palmsource™

Input Services

Exploring Palm OS®

Written by Jean Ostrem
Edited by Greg Wilson
Engineering contributions by Grant Glouser, Joe Onorato, Chris Bark, Greg Wilson, and Ezekiel Sanborn de Asis.

Exploring Palm OS: Input Services
Document Number 3114-003
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

About This Document

This book describes the portions of Palm OS® that receive user input and send it to your application. There are several ways that a user provides input:

- Writing letters, numbers, or symbols in the input area
- Pressing a hardware button on the device
- Tapping the pen (or stylus) on the digitizer

This book covers the Palm OS managers that receive the button presses, pen strokes, and pen taps and translate them into the events that your application receives.

This book focuses on the low-level managers. It does not cover UI controls that receive user input. For information on UI controls, see *Exploring Palm OS: User Interface*. It also does not cover how an application should respond to textual input. See *Exploring Palm OS: Text and Localization* for information on receiving text-based input.

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 and you want to do one of the following:

- Write an application that works on devices that have a **dynamic input area** (one that the user can collapse and expand) and has some level of control over the input area.
- Write a **pinlet**, which is an executable that displays its user interface in the dynamic input area. The pinlet's job is to

receive pen events in the input area and translate them into character input.

- Write a game or some other application that needs input from the hardware buttons.
- Replace the handwriting recognition engine with one of your own.

You can write a full-featured application without using any of the API described in this book. Beginning Palm OS developers may want to delay reading this book until they gain a better understanding of the fundamentals of Palm OS application development. Instead, consider reading *Exploring Palm OS: Programming Basics* to gain a good understanding of event management and *Exploring Palm OS: User Interface* to learn about events generated by standard UI controls. Come back to this book only when you find you need more control than the higher level managers provide.

What This Book Contains

This book contains the following information:

- Part I contains conceptual information and how-to information.
  - Chapter 1, "Receiving Input," on page 3 introduces you to how a Palm Powered™ device receives user input and sends it to your application.
  - Chapter 2, "Working with the Dynamic Input Area," on page 7 explains how an application may interact with the dynamic input area or the pinlet that runs in the dynamic input area.
  - Chapter 3, "Customizing the Dynamic Input Area," on page 17 describes how to create a pinlet that runs in the dynamic input area and how you can replace the handwriting recognition engine if you want to.
  - Chapter 4, "Customizing Hardware Input," on page 37 describes how you might customize the hard keys for your application's use.

- Part II contains reference information organized into the following chapters:
  - Chapter 5, "Low-Level Events Reference," on page 45 describes the lowest level events that an application works with: the key events and the pen events.
  - Chapter 6, "Graffiti 2 Reference," on page 53 describes the function that displays the Graffiti® 2 reference dialog.
  - Chapter 7, "Handwriting Recognition Engine," on page 55 describes the APIs for the handwriting recognition engine.
  - Chapter 8, "Hard Keys Reference," on page 65 describes the APIs that control the hardware buttons.
  - Chapter 9, "Keyboard," on page 71 describes the APIs for the standard keyboard dialog.
  - Chapter 10, "Pen Input Manager," on page 73 describes the APIs for the Pen Input Manager, which controls the pinlet and the dynamic input area.
  - Chapter 11, "Pinlet," on page 89 describes the APIs that you must implement if you write a pinlet.
  - Chapter 12, "Shift Indicator," on page 95 describes the APIs for the shift indicator.

Changes to This Book

3114-003

- Clarified how to change shift indicator location in `GsiSetLocation()` description.

3114-002

- Minor editorial corrections.

3114-001

- 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

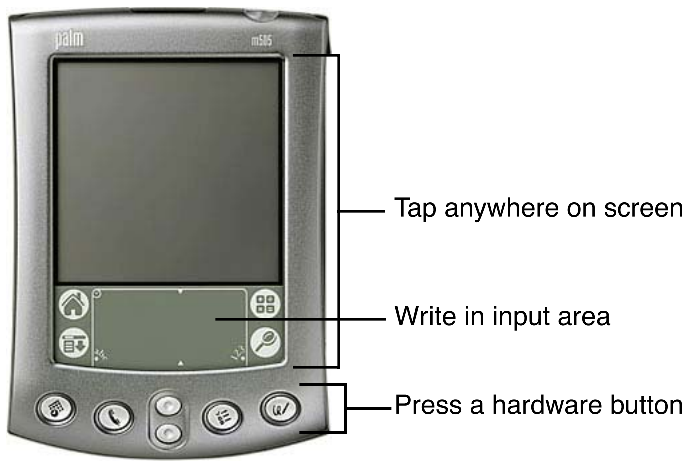
This part contains conceptual information for the input services managers. It covers:

1

Receiving Input

Users send input to an application by tapping the pen inside an application's form, in the input area, or in the status bar, by pressing hardware buttons on the device, and by writing letters, numbers, or symbols in the input area (see Figure 1.1).

Figure 1.1 Receiving input



This chapter describes what happens when the user performs any of the above actions. It covers:

Pen Taps

When the user taps the pen on the device's display, the following happens:

- A `penDownEvent` is generated specifying where the user tapped the pen.

- A `penUpEvent` is generated for the same location.

When the user drags the pen on the device's display, one or more `penMoveEvent`s are generated in between the `penDownEvent` and `penUpEvent`.

Your application, however, often does not care about these events. Instead, they are converted into other events that your application is more likely to handle.

If the user has tapped inside of a form, the `FrmHandleEvent()` checks the coordinates and sends them to the user interface element that was tapped. For example, if the user taps a command button, `FrmHandleEvent()` calls `CtlHandleEvent()`, which enqueues a `ctlSelectEvent` for that button. If the user taps in a text field, `FrmHandleEvent()` calls `FldHandleEvent()`, which enqueues a `fldEnterEvent` and so on.

Your application only receives events intended for it. If the user has tapped an area of the display that is outside of the bounds of your application's forms, your application does not receive the pen events. For example, if the user has tapped the status bar or in a window that is displayed by the status bar, the Status Bar Manager passes the event to the appropriate part of the system.

If the pen events are within the input area, the currently active pinlet receives and handles the events (see "Input Area" on page 5) by converting them into `keyDownEvent`s. Each `keyDownEvent` contains a character. This character may either be text input, such as a letter or number, or a virtual character. A **virtual character** is a character that performs an action, such as moving to the next field or launching a new application.

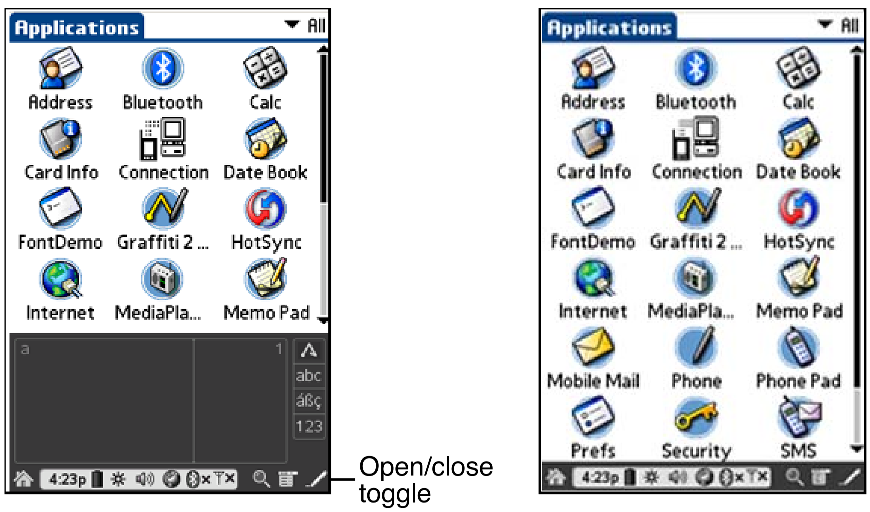
Most applications do not need to receive or handle pen events. Of course, there are exceptions. If you want to write a drawing application, you need to receive pen events to capture what the user has drawn. You might also want to directly receive pen events if you're writing a game.

If you want to capture pen events, you simply need to check for them in your form's event handler, which is called before `FrmHandleEvent()` is called. (`FrmDispatchEvent()` calls `FrmHandleEvent()`.) See *Exploring Palm OS: Programming Basics* for more information about the event loop.

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, such as the input area on the Palm V handheld. A **dynamic input area** is one that is implemented in software. On most devices with a dynamic input area, 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 (see Figure 1.2).

Figure 1.2 Dynamic input area



In either case, when the user taps or drags the pen in the input area, those pen events are turned into a series of `keyDownEvent`s representing textual input that is sent to the active window. The user interface elements handle displaying that data on the screen. You typically do not have to write any extra code to display the text the user has entered.

If your application is running on a device with a dynamic input area, it is strongly recommended that you set up constraint resources for each form so that they can respond by expanding and contracting or by moving as the user opens and closes the input area. Doing so is described in *Exploring Palm OS: User Interface*.

Resizing is the only interaction with the input area that most applications have. It is possible to open and close the input area programmatically and to have some interaction with the pinlet that runs in the dynamic input area. See Chapter 2, "Working with the Dynamic Input Area," on page 7.

Hardware Controls

All Palm Powered™ devices have a power button and some of the following:

- One or more application buttons. Most handhelds have four application buttons for Datebook, Address Book, and so on.
- Scroll buttons that scroll text fields and some forms up or down.
- Contrast/brightness controls.
- A thumb wheel used for form navigation.
- A five-way rocker used for form navigation.
- A built-in keyboard.

Hardware controls or buttons are often called **hard keys**. When a hard key is pressed, Palm OS® generates a `keyDownEvent` containing a virtual character specifying which button was pressed. When a hard key is released, a `keyUpEvent` containing the same information is sent.

Most applications should let the system handle the virtual characters. For hard keys that are intended for form navigation (such as the scroll buttons, thumb wheels, or rocker keys), you may have to write code to perform the navigation. See *Exploring Palm OS: User Interface* for more information.

Sometimes, you may want to have special handling for a hard key. If so, see the chapter Chapter 4, "Customizing Hardware Input," on page 37. It describes how to respond to the virtual characters, remap the hard keys, and intercept them entirely.

2

Working with the Dynamic Input Area

This chapter describes how applications may interact with the dynamic input area. It covers:

Although dynamic input areas are not new to Palm Powered™ devices, Palm OS® Cobalt version 6.0 provides the first built-in support for dynamic input areas.

This chapter does not cover how an application's windows should resize when the user opens and closes the input area; it covers how an application might want to control the input area itself. See *Exploring Palm OS: User Interface* for information about resizing your application.

Checking the Dynamic Input Area Features

Before you can use any of the API described in this chapter, you must make sure that the dynamic input area API is available. Test the `pinFtrAPIVersion` feature as shown in Listing 2.1.

Listing 2.1 Checking the dynamic input area feature

```
err = FtrGet(pinCreator, pinFtrAPIVersion, &version);
if (!err && version) {
   //dynamic input area exists
}
```

If this feature is defined, a manager called the Pen Input Manager controls the input area and notifies the application of any changes in the input area state.

Do not assume that if this feature is not present, the device has a static input area. Some devices have no input area at all. For example, the Handspring Treo has a built-in keyboard and thus forgoes having any input area altogether. All textual user input is typed. On devices that don't have the dynamic input area, the API described in this chapter has no effect.

If you need more information, the `sysFtrNumInputAreaFlags` feature indicates the device-specific capabilities of the input area (see Listing 2.2). "Input Area Flags Constants" on page 75 defines the flags that may be set in this feature constant.

Listing 2.2 Checking the input area capabilities

```
err = FtrGet(sysFtrCreator, sysFtrNumInputAreaFlags,
   &inputAreaFlags);
if (!err) {
   if (inputAreaFlags & grfFtrInputAreaFlagDynamic)
      // device has dynamic input area
   if (inputAreaFlags & grfFtrInputAreaFlagLiveInk)
      // device supports live ink
   if (inputAreaFlags & grfFtrInputAreaFlagCollapsible)
      // dynamic input area is collapsible.
}
```

Programmatically Opening and Closing the Input Area

In rare cases, it may be beneficial for a form to open the input area when the form itself is opened. For example, a password dialog might open the input area while the form is opened to save the user a tap. If your application has a form that *requires* text input (rather than simply having the ability to receive text input), you might also want to open the input area when the form is opened. Only do this if you're certain the user is always going to use the form for text input and never to read what was previously entered.

Opening and closing the input area is controlled by setting the input area state. The function `PINSetInputAreaState()` sets the input area state. Call this function in response to the `winFocusGainedEvent`. Be sure to preserve the previous input area state and restore it when your form is closed. See Listing 2.3.

Listing 2.3 Example of a form controlling input area state

```
uint16_t userInputAreaState = 0;
case winFocusGainedEvent:
   if (eventP->data.winFocusGained.window ==
         FrmGetWindowHandle(frmP)) {
      //First, preserve the current input area state,
      //which is likely to be the one the user prefers.
      userInputAreaState = PINGetInputAreaState();
      PINSetInputAreaState(pinInputAreaOpen);
   }
   break;

case frmCloseEvent:
   PINSetInputAreaState(userInputAreaState);
   break;
```

IMPORTANT: Be careful not to set the input area state too much. If the input area is opened and closed automatically in too many instances, the result may be a jumpy user interface that produces a jarring user experience. It is best to let your users decide what they want to do.

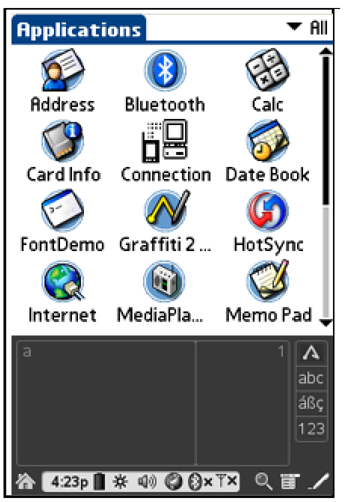
You should open the input area in response to `winFocusGainedEvent` to allow for the possibility that a dialog displayed by another process, such as a system dialog or a slip window, might close it. When the dialog is dismissed and control returns to your application, you then might need to reopen (or reclose) the input area. You won't receive a `frmLoadEvent` or `frmOpenEvent` because your form is already loaded and opened. Instead, you'll get the `winFocusGainedEvent`. Well-behaved dialogs that enforce a certain input state should restore the input area state when they are closed (as shown in Listing 2.3), so you should never encounter a situation where you need to re-open the input area after a dialog is displayed; however, to be on the safe side, open the input area in `winFocusGainedEvent`.

Interacting with Pinlets

A **pinlet** is a module with a user interface that displays in the input area. A pinlet's purpose is to receive pen events in the input area and translate them into character data.

There are two basic types of pinlets. A **handwriting recognition pinlet** converts user pen strokes to characters using either the Graffiti® 2 engine or some other handwriting recognition software (see Figure 2.1).

Figure 2.1 Handwriting recognition pinlet

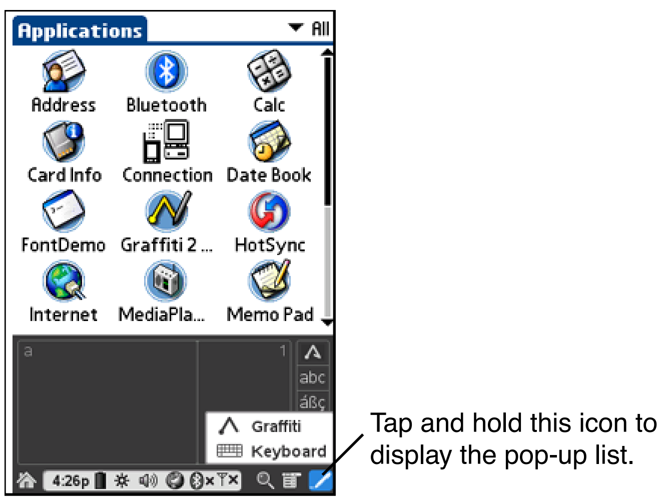


A **keyboard pinlet** provides a set of buttons that the user taps to enter a corresponding character (see Figure 2.2).

Figure 2.2 Keyboard pinlet

On devices with a dynamic input area, there is at least one pinlet available, and there may be several others. The button on the status bar that controls whether the input area is opened or closed also controls which pinlet is active. If the user holds the pen down on that button, it displays a menu from which the user can choose a new pinlet (see Figure 2.3).

Figure 2.3 Changing the active pinlet



Users can also switch between the default handwriting recognition pinlet and the default keyboard pinlet by pressing the buttons shown on the right side of the PalmSource-provided handwriting recognition and keyboard pinlets.

Applications typically do not need to interact with the pinlet. Your application receives the character data that the pinlet produces in the form of `keyDownEvent`s. If your application does need information from or about a pinlet, it uses calls to Pen Input Manager to obtain that information. The next several sections describe how an application might interact with the pinlet through the Pen Input Manager:

Changing the Active Pinlet

An application might want to control which pinlet is active. To do so, it can call `PINSwitchToPinlet()`. Before it can do so, it must know which pinlets are available. Applications can use the following functions:

- `PINGetCurrentPinletName()` returns the name of the currently active pinlet.
- `PINGetPinletInfo()` returns the name or the kind of a specific pinlet. This function references a pinlet by its index in the Pen Input Manager's pinlet list. An application can call `PINCountPinlets()` to obtain the upper limit for the pinlet list.

Suppose an application wants to ensure that the pinlet used with it is a keyboard pinlet. It might contain code similar to that shown in Listing 2.4.

Listing 2.4 Switching to a keyboard pinlet

```
uint16_t index = 0;
uint32_t info;
char *activePinlet;
Boolean found = false;

// First find a keyboard pinlet.
while (!found && (index < PINCountPinlets())) {
```

```
   PINGetPinletInfo(index, pinPinletInfoStyle, &info);
   if (info == pinPinletStyleKeyboard) {
      found = true;
   } else {
      index++;
   }
}
// We now need to see if we need to change pinlets.
if (found) {
   PINGetPinletInfo(index, pinPinletInfoComponentName,
     &info);
   activePinlet = PINGetCurrentPinletName();
   if (strcmp((char *)info, activePinlet)) {
      // If the names are different, we need to switch.
      PINSwitchToPinlet((const char *)info,
         pinInputModeNormal);
   }
}
```

Querying Alternative Input Systems

Many Palm Powered devices come with hardware solutions for text entry. These solutions are called **alternative input systems** because they are not controlled using the Pen Input Manager.

The primary example of an alternative input system is a detachable keyboard that is sold separately from the device, like the keyboards available for many Palm handhelds. The alternative input system is not required to be a keyboard. In the future, it may be some other sort of device such as a speech recognizer. The requirements for an input system to be considered an "alternative input system" are:

- It must be a way for the user to enter textual data. A jog dial is not an alternative input system.
- It must be on a device with an input area. The keyboard on a Handspring Treo is not an alternative input system because there is no other input system available on that device.

Applications might want to decide to open or close the input area based on whether an alternative is available. For example, a password dialog might want to open the dynamic input area to ensure that the user has a means of entering the password, but before it does so, it could check for an alternative input system using `PINAltInputSystemEnabled()`. If that function returns `true`, it could leave the input area state closed because the user already has a means of entering data.

If you use `PINAltInputSystemEnabled()` to decide when to open or close the input area, you should also register to receive the notifications `sysNotifyAltInputSystemEnabled` and `sysNotifyAltInputSystemDisabled` to account for the fact that users might attach or detach this alternative input system while your dialog is being displayed. See Listing 2.5.

Listing 2.5 Registering for alternative input system notifications

```
uint32_t PilotMain(uint16_t cmd, MemPtr, cmdPBP,
  uint16_t launchFlags)
{
  ...
  case sysAppLaunchCmdNormalLaunch:
    SysNotifyRegister(appDBID,
        sysNotifyAltInputSystemEnabled, NULL,
        sysNotifyNormalPriority, NULL, 0);
    SysNotifyRegister(appDBID,
        sysNotifyAltInputSystemDisabled, NULL,
        sysNotifyNormalPriority, NULL, 0);
    ...
    break;
  case sysAppLaunchCmdNotify:
    if (cmdPBP->notify->notifyType ==
        sysNotifyAltInputSystemEnabled)
      PINSetInputAreaState(pinInputAreaClosed);
    else if (cmdPBP->notify->notifyType ==
        sysNotifyAltInputSystemDisabled)
      PINSetInputAreaState(pinInputAreaOpen);
    ...
    break;
}
```

Setting the Pinlet Input Mode

The **input mode** specifies how the pinlet converts the next set of strokes into characters. For example, in the normal input mode on an ISO Latin device, strokes are converted to lowercase letters. If the mode is set to shift, the next stroke is converted into an uppercase letter as if the user has pressed the Shift key on a keyboard. For

Japanese systems, the input mode indicates whether the input is in Hiragana or Katakana characters.

Note that the input mode is different from the FEP mode. The Graffiti 2 handwriting recognition engine does not use Hiragana or Katakana input modes; however, on some Japanese devices writing Graffiti 2 strokes generates Hiragana or Katakana character, but that is dependent on the **FEP mode**, not the pinlet input mode. The same devices might have a Japanese keyboard pinlet that does use the Hiragana and Katakana input modes.

The function `PINSetInputMode()` sets the pinlet input mode, and `PINGetInputMode()` retrieves the current input mode.

The user interface elements use `PINSetInputMode()` to set the shift state automatically. On most ISO Latin 1 devices, the state is set automatically to Shift mode after a period or other sentence terminator followed by a space.

Note that the auto-shifting rules are language-specific, since capitalization differs depending on the region. These rules depend on the version of the ROM, the market into which the device is being sold, and so on.

Earlier releases of Palm OS used something called the **shift state** for the same purpose for which the input mode is now used. You placed a shift indicator (GSI) on all forms that contained an editable text field to show the shift state. If there is a dynamic input area, the pinlet displays its own indication of what the input mode is. However, your forms should still define a GSI to allow for devices with static input areas or no input areas. The GSI is disabled for you if a dynamic input area is present.

Summary

Pen Input Manager Functions	
PINAltInputSystemEnabled()	PINClearPinletState()
PINCountPinlets()	PINGetInputAreaState()
PINGetCurrentPinletName()	PINGetInputMode()
PINGetPinletInfo()	PINSetInputAreaState()
PINSetInputMode()	PINSwitchToPinlet()
PINShowReferenceDialog()	

3

Customizing the Dynamic Input Area

The dynamic input area is simply a window that displays the user interface of a separate executable called a **pinlet**. A pinlet receives user input in the form of pen taps or pen strokes and converts that input into character data that an application can use.

Pinlets come in two general types. **Keyboard pinlets** offer a user interface where users tap the characters they want to enter. **Handwriting recognition pinlets** interpret pen strokes as handwriting. Handwriting recognition pinlets typically use a handwriting recognition engine. You may use the provided Graffiti® 2 engine or replace it with one of your own.

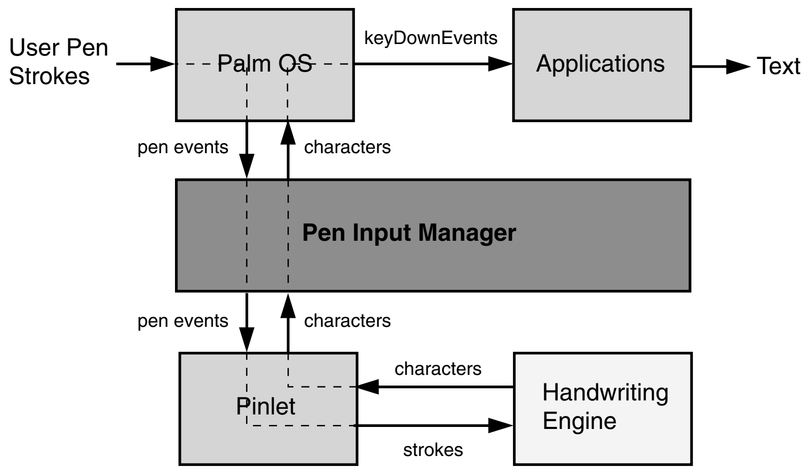
A single device may have more than one pinlet installed on it, and users may control which pinlet they want to use.

This chapter provides guidelines for implementing pinlets. It covers:

How Pinlets Work

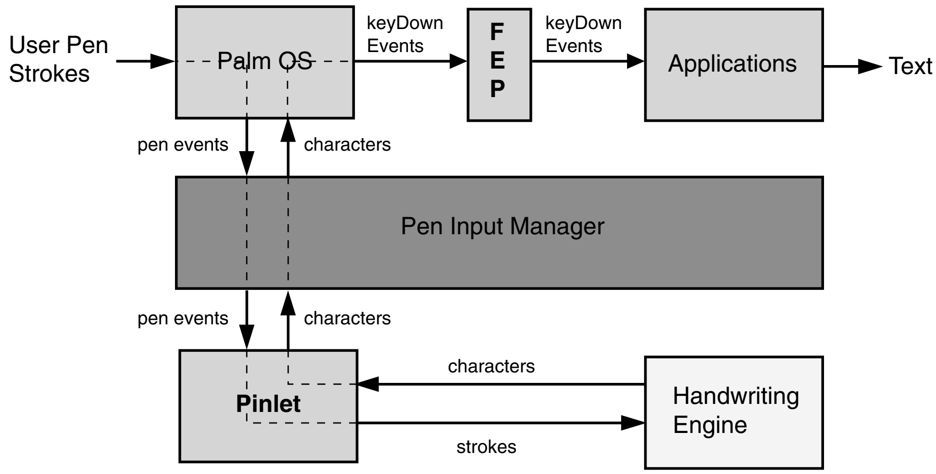
In most cases, pinlets interact only with the Pen Input Manager. The Pen Input Manager provides a window into which the pinlet draws its user interface. Through this window, the pinlet receives all pen events. The pinlet converts the pen events to character data and passes that back to the Pen Input Manager. Handwriting recognition pinlets may also use a handwriting recognition engine to interpret pen strokes. (See Figure 3.1.)

Figure 3.1 Flow of data through pinlets



Some systems use a Front-End Processor (FEP) to convert characters. For example, many existing Japanese devices use handwriting recognition engines to convert pen strokes to Romaji (the Roman alphabet). On these devices, text fields send sequences of characters to the FEP, which converts the characters to Hiragana or Katakana characters. The user can perform a further translation from these alphabets into Kanji. The character recognition performed by the Pen Input Manager is completely separate from the FEP. If you were to add the Pen Input Manager and pinlets to a device with a FEP, the Pen Input Manager still sends its events to Palm OS®, and the text field code in Palm OS sends those characters to the FEP (see Figure 3.2).

Figure 3.2 Flow of data with a FEP



In rare circumstances, a pinlet depends so much on a FEP that it should only be made active if that FEP is active. For this reason, the pinlet has a FEP class attribute to specify which FEPs the pinlet works with. If the FEP class attribute is defined, the status bar does not advertise the pinlet as available unless it works with a FEP that is currently enabled.

A pinlet represents a single means of receiving pen input, and only one pinlet is active at a time. Writing characters is one means of receiving input that is controlled by one pinlet. Tapping characters on an on-screen keyboard is another means of receiving input and thus is controlled by another pinlet. You may have more than one means of receiving input within one pinlet if one pinlet can use multiple FEPs, but in all other cases, you should design one pinlet per character input system.

Building Pinlets and Handwriting Recognition Engines

To build a pinlet, create a shared library of type `'pnlt'` (defined by the constant `sysFileTPinletApp`).

A handwriting recognition engine is also a shared library. If your pinlet is a handwriting recognition pinlet, link it with the engine you want to use. If you want to use the Graffiti 2 engine, link with the `Graffiti2Lib` library, which is included in the SDK.

If you want to replace the Graffiti 2 engine with your own, create a shared library project with type `'libr'` for the engine and export all of the functions described in Chapter 7, "Handwriting Recognition Engine," on page 55. Then link your pinlet with this library.

You do not have to separate your handwriting recognition code from your pinlet. You can interpret strokes in the pinlet directly if you don't intend to reuse or share your handwriting recognition engine.

Starting Up and Shutting Down a Pinlet

Programmatically, a pinlet looks very similar to a Palm OS application: it starts up by receiving and responding to launch codes, and it has an identical event loop that receives the same types of events that an application does.

Startup

The Pen Input Manager starts up a pinlet when the user selects that pinlet from the status bar or an application calls the `PINSwitchToPinlet()` function.

To start up the pinlet, the Pen Input Manager sends launch codes to the pinlet's `PilotMain()` function:

```
uint32_t PilotMain(uint16_t cmd, void *cmdPBP,
uint16_t launchFlags)
```

The pinlet must respond to the following launch codes:

`sysPinletLaunchCmdLoadProcPtrs`
: The Pen Input Manager calls pinlet functions to set and retrieve the input mode, display a help dialog, and clear the input state. The Pen Input Manager sends this launch code to retrieve pointers to those pinlet functions. The pinlet should return its function pointers in a `PinletAPIType` structure and pass it back as the *`cmdPBP`* parameter. (The rest of this chapter describes how and when those functions are called.)

`sysAppLaunchCmdPinletLaunch`
: The pinlet has become the active pinlet, and it should initialize itself.

When the pinlet is launched, it should:

- Initialize its state.
- Display its user interface.
- If it is a handwriting recognition pinlet, it should start up the engine it uses as described in "Starting up the Handwriting Recognition Engine."
- Start its event loop.

  The event loop is identical to the event loop used in an application.

Starting up the Handwriting Recognition Engine

To start up the handwriting recognition engine, you call `HWRInit()` and pass it a `HWRConfig` structure. This structure identifies the following:

- The number of horizontal and vertical pixels per inch.
- The bounds of the area into which the user is allowed to draw strokes.
- The number and bounds of any special areas for special input modes, excluding the writing area for the normal input mode.

  For example, the Graffiti 2 engine in normal mode translates strokes into lowercase letters. The virtual silkscreen pinlet has two writing areas: one for letters and one for numbers. It specifies only one mode area in its `HWRInit()` call because one of its writing areas is for the normal mode.

Note that much of this information is dependent on the size of the pinlet's form. In Palm OS Cobalt, you do not know a form's size until runtime when your pinlet receives the `winResizedEvent`. Therefore, you may have to wait until you receive that event before you can start up the handwriting recognition engine. This is acceptable as long as you call `HWRInit()` before you call any other handwriting function.

Shutdown

When the user switches to a new pinlet, the Pen Input Manager sends the `appStopEvent` to your pinlet. In response to this event, the pinlet must shut down its event loop, clear all state, and exit. If your pinlet works with a handwriting recognition engine, you should call `HWRShutdown()` to shut down the engine.

You don't have to worry that you're getting an `appStopEvent` intended for some other application or process. The pinlet runs in its own thread and has its own event queue. It only receives events that are intended for it.

Presenting a User Interface

You create a pinlet's user interface in a resource file in the same way that you create a user interface for an application. The pinlet's user interface consists of one or more forms containing one or more user interface controls.

The resource file must also contain the following resources that are unique to pinlets:

Main Pinlet Form

The main form for your pinlet must be update-based. That is, it must contain a `WINDOW_CONSTRAINTS_RESOURCE` that does *not* have the back-buffer attribute set. Palm OS places the pinlet form in the appropriate window layer. If you try to specify the window layer or any other window creation attributes in the `WINDOW_CONSTRAINTS_RESOURCE`, they are ignored.

Try to design this form so that a minimum and preferred size of 65 standard coordinates high or less works well. This allows the application to be 160 coordinates high, which is the height used by all legacy application windows. Users prefer to see as much of the application as possible, so your pinlet should be as small as possible while still being usable.

IMPORTANT: Pinlets cannot display other windows. This means that you cannot include a pop-up list in a pinlet because a pop-up list is a window.

Remember that in Palm OS Cobalt, the system controls the size of each window. You cannot guarantee that your pinlet is a specific size. The form's event handler must respond to the `winResizedEvent` to find out what size it actually is, and you must only draw the pinlet's form in response to the `frmUpdateEvent`.

NOTE: When the Pen Input Manager closes the input area, the pinlet receives a `winResizedEvent` specifying a size of 0.

Pinlet Style

The Pen Input Manager needs to know the style of the pinlet. This pinlet style is used as a possible return value to `PINGetPinletInfo()`. Include in your pinlet's resource file a `SOFT_CONSTANT_RESOURCE` with ID 1000. Use one of the constants described in "Pinlet Styles" on page 78 for its value.

Internal Pinlet Name

The Pen Input Manager uses an internal pinlet name to identify each pinlet. Create a `STRING_RESOURCE` of ID 1001 and specify a name of the form:

`com.`*`companyName`*`.pinlet.`*`pinletName`*

where *`companyName`* is the name of your organization and could be used for all pinlets that you write. *`pinletName`* is the unique name for your pinlet.

This name is returned by the functions `PINGetCurrentPinletName()` and `PINGetPinletInfo()` and used as input to `PINSwitchToPinlet()`. It is never shown externally, so there is no need to localize this name.

Status Bar Icons and Name

The status bar displays an icon and name for all pinlets installed on the device. If the user holds the pen down on the input area icon, a list of the names and icons of all pinlets are displayed. This is how the user switches between pinlets.

Use the `APP_ICON_BITMAP_RESOURCE` resource type with ID 1001 to specify the status bar icon. This icon should be 15 standard coordinates wide by 11 coordinates high.

The external pinlet name is contained in an `APP_ICON_NAME_RESOURCE` with ID 1000. Ideally, the width of this name is no more than 100 coordinates, but it can be as wide as the screen if necessary. Because the status bar displays this name, it is localizable.

FEP Creator ID

If the pinlet is associated with a FEP, you must supply an `SOFT_CONSTANT_RESOURCE` with ID 1001 that gives the creator ID of the FEP.

Help Dialog

The function `PINShowReferenceDialog()` calls `PinletShowReferenceDialogProcPtr()` to display a help dialog that specifies how to enter characters.

Pinlets that use a handwriting recognition engine can call through to `HWRShowReferenceDialog()`, which displays the help for you. If you're not using a handwriting recognition engine, you should supply a help dialog in your resource file.

In the `WINDOW_CONSTRAINTS_RESOURCE` for the help dialog, specify `winLayerPriority` as the window layer. This ensures that the help dialog appears on top of the application form if necessary.

The **Edit** menu used in most Palm OS applications has a **Graffiti 2 Help** menu item that displays help for the Graffiti 2 engine only. If you use a different handwriting recognition engine and want to display help for it, include a control on the pinlet that does so.

IMPORTANT: Pinlets cannot display more than one window. To launch a help dialog, you must spawn a separate thread, give that thread a user interface context, and then display the dialog in that thread. See *Exploring Palm OS: System Management* for more information on multithreading.

Input Mode Indicator

The pinlet should display some visual indication of the current input mode. This indication is typically only given for unusual modes (anything other than normal). See "Considering the Input Modes" on page 27 for more information.

Interpreting Pen Strokes

The pinlet receives pen events, translates the events into characters, and passes the characters to the Pen Input Manager. The pinlet must decide how to interpret the pen strokes. It should respect the input mode set by the user or application, and it might need to store internal state or set timers when interpreting the pen events. It should respond to application requests to clear that internal state. This section discusses these issues:

Receiving Pen Events

Your pinlet has a form with its own event handler, just like an application has a form with an event handler. This event handler should check for and interpret user input. How it does so depends on the type of pinlet you are writing and how you've designed it. If you're doing a keyboard pinlet and each key is implemented as a button, you'll receive `ctlSelectEvent`s for each button the user taps.

If you are implementing a handwriting recognition pinlet, you should do something like the following:

1. On `penDownEvent`, check to see if the pen is down within the writing area. If so, track the pen.
2. On `penMoveEvent`s, record the points sent in the event.
3. On the `penUpEvent`, call `HWRProcessStroke()`, sending it the points you recorded upon each `penMoveEvent`.

   The handwriting recognition engine returns a `HWRResult` containing characters, an input mode indication, an inking hint, and a Boolean that indicates if a timeout should be set. More details about each of these are given later in this chapter.

TIP: If you're using a gadget to track the pen, call `FrmSetPenTracking()` in the gadget's event handler in response to `frmGadgetEnterEvent`. See the description of `FormGadgetHandlerType()` for more information.

Sending Results to Pen Input Manager

To send the character input to the Pen Input Manager, use the call `PINFeedChar()` or `PINFeedString()`. You must supply UTF8 data to these functions.

If you are writing a handwriting recognition pinlet that uses the Graffiti 2 engine, that engine returns characters in the device's native encoding. Check the `flags` field of each character that the engine has returned. If it is `pinCharFlagVirtual`, the character is a virtual character and you can pass that directly to `PINFeedChar()` without conversion. If no modifiers are set, the character is textual data. Use `TxtConvertEncoding()` to convert it to UTF8.

Considering the Input Modes

As described in the section "Setting the Pinlet Input Mode" on page 14, the input mode affects how pen events are converted to characters.

The pinlet receives a call to the function `PinletSetInputModeProcPtr()` when the application changes the input mode. It should also implement `PinletGetInputModeProcPtr()`, which is used to retrieve the current input mode from the pinlet.

If the pinlet uses a handwriting recognition engine, these functions can simply call through to the functions `HWRSetInputMode()` and `HWRGetInputMode()`. For the set function, it must also invalidate the display so that it is redrawn in response to the `frmUpdateEvent` to indicate the change in mode.

Note that the engine does not have to respect all input modes. Some modes might not make sense, in which case the engine sets the mode to a reasonably close value. For example, if a handwriting recognition engine does not implement caps lock mode, it might set the mode to shift instead. If the engine interprets the Latin alphabet and receives a request to switch to Hiragana, it could just remain in the default mode. For this reason, the pinlet should always call `HWRGetInputMode()` after calling `HWRSetInputMode()` to see if the change actually took place before invalidating the display.

If the pinlet performs the conversion to character data itself, it should take the input mode into consideration. It should store the input mode in such a way that it will be taken into consideration for the next series of pen events. Like the handwriting recognition engine, the pinlet does not have to respect all input modes.

Many handwriting recognition pinlets will have multiple "mode areas," that is, areas in which the pen stroke is interpreted in a certain way such as numeric or shifted. The effect of `PinletSetInputModeProcPtr()` on these mode areas is implementation-defined.

In addition to receiving `PinletSetInputModeProcPtr()`, a handwriting recognition pinlet may have to change its mode in response to the return value for `HWRProcessStroke()`. If the user has entered a stroke that changes the input mode, the engine sets the `inputMode` field of the returned structure. Pinlets need to check this value and change their input mode accordingly.

Handling Multistroke Characters

Handwriting recognition pinlets might need to deal with multistroke characters. Consider the K character in Graffiti 2 writing. This character takes two strokes to draw, and the first stroke is identical to the stroke for an L character.

If the pinlet is working with a handwriting recognition engine, it should send each stroke to the engine in separate `HWRProcessStroke()` calls. The handwriting recognition engine determines what to do with the information. It might do either or both of the following:

- If the first stroke could be interpreted as a character by itself, the engine might return the character but set the `uncertain` field in the structure to indicate that the character may later have to be erased.
- It might request a timeout value be set by returning `true` for the `timeout` field. The pinlet should set a timeout and if that time period elapses with no other strokes being received, call `HWRTimeout()`.

If the handwriting recognition engine uses the `uncertain` field, it might process an ambiguous character such as the stroke for the letter L as described in Table 3.1.

Table 3.1 Processing an L stroke option 1

User Action	Pinlet Action	Handwriting Recognition Engine Action
Draws the stroke for an L character	Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • a `chars` array containing the L character • the `uncertain` field set to 1 • the `timeout` field set to `true`
	• Stores the L character. • Calls `TimGetTicks()` and records the time. • Passes a timeout value to `EvtGetEvent()`. • Upon each `nilEvent`, checks to see if timeout period has elapsed.	

Then, it would process the second potential stroke as described in Table 3.2. Early versions of the Graffiti 2 engine worked in this manner.

Table 3.2 Processing a stroke after the L stroke option 1

User Action	Pinlet Action	Handwriting Recognition Engine Action
Draws the second stroke of the K character	• Cancels the timeout in response to any pen event. • Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • a `chars` array containing the K character • the `deleteUncertain` field set to 1 • the `timeout` field set to `false`
	• Discards the L character. • Calls `PINFeedChar()` with the K character.	
Draws a different character (instead of the second stroke for the K character)	• Cancels the timeout in response to any pen event. • Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • a `chars` array containing the new character • the `timeout` field set to `false`
	• Calls `PINFeedChar()` with the L character. • Calls `PINFeedChar()` with the new character.	
Does nothing (instead of drawing a second stroke)	• Calls `PINFeedChar()` with the L character. • Calls `HWRTimeout()`	Returns an `HWRResult` structure with the `timeout` field set to `false`.

The current Graffiti 2 engine does not use the `uncertain` field. It processes the L character as described in Table 3.3.

Table 3.3 Processing an L stroke option 2

User Action	Pinlet Action	Handwriting Recognition Engine Action
Draws the stroke for an L character	Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • an empty `chars` array • the `timeout` field set to `true`
	• Calls `TimGetTicks()` and records the time. • Passes a timeout value to `EvtGetEvent()`. • Upon `nilEvent`, check to see if timeout period has elapsed.	

Then it processes the next stroke as described in Table 3.4.

Table 3.4 Processing a stroke after the L stroke option 2

User Action	Pinlet Action	Handwriting Recognition Engine Action
Draws the second stroke of the K character	• Cancels the timeout in response to any pen event. • Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • a `chars` array containing the K character • the `timeout` field set to `false`
	• Calls `PINFeedChar()` with the K character.	

Table 3.4 Processing a stroke after the L stroke option 2

User Action	Pinlet Action	Handwriting Recognition Engine Action
Draws a different character (instead of the second stroke for the K character)	• Cancels the timeout in response to any pen event. • Calls `HWRProcessStroke()` in response to the `penUpEvent`.	Returns an `HWRResult` structure with: • a `chars` array containing the L character and the new character • the `timeout` field set to `false`
	• Calls `PINFeedChar()` with the L character. • Calls `PINFeedChar()` with the new character.	
Does nothing (instead of drawing a second stroke)	Calls `HWRTimeout()`.	Returns an `HWRResult` structure with: • a `chars` array containing the L character • the `timeout` field set to `false`
	• Resets the timeout. • Calls `PINFeedChar()` with the L character.	

Another case that might occur in both scenarios is that the pinlet may receive a `PinletClearStateProcPtr()` call indicating all internal state should be cleared. This occurs when the user has moved to a new text field, tapped a control, switched applications, or performed any other action that indicates that the pinlet should start over when interpreting the next set of events. In response to this call, the pinlet should:

- Clear its timeout, its input mode, and any other internal state that it keeps.

- Call `HWRClearInputState()` to have the handwriting recognition engine do the same.

Implementing Live Ink

Live ink is a popular feature with handwriting recognition pinlets in which the user's pen movement is echoed on the screen. This feature helps users understand which character their strokes become.

If you want to implement a live ink feature, do so in the pinlet. The handwriting recognition engine helps with this feature by returning information in the `inkHint` field. The handwriting recognition engine provides one of the following values:

`hwrInkHintNone`
: The pinlet should erase the last stroke drawn. Any strokes that were previously kept are still kept. This is the default behavior.

`hwrInkHintEraseAll`
: The pinlet should erase all strokes currently being displayed. This is typically sent when the engine has successfully converted a character.

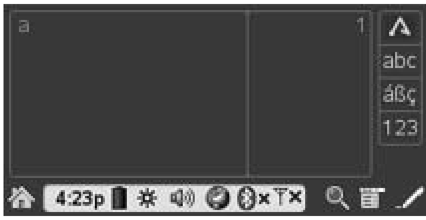
`hwrInkHintKeepAll`
: The pinlet should retain all strokes currently being displayed.

`hwrInkHintKeepLastOnly`
: The user is in the middle of a multistroke character. The pinlet should display only the last stroke. It should erase any previous strokes.

Specifying the Default Pinlet

The PalmSource-provided pinlets have buttons on the right side that allow the user to switch between the default handwriting recognition pinlet and the different panes of the default keyboard pinlet. All other pinlets are only available from the pop-up menu in the status bar.

Figure 3.3 Default pinlet buttons



If you want users to be able to access your pinlet from these buttons, you can set your pinlet as either the default handwriting recognition pinlet or the default keyboard pinlet. To do so, use `PINSetDefaultPinlet()`.

Guidelines for Default Pinlets

If you want your pinlet to be selected as a default, follow these guidelines:

- Users should be allowed to decide which pinlets they want to be the default. You might install a separate application that allows the user to set preferences for your pinlet and allows them to specify your pinlet as the default.
- There are only two possible default pinlets, one handwriting and one keyboard. The three buttons for keyboard pinlets access different input modes of a single default keyboard pinlet, as described in Figure 3.3.
- If your pinlet does something other than recognize handwriting or display a keyboard, do not set it as a default pinlet. If, for example, your pinlet is written specifically to work with a certain set of applications, you should have the applications call `PINSwitchToPinlet()` to make that pinlet active. Do not set the default pinlet.

User Interface Considerations

If you want your pinlet to be chosen as a default, its user interface should include a way to select the other style of default pinlet. That is, if yours is a handwriting recognition pinlet, it should include a button that allows the user to switch to the keyboard pinlet. If yours is a keyboard pinlet, it must include a way to select the default handwriting recognition pinlet.

On devices with no dynamic input area when the user opens the keyboard dialog, the default keyboard pinlet is displayed in the dialog. In this case, the user interface must *not* include a button that switches to the default handwriting recognition pinlet. When a keyboard pinlet is displayed in a dialog, it is opened with the input mode `pinInputAreaNone`.

There is no way to replace the images that appear in the buttons shown in Figure 3.3 on page 34. Therefore, if you are replacing the default keyboard pinlet, you cannot replace the icons in the default handwriting recognition pinlet with your own icons.

Summary

Pinlet Functions

`PINFeedChar()`
`PINFeedString()`
`PINSetDefaultPinlet()`
`PINGetDefaultPinlet()`

`PinletClearStateProcPtr()`
`PinletGetInputModeProcPtr()`
`PinletSetInputModeProcPtr()`
`PinletShowReferenceDialogProcPtr()`

Handwriting Recognition Engine Functions	
`HWRShutdown()`	`HWRGetInputMode()`
`HWRInit()`	`HWRProcessStroke()`
`HWRShowReferenceDialog()`	`HWRClearInputState()`
`HWRSetInputMode()`	`HWRTimeout()`

4

Customizing Hardware Input

This chapter describes how to work with the hard keys on the device. As described in Chapter 1, "Receiving Input," you typically let the system handle all hard keys. However, some applications may want to perform the following tasks:

Replacing a Built-in Application

Palm OS® contains system-wide preferences that the user sets to have the system work they way he or she likes. There are system preferences to remap the application hard keys and have them launch something other than the default applications. You should respect the user's preferences, but you are allowed to set the preference after asking the user if you may do so.

Suppose you are writing a replacement for the built-in Address Book application. You want to make it more convenient for your users to remap the Address Book button, so you might display an alert that asks first-time users if they want the button remapped. If they tap OK, then you should call `PrefSetPreference()` with the new value. See Listing 4.1.

Listing 4.1 Remapping the hard key to launch your application

```
if (PrefGetPreference(prefHard2CharAppCreator !=
    myAppCreatorId)) {
  if (FrmAlert(MakeMeTheDefaultAlert) == 0) {
    /* user said OK to change */
    PrefSetPreference(prefHard2CharAppCreator,
      myAppCreatorId);
  }
}
```

See *Exploring Palm OS: System Management* for more information about setting and getting user preferences.

Remapping the Hard Keys

Sometimes, you want to remap the hard key only while your application is running. A game might want to remap the hard keys to perform some special action such as launching missiles or moving pieces around a board.

As explained previously, when the user presses a hard key, Palm OS creates a `keyDownEvent` containing a virtual character that specifies which key was pressed. The `SysHandleEvent()` function handles most of these `keyDownEvents`. If you want to remap the hard keys, you must intercept them before `SysHandleEvent()`. See Listing 4.2.

Listing 4.2 Intercepting hard key events

```
void EventLoop (void)
{
  uint16_t error;
  EventType event;
  boolean handled = false;

  do {
    EvtGetEvent(&event, evtWaitForever);

    //If user pressed first hard key, do something special.
    if ((event.eType == keyDownEvent) &&
       (event.data.keyDown.chr == vchrHard1) &&
       (event.data.keyDown.modifiers & commandKeyMask)) {
```

```
      handled = HardKeyHandleEvent(&event);
    }

    //Proceed with normal event loop.
    if (!handled) {
      if (!SysHandleEvent(&event))
        if (!MenuHandleEvent(NULL, &event, &error)
          if (!ApplicationHandleEvent(&event))
            FrmDispatchEvent(&event);
  } while (event.eType != appStopEvent);
}
```

In general, only games should remap the hard keys. Users expect the hard keys to behave as they have set them up to behave in the system preferences.

Table 4.1 lists the virtual characters that map to hardware controls on many devices. Remember that not all devices support all of these controls. See `Chars.h` for a complete list of virtual characters.

Table 4.1 Hard key virtual characters

Character	Hard Key
`vchrHard1`	Usually launch Datebook
`vchrHard2`	Usually launches Address Book
`vchrHard3`	Usually launches ToDo
`vchrHard4`	Usually launches Memo
`vchrHardPower`	Power button
`vchrHardCradle`	Button on cradle
`vchrHardCradle2`	Button on cradle
`vchrHardContrast`	Contrast button
`vchrHardAntenna`	Antenna switch
`vchrHardBrightness`	Brightness button
`vchrHard5`	Licensee-specific
`vchrHard6`	Licensee-specific

Table 4.1 Hard key virtual characters *(continued)*

Character	Hard Key
`vchrHard7`	Licensee-specific
`vchrHard8`	Licensee-specific
`vchrHard9`	Licensee-specific
`vchrHard10`	Licensee-specific
`vchrRockerUp`	5-way rocker up
`vchrRockerDown`	5-way rocker down
`vchrRockerLeft`	5-way rocker left
`vchrRockerRight`	5-way rocker right
`vchrRockerCenter`	5-way rocker center
`vchrThumbWheelUp`	Thumb-wheel scroll up
`vchrThumbWheelDown`	Thumb-wheel scroll down
`vchrThumbWheelPush`	Thumb-wheel push center
`vchrThumbWheelBack`	Thumb-wheel back button

Disabling the Hard Keys

In very rare circumstances, you may want to disable the hard keys entirely.

WARNING! Do not disable the hard keys unless you have a very good reason. If you are writing a general-purpose, third-party consumer application, never disable the hard keys. Do so only if you are writing an enterprise-level application and your client insists that the device must never be used as a personal digital assistant.

There are two approaches to disabling the hard keys:

- Intercept the `keyDownEvent`s containing the virtual characters of the keys that you want to disable and ensure that they are never passed to `SysHandleEvent()`. This approach is similar to that shown in Listing 4.2 on page 38.

Keep in mind that modal dialogs and alerts run their own event loops. If the user presses a hard key while an alert is being displayed, the alert calls `SysHandleEvent()`, and allows your application to exit. If you do not want this behavior, look for and discard the `appStopEvent` after your application returns from a dialog or alert handler.

- Use the function `KeySetMask()` to disable the hard keys while your application is active. Listing 4.3 shows an example that disables the four application hard keys while an application is running.

Listing 4.3 Using KeySetMask()

```
void StartApplication(void) {
   uint32_t disableKeyMask = keyBitHard1 | keyBitHard2 |
      keyBitHard3 | keyBitHard4;

   KeySetMask(~disableKeyMask);
   ...
}

void StopApplication(void) {
   KeySetMask(keyBitsAll);
}
```

Keep in mind that `KeySetMask()` only disables hardware buttons. You won't be able to disable the Application Launcher icon in the status bar, for example.

Summary

Hard Key Functions

`KeyCurrentState()`
`KeyRates()`
`KeySetMask()`

Part II
Reference

This part contains reference material for the Input Services managers. It covers:

5

Low-Level Events Reference

This chapter describes the lowest level of events that an application may need to handle. It contains the following sections:

Event Constants

Key Modifier Constants

Purpose Used as the modifiers field of a `keyDownEvent`.

Declared In `CmnKeyTypes.h`

Constants

`#define appEvtHookKeyMask 0x0200`
System use only.

`#define autoRepeatKeyMask 0x0040`
Event was generated due to auto-repeat.

`#define capsLockMask 0x0002`
The handwriting recognition engine is in caps lock mode.

`#define commandKeyMask 0x0008`
The menu command stroke or a virtual key code.

`#define controlKeyMask 0x0020`
Not implemented. Reserved.

`#define doubleTapKeyMask 0x0080`
Not implemented. Reserved.

`#define libEvtHookKeyMask 0x0400`
System use only.

`#define numLockMask 0x0004`
The handwriting recognition engine is in numeric-shift mode.

`#define optionKeyMask 0x0010`
Not implemented. Reserved.

`#define poweredOnKeyMask 0x0100`
The key press caused the system to be powered on.

`#define shiftKeyMask 0x0001`
No longer used.

`#define willGoUpKeyMask 0x0800`
Set if a `keyUpEvent` will be sent.

`#define softwareKeyMask 0x1000`
Set if the key event was generated by software (such as the handwriting recognition engine), or clear if it was generated by pressing an actual hard key.

Events

This section describes the lowest level events that send user input to an application. Further events are described in other *Exploring Palm OS* volumes. See in particular *Exploring Palm OS: Programming Basics* and *Exploring Palm OS: User Interface*.

keyDownEvent

Purpose Sent when the user draws a character in the input area or presses one of the hard keys on the device.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _KeyDownEventType {
  wchar32_t chr;
  uint16_t keyCode;
  uint16_t modifiers;
} keyDown
```

Fields

`chr`
: The character code in the device-specific character encoding.

`keyCode`
: Unused.

`modifiers`
: 0, or one or more of the Key Modifier Constants.

Comments The `chr` field does not necessarily contain a printable character. If the modifiers field has the `commandKeyMask` bit set, then the character is a virtual character. Virtual characters generally correspond to an action that the system should take, such as launching a different application or adjusting the contrast.

Example The structure shown in the prototype section is the definition for the `data` field of the `EventType` structure. Access the information stored in the `data` field in this way:

```
wchar32_t chr = eventP->data.keyDown.chr;
```

keyHoldEvent

Purpose This event is not currently used.

keyHoldEvent5

Purpose Sent when the user holds a hard key on the device.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _KeyHoldEventType {
  wchar32_t chr;
  uint16_t keyCode;
  uint16_t modifiers;
} keyHold
```

Fields

`chr`
: The character code.

`keyCode`
: Unused.

`modifiers`
: 0, or one or more of the Key Modifier Constants.

Comments This event is sent when a hardware key is held for one second. (Note that the one-second timing may be modified by a Palm OS® licensee.)

Unlike `keyDownEvent`, `keyHoldEvent5` is sent for characters corresponding to hardware buttons only. If the device contains a hardware keyboard, holding a key results in a `keyDownEvent` and then, one second later, a `keyHoldEvent5`, and, ultimately, a `keyUpEvent`. If the user is using a keyboard pinlet, or some other software keyboard, the application receives only the `keyDownEvent`.

Only one `keyHoldEvent5` is sent for each press-and-hold of the key. You do not get a `keyHoldEvent5` for each additional second that the key is held.

A `keyHoldEvent5` is sent only for the most recently pressed key. For instance, if a key is pressed and held, and then another key is pressed within a second and before the first key is released, the following happens: a `keyHoldEvent5` is *not* sent for the first key, but a `keyHoldEvent5` *is* sent for the second key if it is held for a full second. Note that a `keyUpEvent` will be generated for both the first and second key as each key is released.

Compatibility By default, Palm OS Cobalt does not generate this event. Some devices that support a 5-way navigation button—such as the Handspring Treo 600 Smartphone—generate this event. Consult the licensee's developer documentation to see whether this event is generated by a particular device.

keyUpEvent

Purpose Sent when the user releases a hard key on the device.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _KeyUpEventType {
  wchar32_t chr;
  uint16_t keyCode;
  uint16_t modifiers;
} keyUp
```

Fields

`chr`
  The character code.

`keyCode`
  Unused.

`modifiers`
  0, or one or more of the Key Modifier Constants.

Comments Unlike the `keyDownEvent`, the `keyUpEvent` is sent for characters corresponding to hardware buttons only. If the device contains a hardware keyboard, the application receives `keyDownEvent` and `keyUpEvent` for each character typed. If the user is using a keyboard pinlet or some other software keyboard, the application only receives the `keyDownEvent`.

Example The structure shown in the prototype section is the definition for the `data` field of the `EventType` structure. Access the information stored in the `data` field in this way:

```
wchar32_t char = eventP->data.keyUp.chr;
```

keyUpEvent5

Purpose Sent when the user releases a hard key on the device.

For this event, the `data` field of the `EventType` structure contains the same structure as is shown for the `keyUpEvent`.

Declared In `Event.h`

Compatibility By default, Palm OS Cobalt does not generate this event. Some devices that support a 5-way navigation button—such as the Handspring Treo 600 Smartphone—generate this event. Consult the licensee's developer documentation to see whether this event is generated by a particular device.

penDownEvent

Purpose Sent when the pen first touches the digitizer.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _PenDownMoveEventType {
  uint16_t flags;
  int16_t pressure;
};
```

Fields

`flags`
: If this field contains `evtPenPressureFlag`, the pressure field is valid.

`pressure`
: The amount of pressure the user applied to the stylus while pressing the pen. If 0, no pressure was applied. A value greater than or equal to 0x1000 is considered heavy pressure.

The following fields in the `EventType` structure are set for this event:

`penDown`
: Always `true`.

`tapCount`
: The number of taps received at this location.

`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 Note that this information is passed with all events.

penMoveEvent

Purpose Sent when the user drags the pen on the digitizer. Note that several kinds of UI objects, such as controls and lists, track the movement directly, and no `penMoveEvent` is generated.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _PenDownMoveEventType {
  uint16_t flags;
  int16_t pressure;
};
```

Fields

`flags`
If this field contains `evtPenPressureFlag`, the pressure field is valid.

`pressure`
The amount of pressure the user applied to the stylus while pressing the pen. If 0, no pressure was applied. A value greater than or equal to 0x1000 is considered heavy pressure.

The following fields in the `EventType` structure are set for this event:

`penDown`
Always `true`.

`tapCount`
The number of taps received at this location.

`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).

penUpEvent

Purpose Sent when the pen is lifted from the digitizer. Note that several kinds of UI objects, such as controls and lists, track the movement directly, and no `penUpEvent` is generated.

For this event, the `data` field of the `EventType` structure contains the structure shown in the Prototype section, below.

Declared In `Event.h`

Prototype

```
struct _PenUpEventType {
  PointType start;
  PointType end;
} penUp
```

Fields

`start`
Draw window-relative start point of the stroke in standard coordinates.

`end`
Draw window-relative end point of the stroke in standard coordinates.

The following fields in the `EventType` structure are set for this event:

`penDown`
Always `false`.

`tapCount`
The number of taps received at this location.

`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).

6

Graffiti 2 Reference

This chapter provides reference material for the header file `GraffitiReference.h`.

Graffiti 2 Reference Functions and Macros

SysGraffitiReferenceDialog Function

Purpose Displays the Graffiti® 2 help dialog.

Declared In `GraffitiReference.h`

Prototype

```
void SysGraffitiReferenceDialog
    (ReferenceType referenceType)
```

Parameters → *referenceType*
Which reference to display. The only valid value is `referenceDefault`.

Returns Nothing.

See Also `HWRShowReferenceDialog()`, `PINShowReferenceDialog()`

7

Handwriting Recognition Engine

This chapter describes the handwriting recognition engine API. It covers:

The header file `HWREngine.h` declares the API that this chapter describes. For more information on using or implementing the handwriting recognition engine APIs, see Chapter 3, "Customizing the Dynamic Input Area," on page 17.

Handwriting Recognition Engine Structures and Types

CharData Struct

Purpose Information about a character. This structure is used as an entry in the `chars` array of `HWRResult`.

Declared In `HWREngine.h`

Prototype

```
typedef struct {
   wchar32_t chr;
   uint32_t flags;
} CharData
```

Fields `chr`

A character code. The Graffiti® 2 engine returns characters in the device-specific encoding.

`flags`
: 0 or `pinCharFlagVirtual` if the character is a virtual character.

HWRConfig Struct

Purpose Information used to initialize the handwriting recognition engine.

Declared In `HWREngine.h`

Prototype

```
typedef struct {
   uint16_t hDotsPerInch;
   uint16_t vDotsPerInch;
   RectangleType writingBounds;
   uint32_t numModeAreas;
   HWRConfigModeArea modeArea[kMaxHWRModeAreas];
} HWRConfig
```

Fields

`hDotsPerInch`
: The number of horizontal pixels per inch of resolution on the device. You can use `WinGetCoordinateSystem()` to determine this value.

`vDotsPerInch`
: The number of vertical pixels per inch of resolution on the device. You can use `WinGetCoordinateSystem()` to determine this value.

`writingBounds`
: The bounds of the writing area within the pinlet.

  The size of a window is not known until runtime. Depending on how you've created your pinlet's form, the size of its writing area bounds may change each time the pinlet is opened.

`numModeAreas`
: The number of areas within the `writingBounds` for different input modes.

`modeArea`
: An array of `HWRConfigModeArea` structures describing each mode area. A pinlet may provide different areas for different

input modes, such as punctuation, numbers, capital letters, accented characters, and so on.

The normal input mode is never given a mode area. For example, if a handwriting recognition engine's normal mode is to return lowercase letters and you've designed a pinlet that has an area for entering numbers on the right and lowercase letters on the left, you would supply only one entry in this array.

Not all handwriting recognition engines support all input modes, so these areas should be considered suggestions only.

Comments When you pass points to the engine using `HWRProcessStroke()`, use the same coordinate system as you use when you specify the bounds and the dots per inch. No handwriting recognition engine should require a particular coordinate system.

See Also `HWRInit()`

HWRConfigModeArea Struct

Purpose Information about a writing mode area. Used for the `modeArea` field within the `HWRConfig` struct.

Declared In `HWREngine.h`

Prototype

```
typedef struct {
   uint16_t writingMode;
   uint16_t reserved;
   RectangleType modeBounds;
} HWRConfigModeArea
```

Fields

`writingMode`
: One of the constants described in "Pinlet Input Modes" on page 76.

`reserved`
: Reserved for future use.

`modeBounds`
: The bounds of the mode area.

HWRResult Struct

Purpose Provides the character result of the pen stroke. Used as a return parameter for `HWRProcessStroke()`.

Declared In `HWREngine.h`

Prototype

```
typedef struct {
   CharData chars[kHWRMaxData];
   uint16_t numChars;
   uint16_t uncertain;
   uint16_t deleteUncertain;
   uint16_t inputMode;
   uint16_t inkHint;
   Boolean timeout;
   uint8_t reserved;
} HWRResult
```

Fields

`chars`
: One or more characters that the user drew. See `CharData`.

`numChars`
: The number of entries in the `chars` field.

`uncertain`
: The number of uncertain characters in the `chars` field. An uncertain character is one that may be the first stroke of a multistroke character. The uncertain characters are always at the end of the `chars` array, so if the value of uncertain is three and the `chars` array has five characters, the uncertain characters are the last three entries in the array.

`deleteUncertain`
: The number of uncertain characters that were previously sent and should be deleted before adding any new characters.

  For example, suppose the K character is a multistroke character that begins with a stroke that could also be the L character. If the user draws the stroke for an L, the engine might return the L and indicate that it is an uncertain character. If the next stroke completes the K character, the engine should return the K and set `deleteUncertain` to 1 to indicate that the pinlet should delete the previous L character.

Engines are not required to return ambiguous characters. Instead, they may hold them and use the `timeout` field to request that a timeout value be set.

`inputMode`
: The engine's current input mode. This is one of the constants described in "Pinlet Input Modes" on page 76. The stroke that was entered may have changed the input mode. If this value changed, the pinlet should change its display of the input mode indicator.

`inkHint`
: One of the Ink Hint Constants. Pinlets that provide live ink (which mirrors the stroke as the user writes it) use this field to determine when to erase the ink.

`timeout`
: `true` if the pinlet should set a timeout and wait for more user input. `false` otherwise. If the pinlet sets a timeout and that value is reached, it should call `HWRTimeout()`.

There is no guarantee that a timeout is reached if one is requested. If the user switches to a new text field or completes the stroke, the timeout is cancelled and the engine receives either a `HWRClearInputState()` call (when the user switches fields) or `HWRProcessStroke()` (if the user completes the stroke).

`reserved`
: Reserved for future use.

Handwriting Recognition Engine Constants

Ink Hint Constants

Purpose Used to set the `inkHint` field of a `HWRResult` structure.

Declared In `HWREngine.h`

Constants

`#define hwrInkHintNone 0`
Erase the last stroke drawn but preserve any previous strokes. This is the default behavior.

`#define hwrInkHintEraseAll 1`
All inking should be erased because either a full character was entered or a timeout value was reached.

`#define hwrInkHintKeepAll 2`
All current inking should remain on the screen.

`#define hwrInkHintKeepLastOnly 3`
Only the inking for the last stroke should remain on the screen.

Maximum Value Constants

Purpose Constants used to size the arrays in `HWRResult` and `HWRConfig`.

Declared In `HWREngine.h`

Constants

`#define kHWRMaxData 32`
The maximum number of bytes allowed in the `chars` field of `HWRResult`.

`#define kMaxHWRModeAreas 4`
The maximum number of input modes allowed in `HWRConfig`.

Handwriting Recognition Engine Functions and Macros

HWRClearInputState Function

Purpose Clears the handwriting recognition engine's internal state.

Declared In `HWREngine.h`

Prototype `status_t HWRClearInputState (void)`

Parameters None.

Returns `errNone` to indicate success or an error code if a failure occurs.

Comments Pinlets should call this function when they receive `PinletClearStateProcPtr()`. Pinlets receive `PinletClearStateProcPtr()` when the user performs an action that indicates that the internal state should be cleared. For example, suppose the user enters the first stroke of a multistroke character and then taps on the next field in the form. The user clearly intends not to complete the character. The engine should respond by clearing the internal state that it keeps to indicate that a character is in progress.

HWRGetInputMode Function

Purpose Returns the current input mode.

Declared In `HWREngine.h`

Prototype `uint16_t HWRGetInputMode (void)`

Parameters None.

Returns One of the constants described in "Pinlet Input Modes" on page 76.

See Also `PINGetInputMode()`, `PinletGetInputModeProcPtr()`

HWRInit Function

Purpose Initializes the handwriting recognition engine.

Declared In `HWREngine.h`

Prototype `status_t HWRInit (const HWRConfig *config)`

Parameters → *config*
Information that should be used to initialize the handwriting recognition engine. See `HWRConfig`.

Returns `errNone` upon success or an error code if a failure occurs.

Comments A pinlet calls this function when it is being launched or at any time before it begins using the engine. It should open any necessary databases and allocate global variables.

See Also `HWRShutdown()`

HWRProcessStroke Function

Purpose Interprets a pen stroke.

Declared In `HWREngine.h`

Prototype
```
status_t HWRProcessStroke(const PointType *points,
   uint32_t numPoints, HWRResult *result)
```

Parameters → *points*
An array of `PointType` structures giving the coordinates of each area of pen movement.

→ *numPoints*
The number of points in the *points* parameter.

← *result*
An `HWRResult` structure indicating what characters, if any, should be sent to Palm OS®.

Returns `errNone` upon success, `hwreErrPointBufferFull` if too many points were specified, or an error if the engine failed to recognize the stroke.

Comments Pinlets should call this function on a `penUpEvent`.

The sending of a pen stroke to this function does not always result in a character being returned. See the description of `HWRResult` for more information.

HWRSetInputMode Function

Purpose Sets the input mode.

Declared In `HWREngine.h`

Prototype `void HWRSetInputMode (uint16_t inputMode)`

Parameters → *`inputMode`*
One of the constants described in "Pinlet Input Modes" on page 76.

Returns Nothing.

Comments The input mode determines how the engine translates the next set of input from the user. The modes that an engine uses or accepts are up to the engine. Typically, in the normal or default input mode, the engine translates user input into lowercase letters. Translation into any other type of character or symbol requires a different input mode.

Not all input modes apply to all handwriting recognition engines. If the engine does not support the specified input mode, it should choose the closest equivalent that is supported, which could be the default mode.

See Also `PINSetInputMode()`, `PinletSetInputModeProcPtr()`

HWRShowReferenceDialog Function

Purpose Displays a dialog that provides user help for the handwriting recognition engine.

Declared In `HWREngine.h`

Prototype `void HWRShowReferenceDialog (void)`

Parameters None.

Returns Nothing.

Comments This function must display a dialog of some form. If no help is available, it should display an alert indicating no help is available.

See Also `PINShowReferenceDialog()`, `PinletShowReferenceDialogProcPtr()`

HWRShutdown Function

Purpose Frees the handwriting recognition resources.

Declared In `HWREngine.h`

Prototype `status_t HWRShutdown (void)`

Parameters None.

Returns `errNone` upon success or an error code if a failure occurs.

Comments A pinlet calls this function in response to the `appStopEvent`. It should close any open databases and deallocate global variables.

See Also `HWRInit()`

HWRTimeout Function

Purpose Specifies that a pinlet's timeout value has been reached.

Declared In `HWREngine.h`

Prototype `status_t HWRTimeout (HWRResult *result)`

Parameters ← *result*

A `HWRResult` structure indicating what characters, if any, should be sent to Palm OS.

Returns `errNone` upon success or an error if a failure occurs.

Comments The `result` parameter may or may not contain a character when a timeout value is reached.

For example, consider the Graffiti 2 handwriting recognition engine. In this engine, some strokes may be strokes for a single character or they may be the first stroke for a multistroke character. For example, the letter L could be the first stroke for several characters, such as a K or an I.

When this engine receives the stroke for an L in a `HWRProcessStroke()` call, it stores that stroke and returns `true` in the `timeout` field of the `HWRResult` structure indicating that the pinlet should set a timeout. If the timeout value is reached, the pinlet calls this function. The engine responds by returning in the *`result`* parameter an `HWRResult` structure that contains the letter L.

See Also "Handling Multistroke Characters" on page 28

8

Hard Keys Reference

This chapter provides reference material for manipulating the hard keys. This chapter covers:

For more information on working with the hard keys, see "Customizing Hardware Input" on page 37.

Hard Key Constants

Key State Values

Purpose Used by `KeyCurrentState()` and `KeySetMask()` to specify the hardware keys.

Declared In `CmnKeyTypes.h`

Constants

`#define keyBitsAll 0xFFFFFFFF`
A bit mask representing all hard keys.

`#define keyBitAntenna 0x00000100`
The antenna "key" on wireless devices that allow a user to go on or off line.

`#define keyBitContrast 0x00000200`
Contract key.

`#define keyBitCradle 0x00000080`
HotSync® button on the cradle.

`#define keyBitHard1 0x00000008`
The leftmost application key. This key often brings up the Datebook application.

`#define keyBitHard2 0x00000010`
The second application key from the left. This key often brings up the Address Book application.

`#define keyBitHard3 0x00000020`
The third application key from the left. This key often brings up the ToDo application.

`#define keyBitHard4 0x00000040`
The fourth application key from the left. This key often brings up a NotePad or Memo application.

`#define keyBitPageDown 0x00000004`
The scroll down button.

`#define keyBitPageUp 0x00000002`
The scroll up button.

`#define keyBitPower 0x00000001`
The power key.

`#define keyBitRockerCenter 0x00100000`
The center button within a five-way rocker.

`#define keyBitRockerDown 0x00020000`
The down button within a five-way rocker.

`#define keyBitRockerLeft 0x00040000`
The left button within a five-way rocker.

`#define keyBitRockerRight 0x00080000`
The right button within a five-way rocker.

`#define keyBitRockerUp 0x00010000`
The up button within a five-way rocker.

`#define keyBitThumbWheelBack 0x00008000`
On devices with a thumb wheel, the back button below the wheel.

`#define keyBitThumbWheelDown 0x00002000`
On devices with a thumb wheel, the thumb wheel has been turned downward.

`#define keyBitThumbWheelPush 0x00004000`
On devices with a thumb wheel, the thumb wheel has been pressed.

`#define keyBitThumbWheelUp 0x00001000`
: On devices with a thumb wheel, the thumb wheel has been turned upward.

Comments Not all devices support all of these keys.

Key Rate Constants

Purpose Specify special values for the key rate set by `KeyRates()`.

Declared In `CmnKeyTypes.h`

Constants `#define slowestKeyDelayRate 0xff`
: Represents the slowest possible delay before the key is recognized as being pressed or before a double-tap is recognized.

`#define slowestKeyPeriodRate 0xff`
: Represents the slowest possible auto-repeat rate.

Hard Key Functions and Macros

KeyCurrentState Function

Purpose Returns a bit field with bits set for each key that is currently depressed.

Declared In `KeyMgr.h`

Prototype `uint32_t KeyCurrentState (void)`

Parameters None.

Returns A bit mask with bits set for keys that are depressed. See Key State Values.

Comments Called by applications that need to poll the hardware keys.

If you want to remap the hardware keys, one way to do so is using this function and `KeySetMask()`. First use `KeySetMask()` to disable the hardware buttons that you want to remap. Then, periodically poll the keys, possibly in response to the `nilEvent` in your event loop, by using `KeyCurrentState()` to see if a hard key has been pressed. Then respond accordingly.

NOTE: This function has high overhead because it performs interprocess-communication. Use it sparingly.

Example To see if the first application hard key has been pressed, do the following:

```
if (KeyCurrentState() & keyBitHard1)
  ...
```

KeyRates Function

Purpose Gets or sets the key repeat rates.

Declared In `KeyMgr.h`

Prototype
```
status_t KeyRates (Boolean set,
   uint16_t *initDelayP, uint16_t *periodP,
   uint16_t *doubleTapDelayP,Boolean *queueAheadP)
```

Parameters

→ *set*
If `true`, settings are changed; if `false`, current settings are returned.

↔ *initDelayP*
Initial delay in ticks for an auto-repeat event.

↔ *periodP*
Auto-repeat rate specified as period in ticks.

↔ *doubleTapDelayP*
Maximum double-tap delay, in ticks.

↔ *queueAheadP*
If `true`, auto-repeating keeps queueing up key events if the queue has keys in it. If `false`, auto-repeat doesn't enqueue keys unless the queue is already empty.

Returns Always returns `errNone`.

Comments This function changes the auto-repeat rate of the hardware buttons. This might be useful to game applications that want to use the hardware buttons for control. The current key repeat rates should be restored before the application exits.

NOTE: This function has high overhead because it performs interprocess-communication. Use it sparingly.

Example The following code shows retrieving the default values, setting the key rates to the slowest possible values for the duration of the game, and then restoring the values.

```
uint16_t initDelay, period, doubleTapDelay;
Boolean queueAhead;

//Retrieve old values.
KeyRates(false, &initDelay, &period, &doubleTapDelay,
  &queueAhead);
// set my values
KeyRates(true, slowestKeyDelayRate, slowestKeyPeriodRate,
  slowestKeyPeriodRate, false);
...
//play game.
...
// game is over. Restore previous values.
KeyRates(true, initDelay, period, doubleTapDelay,
   queueAhead);
```

KeySetMask Function

Purpose Specifies which keys generate `keyDownEvent`s.

Declared In `KeyMgr.h`

Prototype `uint32_t KeySetMask (uint32_t` *`keyMask`*`)`

Parameters → *`keyMask`*

Mask with bits set for those keys to generate `keyDownEvents` for. See Key State Values.

Returns The old key mask.

See Also KeyCurrentState()

9

Keyboard

The chapter describes the API declared in the header file `Keyboard.h`.

Keyboard Functions and Macros

SysKeyboardDialog Function

Purpose Pops up the system keyboard if there is a field object with the focus.

Declared In `Keyboard.h`

Prototype `void SysKeyboardDialog (KeyboardType` *`kbd`*`)`

Parameters → *kbd*

One of the following:

`kbdAlpha`
Show alphabetic characters.

`kbdNumbersAndPunc`
Show numbers and some advanced punctuation.

`kbdAccent`
The International character set, made up of Latin characters with diacritic marks.

`kbdDefault`
Same as `kbdAlpha`.

Returns Nothing.

Comments On devices with a dynamic input area, this function switches the currently active pinlet to the default keyboard style pinlet.

On devices with a static input area (one that is silkscreened onto the device), this function displays a keyboard dialog with a text field and keyboard. The current text field's handle is reassigned to the keyboard dialog's text handle while the dialog is active.

10

Pen Input Manager

This chapter provides reference material for the Pen Input Manager API as declared in the header file `PenInputMgr.h`. It discusses the following topics:

For more information on using the Pen Input Manager, see Chapter 2, "Working with the Dynamic Input Area," on page 7.

Pen Input Manager Constants

Default Pinlet Constants

Purpose Used in `PINGetDefaultPinlet()` and `PINSetDefaultPinlet()` to specify which default should be retrieved or set.

Declared In `PenInputMgr.h`

Constants `#define pinDefaultPinletHWR 0`
The default handwriting recognition pinlet.

`#define pinDefaultPinletKeyboard 1`
The default keyboard pinlet.

Input Area States

Purpose The states that an input area can have.

Declared In `PenInputMgr.h`

Constants

`#define pinInputAreaOpen 0`

The dynamic input area is being displayed.

The dynamic input area is in this state after the user taps the input trigger to open it. An application might also request that the dynamic input area be opened by calling `PINSetInputAreaState()` with this state.

`#define pinInputAreaClosed 1`

The dynamic input area is not being displayed.

The dynamic input area is in this state after the user taps the input trigger to close it. An application also might request that the dynamic input area be closed by calling `PINSetInputAreaState()` with this state.

`#define pinInputAreaNone 2`

The input area is not dynamic, or there is no input area.

See Also `PINGetInputAreaState()`, `PINSetInputAreaState()`

Error Codes

Purpose Error codes returned by Pen Input Manager.

Declared In `PenInputMgr.h`

Constants

`#define pinErrInvalidParam (pinsErrorClass | 0x01)`

An invalid parameter was specified.

`#define pinErrNoSoftInputArea (pinsErrorClass | 0x00)`

This device does not have a dynamic input area.

`#define pinErrPinletNotFound (pinsErrorClass | 0x02)`

The specified pinlet does not exist on this device.

Feature and Version Constants

Purpose Specifies version and creator information.

Declared In `PenInputMgr.h`

Constants `#define pinAPIVersion pinAPIVersion2_0`
The current version of the Pen Input Manager API.

`#define pinAPIVersion1_0 0x01000000`
The first version of the Pen Input Manager API. This version of the API is used on Palm OS® 5.2 devices that have dynamic input areas.

`#define pinAPIVersion1_1 0x01103000`
Version 1.1 of the Pen Input Manager API. This version is used on Palm OS Garnet version 5.3 devices that have dynamic input areas.

`#define pinAPIVersion2_0 0x02003000`
Version 2.0 of the Pen Input Manager API, which is supported on Palm OS Cobalt version 6.0.

`#define pinCreator 'pins'`
The creator code with which a system feature is defined specifying the current Pen Input Manager API version.

`#define pinFtrAPIVersion 1`
The feature constant that stores the Pen Input Manager version number.

Input Area Flags Constants

Purpose Flags set in the `sysFtrNumInputAreaFlags` feature constant to specify the capabilities of a device's input area.

Declared In `PenInputMgr.h`

Constants `#define grfFtrInputAreaFlagDynamic 0x00000001`
The device has a dynamic input area.

`#define grfFtrInputAreaFlagLiveInk 0x00000002`
The device supports live ink.

`#define grfFtrInputAreaFlagCollapsible 0x00000004`
: The dynamic input area is collapsible. Some devices have an input area that is implement in software but that does not collapse.

`#define grfFtrInputAreaFlagLandscape 0x00000008`
: The device supports landscape mode.

`#define grfFtrInputAreaFlagReversePortrait 0x00000010`
: The device supports reverse portrait mode.

`#define grfFtrInputAreaFlagReverseLandscape 0x00000020`
: The device supports reverse landscape mode.

`#define grfFtrInputAreaFlagLefthanded 0x00000040`
: The device supports a special left-handed operation mode.

Comments Palm OS Cobalt version 6.0 currently does not support landscape, reverse portrait, left-handed, or reverse landscape modes. These constants are defined now to support future releases.

Pinlet Input Modes

Purpose Input modes a pinlet might have.

Declared In `PenInputMgr.h`

Constants

`#define pinInputModeNormal 0`
: The default mode. For the ISO-Latin character encoding, the normal mode translates strokes into lowercase letters.

`#define pinInputModeShift 1`
: The next stroke will be translated into an uppercase character rather than the normal lowercase characters.

`#define pinInputModeCapsLock 2`
: All of the characters will be uppercase until the mode is set to something else.

`#define pinInputModePunctuation 3`
: The next stroke will be interpreted as a punctuation mark or symbol, and then the mode is reset to normal.

`#define pinInputModeNumeric 4`
: The stroke will be interpreted as a numeric character.

`#define pinInputModeExtended 5`
The next stroke is a special symbol or part of the extended character set. The Graffiti® 2 handwriting recognition engine uses this mode for special symbols such as the trademark symbol.

`#define pinInputModeHiragana 6`
A Japanese pinlet is active and creates Hiragana characters.

`#define pinInputModeKatakana 7`
A Japanese pinlet is active and creates Katakana characters.

`#define pinInputModeCustomBase 128`
The first value available for custom input modes specific to a pinlet.

`#define pinInputModeCustomMax 255`
The last value available for custom input modes specific to a pinlet.

`#define pinInputModeUnShift 256`
Cancels a shift state. This mode is sent in the UI Library's text field code to set the pinlet back to normal mode after an auto-shift. The pinlet may receive this state when it is already in normal mode.

Comments The input mode is different from the FEP mode. The Graffiti 2 handwriting recognition engine does not use Hiragana or Katakana input modes; however, on some Japanese devices writing Graffiti 2 strokes generates Hiragana or Katakana characters, but that is dependent on the **FEP mode**, not the pinlet input mode. The same devices might have a Japanese keyboard pinlet that does use the Hiragana and Katakana input modes.

See Also `PINGetInputMode()`, `PINSetInputMode()`

Pinlet Information Constants

Purpose Values you pass to `PINGetPinletInfo()` to obtain information about a pinlet.

Declared In `PenInputMgr.h`

Constants `#define pinPinletInfoName 0`
The pinlet's name as it appears in the status bar pop-up menu is returned in the `info` parameter as a `char *`.

`#define pinPinletInfoStyle 1`
: The kind of pinlet is returned in the `info` parameter as an integer constant. See "Pinlet Styles" for the exact constants that could be returned.

`#define pinPinletInfoFEPAssoc 2`
: The creator ID of the FEP associated with this pinlet as an integer constant.

`#define pinPinletInfoIcon 3`
: System use only. Identifies the icon that displays in the status bar pop-up menu.

`#define pinPinletInfoComponentName 4`
: The name used internally for the pinlet is returned as a `char *`.

Pinlet Styles

Purpose Specifies the pinlet style.

Declared In `PenInputMgr.h`

Constants `#define pinPinletStyleHandwriting 0`
: The pinlet is a handwriting recognition pinlet.

`#define pinPinletStyleKeyboard 1`
: The pinlet is a keyboard pinlet.

`#define pinPinletStyleOther 2`
: The pinlet is some other style of pinlet.

Virtual Character Flag

Purpose Flag used to specify that a character is a virtual character.

Declared In `PenInputMgr.h`

Constants `#define pinCharFlagVirtual 0x00000001`
: Used to indicate a virtual character when a character is fed to the Pen Input Manager.

Pen Input Manager Launch Codes

sysAppLaunchCmdPinletLaunch

Purpose Sent when a pinlet has become the active pinlet. The pinlet should respond by initializing itself.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdPinletLaunch 83`

Parameters None.

sysPinletLaunchCmdLoadProcPtrs

Purpose Sent to a pinlet before the pinlet is displayed on the screen, requesting pointers to the functions used by the Pen Input Manager when interacting with this pinlet.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysPinletLaunchCmdLoadProcPtrs 85`

Parameters The launch code's parameter block pointer references an empty `PinletAPIType` structure. Pinlets should fill in the contents of this structure upon receipt of this launch code.

Pen Input Manager Notifications

sysNotifyAltInputSystemDisabled

Purpose Broadcast when an alternative input device has been disabled. For example, if the user detaches an external keyboard from the device, this notification is broadcast.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAltInputSystemDisabled 'aisd'`

See Also "Notification Manager" in *Exploring Palm OS: Programming Basics*, `PINAltInputSystemEnabled()`

sysNotifyAltInputSystemEnabled

Purpose Broadcast when an alternative input device has been enabled. For example, if the user attaches an external keyboard to the device, this notification is broadcast.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAltInputSystemEnabled 'aise'`

See Also "Notification Manager" in *Exploring Palm OS: Programming Basics*, `PINAltInputSystemEnabled()`

Pen Input Manager Functions and Macros

PINAltInputSystemEnabled Function

Purpose Indicates whether an alternative input system is currently available.

Declared In `PenInputMgr.h`

Prototype `Boolean PINAltInputSystemEnabled (void)`

Parameters None.

Returns `true` if an alternative input system is attached and the dynamic input area is available, or `false` if no alternative input system is attached or the dynamic input area is not available.

Comments Applications call this function in rare situations to determine whether an alternative input system is currently available. An alternative input system is a text input device or program that is not controlled using the Pen Input Manager. Applications may want to close the dynamic input area if the user has an alternate way of entering data.

The primary example of an alternative input system is a detachable keyboard that is sold separately from the device, like the keyboards available for many Palm handhelds. The alternative input system is not required to be a keyboard. In the future, it may be some other sort of device such as a speech recognizer. The requirement for an input system to be considered an "alternative input system" is that it must be a way for the user to enter textual data. A jog dial is not an alternative input system.

See Also `sysNotifyAltInputSystemEnabled`, `sysNotifyAltInputSystemDisabled`

PINClearPinletState Function

Purpose Tells the pinlet to clear its internal input state.

Declared In `PenInputMgr.h`

Prototype `void PINClearPinletState (void)`

Parameters None.

Returns Nothing.

Comments Applications and pinlets rarely need to make this call. Palm OS makes this call when the internal state of the pinlet should be cleared, such as when the insertion point is moved to a different text field in the application. The internal state can include temporary shift states, intermediate character results, and so on.

For example, suppose the user has entered the first stroke required to make the "x" character. Now, instead of completing the character, the user taps a new text field. This should be interpreted as the cancellation of the current input mode, so `PINClearPinletState()` is called.

PINCountPinlets Function

Purpose Returns the number of pinlets known to the Pen Input Manager.

Declared In `PenInputMgr.h`

Prototype `uint16_t PINCountPinlets (void)`

Parameters None.

Returns The number of pinlets registered with the Pen Input Manager. Returns 0 if there are no registered pinlets.

Comments In rare cases, a pinlet might be closely associated with a front-end processor (FEP). If the FEP is not active, the Pen Input Manager does not advertise the pinlet as being available in the status bar's menu.

This function, however, returns the count of all pinlets, whether they can be activated or not. Use the `PINGetPinletInfo()` to determine if the pinlet relies on an inactive FEP.

See Also `PINGetPinletInfo()`

PINGetCurrentPinletName Function

Purpose Obtains the internal name of the current pinlet.

Declared In `PenInputMgr.h`

Prototype `const char *PINGetCurrentPinletName (void)`

Parameters None.

Returns An ASCII string containing the name of the current pinlet.

Comments The current pinlet is the one in charge of displaying a user interface in the dynamic input area, receiving any pen strokes or pen taps made in that area, and translating those into textual input from the user.

This identifier is used in `PINSwitchToPinlet()` to change the input system. Do not confuse this identifier with the external pinlet name displayed in the status bar.

PINGetDefaultPinlet Function

Purpose Returns a default pinlet.

Declared In `PenInputMgr.h`

Prototype `uint16_t PINGetDefaultPinlet(uint16_t defaultCode)`

Parameters → *`defaultCode`*
One of the Default Pinlet Constants.

Returns The index of the pinlet used as the specified default. You can pass this index to `PINGetPinletInfo()` to obtain more information about the pinlet.

See Also `PINSetDefaultPinlet()`

PINGetInputAreaState Function

Purpose Returns the current state of the dynamic input area.

Declared In `PenInputMgr.h`

Prototype `uint16_t PINGetInputAreaState (void)`

Parameters None.

Returns One of the constants defined in the section "Input Area States" on page 74.

See Also `PINSetInputAreaState()`

PINGetInputMode Function

Purpose Returns the current input mode of the pinlet.

Declared In `PenInputMgr.h`

Prototype `uint16_t PINGetInputMode (void)`

Parameters None.

Returns One of the input mode constants listed in the "Pinlet Input Modes" section.

Comments Applications call this function to determine the current pinlet input mode. The pinlet input mode determines how the pinlet translates the next set of input from the user. The modes that a pinlet uses or accepts are up to the pinlet. Typically, in the default input mode, the pinlet translates user input into lowercase letters. Translation into any other type of character or symbol requires a different input mode.

The input mode should be considered a hint that the pinlet can use to coordinate with the application. The pinlet should have a visible indication of what its current input mode is.

See Also `PINSetInputMode()`, "Setting the Pinlet Input Mode" on page 14

PINGetPinletInfo Function

Purpose Returns the requested information about the pinlet.

Declared In `PenInputMgr.h`

Prototype
```
status_t PINGetPinletInfo (uint16_t index,
   uint16_t infoSelector, uint32_t *info)
```

Parameters

→ *index*
The index number of the pinlet about which you are requesting information. Valid values are from 0 to `PINCountPinlets()` – 1.

→ *infoSelector*
One of the values described in "Pinlet Information Constants" on page 77.

← *info*
Upon return, contains the information requested by the *infoSelector* parameter.

Returns `errNone` upon success or one of the following error codes:

`pinErrInvalidParam`
An invalid *infoSelector* parameter was specified.

`pinErrPinletNotFound`
An invalid *index* parameter was specified.

PINSetDefaultPinlet Function

Purpose Sets a default pinlet.

Declared In `PenInputMgr.h`

Prototype
```
status_t PINSetDefaultPinlet(uint16_t defaultCode,
   uint16_t index)
```

Parameters

→ *defaultCode*
One of the Default Pinlet Constants.

→ *index*
The index number of the pinlet that you want to make the default. Valid values are from 0 to `PINCountPinlets()` – 1.

Returns `errNone` upon success or one of the following values:

`pinErrInvalidParam`
The *`defaultCode`* parameter is invalid.

`pinErrPinletNotFound`
The index is out of range, or the internal pinlet list has not been created yet.

`sysErrNoInit`
The Pen Input Manager has not been initialized.

Comments This function allows you to set the default handwriting recognition pinlet and the default keyboard pinlet to something other than the system-supplied defaults. The default controls which pinlet is selected when the user taps a button on the system-supplied handwriting recognition and keyboard pinlets.

Before calling this function, you must obtain the index of the pinlet you want o make the default. To do so, you can iterate through the pinlet list from 0 to `PINCountPinlets()` – 1, calling `PINGetPinletInfo()` to obtain the pinlet's name or other identifying information.

See Also `PINGetDefaultPinlet()`

PINSetInputAreaState Function

Purpose Sets the state of the input area.

Declared In `PenInputMgr.h`

Prototype `status_t PINSetInputAreaState (uint16_t state)`

Parameters → *`state`*
The state to which the input area should be set. See "Input Area States" on page 74 for a list of possible values.

Returns `errNone` upon success or one of the following error codes:

`pinErrNoSoftInputArea`
There is no dynamic input area on this device.

`pinErrInvalidParam`
You have entered an invalid `state` parameter.

Comments Applications call this function to open or close the input area.

After opening or closing the input area, all on-screen transitional and update-based windows receive the `winResizedEvent`.

See Also `PINGetInputAreaState()`, "Programmatically Opening and Closing the Input Area" on page 8

PINSetInputMode Function

Purpose Sets the pinlet's input mode.

Declared In `PenInputMgr.h`

Prototype `void PINSetInputMode (uint16_t` *`inputMode`*`)`

Parameters → *`inputMode`*
The mode to which the pinlet should be set. This is one of the constants listed in the "Pinlet Input Modes" section.

Returns Nothing.

Comments Applications call this function to set the pinlet input mode.

The pinlet input mode determines how the pinlet translates the next set of input from the user. The modes that a pinlet uses or accepts are up to the pinlet. Typically, in the default input mode, the pinlet translates user input into lowercase letters. Translation into any other type of character or symbol requires a different input mode.

Not all input modes apply to all pinlets. If the application specifies a mode that the pinlet does not support, the pinlet chooses the closest equivalent that is supported, which could be normal mode.

See Also `PINGetInputMode()`, "Setting the Pinlet Input Mode" on page 14

PINShowReferenceDialog Function

Purpose Displays the reference dialog for the current pinlet.

Declared In `PenInputMgr.h`

Prototype `void PINShowReferenceDialog (void)`

Parameters None.

Returns Nothing.

Comments Applications call this function to show the reference or help dialog for the current pinlet.

PINSwitchToPinlet Function

Purpose Changes the currently active pinlet.

Declared In `PenInputMgr.h`

Prototype
```
status_t PINSwitchToPinlet
    (const char *pinletName,
    uint16_t initialInputMode)
```

Parameters

→ *pinletName*
The name of the pinlet that you want to make active. You can also specify one of the following strings instead:

`"default:hwr"`
Make the default handwriting recognition pinlet active.

`"default:keyboard"`
Make the default keyboard pinlet active.

→ *initialInputMode*
The mode to which the pinlet should initially be set. See "Pinlet Input Modes" on page 76.

Returns `errNone` upon success or one of the following error codes:

`pinErrPinletNotFound`
There is no pinlet with the specified name.

`pinErrNoSoftInputArea`
There is no dynamic input area on the device.

Comments A user can choose a new pinlet by selecting it from a menu that pops up when the user holds the pen down on the input area icon in the status bar. The active pinlet itself might have a button that switches to another pinlet. For example, if the traditional silkscreen area were a pinlet, the on-screen keyboard would be another pinlet. The "abc" button in the silkscreen area switches from the handwriting pinlet to the on-screen keyboard pinlet.

If a pinlet is associated with a FEP, the pinlet will not be activated unless the FEP itself is active.

See Also `PINGetCurrentPinletName()`, "Changing the Active Pinlet" on page 12

11

Pinlet

This chapter describes the APIs that must be implemented in a pinlet. It covers:

The header file `Pinlet.h` declares the API that this chapter describes.

For more information about implementing pinlets, see Chapter 3, "Customizing the Dynamic Input Area," on page 17.

Pinlet Structures and Types

PinletAPIType Struct

Purpose Defines the functions that the pinlet implements.

Declared In `Pinlet.h`

Prototype

```
typedef struct {
   PinletClearStateProcPtr pinletClearState;
   PinletGetInputModeProcPtr pinletGetInputMode;
   PinletSetInputModeProcPtr pinletSetInputMode;
   PinletShowReferenceDialogProcPtr
      pinletShowReferenceDialog;
} PinletAPIType
```

Fields

`pinletClearState`
Pointer to the function that clears the pinlet state. See `PinletClearStateProcPtr()`.

`pinletGetInputMode`
Pointer to the function that returns the pinlet's input mode. See `PinletGetInputModeProcPtr()`.

`pinletSetInputMode`
: Pointer to the function that sets the pinlet's input mode. See `PinletSetInputModeProcPtr()`.

`pinletShowReferenceDialog`
: Pointer to the function that displays pinlet help. See `PinletShowReferenceDialogProcPtr()`.

Comments You pass this structure back to the Pen Input Manager in the parameter block for the `sysPinletLaunchCmdLoadProcPtrs` launch code.

The Pen Input Manager only calls the functions you specify while the pinlet is running.

Pinlet Functions and Macros

PINFeedChar Function

Purpose Sends a character key to Palm OS®.

Declared In `Pinlet.h`

Prototype `void PINFeedChar (wchar32_t chr, uint32_t flags)`

Parameters

→ *chr*
: The character to be sent. This character must use the UTF8 character encoding.

→ *flags*
: 0 or `pinCharFlagVirtual` if the character is a virtual character.

Returns Nothing.

Comments The Pen Input Manager expects characters in the UTF8 encoding. If necessary, use `TxtConvertEncoding()` to convert a character from the device's native encoding to UTF8 before calling this function. Note that the character gets converted back to the device's native encoding before the application receives the `keyDownEvent`.

See Also `PINFeedString()`

PINFeedString Function

Purpose Sends a string of characters to Palm OS.

Declared In `Pinlet.h`

Prototype `void PINFeedString (const char *str)`

Parameters → *str*
A string containing the characters to be sent. The characters must use the UTF8 character encoding.

Returns Nothing.

Comments This function is a convenient way to send more than one character to the Pen Input Manager at once. A `keyDownEvent` is created for each character in the string. You cannot send virtual characters using this function.

The Pen Input Manager expects characters in the UTF8 encoding. If necessary, use `TxtConvertEncoding()` to convert characters from the device's native encoding to UTF8 before calling this function. Note that the characters get converted back to the device's native encoding before the application receives the `keyDownEvent`.

See Also `PINFeedChar()`

Pinlet-Defined Functions

PinletClearStateProcPtr Function

Purpose Clears the current internal state of the pinlet.

Declared In `Pinlet.h`

Prototype `void (*PinletClearStateProcPtr) (void)`

Parameters None.

Returns Nothing.

Comments This function should clear any such internal state and should reset the input mode back to the normal mode. The Pen Input Manager calls this function when it receives the `PINClearPinletState()` call.

As described in "Handling Multistroke Characters" on page 28, a pinlet may need to keep some internal state while it is running. `PinletClearStateProcPtr()` is called when the user has performed an action, such as switching to a new text field, that clearly indicates that they are done writing that character, so the internal state should be cleared.

See Also `HWRClearInputState()`

PinletGetInputModeProcPtr Function

Purpose Returns the current input mode of the pinlet.

Declared In `Pinlet.h`

Prototype `uint16_t (*PinletGetInputModeProcPtr) (void)`

Parameters None.

Returns One of the input mode constants listed in the section "Pinlet Input Modes" on page 76.

Comments The Pen Input Manager calls this function when it receives the `PINGetInputMode()` call. Handwriting recognition pinlets should call through to the `HWRGetInputMode()` function.

See Also `PinletSetInputModeProcPtr()`, "Considering the Input Modes" on page 27

PinletSetInputModeProcPtr Function

Purpose Sets the pinlet input mode.

Declared In `Pinlet.h`

Prototype `void (*PinletSetInputModeProcPtr) (uint16_t mode)`

Parameters → *mode*
The mode to which the pinlet should be set. This is one of the constants listed in the "Pinlet Input Modes" section.

Returns Nothing.

Comments The Pen Input Manager calls this function in response to the `PINSetInputMode()` function.

The pinlet input mode determines how the pinlet translates the next set of input from the user. The modes that a pinlet uses or accepts are up to the pinlet. Typically, in the normal or default input mode, the pinlet translates user input into lowercase letters. Translation into any other type of character or symbol requires a different input mode.

The input mode should be considered a hint that the pinlet can use to coordinate with the application. If the pinlet does respect the input mode, it should have a visible indication of what its current input mode is.

Not all input modes apply to all pinlets. If the pinlet does not support the specified input mode, it should choose the closest equivalent that is supported, which could be the default mode.

Handwriting recognition pinlets can call through to the function `HWRSetInputMode()`.

See Also `PinletGetInputModeProcPtr()`, "Considering the Input Modes" on page 27

PinletShowReferenceDialogProcPtr Function

Purpose Displays the help or reference dialog for the pinlet.

Declared In `Pinlet.h`

Prototype `void (*PinletShowReferenceDialogProcPtr) (void)`

Parameters None.

Returns Nothing.

Comments The Pen Input Manager calls this function in response to the `PINShowReferenceDialog()` function.

The pinlet should display a dialog that teaches the user how to use the pinlet. Handwriting recognition pinlets can call through to `HWRShowReferenceDialog()`.

Some pinlets may not require a help dialog. For example, keyboard pinlets rarely need explanatory text. If the pinlet has no help to display, this function should display an alert that says no help is available for this pinlet.

See Also "Help Dialog" on page 24

12

Shift Indicator

This chapter describes the API for the shift indicator. It covers:

The header file `GraffitiShift.h` declares the API that this chapter describes.

Shift Indicator Constants

Dimension Constants

Purpose Give the size requirements for the shift indicator.

Declared In `GraffitiShift.h`

Constants `#define kMaxGsiHeight 10`

The maximum height for a shift indicator.

`#define kMaxGsiWidth 9`

The maximum width for a shift indicator.

GsiShiftState Typedef

Purpose Shift states. For system use only.

Declared In `GraffitiShift.h`

Prototype `typedef Enum8 GsiShiftState`

Constants `gsiShiftNone`

The default mode. For the ISO-Latin character encoding, the normal mode translates strokes into lowercase letters.

`gsiNumLock`
The strokes will be interpreted as numeric characters.

`gsiCapsLock`
All of the characters will be uppercase until the mode is set to something else.

`gsiShiftPunctuation`
The next stroke will be interpreted as a punctuation mark or symbol, and then the mode is reset to normal.

`gsiShiftExtended`
The next stroke is a special symbol or part of the extended character set. The Graffiti® 2 handwriting recognition engine uses this mode for special symbols such as the trademark symbol.

`gsiShiftUpper`
The next stroke will be translated into an uppercase character rather than the normal lowercase characters.

`gsiShiftLower`
The next stroke will be translated into a lowercase character.

Lock Flag Constants

Purpose Specifies what lock state, if any, the shift state is in.

Declared In `GraffitiShift.h`

Constants `#define glfCapsLock 0x0001`
Turn on the caps lock.

`#define glfNumLock 0x0002`
Turn on the numeric lock.

`#define glfForceUpdate 0x8000`
Forces the shift indicator to update. If this flag is not used, the indicator only updates if you are setting it to a new state.

Temporary Shift State Constants

Purpose Define temporary shift states.

Declared In `GraffitiShift.h`

Constants

`#define gsiTempShiftNone 0`
The default mode. For the ISO-Latin character encoding, the normal mode translates strokes into lowercase letters.

`#define gsiTempShiftPunctuation 1`
The next stroke will be interpreted as a punctuation mark or symbol, and then the mode is reset to normal.

`#define gsiTempShiftExtended 2`
The next stroke is a special symbol or part of the extended character set. The Graffiti 2 handwriting recognition engine uses this mode for special symbols such as the trademark symbol.

`#define gsiTempShiftUpper 3`
The next stroke will be translated into an uppercase character rather than the normal lowercase characters.

`#define gsiTempShiftLower 4`
The next stroke will be translated into a lowercase character.

Shift Indicator Events

gsiStateChangeEvent

Purpose Sent when the shift indicator should change state.

Declared In `Event.h`

Prototype

```
struct gsiStateChange {
   uint16_t lockFlags;
   uint16_t tempShift;
} gsiStateChange
```

Parameters

`lockFlags`
One of the Lock Flag Constants.

`tempShift`
One of the Temporary Shift State Constants.

Shift Indicator Functions and Macros

GsiEnable Function

Purpose Enables or disables the shift indicator.

Declared In `GraffitiShift.h`

Prototype `void GsiEnable (Boolean enableIt)`

Parameters → *`enableIt`*
`true` to enable, `false` to disable.

Returns Nothing.

Comments Enabling the indicator makes it visible, disabling it makes the insertion point invisible.

GsiEnabled Function

Purpose Returns `true` if the shift indicator is enabled, or `false` if it's disabled.

Declared In `GraffitiShift.h`

Prototype `Boolean GsiEnabled (void)`

Parameters None.

Returns `true` if enabled, `false` if not.

GsiInitialize Function

Purpose Initializes the global variables used to manage the shift indicator.

Declared In `GraffitiShift.h`

Prototype `void GsiInitialize (void)`

Parameters None.

Returns Nothing.

GsiSetLocation Function

Purpose Sets the display-relative position of the shift indicator.

Declared In `GraffitiShift.h`

Prototype `void GsiSetLocation (int16_t x, int16_t y)`

Parameters → *x, y*
Coordinate of left side and top of the indicator.

Returns Nothing.

Comments The indicator is not redrawn by this routine.

Do not use this function in application code. It is used internally by the Form Manager. If you need to change the shift indicator's location, do so using the automatic form layout facility described in the section "Laying Out a Form or Dialog" on page 23 in the book *Exploring Palm OS: User Interface*.

GsiSetShiftState Function

Purpose Sets the shift indicator.

Declared In `GraffitiShift.h`

Prototype
```
void GsiSetShiftState (uint16_t lockFlags,
    uint16_t tempShift)
```

Parameters → *lockFlags*
One of the Lock Flag Constants.

→ *tempShift*
One of the Temporary Shift State Constants.

Returns Nothing.

Comments This function affects only the state of the UI element, not the underlying handwriting recognition engine.

Index

I

K

L

N

O

P

W