off text padding. When an application calls `WinSetCoordinateSystem()` to use standard coordinates, the operating system turns text padding on. To use standard coordinates with text padding turned off, or to use native coordinates with text padding turned on, call `WinSetScalingMode()` after calling `WinSetCoordinateSystem()`. Note that using standard coordinates with padding off is not recommended since the Font

Manager functions that measures character widths will return inaccurate widths.

The scaling mode flags are part of the window's draw state. In order to ensure that the draw state is properly restored, save that state using `WinPushDrawState()` before calling functions that alter the draw state—such as `WinSetScalingMode()` or `WinSetCoordinateSystem()`. After drawing is completed, restore the draw state with `WinPopDrawState()`.

When an application quits, the draw state is restored to the default settings.

See Also `WinGetScalingMode()`

WinSetTextColor Function

Purpose Sets the color to use for drawing characters in subsequent draw operations.

Declared In `Window.h`

Prototype

```
IndexedColorType WinSetTextColor
    (IndexedColorType textColor)
```

Parameters

→ *textColor*
The color to set; specify a value of type `IndexedColorType`.

Returns The previous text color index.

Comments This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

To set the foreground color to a predefined UI color default, use `UIColorGetTableEntryIndex()` as an input to this function. For example:

```
curColor = WinSetTextColor
  (UIColorGetTableEntryIndex(UIFieldText));
```

Using `WinSetTextColorRGB()` is preferred over this function. It is faster.

Compatibility Do not use this function with the graphics context. Use `GcSetColor()` instead.

See Also `WinSetBackColor()`, `WinSetForeColor()`

WinSetTextColorRGB Function

Purpose Sets the color to use for drawing characters in subsequent draw operations.

Declared In `Window.h`

Prototype

```
void WinSetTextColorRGB
    (const RGBColorType *newRgbP,
    RGBColorType *prevRgbP)
```

Parameters

→ *newRgbP*
The color to set; specify a value of type `RGBColorType`.

← *prevRgbP*
The previous color; specify a value of type `RGBColorType`.

Returns Nothing.

Comments This function takes new and previous `RGBColorType` arguments. It is acceptable to set *newRgbP* or *prevRgbP* to `NULL`. If an application only wants to get the current color, the *newRgbP* argument is set to `NULL`. If the application does not care about the previous color, *prevRgbP* can be set to `NULL`.

This function sets the `textColorRGB` field to the value specified by *newRgbP*. It then sets the index field of `textColorRGB` to the 8 bit system palette entry that most closely matches the RGB components. Finally, it sets the `textColor` index field to this index value.

This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context. Use `GcSetColor()` instead.

See Also `WinSetBackColorRGB()`, `WinSetForeColorRGB()`

WinSetUnderlineMode Function

Purpose Sets the graphic state to enable or disable the underlining of characters.

Declared In `Window.h`

Prototype
```
UnderlineModeType WinSetUnderlineMode
    (UnderlineModeType mode)
```

Parameters

↔ *mode*
The new underline mode type; see `UnderlineModeType`.

Returns The previous underline mode type.

Comments This function changes the current drawing state. If necessary, use `WinPushDrawState()` to preserve the current drawing state before you set this function and use `WinPopDrawState()` to restore it later.

Compatibility Do not use this function with the graphics context.

See Also `WinDrawChars()`

Application-Defined Functions

winInvalidateFunc Function

Purpose Redraws the specified portion of a window.

Declared In `Window.h`

Prototype
```
Boolean (*winInvalidateFunc) (int32_t cmd,
    WinHandle window,
    const RectangleType *dirtyRect, void *state)
```

Parameters

→ *cmd*
One of the following:

`winInvalidateExecute`
Redraw the region *dirtyRect* in the *window*.

`winInvalidateDestroy`
Free the *state* if necessary.

→ *window*
The window into which to draw.

→ *dirtyRect*
The region of the window that needs to be redrawn (see RectangleType).

→ *state*
Application-specific data necessary for the redraw.

Returns `true` if the function handled the cmd, or `false` if an error occurred.

Comments This function is used as a drawing optimization. You specify this function in a call to `WinInvalidateRectFunc()`. If certain conditions are met, the Window Manager calls this function instead of enqueueing a `winUpdateEvent`.

Even if this function is not used to redraw the specified area, it is always called with the `winInvalidateDestroy` parameter to free the state.

After you receive the call with `winInvalidateDestroy`, the association between the rectangle and this function is destroyed. If you want to keep using your callback function for this region, you should use `WinInvalidateRectFunc()` each time you need to invalidate it.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W