



Palm OS[®] Programmer's Companion

Volume I

CONTRIBUTORS

Written by Greg Wilson and Jean Ostrem

Engineering contributions by Jesse Donaldson, Noah Gibbs, Lee Taylor, Danny Epstein, Peter Epstein, David Fedor, Roger Flores, Steve Lemke, Bob Ebert, Ken Krugler, Bruce Thompson, Tim Wiegman, Gavin Peacock, Ryan Robertson, and Waddah Kudaimi

Copyright © 1996 - 2003, PalmSource, Inc. and its affiliates. All rights reserved. This documentation may be printed and copied solely for use in developing products for Palm OS® software. In addition, two (2) copies of this documentation may be made for archival and backup purposes. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form or by any means or used to make any derivative work (such as translation, transformation or adaptation) without express written consent from PalmSource, Inc.

PalmSource, Inc. reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of PalmSource, Inc. to provide notification of such revision or changes.

PALMSOURCE, INC. AND ITS SUPPLIERS MAKE NO REPRESENTATIONS OR WARRANTIES THAT THE DOCUMENTATION IS FREE OF ERRORS OR THAT THE DOCUMENTATION IS SUITABLE FOR YOUR USE. THE DOCUMENTATION IS PROVIDED ON AN "AS IS" BASIS. PALMSOURCE, INC. AND ITS SUPPLIERS MAKE NO WARRANTIES, TERMS OR CONDITIONS, EXPRESS OR IMPLIED, EITHER IN FACT OR BY OPERATION OF LAW, STATUTORY OR OTHERWISE, INCLUDING WARRANTIES, TERMS, OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND SATISFACTORY QUALITY. TO THE FULL EXTENT ALLOWED BY LAW, PALMSOURCE, INC. ALSO EXCLUDES FOR ITSELF AND ITS SUPPLIERS ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, OR PUNITIVE DAMAGES OF ANY KIND, OR FOR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THIS DOCUMENTATION, EVEN IF PALMSOURCE, INC. OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

PalmSource, the PalmSource logo, AnyDay, EventClub, Graffiti, HandFAX, HandMAIL, HandSTAMP, HandWEB, HotSync, the HotSync logo, iMessenger, MultiMail, MyPalm, Palm, the Palm logo, the Palm trade dress, Palm Computing, Palm OS, Palm Powered, PalmConnect, PalmGear, PalmGlove, PalmModem, PalmPak, PalmPix, PalmPoint, PalmPower, PalmPrint, Palm.Net, Simply Palm, ThinAir, and WeSync are trademarks of PalmSource, Inc. or its affiliates. All other product and brand names may be trademarks or registered trademarks of their respective owners.

IF THIS DOCUMENTATION IS PROVIDED ON A COMPACT DISC, THE OTHER SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT ACCOMPANYING THE COMPACT DISC.

Palm OS Programmer's Companion, Volume I
Document Number 3004-006-HW
April 18, 2003
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

About This Document	xi
Palm OS SDK Documentation	xi
What This Volume Contains	xi
Additional Resources	xiii
 1 Programming Palm OS in a Nutshell	 1
Why Programming for Palm OS Is Different	1
Screen Size	2
Quick Turnaround Expected	2
PC Connectivity	3
Input Methods	3
Power	4
Memory	4
File System	4
Backward Compatibility	4
Palm OS Programming Concepts	5
API Naming Conventions	6
Integrating Programs with the Palm OS Environment	7
Writing Robust Code	9
Assigning a Database Type and Creator ID	11
Making Your Application Run on Different Devices	13
Running New Applications on an Older Device	13
Backward Compatibility with PalmOSGlue	14
Compiling Older Applications with the Latest SDK	15
Programming Tools	15
Where to Go from Here	16
 2 Application Startup and Stop	 19
Launch Codes and Launching an Application	20
Responding to Launch Codes	21
Responding to Normal Launch	23
Responding to Other Launch Codes	25
Launching Applications Programmatically	27
Creating Your Own Launch Codes	28

Stopping an Application	29
Notifications	30
Registering for a Notification	31
Writing a Notification Handler	34
Sleep and Wake Notifications	35
Helper Notifications	38
When to Use the Helper API	39
Requesting a Helper Service	40
Implementing a Helper	42
Launch Code Summary	46
Notification Summary	48
Launch and Notification Function Summary	50

3 Event Loop 51

The Application Event Loop	53
Low-Level Event Management	57
The Graffiti Manager	58
The Key Manager	60
The Pen Manager	61
The System Event Manager	62
System Event Manager Summary	66

4 User Interface 69

Palm OS Resource Summary	70
Drawing on the Palm Powered Handheld	72
The Draw State	73
Drawing Functions	74
High-Density Displays	75
Forms, Windows, and Dialogs	83
Alert Dialogs	85
Progress Dialogs	86
The Keyboard Dialog	87
Offscreen Windows	91
Controls	92
Buttons	92
Pop-Up Trigger	93

Selector Trigger	94
Repeating Button.	95
Push Buttons	96
Check Boxes	97
Sliders and Feedback Sliders.	98
Fields	103
Menus	105
Checking Menu Visibility	107
Dynamic Menus	108
Menu Shortcuts	109
Tables	111
Table Event	112
Lists	112
Using Lists in Place of Tables	114
Categories	116
Initializing Categories in a Database	117
Initializing the Category Pop-up Trigger.	119
Managing a Category Pop-up List	120
Bitmaps	123
Versions of Bitmap Support	124
Bitmap Families	130
Drawing a Bitmap	135
Color Tables and Bitmaps	137
Labels.	137
Scroll Bars	137
Custom UI Objects (Gadgets).	140
Dynamic UI	142
Dynamic User Interface Functions	144
Color and Grayscale Support.	144
Indexed Versus Direct Color Display	145
Color Table	145
UI Color List.	147
Direct Color Functions	149
Pixel Reading and Writing.	149
Direct Color Bitmaps	150

Insertion Point	153
Application Launcher	153
Icons in the Launcher	153
Application Version String.	154
The Default Application Category	154
Opening the Launcher Programmatically	156
Summary of User Interface API.	157

5 Memory 169

Introduction to Palm OS Memory Use	169
Hardware Architecture	169
PC Connectivity	170
Memory Architecture	171
Heap Overview	175
The Memory Manager.	178
Memory Manager Structures.	178
Using the Memory Manager	181
Achieving Optimum Performance	184
Summary of Memory Management	186

6 Files and Databases 189

The Data Manager	189
Records and Databases	190
Structure of a Database Header	191
Using the Data Manager	194
Data Manager Tips	196
The Resource Manager	197
Structure of a Resource Database Header	198
Using the Resource Manager.	199
File Streaming Application Program Interface	201
Using the File Streaming API	201
Summary of Files and Databases	203

7 Expansion 207

Expansion Support	208
Primary vs. Secondary Storage	208

Expansion Slot	209
Universal Connector	209
Architectural Overview	210
Slot Drivers	211
File Systems	211
VFS Manager	212
Expansion Manager	213
Standard Directories	214
Applications on Cards.	216
Card Insertion and Removal	217
Start.prc.	222
Checking for Expansion Cards	223
Verifying Handheld Compatibility	223
Checking for Mounted Volumes	224
Enumerating Slots	225
Determining a Card's Capabilities	226
Volume Operations	227
Hidden Volumes	229
Matching Volumes to Slots	229
Naming Volumes.	230
File Operations.	231
Common Operations	231
Naming Files	233
Working with Palm Databases	234
Directory Operations	240
Directory Paths	240
Common Operations	240
Enumerating the Files in a Directory	241
Determining the Default Directory for a Particular File Type.	242
Default Directories Registered at Initialization	244
Custom Calls.	246
Custom I/O	247
Debugging.	247
Summary of Expansion and VFS Managers	248

8 Text	251
Text Manager and International Manager	252
Characters	253
Declaring Character Variables	253
Using Character Constants	254
Missing and Invalid Characters	255
Retrieving a Character's Attributes	256
Virtual Characters	256
Retrieving the Character Encoding	257
Strings	258
Manipulating Strings	259
Performing String Pointer Manipulation.	260
Truncating Displayed Text.	261
Comparing Strings	262
Global Find	263
Dynamically Creating String Content	265
Using the StrVPrintf Function	267
Fonts	268
Built-in Fonts	269
Selecting Which Font to Use	270
Fonts for High-Density Displays	271
Setting the Font Programmatically	273
Obtaining Font Information	274
Creating Custom Fonts	275
Summary of Text API	279
 9 Attentions and Alarms	 283
Getting the User's Attention	283
The Role of the Attention Manager	283
Attention Manager Operation	285
Getting the User's Attention	291
Attentions and Alarms	301
Detecting and Updating Pending Attentions	302
Detecting Device Capabilities	305
Controlling the Attention Indicator	305

Alarms	306
Setting an Alarm	307
Alarm Scenario	309
Setting a Procedure Alarm.	311
Summary of Attentions and Alarms.	313
10 Palm System Support	315
Features	315
The System Version Feature	316
Application-Defined Features	318
Using the Feature Manager	318
Feature Memory	319
Preferences	321
Accessing System Preferences	321
Setting System Preferences	323
Setting Application-Specific Preferences.	324
Sound.	332
Simple Sound	332
Sampled Sound	333
Simple vs Sampled	333
Sound Preferences	334
Standard MIDI Files	335
Creating a Sound Stream	337
System Boot and Reset	338
Soft Reset	338
Soft Reset + Up Arrow	338
Hard Reset	339
System Reset Calls	339
ARM-Native Functions	339
Calling an ARM Function	340
ARM Function Definition	342
Accessing 68K Data From an ARM Function	342
Embedding ARM Code in a 68K Application.	344
Calling Palm OS Functions From ARM Code.	345
Hardware Interaction	349
Palm OS Power Modes	350

Guidelines for Application Developers	351
Power Management Calls	352
The Microkernel	352
Retrieving the ROM Serial Number	353
Time	355
Using Real-Time Clock Functions.	356
Using System Ticks Functions	356
Floating-Point	356
Summary of System Features.	359
11 Localized Applications	363
Localization Guidelines	364
Using Overlays to Localize Resources	365
Dates	367
Numbers	368
Obtaining Locale Information	369
Notes on the Japanese Implementation	372
Japanese Character Encoding	372
Japanese Character Input	372
The Calculator Button.	373
Displaying Japanese Strings on UI Objects.	373
Displaying Error Messages	373
Summary of Localization	374
12 Debugging Strategies	375
Displaying Development Errors	376
Using the Error Manager Macros	377
The Try-and-Catch Mechanism	378
Using the Try and Catch Mechanism	378
Summary of Debugging API	380
13 Standard IO Applications	381
Creating a Standard IO Application	382
Creating a Standard IO Provider Application.	383
Summary of Standard IO	386
Index	387

About This Document

Palm OS Programmer's Companion is part of the Palm OS® Software Development Kit. This introduction provides an overview of SDK documentation, discusses what materials are included in this document, and what conventions are used.

Palm OS SDK Documentation

The following documents are part of the SDK:

Document	Description
<i>Palm OS Programmer's API Reference</i>	An API reference document that contains descriptions of all Palm OS function calls and important data structures.
<i>Palm OS Programmer's Companion</i>	A multi-volume guide to application programming for Palm OS. This guide contains conceptual and "how-to" information that complements the Reference.
<i>Constructor for Palm OS</i>	A guide to using Constructor to create Palm OS resource files.
<i>Palm OS Programming Development Tools Guide</i>	A guide to writing and debugging Palm OS applications with the various tools available.

What This Volume Contains

This volume is designed for random access. That is, you can read any chapter in any order. You don't necessarily have to read some before others, though the first few chapters are designed for programmers who are new to the Palm OS. The first three chapters help you learn necessary tasks and possible features for your application.

About This Document

What This Volume Contains

Note that each chapter ends with a list of hypertext links into the relevant function descriptions in the Reference book.

Here is an overview of this volume:

- [Chapter 1, “Programming Palm OS in a Nutshell.”](#) Provides new Palm OS programmers with a summary of what tasks and tools are involved in writing a Palm OS application and provides pointers to where to look for more information.
- [Chapter 2, “Application Startup and Stop.”](#) Describes how to use and respond to launch codes to start and stop an application and perform other actions. Describes how to implement the `PilotMain` function, the entry point for all applications.
- [Chapter 3, “Event Loop.”](#) Describes the Event Manager, events, the event loop, and how to implement the event loop in your application. Discusses how your application and the system interact to handle events.
- [Chapter 4, “User Interface.”](#) Describes the user interface elements that you can use in your application and how to use them. Also covers related topics such as drawing, dynamic UI, receiving user input, and the Application Launcher.
- [Chapter 5, “Memory.”](#) Describes the memory architecture, memory use on Palm Powered™ handhelds, and the Memory Manager.
- [Chapter 6, “Files and Databases.”](#) Describes the data storage system, the Data Manager, Resource Manager, and the file streaming API.
- [Chapter 7, “Expansion.”](#) Describes how to work with expansion cards and add-on devices using the Palm OS Expansion and Virtual File System (VFS) Managers.
- [Chapter 8, “Text.”](#) Describes how to manipulate characters and strings in a way that makes your application easily localizable.
- [Chapter 9, “Attentions and Alarms.”](#) Describes the Attention Manager, which applications use to bring important events to the user’s attention, and the Alarm Manager, which allows applications to receive notification at some future point in time.

- [Chapter 10, “Palm System Support.”](#) Describes features unique to the Palm hardware and OS such as the Feature Manager, preferences, the Sound Manager, system boot and reset, the microkernel, time, and floating point arithmetic.
- [Chapter 11, “Localized Applications.”](#) Discusses how to make your application localizable. Includes information on the Overlay Manager and the Locale Manager, and how to work with numbers and dates.
- [Chapter 12, “Debugging Strategies.”](#) Describes programmatic approaches to debugging your application; that is, using the Error Manager and the Palm OS try and catch mechanism for debugging.
- [Chapter 13, “Standard IO Applications.”](#) Describes how to create a command line application. On Palm OS, command line applications are typically used by developers for debugging purposes only.

Volume II of the *Palm OS Programmer’s Companion* discusses communications.

Additional Resources

- Documentation
PalmSource publishes its latest versions of this and other 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/>

Programming Palm OS in a Nutshell

This chapter is the place to start if you're new to Palm™ programming. It summarizes what's unique about writing applications for Palm Powered™ handhelds and tells you where to go for more in-depth information. It covers:

- [Why Programming for Palm OS Is Different](#)
- [Palm OS Programming Concepts](#)
- [Assigning a Database Type and Creator ID](#)
- [Making Your Application Run on Different Devices](#)
- [Programming Tools](#)
- [Where to Go from Here](#)

Read this chapter for a high-level introduction to Palm programming. The rest of this book provides the details.

Why Programming for Palm OS Is Different

Like most programmers, you have probably written a desktop application—an application that is run on a desktop computer such as a PC or a Macintosh computer. Writing applications for handhelds, specifically Palm Powered handhelds, is a bit different from writing desktop applications because the Palm Powered handheld is designed differently than a desktop computer. Also, users simply interact with the handheld differently than they do desktop computers.

This section describes how these differences affect the design of a Palm OS® application.

Screen Size

Most Palm Powered handheld screens are only 160x160 pixels, so the amount of information you can display at one time is limited.

For this reason, you must design your user interface carefully with different priorities and goals than are used for large screens. Strive for a balance between providing enough information and overcrowding the screen. See the book *Palm OS User Interface Guidelines* for more detailed guidelines on designing the user interface.

Note that screen sizes of future Palm Powered handhelds may vary. The Sony Clie already has a different screen resolution (320 X 320 pixels) than other Palm Powered handhelds although its screen is still the same size as other handhelds. The HandEra 330 has introduced the ability to rotate the display and the ability to collapse the input area. If the user collapses the input area, there is more space available to the application.

Quick Turnaround Expected

On a PC, users don't mind waiting a few seconds while an application loads because they plan to use the application for an extended amount of time.

By contrast, the average handheld user uses a handheld application 15 to 20 times per day for much briefer periods of time, usually just a few seconds. Speed is therefore a critical design objective for handhelds and is not limited to execution speed of the code. The total time needed to navigate, select, and execute commands can have a big impact on overall efficiency. (Also consider that Palm OS does not provide a wait cursor.)

To maximize performance, the user interface should minimize navigation between windows, opening of dialogs, and so on. The layout of application screens needs to be simple so that the user can pick up the product and use it effectively after a short time. It's especially helpful if the user interface of your application is consistent with other applications on the handheld so users work with familiar patterns.

The Palm OS development team has put together a set of design guidelines that were used as the basis for the applications resident

on the handheld (Memo Pad, Address Book, and so on). These guidelines are summarized in the book *Palm OS User Interface Guidelines*.

PC Connectivity

PC connectivity is an integral component of the Palm Powered handheld. The handheld comes with a cradle that connects to a desktop PC and with software for the PC that provides “one-button” backup and synchronization of all data on the handheld with the user’s PC.

Many Palm OS applications have a corresponding application on the desktop. To share data between the handheld’s application and the desktop’s application, you must write a **conduit**. A conduit is a plug-in to the HotSync® technology that runs when you press the HotSync button. A conduit synchronizes data between the application on the desktop and the application on the handheld. To write a conduit, you use the Conduit SDK, which provides its own documentation.

Input Methods

Most users of Palm Powered handhelds don’t have a keyboard or mouse. Users enter data into the handheld using a pen. They can either write characters in the input area or use the keyboard dialog provided on the handheld.

While Graffiti® or Graffiti 2 strokes and the keyboard dialog are useful ways of entering data, they are not as convenient as using the full-sized desktop computer with its keyboard and mouse. Therefore, you should not require users to enter a lot of data on the handheld itself.

Many Palm Powered handhelds support external keyboards, which are sold separately. Do not rely on your users having an external keyboard.

Power

Palm Powered handhelds run on batteries and thus do not have the same processing power as a desktop PC. The handheld is intended as a satellite viewer for corresponding desktop applications.

If your application needs to perform a computationally intensive task, you should implement that task in the desktop application instead of the handheld application.

Memory

Palm Powered handhelds have limited heap space and storage space. Different versions of the handheld have between 512K and 8MB total of dynamic memory and storage available. The handheld does not have a disk drive or PCMCIA support.

Because of the limited space and power, optimization is critical. To make your application as fast and efficient as possible, optimize for heap space first, speed second, code size third.

File System

Because of the limited storage space, and to make synchronization with the desktop computer more efficient, Palm OS does not use a traditional file system. You store data in memory chunks called **records**, which are grouped into **databases**. A database is analogous to a file. The difference is that data is broken down into multiple records instead of being stored in one contiguous chunk. To save space, you edit a database in place in memory instead of creating it in RAM and then writing it out to storage.

Backward Compatibility

Different versions of Palm Powered handhelds are available, and each runs a different version of Palm OS. Users are not expected to upgrade their versions of Palm OS as rapidly as they would an operating system on a desktop computer. Updates to the OS are designed in such a way that you can easily maintain backward compatibility with previous versions of the OS, and thus, your application is available to more users. See “[Making Your Application Run on Different Devices](#)” on page 13 for details.

Palm OS Programming Concepts

Palm OS applications are generally single-threaded, event-driven programs. Only one program runs at a time. To successfully build a Palm OS application, you have to understand how the system itself is structured and how to structure your application.

- Each application has a [PilotMain](#) function that is equivalent to `main` in C programs. To launch an application, the system calls `PilotMain` and sends it a **launch code**. The launch code may specify that the application is to become active and display its user interface (called a normal launch), or it may specify that the application should simply perform a small task and exit without displaying its user interface.

The sole purpose of the `PilotMain` function is to receive launch codes and respond to them. (See [Chapter 2](#), “[Application Startup and Stop](#).”)

- Palm OS is an event-based operating system, so Palm OS applications contain an event loop; however, this event loop is only started in response to the normal launch. Your application may perform work outside the event loop in response to other launch codes. [Chapter 3](#), “[Event Loop](#),” describes the main event loop.
- Most Palm OS applications contain a user interface made up of **forms**, which are analogous to windows in a desktop application. The user interface may contain both predefined UI elements (sometimes referred to as **UI objects**), and custom UI elements. (See [Chapter 4](#), “[User Interface](#).”)
- All applications should use the memory and data management facilities provided by the system. (See [Chapter 5](#), “[Memory](#),” and [Chapter 6](#), “[Files and Databases](#).”)
- You implement an application’s features by calling Palm OS functions. Palm OS consists of several managers, which are groups of functions that work together to implement a feature. As a rule, all functions that belong to one manager use the same prefix and work together to implement a certain aspect of functionality.

Managers are available to, for example, generate sounds, send alarms, perform network communication, and beam information through an infrared port. A good way to find out

the capabilities of the Palm OS is to scan the [Table of Contents](#) of both this book and *Palm OS Programmer's Companion*, vol. II, *Communications*.

IMPORTANT: The ANSI C libraries are not part of the Palm development platform. In many cases, you can perform the same function using a Palm OS API call as you can with a call to a ANSI C function. For example, the Palm OS provides a string manager that performs many of the string functions you'd expect to be able to perform in an ANSI C program. If you do use a standard C function, the code for the function is linked into your application and results in a bigger executable.

API Naming Conventions

The following conventions are used throughout the Palm OS API:

- Functions start with a capital letter.
- All functions belonging to a particular manager start with a two- or three-letter prefix, such as "Ctl" for control functions or "Ftr" for functions that are part of the Feature Manager.
- Events and other constants start with a lowercase letter.
- Structure elements start with a lowercase letter.
- Global variables start with a capital letter.
- Typedefs start with a capital letter and end with "Type" (for example, `DateFormatType`, found in `DateTime.h`).
- Macintosh ResEdit resource types usually start with a lowercase letter followed by three capital letters, for example `tSTR` or `tTBL`. (Customized Macintosh resources provided with your developer package are all uppercase, for example, `MENU`. Some resources, such as `Talt`, don't follow the conventions.)
- Members of an enumerated type start with a lowercase prefix followed by a name starting with a capital letter, as follows:

```
enum formObjects {  
    frmFieldObj,  
    frmControlObj,
```

```
frmListObj,  
frmTableObj,  
frmBitmapObj,  
frmLineObj,  
frmFrameObj,  
frmRectangleObj,  
frmLabelObj,  
frmTitleObj,  
frmPopupObj,  
frmGraffitiStateObj,  
frmGadgetObj};  
typedef enum formObjects FormObjectKind;
```

Integrating Programs with the Palm OS Environment

When users work with a Palm OS application, they expect to be able to switch to other applications, have access to Graffiti or Graffiti 2 power writing software and the onscreen keyboard, access information with the global find, receive alarms, and so on. Your application will integrate well with others if you follow the guidelines in this section. Integrate with the system software as follows:

- Handle `sysAppLaunchCmdNormalLaunch`
- Handle or ignore other application launch codes as appropriate. For more information, see the next chapter, [Chapter 2, “Application Startup and Stop.”](#)
- Handle system preferences properly. System preferences determine the display of
 - Date formats
 - Time formats
 - Number formats
 - First day of week (Sunday or Monday)

Be sure your application uses the system preferences for numeric formats, date, time, and start day of week. See [“Accessing System Preferences”](#) on page 321 for instructions on how to do so.

Programming Palm OS in a Nutshell

Palm OS Programming Concepts

- Allow the system to post these messages:
 - alarms
 - low-battery warnings
 - system messages during synchronization

The normal event loop used by virtually all Palm OS applications allows ample time for the system to post messages and handle necessary events. You only need to take special care if your application performs a lengthy computational task. For example, if your application has a large database with greater than 20,000 records and it must search through each of these database records, you might want to check for system events every so often during this loop.

- Be sure your application does not obscure or change the input area, input area icons, and power button.
- Don't obscure shift indicators.

In addition, follow these rules:

- Store state information in the application preferences database, not in the application record database. See "[Setting Application-Specific Preferences](#)" on page 324 for more information.
- If your application uses the serial port, be sure to close the port when you no longer need it so that the HotSync application can use it.
- Ensure that your application properly handles the global find. Generally, searches and sorts aren't case sensitive.
- If your application supports private records, be sure they are unavailable to the global find when they should be hidden.
- Integrate with the Launcher application by providing an application name, two application icons, and a version string as described in "[Application Launcher](#)" on page 153.
- Follow the guidelines detailed in the book *Palm OS User Interface Guidelines*.
- Ensure that your application properly handles system messages during and after synchronization.
- Ensure that deleted records are not displayed.

- Ensure that your application doesn't exceed the maximum number of categories: 15 categories and the obligatory category "Unfiled" for a total of 16.
- Ensure that your application uses a consistent default state when the user enters it:
 - Some applications have a fixed default; for example, the Date Book always displays the current day when launched.
 - Other applications return to the place the user exited last. In that case, remember to provide a default if that place is no longer available. Because of HotSync operations and Preferences, don't assume the application data is the same as it was when the user looked at it last.
- If your application uses sounds, be sure it uses the Warning and Confirmation sounds properly.

Writing Robust Code

To make your programs more robust and to increase their compatibility with the next generation of Palm OS products, it is strongly recommended that you follow the guidelines and practices outlined in this section.

- Check assumptions

You can write defensive code by adding frequent calls to the [ErrNonFatalDisplayIf](#) function, which enables your debug builds to check assumptions. Many bugs are caught in this way, and these "extra" calls don't weigh down your shipping application. You can keep more important checks in the release builds by using the [ErrFatalDisplayIf](#) function.

- Avoid continual polling

To conserve the battery, avoid continual polling. If your application is in a wait loop, poll at short intervals (for example, every tenth of a second) instead. The event loop of the Hardball example application included with your Palm OS SDK illustrates how to do this.

Programming Palm OS in a Nutshell

Palm OS Programming Concepts

- Avoid reading and writing to NULL (or low memory)

When calling functions that allocate memory ([MemSet](#), [MemMove](#) and similar functions) make sure that the pointers they return are non-NULL. (If you can do better validation than that, so much the better.) Also check that pointers your code obtains from structures or other function calls are not NULL. Consider adding to your debug build a `#define` that overrides [MemMove](#) (and similar functions) with a version that validates the arguments passed to it.

- Use dynamic heap space frugally

It is important not to use the extra dynamic heap space available on Palm units running 2.0 and higher unless it is truly necessary to do so. Wasteful use of heap space may limit your application to running only on the latest handhelds—which prevents it from running on the very large number of units already in the marketplace.

Note that some system services, such as the IrDA stack or the Find window, can require additional memory while your application is running; for example, if the unit starts to receive a beam or other external input, the system may need to allocate additional heap space for the incoming data. Don't use all available dynamic memory just because it's there; instead, consider using the storage heap for working with large amounts of temporary data.

- Check result codes when allocating memory

Because future handhelds may have larger or smaller amounts of available memory, it is always a good idea to check result codes carefully when allocating memory. It's also good practice to use the storage heap (and possibly file streams) to work with large objects.

- Avoid allocating zero-length objects

It's not valid to allocate a zero-byte buffer, or to resize a buffer to zero bytes. Palm OS 2.0 and previous releases allowed this practice, but later revisions of the OS do not permit zero-length objects.

- Avoid making assumptions about the screen

The location of the screen buffer, its size, and the number of pixels per bit aren't set in stone—they might well change. Don't hack around the windowing and drawing functions. If you are going to hack the hardware to circumvent the APIs, save the state and return the system to that saved state when you quit.

- Don't access globals or hardware directly

Global variables and their locations can change; to avoid mishaps, use the documented API functions and disable your application if it is run on anything but a tested version of the OS. Future handhelds might run on a different processor than the current one.

Similarly, don't hardcode references to cards. Although current Palm OS hardware provides only a single card slot, this may not always be the case. Thus, when calling functions that manipulate cards, such as Data Manager and file streaming functions, pass a variable that references the target card, rather than passing a hardcoded reference to card 0.

- Built-in applications can change

The format and size of the preferences (and data) for the built-in applications is subject to change. Write your code defensively, and consider disabling your application if it is run on an untested version of the OS.

Assigning a Database Type and Creator ID

Each Palm OS application is uniquely identified by a four-byte creator ID. Assigning this same creator ID to all of the databases related to an application associates those databases with the application. The OS takes advantage of this; for instance, the launcher's Info panel uses the creator ID to calculate the total memory used by each application.

Each database on the Palm Powered handheld has a type as well as a creator ID. The database type allows applications and the OS to distinguish among multiple databases with the same creator ID. For applications, set the database type to `sysFileTApplication`

Programming Palm OS in a Nutshell

Assigning a Database Type and Creator ID

(`'appl'`). For each database associated with an application, set the database type to any other value (as long as it isn't composed entirely of lowercase letters, since those are reserved by Palm). Certain predefined types—such as `'appl'` (application) or `'libr'` (library)—have special meaning to Palm OS. For instance, the launcher looks at the database type to determine which databases are applications.

Types and creator IDs are case-sensitive, and are composed of four ASCII characters in the range 32-126 (decimal). Types and creator IDs consisting of all lowercase letters are reserved for use by Palm Inc., so any type or creator ID that you choose must contain at least one uppercase letter, digit, or symbol¹.

To protect your application from conflicting with others, you need to register your creator ID with Palm, which maintains a database of registered IDs. To choose and register a creator ID, see this web page:

<http://www.palmos.com/dev/creatorid/>

Note that you don't need to register database types as you do creator IDs. Each creator ID in effect defines a new space of types, so there is no connection between two databases with type `'Data'` but with different creator IDs.

IMPORTANT: Applications with identical creator IDs cannot coexist on the same handheld; during installation the new application will replace the existing application that possesses the same creator ID. Further, the new application could well corrupt any databases that were associated with the preexisting application. For this reason, all applications should have their own unique creator ID.

Finally, creator IDs aren't used only to identify databases. They are also used, among other things, when getting or setting application preferences, to register for notifications, and to identify features.

1. Palm has also reserved `'pqa'`.

Making Your Application Run on Different Devices

There are many different handhelds that run Palm OS, and each may have a different version of the OS installed on it. Users are not expected to upgrade the Palm OS as frequently as they would an OS on a desktop computer. This fact makes backward compatibility more crucial for Palm OS applications.

This section describes how to make sure your application runs on as many handhelds as possible by discussing:

- [Running New Applications on an Older Device](#)
- [Backward Compatibility with PalmOSGlue](#)
- [Compiling Older Applications with the Latest SDK](#)

Running New Applications on an Older Device

Releases of the Palm OS are binary compatible with each other. If you write a brand new application today, it can run on all versions of the operating system provided the application doesn't use any new features. In other words, if you write your application using only features available in Palm OS 1.0, then your application runs on all handhelds. If you use 2.0 features, your application won't run on the earliest Palm Powered handhelds, but it will run on all others, and so on.

How can you tell which features are available in each version of the operating system? There are a couple of ways to do so:

- The *Palm OS Programmer's API Reference* has a "[Compatibility Guide](#)" appendix. This guide lists the features and functions introduced in each operating system version greater than 1.0.
- The header file `SysTraps.h` (or `CoreTraps.h` on Palm OS 3.5 and higher) lists all of the system traps available. Traps are listed in the order in which they were introduced to the system, and comments in the file clearly mark where each operating system version begins.

Programmatically, you can use the Feature Manager to determine which features are available on the system the application is running on. Note that you can't always rely on the operating system

Programming Palm OS in a Nutshell

Making Your Application Run on Different Devices

version number to guarantee that a feature exists. For example, Palm OS version 3.2 introduces wireless support, but not all Palm Powered handhelds have that capability. Thus, checking that the system version is 3.2 does not guarantee that wireless support exists. Consult the “[Compatibility Guide](#)” in the *Palm OS Programmer’s API Reference* to learn how to check for the existence of each specific feature.

Backward Compatibility with PalmOSGlue

The PalmOSGlue library can help you maintain backward compatibility with earlier releases while still allowing you to use the latest set of APIs. PalmOSGlue provides backward compatibility for some of the user interface manager calls and the managers that enable localization and internationalization.

PalmOSGlue can be used in any application that runs on Palm OS 2.0 and later. The library provides the latest support for localization features and for accessing internal UI data structures. Link your application with the library PalmOSGlue (`PalmOSGlue.lib` or `libPalmOSGlue.a`).

When you use PalmOSGlue, you use the functions in the same way as described they are described in the *Palm OS Programmer’s API Reference*, but their names are different. For example, `TxtFindString` is named `TxtGlueFindString` in PalmOSGlue. When you make a call to a glue function (for example, `TxtGlueFunc`, `FntGlueFunc`, or `WinGlueFunc`), the code in PalmOSGlue either uses the appropriate function found in the ROM or, if the function don’t exist, executes a simple equivalent of the function.

To see a complete list of functions in PalmOSGlue, see the chapter “[PalmOSGlue Library](#)” on page 1845 of the *Palm OS Programmer’s API Reference*.

PalmOSGlue is a linkable library that is bound to your project at link time. It is not a shared library. PalmOSGlue will increase your application’s code size. The exact amount by which your code size increases depends on the number of library functions you call; the linker strips any unused routines and data.

Compiling Older Applications with the Latest SDK

As a rule, all Palm OS applications developed with an earlier version of the Palm OS platform SDK should run error-free on the latest release.

If you want to compile your older application under the latest release, you need to look out for functions with a changed API. For any of these functions, the old function still exists with an extension noting the release that supports it, such as V10 or V20.

You can choose one of two options:

- Change the function name to keep using the old API. Your application will then run error free on the newer handhelds.
- Update your application to use the new API. The application will then run error free and have access to some new functionality; however, it will no longer run on older handhelds that use prior releases of the OS.

NOTE: If you want to compile a legacy application with the Palm OS 3.5 or later SDK, note that some header file names have changed, and the names used for basic types have changed. For example, parameters previously declared as `Word` are now `UInt16` or `Int16`. To compile existing applications, you'll need to make these changes in your code or include the header file `PalmOSCompatibility.h`. See the "[Compatibility Guide](#)" in the *Palm OS Programmer's API Reference* for further details.

Programming Tools

Several tools are available that help you build, test, and debug Palm OS applications. The most widely used tool is the CodeWarrior Interactive Development Environment (IDE). Documentation for the CodeWarrior IDE is provided with CodeWarrior. (See <http://www.palmos.com/dev/tools/> for information about other development tools.)

Programming Palm OS in a Nutshell

Where to Go from Here

As with most applications, the user interface is generally stored in one or more resource files. You use the Constructor for Palm OS to create these resources. To learn how, refer to the Constructor documentation.

To debug and test your application, there are several tools available:

- The CodeWarrior Debugger handles source-level debugging. You can use it with an application running on the Palm Powered handheld, or you can use it in conjunction with one of the other debugging tools below.
- The Palm OS Emulator tests your application on the desktop computer before downloading it onto the handheld.
- On the Macintosh, you can build a Simulator version of your application to test it. This is a standalone Mac OS application that runs your Palm OS application on a Macintosh computer.
- The Palm Debugger is an assembly-level tool. You can also use it to enter commands directly to the Palm Powered handheld.

The book *Palm OS Programming Development Tools Guide* describes the Palm-provided debugging tools available on your development platform. For CodeWarrior Debugger documentation, refer to the CodeWarrior CD.

Where to Go from Here

This chapter provided you only with a general outline of the issues involved in writing a Palm OS application. To learn the specifics, refer to the following resources:

- This book

The rest of this book provides details on how to implement common application features using the Palm OS SDK. If you're new to Palm OS programming, you need to read the next two chapters to learn the principles of Palm OS application design, how to implement the main function, and how to implement the standard event loop. The remaining chapters you can read on an as-needed basis.

- Example applications

The actual source code for the applications on the Palm Powered handheld is included with your SDK as examples. The code can be a valuable aid when you develop your own program. The software development kit provides a royalty-free license that permits you to use any or all of the source code from the examples in your application.

- *Palm OS Programming Development Tools Guide*

The *Palm OS Programming Development Tools Guide* provides more details on using the tools to debug programs. (You might also be interested in the “[Debugging Strategies](#)” chapter in this book, which describes programmatic debugging solutions.)

- *Palm OS Programmer’s API Reference*

The reference book provides the details on all of the public data structures and API calls.

- *Palm OS User Interface Guidelines*

The *Palm OS User Interface Guidelines* provides detailed guidelines for creating a user interface that conforms to Palm standards. You should read this book before you begin designing your application’s interface.

- Conduit Development Kits and documentation

If you need to write a conduit for your application, see the documentation provided with the Conduit Development Kits.

Application Startup and Stop

On desktop computers, an application starts up when a user launches it and stops when the user chooses the Exit or Quit command. These things occur a little bit differently on the Palm OS® handheld. A Palm OS application does launch when the user requests it, but it may also launch in response to some other user action, such as a request for the global find facility. Palm OS applications don't have an Exit command; instead they exit when a user requests another application.

This chapter describes how an application launches, how an application stops, and the code you must write to perform these tasks properly. It also covers notifications, which is another way for the system to launch your code when certain events occur. Notifications are available in later releases of the Palm OS. This chapter covers:

- [Launch Codes and Launching an Application](#)
- [Responding to Launch Codes](#)
- [Launching Applications Programmatically](#)
- [Creating Your Own Launch Codes](#)
- [Stopping an Application](#)
- [Notifications](#)
- [Helper Notifications](#)
- [Launch Code Summary](#)
- [Notification Summary](#)
- [Launch and Notification Function Summary](#)

This chapter does not cover the main application event loop. The event loop is covered in [Chapter 3](#), “[Event Loop](#).”

Launch Codes and Launching an Application

An application launches when it receives a **launch code**. Launch codes are a means of communication between the Palm OS and the application (or between two applications).

For example, an application typically launches when a user presses one of the buttons on the device or selects an application icon from the application launcher screen. When this happens, the system generates the launch code `sysAppLaunchCmdNormalLaunch`, which tells the application to perform a full launch and display its user interface.

Other launch codes specify that the application should perform some action but not necessarily become the current application (the application the user sees). A good example of this is the launch code used by the global find facility. The global find facility allows users to search all databases for a certain record, such as a name. In this case, it would be very wasteful to do a full launch—including the user interface—of each application only to access the application's databases in search of that item. Using a launch code avoids this overhead.

Each launch code may be accompanied by two types of information:

- A parameter block, a pointer to a structure that contains several parameters. These parameters contain information necessary to handle the associated launch code.
- Launch flags indicate how the application should behave. For example, a flag could be used to specify whether the application should display UI or not. (See “[Launch Flags](#)” in the *Palm OS Programmer's API Reference*.)

A complete list of all launch codes is provided at the end of this chapter in the section “[Launch Code Summary](#).” That section contains links into where each launch code is described in the *Palm OS Programmer's API Reference*.

Responding to Launch Codes

Your application should respond to launch codes in a function named [PilotMain](#). PilotMain is the entry point for all applications.

When an application receives a launch code, it must first check whether it can handle this particular code. For example, only applications that have text data should respond to a launch code requesting a string search. If an application can't handle a launch code, it exits without failure. Otherwise, it performs the action immediately and returns.

[Listing 2.1](#) shows parts of PilotMain from the Datebook application as an example. To see the complete example, go to the examples folder in the Palm OS SDK and look at the file Datebook.c.

Listing 2.1 PilotMain in Datebook.c

```
UInt32 PilotMain (UInt16 cmd, void *cmdPBP,
UInt16 launchFlags)
{
    return DBPilotMain(cmd, cmdPBP, launchFlags);
}

static UInt32 DBPilotMain (UInt16 cmd, void *cmdPBP,
UInt16 launchFlags)
{
    UInt16 error;
    Boolean launched;

    // This app makes use of PalmOS 2.0 features. It will crash
    // if run on an earlier version of PalmOS. Detect and warn
    // if this happens, then exit.
    error = RomVersionCompatible (version20, launchFlags);
    if (error)
        return error;

    // Launch code sent by the launcher or the datebook
    // button.
    if (cmd == sysAppLaunchCmdNormalLaunch) {
        error = StartApplication ();
        if (error) return (error);

        FrmGotoForm (DayView);
        EventLoop ();
    }
}
```

Application Startup and Stop

Responding to Launch Codes

```
        StopApplication ();
    }

    // Launch code sent by text search.
    else if (cmd == sysAppLaunchCmdFind) {
        Search ((FindParamsPtr) cmdPBP);
    }

    // This launch code might be sent to the app when it's
    // already running if the user hits the "Go To" button in
    // the Find Results dialog box.
    else if (cmd == sysAppLaunchCmdGoTo) {
        launched = launchFlags & sysAppLaunchFlagNewGlobals;
        if (launched) {
            error = StartApplication ();
            if (error) return (error);

            GoToItem ((GoToParamsPtr) cmdPBP, launched);

            EventLoop ();
            StopApplication ();
            else
                GoToItem ((GoToParamsPtr) cmdPBP, launched);
        }

        // Launch code sent by sync application to notify the
        // datebook application that its database has been synced.
        // ...
        // Launch code sent by Alarm Manager to notify the
        // datebook application that an alarm has triggered.
        // ...
        // Launch code sent by Alarm Manager to notify the
        // datebook application that it should display its alarm
        // dialog.
        // ...
        // Launch code sent when the system time is changed.
        // ...
        // Launch code sent after the system is reset. We use this
        // time to create our default database if this is a hard
        // reset
        // ...
        // Launch code sent by the DesktopLink server when it
        // creates a new database. We will initialize the new
        // database.
        return (0);
    }
}
```

Responding to Normal Launch

When an application receives the launch code `sysAppLaunchCmdNormalLaunch`, it begins with a startup routine, then goes into an event loop, and finally exits with a stop routine. (The event loop is described in [Chapter 3](#), “[Event Loop](#).” The stop routine is shown in the section “[Stopping an Application](#)” at the end of this chapter.)

During the startup routine, your application should perform these actions:

1. Get system-wide preferences (for example for numeric or date and time formats) and use them to initialize global variables that will be referenced throughout the application.
2. Find the application database by creator type. If none exists, create it and initialize it.
3. Get application-specific preferences and initialize related global variables.
4. Initialize any other global variables.

As you saw in [Listing 2.1](#), the Datebook application example responds to `sysAppLaunchCmdNormalLaunch` by calling a function named `StartApplication`. [Listing 2.2](#) shows the `StartApplication` function.

Listing 2.2 StartApplication from Datebook.c

```
static UInt16 StartApplication (void)
{
    UInt16 error = 0;
    Err err = 0;
    UInt16 mode;
    DateTimeType dateTime;
    DatebookPreferenceType prefs;
    SystemPreferencesType sysPrefs;
    UInt16 prefsSize;

    // Step 1: Get system-wide preferences.
    PrefGetPreferences (&sysPrefs);
    // Determine if secret records should be
    // displayed.
    HideSecretRecords = sysPrefs.hideSecretRecords;

    if (HideSecretRecords)
```

Application Startup and Stop

Responding to Launch Codes

```
        mode = dmModeReadWrite;
    else
        mode = dmModeReadWrite | dmModeShowSecret;

    // Get the time formats from the system preferences.
    TimeFormat = sysPrefs.timeFormat;

    // Get the date formats from the system preferences.
    LongDateFormat = sysPrefs.longDateFormat;
    ShortDateFormat = sysPrefs.dateFormat;

    // Get the starting day of the week from the system
    // preferences.
    StartDayOfWeek = sysPrefs.weekStartDay;

    // Get today's date.
    TimSecondsToDateTime (TimGetSeconds(), &dateTime);
    Date.year = dateTime.year - firstYear;
    Date.month = dateTime.month;
    Date.day = dateTime.day;

    // Step 2. Find the application's data file.  If it
    // doesn't exist, create it.
    ApptDB = DmOpenDatabaseByTypeCreator(datebookDBType,
        sysFileCDatebook, mode);
    if (! ApptDB) {
        error = DmCreateDatabase (0, datebookDBName,
            sysFileCDatebook, datebookDBType, false);
        if (error) return error;

        ApptDB =
            DmOpenDatabaseByTypeCreator(datebookDBType,
                sysFileCDatebook, mode);
        if (! ApptDB) return (1);

        error = ApptAppInfoInit (ApptDB);
        if (error) return error;
    }

    // Step 3. Get application-specific preferences.
    // Read the preferences/saved-state information. There is
    // only one version of the DateBook preferences so don't
    // worry about multiple versions.
    prefsSize = sizeof (DatebookPreferenceType);
    if (PrefGetAppPreferences (sysFileCDatebook,
```

```
        datebookPrefID, &prefs, &prefsSize, true)
        != noPreferenceFound) {
    DayStartHour = prefs.dayStartHour;
    DayEndHour = prefs.dayEndHour;
    AlarmPreset = prefs.alarmPreset;
    NoteFont = prefs.noteFont;
    SaveBackup = prefs.saveBackup;
    ShowTimeBars = prefs.showTimeBars;
    CompressDayView = prefs.compressDayView;
    ShowTimedAppts = prefs.showTimedAppts;
    ShowUntimedAppts = prefs.showUntimedAppts;
    ShowDailyRepeatingAppts =
        prefs.showDailyRepeatingAppts;
}

// Step 4. Initialize any other global variables.
TopVisibleAppt = 0;
CurrentRecord = noRecordSelected;

// Load the far call jump table.
FarCalls.apptGetAppointments = ApptGetAppointments;
FarCalls.apptGetRecord = ApptGetRecord;
FarCalls.apptFindFirst = ApptFindFirst;
FarCalls.apptNextRepeat = ApptNextRepeat;
FarCalls.apptNewRecord = ApptNewRecord;
FarCalls.moveEvent = MoveEvent;

return (error);
}
```

Responding to Other Launch Codes

If an application receives a launch code other than `sysAppLaunchCmdNormalLaunch`, it decides if it should respond to that launch code. If it responds to the launch code, it does so by implementing a launch code handler, which is invoked from its [PilotMain](#) function.

In most cases, when you respond to other launch codes, you are not able to access global variables. Global variables are generally only allocated after an application receives `sysAppLaunchCmdNormalLaunch` (see [Listing 2.2](#)) or [sysAppLaunchCmdGoto](#); so if the application hasn't received either of these launch codes, its global variables are usually not

Application Startup and Stop

Responding to Launch Codes

allocated and not accessible. In addition, if the application has multiple code segments, you cannot access code outside of segment 0 (the first segment) if the application has no access to global variables.

There is one other case where an application may have access to its global variables (and to code segments other than 0). This is when an application is launched with the code [sysAppLaunchCmdURLParams](#). If this launch code results from a `palm` URL, then globals are available. If the launch code results from a `palmcall` URL, then globals are not available. The URL is passed to your application in the launch parameter block.

NOTE: Static local variables are stored with the global variables on the system's dynamic heap. They are not accessible if global variables are not accessible.

Checking launch codes is generally a good way to determine if your application has access to global variables. However, it actually depends on the setting of the launch flags that are sent with the launch code. In particular, if the `sysAppLaunchFlagNewGlobals` flag is set, then your application's global variables have been allocated on this launch. This flag is set by the system and isn't (and shouldn't be) set by the sender of a launch code.

```
Boolean appHasGlobals = launchFlags & sysAppLaunchFlagNewGlobals;
```

There's one case where this flag won't be set and your application will still have access to global variables. This is when your application is already running as the current application. In this case, its global variables have already been allocated through a previous launch.

If your application receives a launch code other than `sysAppLaunchCmdNormalLaunch` or `sysAppLaunchCmdGoTo`, you can find out if it is the current application by checking the launch flags that are sent with the launch code. If the application is the currently running application, the

`sysAppLaunchFlagSubCall` flag is set. This flag is set by the system and isn't (and shouldn't be) set by the sender of a launch code.

```
Boolean appIsActive = launchFlags & sysAppLaunchFlagSubCall;
```

Launching Applications Programmatically

Applications can send launch codes to each other, so your application might be launched from another application or it might be launched from the system. An application can use a launch code to request that another application perform an action or modify its data. For example, a data collection application could instruct an email application to queue up a particular message to be sent.

TIP: In Palm OS 4.0 and higher, there are other ways for applications to communicate. See the section “[When to Use the Helper API](#)” to help you decide which method to use.

Sending a launch code to another application is like calling a specific subroutine in that application: the application responding to the launch code is responsible for determining what to do given the launch code constant passed on the stack as a parameter.

To send a launch code to another application, use the system manager function [SysAppLaunch](#). Use this routine when you want to make use of another application's functionality and eventually return control of the system to your application. Usually, applications use it only for sending launch codes to other user-interface applications.

For example, you would use this function to request that the built in Address Book application search its databases for a specified phone number and return the results of the search to your application. When calling [SysAppLaunch](#) do not set launch flags yourself—the [SysAppLaunch](#) function sets launch flags appropriately for you.

An alternative, simpler method of sending launch codes is the [SysBroadcastActionCode](#) call. This routine automatically finds

Application Startup and Stop

Creating Your Own Launch Codes

all other user-interface applications and calls [SysAppLaunch](#) to send the launch code to each of them.

When an application is called using `SysAppLaunch`, the system considers that application to be the current application even though the application has not switched from the user's perspective. Thus, if your application is called from another application, it can still use the function [SysCurAppDatabase](#) to get the card number and database ID of its own database.

If you want to actually close your application and open another application, use [SysUIAppSwitch](#) instead of `SysAppLaunch`. This routine notifies the system which application to launch next and feeds an application-quit event into the event queue. If and when the current application responds to the quit event and returns, the system launches the new application.

When you allocate a parameter block to pass to `SysUIAppSwitch`, you must call [MemPtrSetOwner](#) to grant ownership of the parameter block chunk to the OS (your application is originally set as the owner). If the parameter block structure contains references by pointer or handle to any other chunks, you also must set the owner of those chunks by calling `MemPtrSetOwner` or [MemHandleSetOwner](#). If you don't change the ownership of the parameter block, it will get freed before the application you're launching has a chance to use it.

In Palm OS 3.0 and higher, you can also use the Application Launcher to launch any application. For more information, see the section "[Application Launcher](#)" in the "[User Interface](#)" chapter.

WARNING! Do not use the [SysUIAppSwitch](#) or [SysAppLaunch](#) functions to open the Application Launcher application.

Creating Your Own Launch Codes

The Palm OS contains predefined launch codes, which are listed in [Table 2.1](#) at the end of this chapter. In addition, developers can create their own launch codes to implement specific functionality.

Both the sending and the receiving application must know about and handle any developer-defined launch codes.

The launch code parameter is a 16-bit word value. All launch codes with values 0–32767 are reserved for use by the system and for future enhancements. Launch codes 32768–65535 are available for private use by applications.

Stopping an Application

An application shuts itself down when it receives the event [appStopEvent](#). Note that this is an event, not a launch code. The application must detect this event and terminate. (You'll learn more about events in the next chapter.)

When an application stops, it is given an opportunity to perform cleanup activities including closing databases and saving state information.

In the stop routine, an application should first flush all active records, then close the application's database, and finally save those aspects of the current state needed for startup. [Listing 2.3](#) is an example of a `StopApplication` routine from `Datebook.c`.

Listing 2.3 StopApplication from Datebook.c

```
static void StopApplication (void)
{
    DatebookPreferenceType prefs;

    // Write the preferences / saved-state information.
    prefs.noteFont = NoteFont;
    prefs.dayStartHour = DayStartHour;
    prefs.dayEndHour = DayEndHour;
    prefs.alarmPreset = AlarmPreset;
    prefs.saveBackup = SaveBackup;
    prefs.showTimeBars = ShowTimeBars;
    prefs.compressDayView = CompressDayView;
    prefs.showTimedAppts = ShowTimedAppts;
    prefs.showUntimedAppts = ShowUntimedAppts;
    prefs.showDailyRepeatingAppts = ShowDailyRepeatingAppts;

    // Write the state information.
    PrefSetAppPreferences (sysFileCDatebook, datebookPrefID,
```

Application Startup and Stop

Notifications

```
        datebookVersionNum, &prefs, sizeof
        (DatebookPreferenceType), true);

    // Send a frmSave event to all the open forms.
    FrmSaveAllForms ();

    // Close all the open forms.
    FrmCloseAllForms ();

    // Close the application's data file.
    DmCloseDatabase (ApptDB);
}
```

Notifications

On systems where the [Notification Feature Set](#) is present, your application can receive **notifications** and launch when certain system-level events or application-level events occur. Notifications are similar to application launch codes, but differ from them in two important ways:

- Notifications can be sent to any code resource, such as a shared library or a system extension (for example, a hack installed with the HackMaster program). Launch codes can only be sent to applications. Any code resource that is registered to receive a notification is called a **notification client**.
- Notifications are only sent to applications or code resources that have specifically registered to receive them, making them more efficient than launch codes. Many launch codes are sent to all installed applications to give each application a chance to respond.

The Palm OS system and the built-in applications send notifications when certain events occur. See the “[Notification Summary](#)” in this chapter for a complete list.

It’s also possible for your application to create and broadcast its own notifications. However, doing so is rare. It’s more likely that you’ll want to register to receive the predefined notifications or that you’ll broadcast the predefined [sysNotifyHelperEvent](#) described in the “[Helper Notifications](#)” section.

Three general types of event flow are possible using the notification manager:

- Single consumer

Each client is notified that the event has occurred and handles it in its own way without modifying any information in the parameter block.

- Collaborative

The notification's parameter block contains a handled flag. Clients can set this flag to communicate to other clients that the event has been handled, while still allowing them to receive the notification. An example of this is the `sysNotifyAntennaRaisedEvent` for Palm VII™ series handhelds. A client might decide to handle the antenna key down event and in this case, sets `handled` to `true` to inform other clients that the event has been handled.

- Collective

Each client can add information to the notification's parameter block, allowing the data to be accumulated for all clients. This style of notification could be used, for example, to build a menu dynamically by letting each client add its own menu text. The `sysNotifyMenuCmdBarOpenEvent` is similar to this style of notification.

Registering for a Notification

To receive notification that an event has occurred, you must register for it using the [SysNotifyRegister](#) function. Once you register for a notification, you remain registered until the system is reset or until you explicitly unregister for this notification using [SysNotifyUnregister](#).

To register an application for the HotSync® notification, you'd use a function call similar to the one in [Listing 2.4](#).

Listing 2.4 Registering an application for a notification

```
SysNotifyRegister(myCardNo, appDBID, sysNotifySyncStartEvent,  
                NULL, sysNotifyNormalPriority, myDataP);
```

Application Startup and Stop

Notifications

If you are writing a shared library instead of an application and you want to be notified about the HotSync event, your call to `SysNotifyRegister` looks slightly different. See [Listing 2.5](#).

Listing 2.5 Registering a shared library for a notification

```
SysNotifyRegister(myCardNo, shlibDBID,  
    sysNotifySyncStartEvent, SyncNotifyHandler,  
    sysNotifyNormalPriority, myDataP);
```

The parameters you pass to the `SysNotifyRegister` function specify the following:

- The first two parameters are the card number and database ID for the `prc` file. Be sure you're not passing the local ID of the record database that your application accesses. You use the record database's local ID more frequently than you do the application's local ID, so this is a common mistake to make.
- `sysNotifySyncStartEvent` specifies that you want to be informed when a HotSync operation is about to start. There is also a `sysNotifySyncFinishEvent` that specifies that a HotSync operation has ended.
- The next parameter specifies how the notification should be received. This is where [Listing 2.4](#) and [Listing 2.5](#) differ.

Applications use `NULL` for this parameter to specify that they should be notified through the application launch code [sysAppLaunchCmdNotify](#). As with all other launch codes, the system passes this to the application's [PilotMain](#) function.

The shared library has no `PilotMain` function and therefore no way to receive a launch code, so it passes a pointer to a callback function. Only use a callback function if your code doesn't have a `PilotMain`.

Note that it's always necessary to pass the card number and database ID of your `prc` file even if you specify a callback function.

- `sysNotifyNormalPriority` means that you don't want your code to receive any special consideration when receiving the notification. Notifications are broadcast synchronously in priority order. The lower the number you specify here, the earlier you receive the notification in the list.

In virtually all cases, you should use `sysNotifyNormalPriority`. If you absolutely must ensure that your code is notified in a certain order (either before most notifications or after most notifications), be sure to leave some space between priority values so that your code won't collide with the system's handling of notifications or with another application's handling of notifications. Never use the extreme maximum or minimum allowed value. In general, Palm™ recommends using a value whose least significant bits are 0 (such as 32, 64, 96, and so on).

- `myDataP` is a pointer to any data you need to access in your notification handler function. As with most launch codes, `sysAppLaunchCmdNotify` does not provide access to global variables, so you should use this pointer to pass yourself any needed data.

After you've made the calls shown in [Listing 2.4](#) and [Listing 2.5](#) and the system is about to begin a HotSync operation, it broadcasts the `sysNotifySyncStartEvent` notification to both clients.

The application is notified through the `sysAppLaunchCmdNotify` launch code. This launch code's parameter block is a [SysNotifyParamType](#) structure containing the notification name, the broadcaster, and a pointer to your specific data (`myDataP` in the example above). Some notifications contain extra information in a `notifyDetailsP` field in this structure. The HotSync notifications do not use the `notifyDetailsP` field.

The shared library is notified by a call to its `SyncNotifyHandler` function. This function is passed the same `SysNotifyParamType` structure that is passed through the launch code mechanism.

IMPORTANT: Because the callback pointer is used to directly call the function, the pointer must remain valid from the time `SysNotifyRegister` is called to the time the notification is broadcast. If the function is in a shared library, you must keep the library open. If the function is in a separately loaded code resource, the resource must remain locked while registered for the notification. When you close a library or unlock a resource, you must first unregister for any notifications. If you don't, the system will crash when the notification is broadcast.

Writing a Notification Handler

The application's response to `sysAppLaunchCmdNotify` and the shared library's callback function are called **notification handlers**. A notification handler may perform any processing necessary, including displaying a user interface or broadcasting other notifications.

When displaying a user interface, consider the possibility that you may be blocking other applications from receiving the notification. For this reason, it's generally not a good idea to display a modal form or do anything else that requires waiting for the user to respond. Also, many of the notifications are broadcast during [SysHandleEvent](#), which means your application event loop may not have progressed to the point where it is possible for you to display a user interface, or that you may overflow the stack.

If you need to perform some lengthy process in a notification handler, one way to ensure that you aren't blocking other events is to send yourself a deferred notification. For example, [Listing 2.6](#) shows a notification handler for the `sysNotifyTimeChangeEvent` notification that performs no work other than setting up a deferred notification (`myDeferredNotifyEvent`) and scheduling it for broadcast. When the application receives the `myDeferredNotifyEvent`, it calls the `MyNotifyHandler` function, which is where the application really handles the time change event.

Listing 2.6 Deferring notification within a handler

```
case sysAppLaunchCmdNotify :
    if (cmdPBP->notify->notifyType == sysNotifyTimeChangeEvent) {
        SysNotifyParamType notifyParam;
        MyGlobalsToAccess myData;

        /* initialize myData here */

        /* Create the notification block. */
        notifyParam.notifyType = myDeferredNotifyEvent;
        notifyParam.broadcaster = myCreatorID;
        notifyParam.notifyDetailsP= NULL;
        notifyParam.handled = false;

        /* Register for my notification */
        SysNotifyRegister(myCardNo, appDBID, myDeferredNotifyEvent, NULL,
            sysNotifyNormalPriority, &myData);

        /* Broadcast the notification */
        SysNotifyBroadcastDeferred(&notifyParam, NULL);

    } else if (cmdPBP->notify->notifyType == myDeferredNotifyEvent)
        MyNotifyHandler(cmdPBP->notify);
break;
```

The [SysNotifyBroadcastDeferred](#) function broadcasts the specified notification to all interested parties; however, it waits to do so until the current event has completed processing. Thus, by using a separate deferred notification, you can be sure that all other clients have had a chance to respond to the first notification.

There are several functions that broadcast notifications. Notification handlers should use `SysNotifyBroadcastDeferred` to avoid the possibility of overflowing the notification stack.

A special case of dealing with lengthy computations in a notification handler occurs when the system is being put to sleep. See “[Sleep and Wake Notifications](#)” below.

Sleep and Wake Notifications

Several notifications are broadcast at various stages when the system goes to sleep and when the system wakes up. These are:

- [sysNotifySleepRequestEvent](#)

Application Startup and Stop

Notifications

- [sysNotifySleepNotifyEvent](#)
- [sysNotifyEarlyWakeupEvent](#)
- [sysNotifyLateWakeupEvent](#)

These notifications are **not** guaranteed to be broadcast. For example, if the system goes to sleep because the user removes the batteries, sleep notifications are not sent. Thus, these notifications are unsuitable for applications where external hardware must be shut off to conserve power before the system goes to sleep.

If you want to know when the system is going to sleep because you have a small amount of cleanup that should occur beforehand, then register for `sysNotifySleepNotifyEvent`.

It is recommended that you not perform any sort of prolonged activity, such as displaying an alert panel that requests confirmation, in response to a sleep notification. If you do, the alert might be displayed long enough to trigger another auto-off event, which could be detrimental to other handlers of the sleep notify event.

In a few instances, you might need to prevent the system from going to sleep. For example, your code might be in the middle of performing some lengthy computation or in the middle of attempting a network connection. If so, register for the `sysNotifySleepRequestEvent` instead. This notification informs all clients that the system might go to sleep. If necessary, your handler can delay the sleep request by doing the following:

```
((SleepEventParamType *)
(notify->notifyDetailsP))->deferSleep++;
```

The system checks the `deferSleep` value when each notification handler returns. If it is nonzero, it cancels the sleep event.

After you defer sleep, your code is free to finish what it was doing. When it is finished, you must allow the system to continue with the sleep event. To do so, create a [keyDownEvent](#) with the `resumeSleepChr` and the command key bit set (to signal that the character is virtual) and add it to the event queue. When the system receives this event, it will again broadcast the `sysNotifySleepRequestEvent` to all clients. If `deferSleep` is 0 after all clients return, then the system knows it is safe to go to

sleep, and it broadcasts the `sysNotifySleepNotifyEvent` to all of its clients.

Notice that you may potentially receive the `sysNotifySleepRequestEvent` many times before the system actually goes to sleep, but you receive the `sysNotifySleepNotifyEvent` exactly once.

During a wake-up event, the other two notifications listed above are broadcast. The `sysNotifyEarlyWakeupEvent` is broadcast very early on in the wakeup process, generally before the screen has turned on. At this stage, it is not guaranteed that the system will fully wake up. It may simply handle an alarm or a battery charger event and go back to sleep. Most applications that need notification of a wakeup event will probably want to register for `sysNotifyLateWakeupEvent` instead. At this stage, the screen has been turned on and the system is guaranteed to fully wake up.

When the handheld receives the `sysNotifyLateWakeupEvent` notification, it may be locked and waiting for the user to enter the password. If this is the case, you must wait for the user to unlock the handheld before you display a user interface. Therefore, if you intend to display a user interface when the handheld wakes up, you should make sure the handheld is not locked. If the handheld is locked, you should register for [sysNotifyDeviceUnlocked](#) notification and display your user interface when it is received. See [Listing 2.7](#).

Listing 2.7 Responding to Late Wakeup Notification

```
case sysNotifyLateWakeupEvent:
    if ((Boolean)
        PrefGetPreference(prefDeviceLocked)) {
        SysNotifyRegister(myCardNo, myDbID,
            sysNotifyDeviceUnlocked, NULL,
            sysNotifyNormalPriority, NULL);
    } else {
        HandleDeviceWakeup();
    }
case sysNotifyDeviceUnlocked:
    HandleDeviceWakeup();
```

Helper Notifications

If the [4.0 New Feature Set](#) is present, the helper notification, [sysNotifyHelperEvent](#), is defined. This notification is a way for one application to request a service from another application. On Palm OS 4.0, the Dial application is the only application that performs a service through `sysNotifyHelperEvent`. Specifically, the Dial application dials a phone in response to this notification. The Address Book uses the Dial application to dial the phone number that the user has selected. You can use the Dial application in a similar way by broadcasting the `sysNotifyHelperEvent` from your application. You may also choose to write a provider of services.

In this section, the application that responds to the `sysNotifyHelperEvent` notification is called the **helper**, and the application that broadcasts the notification is called the **broadcaster**.

A helper registers for the [sysNotifyHelperEvent](#) notification. In the notification handler, the helper responds to action requests pertaining to the **service** that it provides.

Actions are requests to provide information about the service or to perform the service. The details structure for [sysNotifyHelperEvent](#) (a [HelperNotifyEventType](#) structure) defines three possible actions:

- `kHelperNotifyActionCodeEnumerate` is a request for the helper to list the services that it can perform.
- `kHelperNotifyActionCodeValidate` is a request for the helper to make sure that it can perform the service.
- `kHelperNotifyActionCodeExecute` is a request to actually perform the service.

The possible **services** are defined in `HelperServiceClass.h` and described in the chapter “[Helper API](#)” on page 719 of the *Palm OS Programmer’s API Reference*. These services are to dial a number, email a message, send an SMS message, or send a fax. If you want to define your own service, you must register a unique creator ID for that service. Alternatively, you can use the creator ID of your application.

This section discusses the helper APIs, which include the `sysNotifyHelperEvent` notification and the data structures that it passes as the `notifyDetailsP` portion of the [SysNotifyParamType](#) structure. It covers:

- [When to Use the Helper API](#)
- [Requesting a Helper Service](#)
- [Implementing a Helper](#)

When to Use the Helper API

If the [4.0 New Feature Set](#) is present, there are several means by which one application can communicate with another application on the same handheld. Specifically, an application can send a launch code to another application (see “[Launching Applications Programmatically](#)” in this chapter), can use the Exchange Manager and Local Exchange Library to send data to another application (see the “[Object Exchange](#)” chapter), or can use the helper API to request that a service be performed. It can be difficult to determine which is the best method to use for your particular situation.

The helper API is best used in these circumstances:

- The [4.0 New Feature Set](#) is present.
- You do not know anything about the receiving application.

The helper API provides a means of communication where the sending and receiving application do not need to know anything about each other. This contrasts with the launch code mechanism, in which the sending application must know the card number and local ID of the receiving database as well as which launch code to send.

- You want to communicate with any type of program.

Because the helper API uses a notification, the helper can be a shared library or another separately loaded code resource. Launch codes can only be received by applications. Because the Exchange Manager works through launch codes, it also only works with applications.

Requesting a Helper Service

[Listing 2.8](#) shows how an application should request the dial service. In general, you should do the following to request a service:

- Broadcast a [sysNotifyHelperEvent](#) with a `kHelperNotifyActionCodeValidate` action each time you want to advertise that the service is available.

For example, when the Address Book initializes the List view form, it checks to see if the dial service is available by broadcasting the notification with the action code `kHelperNotifyActionCodeValidate`. The Dial application makes sure the Telephony Library is open. If so, it sets `handled` to `true` in the [SysNotifyParamType](#) structure. If not, it sets `handled` to `false`. If `handled` is `false` after the notification is broadcast, the Address Book does not display the Dial menu item.

- Broadcast a `sysNotifyHelperEvent` with a `kHelperNotifyActionCodeExecute` action when you want the service performed. See [Listing 2.8](#).
- If you want to obtain a list of all possible services, broadcast a `sysNotifyHelperEvent` with a `kHelperNotifyActionCodeEnumerate` action. You might do so when your application is launched, upon system reset, or any time the user performs a task where you might want to provide a service.

Listing 2.8 Requesting a helper service

```
Boolean PrvDialListDialSelected(FormType* frmP) {
    SysNotifyParamType param;
    HelperNotifyEventType details;
    HelperNotifyExecuteType execute;

    param.notifyType = sysNotifyHelperEvent;
    param.broadcaster = sysFileCAddress;
    param.notifyDetailsP = &details;
    param.handled = false;

    details.version = kHelperNotifyCurrentVersion;
    details.actionCode = kHelperNotifyActionCodeExecute;
    details.data.executeP = &execute;
```

```
execute.serviceClassID = kHelperServiceClassIDVoiceDial;
execute.helperAppID = 0;
execute.dataP = FldGetTextPtr(ToolsGetFrmObjectPtr(frmP,
    DialListNumberField));
execute.displayedName = gDisplayName;
execute.detailsP = 0;
execute.err = errNone;

SysNotifyBroadcast(&param);

// Check error code
if (!param.handled)
// Not handled so exit the list - Unexpected error
    return true;
else
    return (execute.err == errNone);
}
```

When you broadcast the `sysNotifyHelperEvent`, it's important to note the following:

- Always use [SysNotifyBroadcast](#), which broadcasts the notification synchronously.
- The notification's `notifyDetailsP` parameter points to a [HelperNotifyEventType](#). This structure allows the broadcaster to communicate with the helper.
- The helper may allocate memory and add it to the `HelperNotifyEventType` structure. In particular, if the action code is `kHelperNotifyActionCodeEnumerate`, the helper allocates at least one structure of type [HelperNotifyEnumerateListType](#) and adds it to the `data` field in the `HelperNotifyEventType` structure. The broadcaster must free this memory, even though the helper allocated it.
- The broadcaster uses the `helperAppID` field to communicate directly with a particular provider of the requested service. For example, suppose two applications provide a dial service. The broadcaster might discover these two applications through the `enumerate` action and then allow the user to specify which application should dial the phone number. When broadcasting the `enumerate` action, no helper ID is specified, so all helpers respond. After the user has set the preferred helper, the broadcaster sets the

Application Startup and Stop

Helper Notifications

`helperAppID` field for the validate and execute actions to that helper's creator ID. A helper must check the `helperAppID` field and only respond to the notification if its creator ID matches the value in that field or if that field is 0.

- The `dataP` field contains the data required to perform the service. For the dial service, `dataP` contains the phone number to dial. If any extra information is required or desired, then it is provided in the `detailsP` field. If you're requesting the email or SMS service, you use `detailsP` to provide the message to be sent. See the chapter "[Helper API](#)" on page 719 of the *Palm OS Programmer's API Reference* for more information.
- The `handled` field of `SysNotifyParamType` and the `err` field of the `HelperNotifyEventType` structure are used to return the result. Always set `handled` to false and `err` to `errNone` before broadcasting and check their values after the broadcast is complete. The helper uses `handled` to indicate if it attempted to handle the service. If `handled` is true, it uses `err` to indicate the success or failure of performing that service.

Implementing a Helper

To implement a helper, do the following:

- Register to receive the [sysNotifyHelperEvent](#). It is best to register for this notification in response to the [sysAppLaunchCmdSyncNotify](#) and [sysAppLaunchCmdSystemReset](#) launch codes. This registers your helper when it is first installed and re-registers it upon each system reset.
- In the notification handler, handle the three possible actions: enumerate, execute, and validate. Note that even though the enumerate action is optional and not currently used by Address Book, a helper must respond to this action in its handler because another third party application might send the enumerate action.

[Listing 2.9](#) and [Listing 2.10](#) show how the Dial application responds to the enumerate and validate actions. Note that the enumerate action requires the helper to allocate memory and add that memory to the [HelperNotifyEventType](#) structure pointed to by

notifyDetailsP in the [SysNotifyParamType](#) parameter block. In this case, the notifyDetailsP->dataP field is a linked list of [HelperNotifyEnumerateListType](#) structures. Each helper must allocate one of these structure per service and add it to the end of the list. The broadcaster is responsible for freeing all of these structures after the notification broadcast is complete.

Listing 2.9 Enumerating services provided

```
Boolean PrvAppEnumerate
(helperNotifyEventType *helperNotifyEventP)
{
    HelperNotifyEnumerateListType* newNodeP;
    MemHandle handle;
    MemPtr stringP;

    newNodeP = MemPtrNew
        (sizeof(HelperNotifyEnumerateListType));

    // Get name to display in user interface.
    handle = DmGetResource(strRsc, HelperAppNameString);
    stringP = MemHandleLock(handle);
    StrCopy(newNodeP->helperAppName, stringP);
    MemHandleUnlock(handle);
    DmReleaseResource(handle);

    // Get name of service to display in UI.
    handle = DmGetResource(strRsc, HelperActionNameString);
    stringP = MemHandleLock(handle);
    StrCopy(newNodeP->actionName, stringP);
    MemHandleUnlock(handle);
    DmReleaseResource(handle);

    newNodeP->serviceClassID = kHelperServiceClassIDVoiceDial;
    newNodeP->helperAppID = kDialCreator;
    newNodeP->nextP = 0;

    // Add the new node.
    if (helperNotifyEventP->data.enumerateP == 0) {
        helperNotifyEventP->data.enumerateP = newNodeP;
    } else {
        HelperNotifyEnumerateListType* nodeP;
        nodeP = helperNotifyEventP->data.enumerateP;
        //Look for the end of the list.
        while ( nodeP->nextP != 0 )
```

Application Startup and Stop

Helper Notifications

```
        nodeP = nodeP->nextP;
        nodeP->nextP = newNodeP;
    }

    return true;
}
```

[Listing 2.10](#) show how the Dial application responds to the validate action.

Listing 2.10 Responding to validate action

```
Boolean PrvAppValidate (SysNotifyParamType *sysNotifyParamP)
{
    HelperNotifyEventType* helperNotifyEvent;

    helperNotifyEvent = sysNotifyParamP->notifyDetailsP;
    // Check version
    if (helperNotifyEvent->version < 1)
        return false;

    // Check service
    if (helperNotifyEvent-> data.validateP->serviceClassID
        != kHelperServiceClassIDVoiceDial)
        return false;

    // check appId (either null or me)
    if ((helperNotifyEvent->data.validateP->helperAppID != 0)
        && (helperNotifyEvent->data.validateP->helperAppID !=
            kDialCreator))
        return false;

    // Check Telephony library presence
    if (!PrvAppCheckTelephony())
        return false;

    sysNotifyParamP->handled = true;
    return true;
}
```

When writing a helper, it is also important to note the following:

- Always check the helperAppID field and only respond if it is 0 or if it matches your creator ID. For the validate and

execute actions, a broadcaster may use `helperAppID` to only communicate with the desired helper.

- If you handle the action, set `handled` to `true`. If the handling of the service was unsuccessful, set the `err` field in `notifyDetailsP`.
- Always check the `handled` field before performing the service. If any helper can perform the service, you must make sure that the service has not already been performed before you perform it. If `handled` is `true`, the service has already been performed.
- Remember that, as with all notifications, your notification handler does not have access to global variables. If there is data you need to access, pass it in the `userDataP` parameter to [SysNotifyRegister](#). If you want to have the notification handler return before the service is fully complete, make a copy of any data in the parameter block that you will need to complete the service.

Launch Code Summary

[Table 2.1](#) lists all Palm OS standard launch codes. These launch codes are declared in the header `SystemMgr.h`. All the parameters for a launch code are passed in a single parameter block, and the results are returned in the same parameter block.

Table 2.1 Palm OS Launch Codes

Code	Request
scptLaunchCmdExecuteCmd	Execute the specified Network login script plugin command.
scptLaunchCmdListCmds	Provide information about the commands that your Network script plugin executes.
sysAppLaunchCmdAddRecord	Add a record to a database.
sysAppLaunchCmdAlarmTriggered	Schedule next alarm or perform quick actions such as sounding alarm tones.
sysAppLaunchCmdAttention	Perform the action requested by the attention manager.
sysAppLaunchCmdCardLaunch	Launch the application. This launch code signifies that the application is being launched from an expansion card.
sysAppLaunchCmdCountryChange	Respond to country change.
sysAppLaunchCmdDisplayAlarm	Display specified alarm dialog or perform time-consuming alarm-related actions.
sysAppLaunchCmdExgAskUser	Let application override display of dialog asking user if they want to receive incoming data via the Exchange Manager.
sysAppLaunchCmdExgGetData	Notify application that it should send data using the Exchange Manager.

Table 2.1 Palm OS Launch Codes (*continued*)

Code	Request
<u>sysAppLaunchCmdExgPreview</u>	Notify application that it should display a preview using the Exchange Manager.
<u>sysAppLaunchCmdExgReceiveData</u>	Notify application that it should receive incoming data using the Exchange Manager.
<u>sysAppLaunchCmdFind</u>	Find a text string.
<u>sysAppLaunchCmdGoto</u>	Go to a particular record, display it, and optionally select the specified text.
<u>sysAppLaunchCmdGoToURL</u>	Launch an application and open a URL.
<u>sysAppLaunchCmdHandleSyncCallApp</u>	Perform some application-specific operation at the behest of the application's conduit.
<u>sysAppLaunchCmdInitDatabase</u>	Initialize database.
<u>sysAppLaunchCmdLookup</u>	Look up data. In contrast to <code>sysAppLaunchCmdFind</code> , a level of indirection is implied. For example, look up a phone number associated with a name.
<code>sysAppLaunchCmdNormalLaunch</code>	Launch normally.
<u>sysAppLaunchCmdNotify</u>	Broadcast a notification.
<u>sysAppLaunchCmdOpenDB</u>	Launch application and open a database.
<u>sysAppLaunchCmdPanelCalledFromApp</u>	Tell preferences panel that it was invoked from an application, not the Preferences application.
<u>sysAppLaunchCmdReturnFromPanel</u>	Tell an application that it's restarting after preferences panel had been called.

Application Startup and Stop

Notification Summary

Table 2.1 Palm OS Launch Codes (*continued*)

Code	Request
sysAppLaunchCmdSaveData	Save data. Often sent before find operations.
sysAppLaunchCmdSyncNotify	Notify applications that a HotSync has been completed.
sysAppLaunchCmdSystemLock	Sent to the Security application to request that the system be locked down.
sysAppLaunchCmdSystemReset	Respond to system reset. No UI is allowed during this launch code.
sysAppLaunchCmdTimeChange	Respond to system time change.
sysAppLaunchCmdURLParams	Launch an application with parameters from the Web Clipping Application Viewer.

Notification Summary

[Table 2.2](#) lists all Palm OS standard notifications. These notifications are declared in the header `NotifyMgr.h`. All the parameters for a notification are passed in a [SysNotifyParamType](#) structure and the results are returned in that same structure.

Table 2.2 Notification Constants

Constant	Description
cncNotifyProfileEvent	The connection profile used by the Connection Panel has changed.
sysExternalConnectorAttachEvent	A device has been attached to an external connector.
sysExternalConnectorDetachEvent	A device has been detached from an external connector.

Table 2.2 Notification Constants (*continued*)

Constant	Description
<u>sysNotifyAntennaRaisedEvent</u>	The antenna has been raised on a Palm VII series handheld.
<u>sysNotifyCardInsertedEvent</u>	An expansion card has been inserted into the expansion slot.
<u>sysNotifyCardRemovedEvent</u>	An expansion card has been removed from the expansion slot.
<u>sysNotifyDBCreatedEvent</u>	A database has been created.
<u>sysNotifyDBChangedEvent</u>	Database info has been set on a database, such as with <u>DmSetDatabaseInfo</u> .
<u>sysNotifyDBDeletedEvent</u>	A database has been deleted.
<u>sysNotifyDBDirtyEvent</u>	An overlay has been opened, a database has been opened for write, or another event has occurred which has made the database info “dirty.”
<u>sysNotifyDeleteProtectedEvent</u>	The Launcher has attempted to delete a protected database.
<u>sysNotifyDeviceUnlocked</u>	The user has unlocked the handheld.
<u>sysNotifyDisplayChangeEvent</u>	The color table or bit depth has changed.
<u>sysNotifyEarlyWakeupEvent</u>	The system is starting to wake up.
<u>sysNotifyForgotPasswordEvent</u>	The user has tapped the Lost Password button in the Security application.
<u>sysNotifyGotUsersAttention</u>	The Attention Manager has informed the user of an event.
<u>sysNotifyHelperEvent</u>	An application has requested that a particular service be performed.
<code>sysNotifyIrDASniffEvent</code>	Not used.
<u>sysNotifyLateWakeupEvent</u>	The system has finished waking up.

Table 2.2 Notification Constants (*continued*)

Constant	Description
sysNotifyLocaleChangedEvent	The system locale has changed.
sysNotifyMenuCmdBarOpenEvent	The system is about to display the menu command toolbar.
sysNotifyNetLibIFMediaEvent	The system has been connected to or disconnected from the network.
<code>sysNotifyPhoneEvent</code>	Reserved for future use.
<code>sysNotifyPOSEMountEvent</code>	System use only.
sysNotifyResetFinishedEvent	The system has finished a reset.
sysNotifyRetryEnqueueKey	The Attention Manager has failed to post a virtual character to the key queue.
sysNotifySleepNotifyEvent	The system is about to go to sleep.
sysNotifySleepRequestEvent	The system has decided to go to sleep.
sysNotifySyncFinishEvent	A HotSync operation has just completed.
sysNotifySyncStartEvent	A HotSync operation is about to begin.
sysNotifyTimeChangeEvent	The system time has just changed.
sysNotifyVolumeMountedEvent	A file system has been mounted.
sysNotifyVolumeUnmountedEvent	A file system has been unmounted.

Launch and Notification Function Summary

Launching Applications

SysAppLaunch	SysUIAppSwitch SysBroadcastActionCode
------------------------------	--

Notification Manager Functions

SysNotifyRegister SysNotifyBroadcast	SysNotifyUnregister SysNotifyBroadcastDeferred SysNotifyBroadcastFromInterrupt
---	--

Event Loop

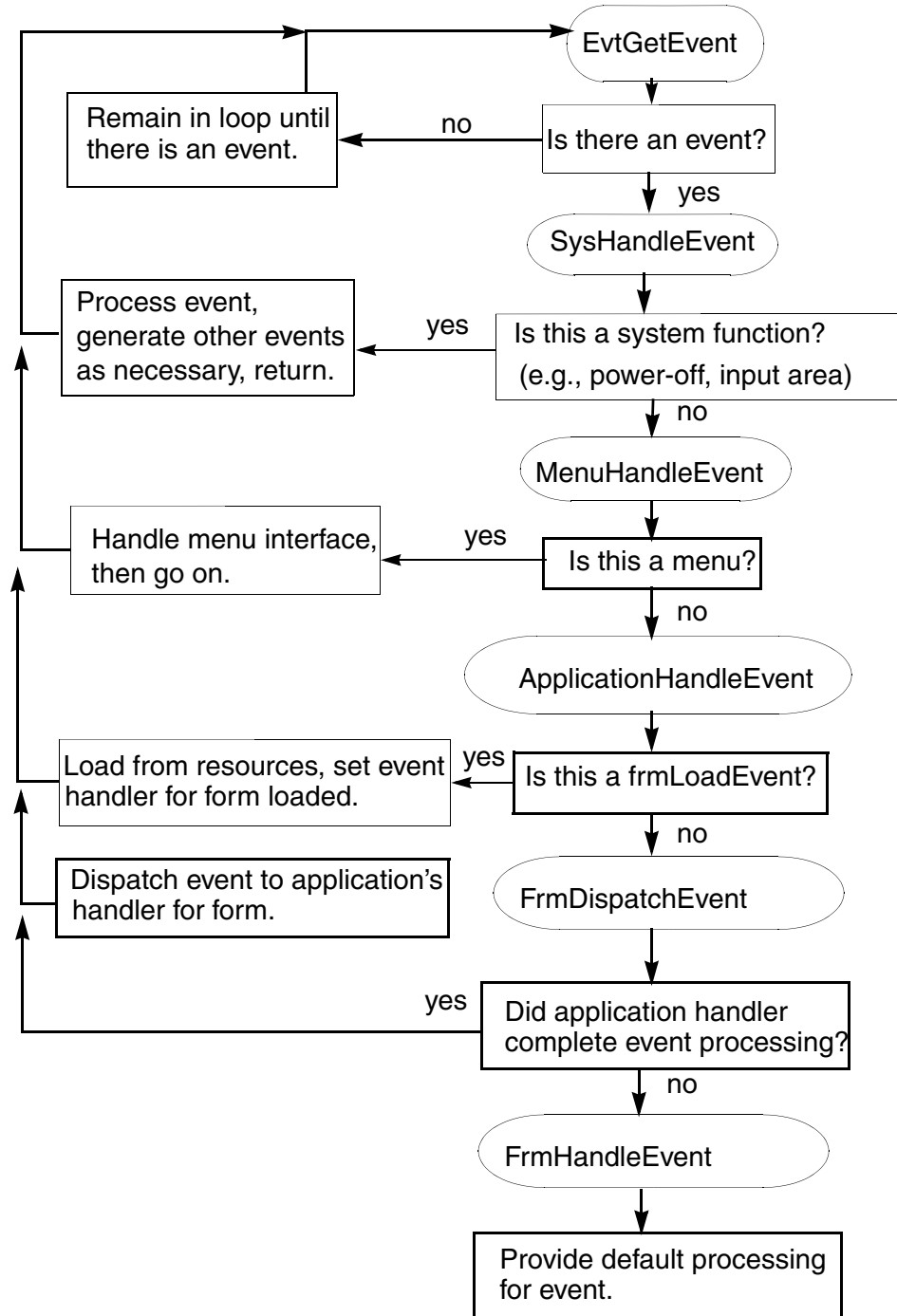
This chapter discusses the Event Manager, the main interface between the Palm OS[®] system software and the application. It discusses in some detail what an application does in response to user input, providing code fragments as examples where needed. The topics covered are:

- [The Application Event Loop](#)
- [Low-Level Event Management](#)

This chapter's focus is on how to write your applications main event loop. For more detailed information on events, consult the *Palm OS Programmer's API Reference*. Details for each event are given in [Chapter 2](#), "[Palm OS Events](#)." In addition to the reference material, consult the chapter "[User Interface](#)" in this book. It provides the event flow for each user interface element.

[Figure 3.1](#) illustrates control flow in a typical application.

Figure 3.1 Control Flow in a Typical Application



The Application Event Loop

As described in the previous chapter, “[Application Startup and Stop](#),” an application performs a full startup when it receives the launch code `sysAppLaunchCmdNormalLaunch`. It begins with a startup routine, then goes into an event loop, and finally exits with a stop routine.

In the event loop, the application fetches events from the queue and dispatches them, taking advantage of the default system functionality as appropriate.

While in the loop, the application continuously checks for events in the event queue. If there are events in the queue, the application has to process them as determined in the event loop. As a rule, the events are passed on to the system, which knows how to handle them. For example, the system knows how to respond to pen taps on forms or menus.

The application typically remains in the event loop until the system tells it to shut itself down by sending an [appStopEvent](#) (not a launch code) through the event queue. The application must detect this event and terminate.

Listing 3.1 Top-Level Event Loop Example from Datebook.c

```
static void EventLoop (void)
{
    UInt16 error;
    EventType event;
    do
    {
        EvtGetEvent (&event, evtWaitForever);

        PreprocessEvent (&event);

        if (! SysHandleEvent (&event))

            if (! MenuHandleEvent (NULL, &event, &error))

                if (! ApplicationHandleEvent (&event))
                    FrmDispatchEvent (&event);
    } while (true);
}
```

Event Loop

The Application Event Loop

```
        #if EMULATION_LEVEL != EMULATION_NONE
            ECApptDBValidate (ApptDB);
        #endif
    }
    while (event.eType != appStopEvent);
}
```

In the event loop, the application iterates through these steps (see [Figure 3.1](#) and [Listing 3.1](#))

1. Fetch an event from the event queue.
2. Call `PreprocessEvent` to allow the datebook event handler to see the command keys before any other event handler gets them. Some of the datebook views display UI that disappears automatically; this UI needs to be dismissed before the system event handler or the menu event handler display any UI objects.

Note that not all applications need a `PreprocessEvent` function. It may be appropriate to call `SysHandleEvent` right away.

3. Call [SysHandleEvent](#) to give the system an opportunity to handle the event.

The system handles events like power on/power off, Graffiti® or Graffiti 2 input, tapping input area icons, or pressing buttons. During the call to `SysHandleEvent`, the user may also be informed about low-battery warnings or may find and search another application.

Note that in the process of handling an event, `SysHandleEvent` may generate new events and put them on the queue. For example, the system handles Graffiti 2 input by translating the pen events to key events. Those, in turn, are put on the event queue and are eventually handled by the application.

`SysHandleEvent` returns `true` if the event was completely handled, that is, no further processing of the event is required. The application can then pick up the next event from the queue.

4. If `SysHandleEvent` did not completely handle the event, the application calls [MenuHandleEvent](#).
`MenuHandleEvent` handles two types of events:
 - If the user has tapped in the area that invokes a menu, `MenuHandleEvent` brings up the menu.
 - If the user has tapped inside a menu to invoke a menu command, `MenuHandleEvent` removes the menu from the screen and puts the events that result from the command onto the event queue.

`MenuHandleEvent` returns true if the event was completely handled.

5. If `MenuHandleEvent` did not completely handle the event, the application calls `ApplicationHandleEvent`, a function your application has to provide itself. `ApplicationHandleEvent` handles only the [frmLoadEvent](#) for that event; it loads and activates application form resources and sets the event handler for the active form by calling the function [FrmSetEventHandler](#).
6. If `ApplicationHandleEvent` did not completely handle the event, the application calls [FrmDispatchEvent](#). `FrmDispatchEvent` first sends the event to the application's event handler for the active form. This is the event handler routine that was established in `ApplicationHandleEvent`. Thus the application's code is given the first opportunity to process events that pertain to the current form. The application's event handler may completely handle the event and return true to calls from `FrmDispatchEvent`. In that case, `FrmDispatchEvent` returns to the application's event loop. Otherwise, `FrmDispatchEvent` calls [FrmHandleEvent](#) to provide the system's default processing for the event.

For example, in the process of handling an event, an application frequently has to first close the current form and then open another one, as follows:

- The application calls [FrmGotoForm](#) to bring up another form. `FrmGotoForm` queues a [frmCloseEvent](#) for the currently active form, then queues [frmLoadEvent](#) and [frmOpenEvent](#) for the new form.
- When the application gets the `frmCloseEvent`, it closes and erases the currently active form.

Event Loop

The Application Event Loop

- When the application gets the `frmLoadEvent`, it loads and then activates the new form. Normally, the form remains active until it's closed. (Note that this wouldn't work if you preload all forms, but preloading is really discouraged. Applications don't need to be concerned with the overhead of loading forms; loading is so fast that applications can do it when they need it.) The application's event handler for the new form is also established.
- When the application gets the `frmOpenEvent`, it performs any required initialization of the form, then draws the form on the display.

After `FrmGotoForm` has been called, any further events that come through the main event loop and to `FrmDispatchEvent` are dispatched to the event handler for the form that's currently active. For each dialog or form, the event handler knows how it should respond to events, for example, it may open, close, highlight, or perform other actions in response to the event. [FrmHandleEvent](#) invokes this default UI functionality.

After the system has done all it can to handle the event for the specified form, the application finally calls the active form's own event handling function. For example, in the datebook application, it may call `DayViewHandleEvent` or `WeekViewHandleEvent`.

Notice how the event flow allows your application to rely on system functionality as much as it wants. If your application wants to know whether a button is pressed, it has only to wait for [ctlSelectEvent](#). All the details of the event queue are handled by the system.

Some events are actually requests for the application to do something, for example, [frmOpenEvent](#). Typically, all the application does is draw its own interface, using the functions provided by the system, and then waits for events it can handle to arrive from the queue.

Only the active form should process events.

Low-Level Event Management

The five ways that a user interacts with an application are:

- by writing letters, numbers, or symbols in the input area
- by pressing a hardware button on the handheld
- by tapping the pen on a control in a form or dialog
- by tapping on an onscreen keyboard in the keyboard dialog.
- by tapping in the menu bar or in a particular menu.

For the first three types of input, the Palm OS provides a dedicated manager: the Graffiti Manager, the Key Manager, and the Pen Manager, respectively. Most applications do not need to access these managers directly; instead, applications receive events from these managers and respond to the events. There are cases, however, where you might need to interact with one of these managers. The following pages describe each of these managers and when you might need to use them. To learn how to obtain user input from a UI object, refer to the section in [Chapter 4, “User Interface,”](#) on page 69 that covers that object.

The keyboard dialog allows users to input characters into a text field by tapping an onscreen keyboard. When the keyboard dialog is closed, the amended text is automatically displayed in the original field. As with the three managers just mentioned, you will probably not need to access the keyboard dialog directly. The user can open the keyboard from any text field. In certain limited circumstances, however, you may wish to display the keyboard dialog programmatically. For more information, see “[The Keyboard Dialog](#)” on page 87.

The Menu Manager handles taps that display a menu and those that select an item from a menu. For details, see “[Menus](#)” on page 105.

In addition to these managers, the System Event Manager is another manager involved in low-level event handling. Most applications have no need to call the System Event Manager directly because most of the functionality they need comes from the higher-level Event Manager or is automatically handled by the system.

Event Loop

Low-Level Event Management

This section provides information about the following managers:

- [The Graffiti Manager](#)
- [The Key Manager](#)
- [The Pen Manager](#)
- [The System Event Manager](#)

The Graffiti Manager

The Graffiti Manager provides an API to the Palm OS Graffiti or Graffiti 2 recognizer. The recognizer converts pen strokes into key events, which are then fed to an application through the Event Manager.

IMPORTANT: If the [Graffiti 2 Feature Set](#) is present, many of the Graffiti Manager API calls are deprecated or work differently. Avoid using the Graffiti Manager API.

Most applications never need to call the Graffiti Manager directly because it's automatically called by the Event Manager whenever it detects pen strokes in the input area of the digitizer.

Special-purpose applications, such as a Graffiti tutorial, may want to call the Graffiti Manager directly to recognize strokes in other areas of the screen or to customize the Graffiti behavior.

Using GrfProcessStroke

[GrfProcessStroke](#) is a high-level Graffiti Manager call used by the Event Manager for converting pen strokes into key events. The call

- Removes pen points from the pen queue
- Recognizes the stroke
- Puts one or more key events into the key queue

`GrfProcessStroke` automatically handles shortcut strokes and calls the user interface as appropriate to display shift indicators in the current window.

An application can call `GrfProcessStroke` when it receives a [penUpEvent](#) from the Event Manager if it wants to recognize strokes entered into its application area (in addition to the input area).

Using Other High-Level Graffiti Manager Calls

Other high-level calls provided by the Graffiti Manager include routines for

- Getting and setting the current shift state (caps lock on/off, temporary shift state, etc.)
- Notifying the handwriting recognizer when the user selects a different field. The handwriting recognizer needs to be notified when a field change occurs so that it can cancel out of any partially entered shortcut and clear its temporary shift state if it's showing a potentially accented character.

Note that if the [Graffiti 2 Feature Set](#) is present, the caps lock state is not supported. In Graffiti 2 handwriting, users write uppercase letters by writing between the dividers that separate the letter area from the number area. Because users can easily write a succession of uppercase letters, the caps lock state is unnecessary.

Special-Purpose Graffiti Manager Calls

The remainder of Graffiti Manager API routines are for special-purpose use. They are basically all the entry points into the Graffiti recognizer engine and are usually called only by [GrfProcessStroke](#). These special-purpose uses include calls to add pen points to the Graffiti recognizer's stroke buffer, to convert the stroke buffer into a Graffiti glyph ID, and to map a glyph into a string of one or more key strokes.

IMPORTANT: If the [Graffiti 2 Feature Set](#) is present, the special-purpose Graffiti Manager calls are deprecated.

Accessing Shortcuts

Other routines provide access to the Graffiti or Graffiti 2 Shortcuts database. This is a separate database owned and maintained by the Graffiti Manager that contains all of the shortcuts. In Palm OS

Event Loop

Low-Level Event Management

version 3.5 and earlier releases, this database is opened by the Graffiti Manager when it initializes and stays open even after applications quit. Starting in Palm OS 4.0, the database is only opened when necessary and is closed when it is no longer needed.

The only way to modify this database is through the Graffiti Manager API. It provides calls for getting a list of all shortcuts, and for adding, editing, and removing shortcuts. The Shortcuts screen of the Preferences application provides a user interface for modifying this database.

Note on Auto Shifting

The Palm OS 2.0 and later automatically uses an upper-case letter under the following conditions:

- Period and space or Return.
- Other sentence terminator (such as ? or !) and space

This functionality requires no changes by the developer, but should be welcome to the end user.

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 handheld is being sold, and so on.

Note on Graffiti Help

In Palm OS 2.0 and later, applications can pop up Graffiti help by calling [`SysGraffitiReferenceDialog`](#) or by putting a virtual character—`graffitiReferenceChr` from `Chars.h`—on the queue.

Graffiti help is also available through the system Edit menu. As a result, any application that includes the system Edit menu allows users to access Graffiti help that way.

The Key Manager

The Key Manager manages the hardware buttons on the Palm Powered™ handheld. It converts button presses into key events and implements auto-repeat of the buttons. Most applications never

need to call the Key Manager directly except to change the key repeat rate or to poll the current state of the keys.

The Event Manager is the main interface to the keys; it returns a [keyDownEvent](#) to an application whenever a button is pressed. Normally, applications are notified of key presses through the Event Manager. Whenever a hardware button is pressed, the application receives an event through the Event Manager with the appropriate key code stored in the event record. The state of the hardware buttons can also be queried by applications at any time through the [KeyCurrentState](#) function call.

The [KeyRates](#) call 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 are stored in the Key Manager globals and should be restored before the application exits.

The Pen Manager

The Pen Manager manages the digitizer hardware and converts input from the digitizer into pen coordinates. The Palm Powered handheld has a built-in digitizer overlaid onto the LCD screen and extending about an inch below the screen. This digitizer is capable of sampling accurately to within 0.35 mm (.0138 in) with up to 50 accurate points/second. When the handheld is in doze mode, an interrupt is generated when the pen is first brought down on the screen. After a pen down is detected, the system software polls the pen location periodically (every 20 ms) until the pen is again raised.

Most applications never need to call the Pen Manager directly because any pen activity is automatically returned to the application in the form of events.

Pen coordinates are stored in the pen queue as raw, uncalibrated coordinates. When the System Event Manager routine for removing pen coordinates from the pen queue is called, it converts the pen coordinate into screen coordinates before returning.

The Preferences application provides a user interface for calibrating the digitizer. It uses the Pen Manager API to set up the calibration which is then saved into the Preferences database. The Pen Manager assumes that the digitizer is linear in both the x and y directions; the

calibration is therefore a simple matter of adding an offset and scaling the x and y coordinates appropriately.

The System Event Manager

The System Event Manager:

- manages the low-level pen and key event queues.
- translates taps on input area icons into key events.
- sends pen strokes in the input area to the Graffiti or Graffiti 2 recognizer.
- puts the system into low-power doze mode when there is no user activity.

Most applications have no need to call the System Event Manager directly because most of the functionality they need comes from the higher-level Event Manager or is automatically handled by the system.

Applications that do use the System Event Manager directly might do so to enqueue key events into the key queue or to retrieve each of the pen points that comprise a pen stroke from the pen queue.

Event Translation: Pen Strokes to Key Events

One of the higher-level functions provided by the System Event Manager is conversion of pen strokes on the digitizer to key events. For example, the System Event Manager sends any stroke in the input area of the digitizer automatically to the Graffiti or Graffiti 2 recognizer for conversion to a key event. Taps on input area icons, such as the Application button, Menu button, and Find button, are also intercepted by the System Event Manager and converted into the appropriate key events.

When the system converts a pen stroke to a key event, it:

- Retrieves all pen points that comprise the stroke from the pen queue
- Converts the stroke into the matching key event
- Enqueues that key event into the key queue

Eventually, the system returns the key event to the application as a normal result of calling [EvtGetEvent](#).

Most applications rely on the following default behavior of the System Event Manager:

- All strokes in the predefined input area of the digitizer are converted to key events
- All taps on the input area icons are convert to key events
- All other strokes are passed on to the application for processing

If the [Graffiti 2 Feature Set](#) is present, you should be careful when processing key events one at a time. Graffiti 2 contains several multi-stroke characters. If the first stroke matches the stroke for another letter, the Graffiti 2 engine first enqueues the single-stroke character followed by a backspace and the multi-stroke character.

For example, The first stroke of a “k” character matches the stroke for the “l” character. When the user draws the first stroke of the “k,” the Graffiti 2 engine processes that stroke and sends the “l” character to the key queue. When the user draws the second stroke of the “k,” the Graffiti 2 engine sends a backspace to erase the “l” and then sends the “k” character. If your application has processed the “l” in the meantime, it has done so in error.

If your application processes characters as they are received, you should be aware that the following characters could be the first stroke of a multi-stroke character:

- The letter l
- The space character
- The single quote
- The minus sign

When your application receives these characters, it should wait to see if the next character received is a backspace.

Pen Queue Management

The pen queue is a preallocated area of system memory used for capturing the most recent pen strokes on the digitizer. It is a circular queue with a first-in, first-out method of storing and retrieving pen points. Points are usually enqueued by a low-level interrupt routine and dequeued by the System Event Manager or application.

Event Loop

Low-Level Event Management

[Table 3.1](#) summarizes pen management.

Table 3.1 Pen queue management

The user...	The system...
Brings the pen down on the digitizer.	Stores a pen-down sequence in the pen queue and starts the stroke capture.
Draws a character.	Stores additional points in the pen queue periodically.
Lifts the pen.	Stores a pen-up sequence in the pen queue and turns off stroke capture.

The System Event Manager provides an API for initializing and flushing the pen queue and for queuing and dequeuing points. Some state information is stored in the queue itself: to dequeue a stroke, the caller must first make a call to dequeue the stroke information ([EvtDequeuePenStrokeInfo](#)) before the points for the stroke can be dequeued. Once the last point is dequeued, another `EvtDequeuePenStrokeInfo` call must be made to get the next stroke.

Applications usually don't need to call `EvtDequeuePenStrokeInfo` because the Event Manager calls this function automatically when it detects a complete pen stroke in the pen queue. After calling `EvtDequeuePenStrokeInfo`, the System Event Manager stores the stroke bounds into the event record and returns the pen-up event to the application. The application is then free to dequeue the stroke points from the pen queue, or to ignore them altogether. If the points for that stroke are not dequeued by the time [EvtGetEvent](#) is called again, the System Event Manager automatically flushes them.

Key Queue Management

The key queue is an area of system memory preallocated for capturing key events. Key events come from one of two occurrences:

- As a direct result of the user pressing one of the buttons on the case

- As a side effect of the user drawing a Graffiti or Graffiti 2 stroke on the digitizer, which is converted in software to a key event

[Table 3.2](#) summarizes key management.

Table 3.2 Key queue management

User action	System response
Hardware button press.	Interrupt routine enqueues the appropriate key event into the key queue, temporarily disables further hardware button interrupts, and sets up a timer task to run every 10 ms.
Hold down key for extended time period.	Timer task to supports auto-repeat of the key (timer task is also used to debounce the hardware).
Release key for certain amount of time.	Timer task reenables the hardware button interrupts.
Pen stroke in input area of digitizer.	System Manager calls the Graffiti or Graffiti 2 recognizer, which then removes the stroke from the pen queue, converts the stroke into one or more key events, and finally enqueues these key events into the key queue.
Pen stroke on silk-screened icons.	System Event Manager converts the stroke into the appropriate key event and enqueues it into the key queue.

The System Event Manager provides an API for initializing and flushing the key queue and for enqueueing and dequeuing key events. Usually, applications have no need to dequeue key events; the Event Manager does this automatically if it detects a key in the queue and returns a `keyDownEvent` to the application through the [EvtGetEvent](#) call.

Auto-Off Control

Because the System Event Manager manages hardware events like pen taps and hardware button presses, it's responsible for resetting the auto-off timer on the handheld. Whenever the system detects a hardware event, it automatically resets the auto-off timer to 0. If an application needs to reset the auto-off timer manually, it can do so through the System Event Manager call [EvtResetAutoOffTimer](#).

Event Loop

System Event Manager Summary

System Event Manager Summary

System Event Manager Functions

Main Event Queue Management

[EvtGetEvent](#)

[EvtEventAvail](#)

[EvtSysEventAvail](#)

[EvtAddEventToQueue](#)

[EvtAddUniqueEventToQueue](#)

[EvtCopyEvent](#)

[EvtSetNullEventTick](#)

Pen Queue Management

[EvtPenQueueSize](#)

[EvtDequeuePenPoint](#)

[EvtDequeuePenStrokeInfo](#)

[EvtFlushNextPenStroke](#)

[EvtFlushPenQueue](#)

[EvtGetPen](#)

[EvtGetPenBtnList](#)

Key Queue Management

[EvtKeyQueueSize](#)

[EvtEnqueueKey](#)

[EvtFlushKeyQueue](#)

[EvtKeyQueueEmpty](#)

[EvtKeydownIsVirtual](#)

Handling pen strokes and key strokes

[EvtEnableGraffiti](#)

[EvtProcessSoftKeyStroke](#)

Handling input area

[EvtGetSilkscreenAreaList](#)

Handling power on and off events

[EvtResetAutoOffTimer](#)

[EvtSetAutoOffTimer](#)

[EvtWakeup](#)

[EvtWakeupWithoutNilEvent](#)

Graffiti Manager Functions

Translate a Stroke into Keyboard Events

[GrfProcessStroke](#)

Shift State

[GrfInitState](#)

[GrfGetState](#)

[GrfCleanState](#)

[GrfSetState](#)

[GrfFindBranch](#)

Point Buffer

[GrfGetNumPoints](#)

[GrfGetPoint](#)

[GrfAddPoint](#)

[GrfFilterPoints](#)

[GrfFlushPoints](#)

[GrfGetGlyphMapping](#)

[GrfMatch](#)

[GrfMatchGlyph](#)

Working with Macros

[GrfGetAndExpandMacro](#)

[GrfAddMacro](#)

[GrfDeleteMacro](#)

[GrfGetMacro](#)

[GrfGetMacroName](#)

Key Manager Functions

[KeyCurrentState](#)

[KeyRates](#)

[KeySetMask](#)

Pen Manager Functions

[PenCalibrate](#)

[PenResetCalibration](#)

Event Loop

System Event Manager Summary

User Interface

This chapter describes the user interface (UI) elements that you can use in your application. To create a user interface element, you create a resource that defines what that element looks like and where it is displayed. You interact with the element programmatically as a UI object. A Palm OS® UI object is a C structure that is linked with one or more items on the screen. Note that Palm UI objects are just structures, not the more elaborate objects found in some systems. This is useful because in general a C structure is more compact than these other objects.

This chapter introduces each of the user interface objects. It also describes Palm system managers that aid in working with the user interface. The chapter covers:

- [Palm OS Resource Summary](#)
- [Drawing on the Palm Powered Handheld](#)
- [Forms, Windows, and Dialogs](#)
- [Controls](#)
- [Fields](#)
- [Menus](#)
- [Tables](#)
- [Lists](#)
- [Categories](#)
- [Bitmaps](#)
- [Labels](#)
- [Scroll Bars](#)
- [Custom UI Objects \(Gadgets\)](#)
- [Dynamic UI](#)
- [Color and Grayscale Support](#)
- [Insertion Point](#)
- [Application Launcher](#)

User Interface

Palm OS Resource Summary

For guidelines on designing a user interface, see the book *Palm OS User Interface Guidelines*.

TIP: The Palm OS web site contains recipes for writing code to work with the various user interface objects. See the following URL: <http://www.palmos.com/dev/tech/docs/recipes>

Palm OS Resource Summary

The Palm OS development environment provides a set of resource templates that application developers use to implement the buttons, dialogs, and other UI elements. [Table 4.1](#) maps user interface elements to resources. The ResEdit name is included for developers using that tool. It is not relevant for users of Constructor for Palm OS.

All resources are discussed in detail in the book *Palm OS User Interface Guidelines*. In addition, specific design recommendations are provided for some of the elements.

Table 4.1 UI resource summary









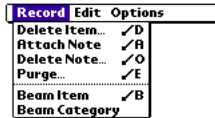



UI Element and Functionality	Example	Resource(s)
Alert— Display a warning, error, or confirmation message	 A dialog box titled "Memo Delete" with a question mark icon. The text inside says "Do you really want to delete this memo?". There are two buttons at the bottom: "OK" and "Cancel".	Alert (tAlt)
Application icon— Icon to display in Launcher	 A blue circular icon with a white telephone handset inside.	Application icon (tAIB) Application icon family (taif)
Bitmap— Display a bitmap	 A small square icon with a red border and a white background, containing a small red square and a small green square.	Form bitmap (tFBM) Bitmap (Tbmp) Bitmap family (tbmf)
Command button— Execute command	 A rectangular button with a black border and the text "OK" in the center.	Button (tBTN) Graphic button (tgbn)






Table 4.1 UI resource summary (*continued*)

UI Element and Functionality	Example	Resource(s)
Check box— Toggle on or off	<input type="checkbox"/> Show Due Dates <input checked="" type="checkbox"/> Show Priorities	Checkbox (tCBX)
Form— Window that displays other UI objects		
Gadget— Custom control		Gadget (tGDT)
Shift Indicator— Display shift status		Shift Indicator (tGSI)
Label— Display noneditable text	Set Date:	Label (tLBL)
List— Display a series of items		List (tLST)
Menu— Execute commands		Menu Bar (MBAR) Menu (MENU)
Pop-up list— Choose a setting from a list		Pop-up trigger (tPUT) Pop-up list (tPUL) List (tLST)
Push button— Select a value		Push button (tPBN) Graphic push button (tgpB)
Repeating button — Increment/decrement values or scroll		Repeating button (tREP) Graphic repeating button (tgrb)

User Interface

Drawing on the Palm Powered Handheld

Table 4.1 UI resource summary (*continued*)

UI Element and Functionality	Example	Resource(s)
Scroll bar— Scroll fields or tables		Scroll bar (tSCL)
Selector trigger — Invoke dialog that changes text of the control		Selector trigger (tSLT)
Slider— Adjust a setting		Slider (tsld) Feedback slider (tslf)
Table— Display columns of data		Table (tTBL)
Text field— Display text (single or multiple lines)		Field (tFLD)

Drawing on the Palm Powered Handheld

The first Palm Powered™ handhelds had an LCD screen of 160x160 pixels. Since then, handhelds with screens resolutions of 320 X 320 pixels have been introduced. The built-in LCD controller maps a portion of system memory to the LCD. The capabilities of the controller depend on the particular handheld, but be aware that hardware may still limit the actual displayable depths. Given that, [Table 4.2](#) lists the screen bit depths that the Palm OS supports.

Table 4.2 Supported bit depths

Palm OS Version	Supported Resolutions
1.0	1 bit/pixel
2.0	1 bit/pixel
3.0	1 or 2 bits/pixel

Table 4.2 Supported bit depths (*continued*)

Palm OS Version	Supported Resolutions
3.3	1, 2, or 4 bits/pixel
3.5	1, 2, 4 or 8 bits/pixel
4.0, 5.0	1, 2, 4, 8, or 16 bits/pixel (See “ Color and Grayscale Support ” for more information.)

Usually, the Form Manager handles all necessary drawing and redrawing to the screen when it receives certain events. (In Palm OS, a **form** is analogous to a window in a desktop application, and a **window** is an abstract drawing region.) You don’t have to explicitly call drawing routines. However, if you’re performing animation or if you have any custom user interface objects (known as gadgets), you might need to use the drawing functions provided by the Window Manager.

The Window Manager defines a window object and provides functions for drawing to windows. A window is a drawing region that is either onscreen or offscreen. The window’s data structure contains a bitmap that contains the actual data displayed in the window. Windows add clipping regions over the top of bitmaps.

The Draw State

The Window Manager also defines a draw state: pen color, pattern, graphics state, and so on. The draw state is handled differently depending on the operating system version.

On pre-3.5 versions of Palm OS, the system maintains several individual global variables that each track an element of the draw state. If you want to change some aspect of the draw state, you use a WinSet... function (such as [WinSetUnderlineMode](#)). Each WinSet... function returns the old value. It’s your responsibility to save the old value returned by the function and to restore the variable’s value when you are finished by calling the function again. Using such routines can be inconvenient because it means using application stack space to track system state. Further, if a caller

User Interface

Drawing on the Palm Powered Handheld

forgets to restore the value, the entire look and feel of the handheld may be altered.

Palm OS 3.5 and later has two improvements to make tracking changes to the draw state easier. First, it groups the drawing-related global variables and treats them as a single unit. This draw state is the Palm OS implementation of a **pen**. It contains the current transfer (or draw) mode, pattern mode and pattern data for `WinFill...` routines, and foreground and background colors. It also contains text-related drawing information: the font ID, the font pointer, the underline mode, and the text color. (Palm OS does not currently support other common pen-like concepts such as line width, pen shape, or corner join.) Only one draw state exists in the system.

Second, Palm OS 3.5 can track changes to the draw state by storing states on a stack. Your application no longer needs to use its own stack for pieces of the draw state. Instead, use [WinPushDrawState](#) to push a copy of the current draw state on the top of the stack. Then use the existing `WinSet...` functions to make your changes. When you've finished your drawing and want to restore the draw state, call [WinPopDrawState](#).

The new drawing state stack allows for additional debugging help. If an application exits without popping sufficiently or it pops too much, this is detected and flagged on debug ROMs. When switching applications, the system pops to a default state on application exit, guaranteeing a consistent draw state when a new application is launched.

Drawing Functions

The Window Manager can draw rectangles, lines, characters, bitmaps, and (starting in version 3.5) pixels. The Window Manager supports five flavors of most calls, as described in [Table 4.3](#).

Table 4.3 Window Manager drawing operations

Mode	Operation
Draw	Render object outline only, using current foreground color and pattern. For a bitmap, draws the bitmap.
Fill	Render object internals only, using current foreground color and pattern.
Erase	Erase object outline and internals to window background color.
Invert	Swap foreground and background colors within region defined by object.
Paint	Supported only in version 3.5 and higher. Render object using all of the current draw state: transfer mode, foreground and background colors, pattern, font, text color, and underline mode.

The drawing functions always draw to the current draw window. This window may be either an onscreen window or an offscreen window. Use [WinSetDrawWindow](#) to set the draw window before calling these functions.

High-Density Displays

The screens on most Palm Powered handhelds are 160 pixels wide and 160 pixels high. Prior to High-Density Display feature set, the operating system provided little support for other screen sizes. Palm OS 5, with the addition of the [High-Density Display Feature Set](#), adds support for 320 by 320, or *double-density*, screens and resources. This support is designed so that a Palm Powered handheld with a double-density screen runs, unaltered, nearly all applications designed for a single-density (160 by 160) screen.

Display Density

The density of a display refers to the ratio of the screen's width and height, in number of pixels, to the width and height of a standard 160 by 160 pixel screen. The screen's density has no relation to the

User Interface

Drawing on the Palm Powered Handheld

screen's physical size; given the form factor of the typical Palm Powered handheld screens tend to be roughly the same size regardless of the display density.

A double-density screen packs twice as many pixels in each dimension into the same physical space as a single-density screen. Regardless of the screen density, graphic primitives have the same footprint, taking up the same percentage of screen space. The Address Book application, for example, shows 11 lines of text on both single- and double-density screens. The text looks better on the double-density screen, however, because the blitter uses double-density font data when drawing the text, and each character is composed of more pixels.

NOTE: The High-Density Display feature set is designed to allow screen sizes of various densities. The blitter that is part of the Palm OS 5 reference platform, however, only supports single- and double-density displays.

When writing applications for Palm OS 5, you generally can stop thinking in terms of pixels and start thinking in terms of screen coordinates. The operating system takes care of mapping coordinates to physical pixels, and drawing functions now work in terms of coordinates. So, for example, if you draw a line of text using one of the built-in fonts, that line of text is 12 standard coordinates high. Depending on the display density, that text might be 12 pixels high, 24 pixels high (on a double-density display), or even some other multiple of 12.

Terminology

A clear understanding of the following terms is essential to understanding the concepts introduced by the High-Density Display feature set.

default density – a pixel layout with one pixel per standard coordinate.

high density – a pixel layout that uses more pixels per standard coordinate than a low-density layout.

low density – equivalent to default density; one pixel per standard coordinate.

native coordinate system – a coordinate system based on physical screen pixels. For offscreen windows, the native coordinates are based on the offscreen bitmap rather than the physical screen.

standard coordinate system – The coordinate system used by most handhelds that don't have the High-Density Display feature set installed. On a single density screen, there is one screen pixel per standard coordinate. On a high-density screen, there is more than one screen pixel per standard coordinate.

Implementation

Applications running on Palm OS 5 default to the standard coordinate system, even if the handheld has a high-density screen. When creating forms, you continue to use the standard coordinate system for form dimensions and for the placement of UI widgets.

In Palm OS 5, every drawing operation uses a draw state. Added to this draw state by the High-Density Display feature set is a scaling factor, which is used by the Window Manager to transform coordinates passed as arguments to the drawing functions (such as [WinDrawLine](#), [WinCopyRectangle](#), and [WinPaintPixel](#)) into native coordinates. This scaling factor is a ratio of the active coordinates to native coordinates.

Drawing is a function of the Window Manager, which initializes the graphic state and then calls the blitter—the low-level code that draws lines and rectangles and places all primitives at the appropriate location on the screen. The Window Manager converts coordinates passed as drawing function arguments from the window's coordinate system to the native coordinate system used by the blitter. Because the blitter needs with integer coordinates, most of the `WinScale...` and `WinUnscale...` functions have a `ceiling` parameter that lets you control whether the integer results are obtained by truncation or rounding. [WinScaleRectangle](#) and [WinUnscaleRectangle](#) are the exceptions to this rule: the calculated extent is always rounded up if either the extent or the top left coordinate has a fractional part after scaling.

Using [WinSetCoordinateSystem](#), an application can define the coordinate system used by its calls to the drawing functions. On a

User Interface

Drawing on the Palm Powered Handheld

handheld with a high-density screen, this allows applications to draw using either the standard coordinate system or the native coordinate system. Note that the `bounds` and `clippingBounds` fields in the [WindowType](#) data structure are always stored using native coordinates. The various functions that access these fields convert the native coordinates to the coordinate system being used by the window.

Which coordinate system a window uses affects the placement and dimensions of graphic primitives. It does not affect bitmap contents, however. You can create bitmaps that contain either low- or high-density bitmap data; the bitmap's density is recorded in the [BitmapTypeV3](#) data structure. The Window Manager uses the window's coordinate system to determine where to place the top left corner of the bitmap on the screen, while the blitter uses the bitmap structure's `density` field to determine if it needs to stretch or shrink the bitmap data as it blits.

As an example, suppose you have an application that draws a low-density bitmap containing data 30 pixels wide and 70 pixels high on a Palm Powered handheld with a double-density screen. Using the standard coordinate system, the application instructs the Window Manager to place the bitmap at window coordinates (10,20). The Window Manager converts (10,20) to native coordinates and instructs the blitter to draw the bitmap at native coordinates (20,40). Recognizing that the bitmap is low density, the blitter pixel-doubles the source data as it is blitted. The result is that on the double-density screen, the bitmap is displayed with 60 pixels per row and contains 140 rows.

Regardless of the coordinate system used for the placement and dimensions of graphic primitives, no new functions are needed to take advantage of high-density fonts. High-density fonts are used by default when drawing text in a high-density window.

Maintaining Compatibility

The [High-Density Display Feature Set](#) is designed to ensure the greatest degree of compatibility with applications that weren't written using the High-Density Display feature set. When running in low-density mode on a handheld with a high-density screen, the window's `scale` attribute is set to the ratio of the handheld's screen density to the default density. This causes the Window Manager to

scale the low-density coordinates used to position graphic primitives into high-density ones used by the blitter. Because low-density mode is the default on all Palm Powered handhelds, applications not designed for high-density screens behave as expected on handhelds with screens of both low and high-density.

Offscreen windows are allocated by default with low-density bitmaps, so direct manipulation of offscreen bitmaps by applications unaware of the High-Density Displays feature set works consistently on handhelds with either low- high-density screens.

On the other hand, applications that employ the High-Density Display feature set need to include both low-density and high-density bitmaps in order to function consistently on handhelds that don't have high-density displays.

Some Sony CLIE™ handhelds have a double-density screen but don't have the High-Density Display feature set. The High-Density Display feature set recognizes bitmaps created for these handhelds and properly displays them as double-density bitmaps.

The one area of incompatibility involves applications that directly access the handheld's screen. **Applications not designed for a high-density screen that directly access the screen give unexpected results when accessing a high-density screen.** For example, if such an application directly manipulates the screen pixels, expecting a 160 by 160 screen, modifying pixel 161 on a double-density screen modifies a pixel in the middle of the first row, not the first pixel on the second row. As well, **if the handheld's processor is ARM-based, improper drawing can also result due to differences in endianness between ARM-based platforms and those based upon a 68k-family processor.**

Text

By default, text is always drawn at the best possible density, even for applications unaware of a handheld's high-density capability. Because of this, handhelds contain system fonts that match the density of the screen. The high-density font metrics match those of the low-density system fonts.

User Interface

Drawing on the Palm Powered Handheld

Because the system fonts match the screen density, the blitter does not need to perform scaling when blitting text to the screen. If an application running on a handheld with a high-density screen allocates a low-density offscreen window, however, and there are no low-density fonts available, the blitter shrinks the high-density system font bitmaps. This results in poor quality text when the offscreen bitmap is subsequently transferred, and pixel-doubled, to a high-density display. If a low-density font is available, the blitter substitutes a low-density font when drawing text to the window in an attempt to produce the best possible aesthetic result.

The blitter uses the following selection algorithm when selecting the font, from high to low priority:

1. Select the font with the correct density.
2. Select the font whose density is one-half of the correct density.
3. Select the font with the closest density, with a tie going to the lower-density font.

The Font Manager uses the `stdToActiveScale` field in the offscreen window's draw state to transform the font metrics. To draw text using high-density coordinates, set the high-density coordinate system by calling [WinSetCoordinateSystem](#) before using the Font Manager functions to position text or extract font metrics.

On a high-density screen, underline mode is always drawn using a high-density pattern.

Lines and Rectangles

When drawing lines and rectangles with the standard coordinate system on a double-density screen, the primitives are drawn with improved resolution. This behavior prevents an inconsistent appearance when drawing to and from offscreen windows, and prevents unintended overlap and unintended gaps between primitives.

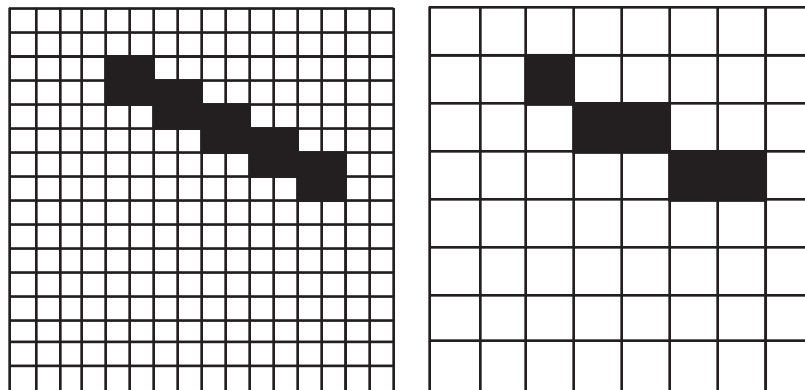
As with other primitives, the Window Manager performs the conversion to the destination coordinate system before calling the blitter. This converts line coordinates as well as a rectangle's `topLeft` and `extent` fields.

The diagram on the left in [Figure 4.1](#) results from drawing a diagonal line from (2,1) to (6,3) in the screen's standard coordinate system. The diagram on the right shows the same line drawn in the screen's native coordinate system with the following code:

```
WinPushDrawState();
oldScale = WinSetCoordinateSystem(kCoordinatesNative);
WinDrawLine(2, 1, 6, 3); // x1, y1, x2, y2
WinPopDrawState();
```

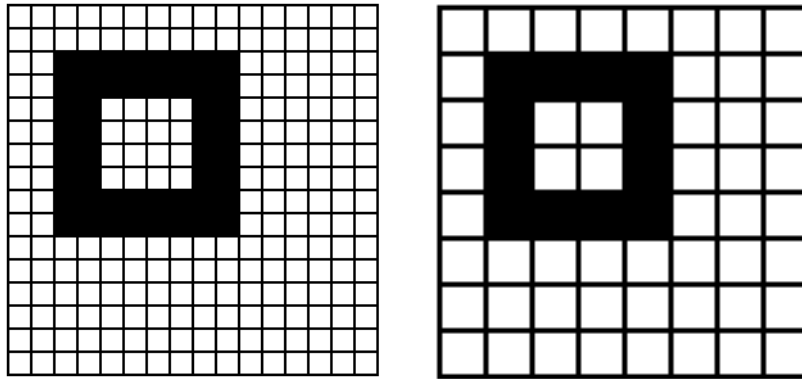
NOTE: In these illustrations, the top-left grid coordinate is the screen origin (0,0) on a double-density screen.

Figure 4.1 A diagonal line drawn using the standard coordinate system (left) and the native coordinate system (right)



The result of drawing a rectangle with a `topLeft` of (1,1) and an `extent` of (4,4) when using the standard and native coordinate systems on a double-density screen is shown in [Figure 4.2](#).

Figure 4.2 A rectangle drawn using the standard coordinate system (left) and the native coordinate system (right)



A rounded rectangle would be pixel-doubled in a similar fashion when drawn using the standard coordinate system on a double-density screen. In double-density mode, the rounded corners are drawn in the native double-density coordinates, resulting in more detailed corners.

Patterns

In prior versions of the Palm OS, patterns are 8 by 8 bits, and are 1 bit deep. To support high-density patterns, a new Window Manager function, [WinPaintTiledBitmap](#), gives applications the ability to fill a rectangle with an arbitrary pattern defined in a bitmap argument.

Patterns are expanded to the destination bit depth by the blitter when drawing patterned lines and filled rectangles. The blitter uses the density fields in the pattern's source bitmap and the destination bitmap so that the pattern is drawn using the appropriate density. This makes it possible for an application to define both low-density and high-density patterns.

To supplement the standard [PatternType](#) grayPattern, the High-Density Display feature set defines two additional gray patterns: `lightGrayPattern` and `darkGrayPattern`. These patterns are shown in [Figure 4.3](#) and [Figure 4.4](#), respectively.

Figure 4.3 lightGrayPattern

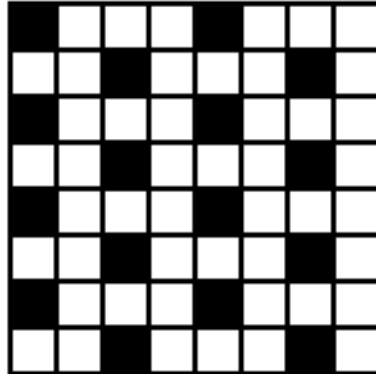
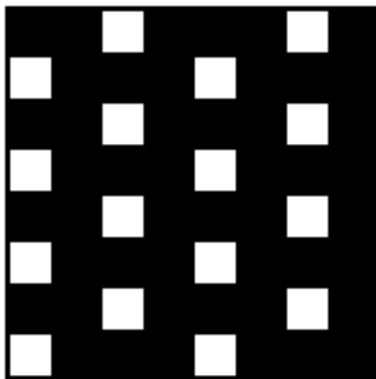


Figure 4.4 darkGrayPattern



These three standard gray patterns are always drawn by the blitter using the screen density, improving the appearance of gray fills. Custom 8 by 8 patterns, however, are stretched as appropriate by the blitter based on the ratio of the destination density and `kDensityLow`.

Forms, Windows, and Dialogs

A form is the GUI area for each view of your application. For example the Address Book offers an Address List view, Address Edit view, and so on. Each application has to have one form, and most applications have more than one. To actually create the view, you have to add other UI elements to the form; either by dragging

User Interface

Forms, Windows, and Dialogs

them onto the form from the catalog or by providing their ID as the value of some of the form's fields.

[Figure 4.5](#) shows an example of a form. Typical forms are as large as the screen, as shown here. Other forms are modal dialogs, which are shorter than the screen but just as wide.

Figure 4.5 Form



A window defines a drawing region. This region may be on the display or in a memory buffer (an off-screen window). Off-screen windows are useful for saving and restoring regions of the display that are obscured by other UI objects. All forms are windows, but not all windows are forms.

The window object is the portion of the form object that determines how the form's window looks and behaves. A window object contains viewing coordinates of the window and clipping bounds.

When a form is opened, a [frmOpenEvent](#) is triggered and the form's ID is stored. A [winExitEvent](#) is triggered whenever a form is closed, and a [winEnterEvent](#) is triggered whenever a form is drawn.

The following sections describe special types of forms:

- [Alert Dialogs](#)
- [Progress Dialogs](#)
- [The Keyboard Dialog](#)

Alert Dialogs

If you want to display an alert dialog (see [Figure 4.6](#)) or prompt the user for a response to a question, use the alert manager. The alert manager defines the following functions:

- [FrmAlert](#)
- [FrmCustomAlert](#)

Figure 4.6 Alert dialog



Given a resource ID that defines an alert, the alert manager creates and displays a modal dialog box. When the user taps one of the buttons in the dialog, the alert manager disposes of the dialog box and returns to the caller the item number of the button the user tapped.

There are four types of system-defined alerts:

- Question
- Warning
- Notification
- Error

The alert type determines which icon is drawn in the alert window and which sound plays when the alert is displayed.

When the alert manager is invoked, it's passed an alert resource that contains the following information:

- The rectangle that specifies the size and position of the alert window
- The alert type (question, warning, notification, or error)
- The null-terminated text string; that is, the message the alert displays
- The text labels for one or more buttons

Progress Dialogs

If your application performs a lengthy process, such as data transfer during a communications session, consider displaying a progress dialog to inform the user of the status of the process. The Progress Manager provides the mechanism to display progress dialogs.

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

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

The dialog can display a few lines of text that are automatically centered and formatted. You can also specify an icon that identifies the operation in progress. The dialog has one optional button that can be a cancel or an OK button. The type of the button is automatically controlled by the Progress Manager and depends on the current progress state (no error, error, or user canceled operation).

Progress `textCallback` Function

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

The system passes the `textCallback` function one parameter, a pointer to a `PrgCallbackData` structure. To learn what type of information is passed in this structure, see the chapter “[Progress Manager](#)” in the *Palm OS Programmer’s API Reference*.

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

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

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

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

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

The Keyboard Dialog

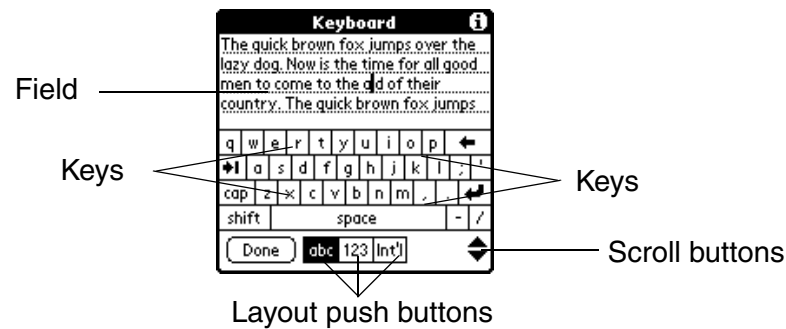
The keyboard dialog is an onscreen keyboard on which the user taps to input information into a text field. When the insertion point is in a text field, the user can open the onscreen keyboard by tapping on

User Interface

Forms, Windows, and Dialogs

the silk-screen letters (“abc” or “123”) in the lower corners of the input area. The keyboard dialog appears:

Figure 4.7 The Keyboard Dialog



The keyboard dialog’s text field contains the full text of the original field, with the insertion point in the same position as in the original field. Users can start inserting and deleting characters immediately, or they can scroll up or down and then insert and delete. When one of the software keys is tapped, the corresponding character is inserted in the text field of the keyboard dialog.

As the user taps, the keyboard dialog code captures pen events, maps them to characters, and posts `keyDownEvents`. The text field in the keyboard dialog handles each [keyDownEvent](#), displaying the character onscreen and inserting it into the memory chunk associated with the field. Since the keyboard dialog has its own event loop, you cannot handle the key events yourself. If you need to capture the key events, you should consider creating a custom version of the onscreen keyboard dialog, as outlined in “[Creating a Custom Keyboard Layout](#)” on page 90.

The keyboard code edits the text string of the original field in-place. Each field has a text handle that points to the memory chunk containing the text string for the field. When the keyboard dialog is opened, the association between the text handle and the original field is removed. The text handle is then assigned to the text field of the keyboard dialog and edited according to user input. When the keyboard dialog is closed, the text handle is reassigned to the original field.

For details on how [FldHandleEvent](#) manipulates the memory chunk that holds a field's text string, see [Chapter 9](#), “[Fields](#),” in the *Palm OS Programmer's API Reference*. Read about the [FldGetTextHandle](#) and [FldSetText](#) functions.

Opening the Keyboard Dialog Programmatically

In most applications, the keyboard dialog appears only when explicitly opened by the user. There are occasions, however, when you may wish to force the user to input characters via the onscreen keyboard. For example, the service activation application shipped with all Palm VIs displays the keyboard dialog automatically. Palm made this choice because the activation application must be usable by completely new Palm users, who may not know how to write Graffiti® or Graffiti 2 or open the keyboard dialog themselves. Other reasons for imposing the keyboard dialog include accurate input of passwords or account numbers.

To display the keyboard dialog programmatically, use one of the following functions:

- [SysKeyboardDialog](#)
- [SysKeyboardDialogV10](#)

Normally, use `SysKeyboardDialog` only.

`SysKeyboardDialogV10` is for compatibility with Palm OS 1.0.

See `Keyboard.h` for the function prototypes and the `KeyboardType` they use. Note that the rest of the functions listed in `Keyboard.h` are for system use only and do not form part of the Palm OS API.

Keyboard Layouts

The keyboard dialog has three views, one for each of the pre-defined layouts: the English alphabet, numerals and punctuation, and Latin characters with diacritic marks. The default is the English alphabet. To display a particular layout, call the `SysKeyboardDialog` function and pass it one of the following constants, which are defined in an enumeration named `KeyboardType`:

Table 4.4 Constants defined in KeyboardType

Constant	Character Set
<code>kbdAlpha</code>	The English language character set.
<code>kbdNumbersAndPunc</code>	A set containing numbers and some advanced punctuation.
<code>kbdAccent</code>	The International character set, made up of Latin characters with diacritic marks.
<code>kbdDefault</code>	The value of <code>kbdDefault</code> is the same as <code>kdbAlpha</code> and cannot be changed.

Creating a Custom Keyboard Layout

You cannot add an extra keyboard layout or modify an existing one. You can, however, create your own keyboard dialog module that implements the functionality outlined below.

First, your application should intercept the [keyDownEvent](#) generated when the user taps the “abc” or “123” letters in the input area. Create a custom keyboard dialog loader routine to handle it. Your keyboard code should then do the following:

- Get the text handle of the original field and save it in a variable. Use [FldGetTextHandle](#).
- Remove the association between the text handle and the original field. Use [FldSetTextHandle](#) or [FldSetText](#), passing NULL as the second argument.
- Assign the text handle to the text field of the keyboard dialog.
- Define a Keyboard event handler that:
 - captures pen events in your onscreen keyboard region, which may be a bitmap of a keyboard or may consist of individual push buttons,
 - maps pen events to characters,

- creates `keyDownEvents` and posts them to the event queue so that the dialog's text field can automatically handle them and insert them in its text chunk.
- When the dialog is closed, remove the association between the text handle and keyboard's field, and then re-assign the text handle to the original text field.

Finally, if you wish more than one layout, your custom keyboard dialog must contain a button to open each layout.

Offscreen Windows

Offscreen windows are generally used for one of two reasons: to do offscreen drawing (for double-buffering, smooth animations, or to reduce flicker) or so that the application can capture or import graphics.

[`WinCreateOffscreenWindow`](#) allocates a bitmap for the offscreen window. Unless you specify a format of `nativeFormat`, the offscreen window's bitmap is always low density. This allows applications that expect low-density bitmaps, and that directly manipulate bitmap data, to still function as expected. If you call `WinCreateOffscreenWindow` and specify a format of `nativeFormat`, *do not access the data in the offscreen window's bitmap directly*: the format of bitmaps created by Palm OS can change from release to release, from device to device, and may even differ on a single device depending on the screen depth or compatibility mode.

Functions that return window dimensions—such as [`WinScreenMode`](#), [`WinGetBounds`](#), and [`WinGetDrawWindowBounds`](#)—use the window's scaling factor to return coordinates in terms of the active coordinate system. This ensures that the window has the expected dimensions and that graphic primitives have coordinates expected by the application.

[`WinCreateBitmapWindow`](#) gives you the ability to allocate a high-density bitmap for an offscreen window. Use [`BmpCreate`](#) and [`BmpSetDensity`](#) to allocate a high-density bitmap, then associate it with a window by calling `WinCreateBitmapWindow`.

Controls

Control objects allow for user interaction when you add them to the forms in your application. Events in control objects are handled by [CtlHandleEvent](#). There are several types of control objects, which are all described in this section.

NOTE: Palm OS 3.5 and later support graphical controls for all control types other than check box. Graphical controls behave the same as their non-graphical counterparts, but they display a bitmap instead of a text label. On releases prior to Palm OS 3.5, you can create a graphical control by setting the text label to the empty string and placing the control on top of a bitmap.

Buttons

Buttons (see [Figure 4.8](#)) display a text or graphic label in a box. The default style for a button is a text string centered within a rounded rectangle. Buttons have rounded corners unless a rectangular frame is specified. A button without a frame inverts a rectangular region when pressed.

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

[Table 4.5](#) shows the system events generated when the user interacts with the button and `CtlHandleEvent`'s response to the events.

Figure 4.8 Buttons



Table 4.5 Event flow for buttons

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

Pop-Up Trigger

A pop-up trigger (see [Figure 4.9](#)) displays a text label and a graphic element (always on the left) that signifies the control initiates a pop-up list. If the text label changes, the width of the control expands or contracts to the width of the new label plus the graphic element.

[Table 4.6](#) shows the system events generated when the user interacts with the pop-up trigger and CtlHandleEvent's response to the events. Because pop-up triggers are used to display list objects, also see the section "[Lists](#)" in this chapter.

Figure 4.9 Pop-up trigger

▼ Work

Table 4.6 Event flow for pop-up triggers

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

TIP: To create a pop-up list in Constructor for Palm OS, add a pop-up trigger to your form, then add a list at the same coordinates, uncheck the usable check box in the list resource settings, and then set the List ID in the pop-up trigger to match the ID of the list resource.

Selector Trigger

A selector trigger (see [Figure 4.10](#)) displays a text label surrounded by a gray rectangular frame. If the text label changes, the width of the control expands or contracts to the width of the new label.

[Table 4.7](#) shows the system events generated when the user interacts with the selector trigger and `CtlHandleEvent`'s response to the events.

Figure 4.10 Selector trigger

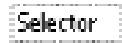


Table 4.7 Event flow for selector triggers

User Action	System Response	CtlHandleEvent Response
Pen goes down on a selector trigger.	penDownEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>ctlEnterEvent</code> to the event queue.
	ctlEnterEvent with selector trigger's ID number.	Inverts the button's display.
Pen is lifted from the selector trigger.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>ctlSelectEvent</code> to the event queue.
	ctlSelectEvent with selector trigger's ID number.	Adds a frmOpenEvent followed by a winExitEvent to the event queue. Control is passed to the form object.

Repeating Button

A repeat control looks like a button. In contrast to buttons, however, users can repeatedly select repeat controls if they don't lift the pen when the control has been selected. The object is selected repeatedly until the pen is lifted.

[Table 4.8](#) shows the system events generated when the user interacts with the repeating button and `CtlHandleEvent`'s response to the events.

User Interface

Controls

Table 4.8 Event flow for repeating buttons

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

Push Buttons

Push buttons (see [Figure 4.11](#)) look like buttons, but the frame always has square corners. Touching a push button with the pen inverts the bounds. If the pen is released within the bounds, the button remains inverted.

[Table 4.9](#) shows the system events generated when the user interacts with the push button and CtlHandleEvent's response to the events.

Figure 4.11 Push buttons

Priority: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Sort by: ☐ Priority ☐ Due Date

Table 4.9 Event flow for push buttons

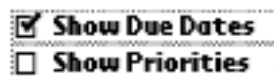
User Action	System Response	CtlHandleEvent Response
Pen goes down on a push button.	penDownEvent with the x and y coordinates stored in EventType.	Adds the ctlEnterEvent to the event queue.
	ctlEnterEvent with push button's ID number.	If push button is grouped and highlighted, no change. If push button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

Check Boxes

Check boxes (see [Figure 4.12](#)) display a setting, either on (checked) or off (unchecked). Touching a check box with the pen toggles the setting. The check box appears as a square, which contains a check mark if the check box's setting is on. A check box can have a text label attached to it; selecting the label also toggles the check box.

[Table 4.10](#) shows the system events generated when the user interacts with the check box and CtlHandleEvent's response to the events.

Figure 4.12 Check boxes



User Interface

Controls

Table 4.10 Event flow for check boxes

User Action	Event Generated	CtlHandleEvent Response
Pen goes down on check box.	penDownEvent with the x and y coordinates stored in EventType.	Adds the ctlEnterEvent to the event queue.
	ctlEnterEvent with check box's ID number.	Tracks the pen until the user lifts it.
Pen is lifted from check box.	penUpEvent with the x and y coordinates stored in EventType.	<ul style="list-style-type: none">• If the check box is unchecked, a check appears.• If the check box is already checked and is grouped, there is no change in appearance.• If the check box is already checked and is ungrouped, the check disappears. Adds the ctlSelectEvent to the event queue.
Pen is lifted outside box.	penUpEvent with the x and y coordinates stored in EventType.	Adds the ctlExitEvent to the event queue.

Sliders and Feedback Sliders

Starting in Palm OS 3.5, slider controls (see [Figure 4.13](#)) are supported. Sliders represent a value that falls within a particular range. For example, a slider might represent a value that can be between 0 and 10.

Figure 4.13 Slider



There are four attributes that are unique to slider controls:

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

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

[Table 4.11](#) shows the system events generated when the user interfaces with a slider and how `CtlHandleEvent` responds to the events.

Table 4.11 Event flow for sliders

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

User Interface

Controls

Table 4.11 Event flow for sliders (*continued*)

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

[Table 4.12](#) shows the system events generated when the user interacts with a feedback slider and CtlHandleEvent's response to the events.

Table 4.12 Event flow for feedback sliders

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

Table 4.12 Event flow for feedback sliders (*continued*)

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

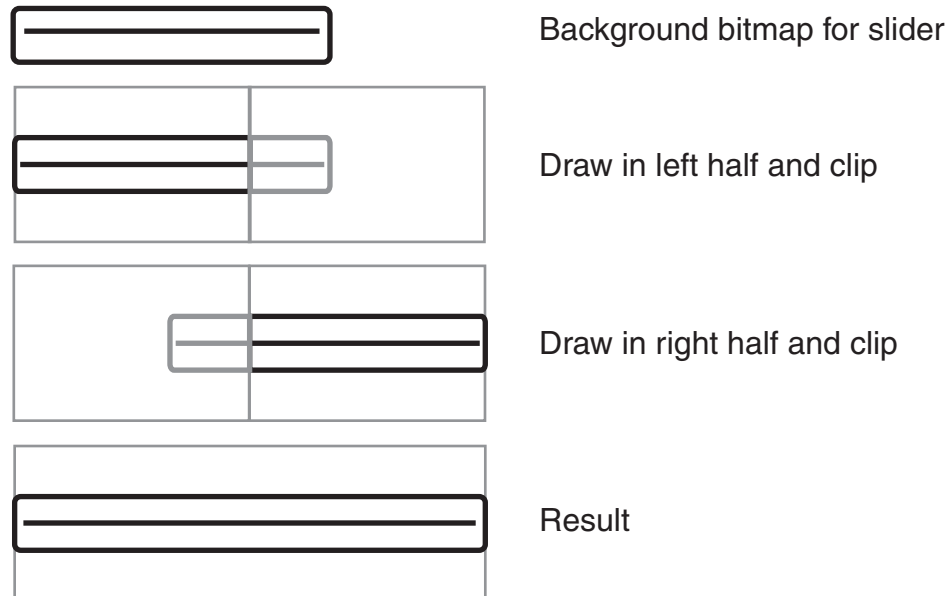
User Interface

Controls

Sliders are drawn using two bitmaps: one for the slider background, and the other for the thumb. You may use the default bitmaps to draw sliders, or you may specify your own bitmaps when you create the slider.

The background bitmap you provide can be smaller than the slider's bounding rectangle. This allows you to provide one bitmap for sliders of several different sizes. If the background bitmap isn't as tall as the slider's bounding rectangle, it's vertically centered in the rectangle. If the bitmap isn't as wide as the slider's bounding rectangle, the bitmap is drawn twice. First, it's drawn left-justified in the left half of the bounding rectangle and clipped to exactly half of the rectangle's width. Then, it's drawn right-justified in the right half of the bounding rectangle and clipped to exactly half of the rectangle's width. (See [Figure 4.14](#).) Note that this means that the bitmap you provide must be at least half the width of the bounding rectangle.

Figure 4.14 Drawing a slider background



Fields

A field object displays one or more lines of text. [Figure 4.15](#) is an underlined, left-justified field containing data.

Figure 4.15 Field



Look Up: Text.....

The field object supports these features:

- Proportional fonts (only one font per field)
- Drag-selection
- Scrolling for multiline fields
- Cut, copy, and paste
- Left and right text justification
- Tab stops
- Editable/noneditable attribute
- Expandable field height (the height of the field expands as more text is entered)
- Underlined text (each line of the field is underlined)
- Maximum character limit (the field stops accepting characters when the maximum is reached)
- Special keys (Graffiti and Graffiti 2 strokes) to support cut, copy, and paste
- Insertion point positioning with pen (the insertion point is positioned by touching the pen between characters)
- Scroll bars

The field object does **not** support overstrike input mode; horizontal scrolling; numeric formatting; or special keys for page up, page down, left word, right word, home, end, left margin, right margin, and backspace. On Palm OS versions earlier than 3.5, the field object also does not support word selection. Starting in version 3.5, double-tapping a word selects that word, and triple-tapping selects the entire line.

User Interface

Fields

NOTE: Field objects can handle line feeds—\0A—but not carriage returns—\0D. PalmRez translates any carriage returns it finds in any Palm OS resources into line feeds, but doesn't touch static data.

Events in field objects are handled by [FldHandleEvent](#). [Table 4.13](#) provides an overview of how `FldHandleEvent` deals with the different events

Table 4.13 Event flow for fields

User Action	Event Generated	FldHandleEvent Response
Pen goes down on a field.	penDownEvent with the x and y coordinates stored in <code>EventType</code> . fldEnterEvent with the field's ID number.	Adds the <code>fldEnterEvent</code> to the event queue. Sets the insertion point position to the position of the pen and tracks the pen until it is released. Drag-selection and drag-scrolling are supported. Starting in Palm OS 3.5, double-tapping in a field selects the word at that location, and triple-tapping selects the line.
Pen is lifted.	penUpEvent with the x and y coordinates.	Nothing happens; a field remains selected until another field is selected or the form that contains the field is closed.
Enters characters into selected field.	keyDownEvent with character value in <code>EventType</code> .	Character added to field's text string.
Presses up arrow key	keyDownEvent	Moves insertion point up a line.

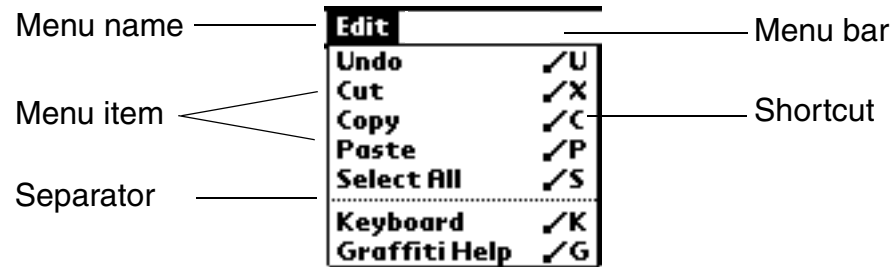
Table 4.13 Event flow for fields (*continued*)

User Action	Event Generated	FldHandleEvent Response
Presses down arrow	keyDownEvent	Moves insertion point down a line; the insertion point doesn't move beyond the last line that contains text.
Presses left arrow	keyDownEvent	Moves insertion point one character position to the left. When the left margin is reached, move to the end of the previous line.
Presses right arrow	keyDownEvent	Moves insertion point one character position to the right. When the right margin is reached, move to the start of the next line.
Cut command	keyDownEvent	Cuts the current selection to the text clipboard.
Copy command	keyDownEvent	Copies the current selection to the text clipboard.
Paste command	keyDownEvent	Inserts clipboard text into the field at insertion point.

Menus

A menu bar is displayed whenever the user taps a menu icon. Starting in Palm OS 3.5, the menu bar is also displayed when the user taps in a form's titlebar. The menu bar, a horizontal list of menu titles, appears at the top of the screen in its own window, above all application windows. Pressing a menu title highlights the title and "pulls down" the menu below the title (see [Figure 4.16](#)).

Figure 4.16 Menu



User actions have the following effect on a menu:

When...	Then...
User drags the pen through the menu	Command under the pen is highlighted
Pen is released over a menu item	That item is selected and the menu bar and menu disappear
Pen is released outside both the menu bar and the menu	Both menu and menu bar disappear and no selection is made
Pen is released in a menu title (Palm OS 3.5 and later only)	Menu bar and Menu remain displayed until a selection is made from the menu.
Pen is tapped outside menu and menu bar	Both menu and menu bar are dismissed

A menu has the following features:

- Item separators, which are lines to group menu items.
- Menu shortcuts; the shortcut labels are right justified in menu items.
- A menu remembers its last selection; the next time a menu is displayed the prior selection appears highlighted.
- The bits behind the menu bar and the menus are saved and restored by the menu routines.

- When the menu is visible, the insertion point is turned off.

Menu events are handled by [MenuHandleEvent](#). [Table 4.14](#) describes how user actions get translated into events and what `MenuHandleEvent` does in response.

Table 4.14 Event flow for menus

User Action	Event Generated	MenuHandleEvent Response
Pen enters menu bar.	winEnterEvent identifying menu's window.	Tracks the pen.
User selects a menu item.	penUpEvent with the x and y coordinates.	Adds a menuEvent with the item's ID to the event queue.

Checking Menu Visibility

When the operating system draws a menu, the menu's window becomes the active drawing window. The operating system generates a [winExitEvent](#) for the previous active drawing window and a [winEnterEvent](#) for the menu's window. When the menu is erased, the system generates a `winExitEvent` for the menu's window and a `winEnterEvent` for the window that was active before the menu was drawn.

It's common to want to check if the menu is visible in applications that perform custom drawing to a window. Such applications want to make sure that they don't draw on top of the menu. The recommended way to do this is to stop drawing when you receive a `winExitEvent` matching your drawing window and resume drawing when you receive the corresponding `winEnterEvent`. For example, the following code is excerpted from the Reptoids example application's main event loop:

```

EvtGetEvent (&event, TimeUntillNextPeriod());

if (event.eType == winExitEvent) {
    if (event.data.winExit.exitWindow ==
        (WinHandle) FrmGetFormPtr(MainView)) {
        // stop drawing.
    }
}

```

User Interface

Menus

```
else if (event.eType == winEnterEvent) {
    if (event.data.winEnter.enterWindow ==
        (WinHandle) FrmGetFormPtr(MainView) &&
        event.data.winEnter.enterWindow ==
        (WinHandle) FrmGetFirstForm ()) {
        // start drawing
    }
}
```

Note that this technique is not specific to menus—your application should stop drawing if any window obscures your drawing window, and it will do so if you check for `winEnterEvent` and `winExitEvent`.

Dynamic Menus

In releases of Palm OS prior to release 3.5, the menu was loaded from a menu resource (created with Constructor or some other tool) and could not be modified in code. Starting in Palm OS 3.5, you can add, hide, or unhide menu items while the menu resource is being loaded.

A [menuOpenEvent](#) is sent when the menu resource is loaded. (Note that this event is new in version 3.5. Previous releases do not use it.) In response to this event, you can call [MenuAddItem](#) to add a menu item to one of the pull-down menus, [MenuHideItem](#) to hide a menu item, or [MenuShowItem](#) to display a menu item.

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

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

Suppose a user selects your application's About item from the Options menu then clicks the OK button to return to the main form.

When the About dialog is displayed, it becomes the active form, which causes the main form's menu state to be erased. This menu state is not restored when the main form becomes active again. The next time the user requests the menu, the menu resource is reloaded, and a new `menuOpenEvent` is queued.

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

Menu Shortcuts

As an alternative to selecting a menu command through the user interface, users can instead enter a menu shortcut. This support is present in all versions of the Palm OS, but it was extended in Palm OS 3.5.

On all versions of Palm OS, the user can enter a Graffiti or Graffiti 2 command keystroke followed by another character. If the next character matches one of the shortcut characters for an item on the active menu, a [menuEvent](#) with that menu item is generated. To support this behavior, you simply specify a shortcut character when you create a menu item resource. The default behavior of Palm OS handles this shortcut appropriately.

NOTE: If the [Graffiti 2 Feature Set](#) is present, avoid using a multi-stroke character as a keyboard shortcut if its first stroke matches the stroke of another letter (such as the letter "l" or the space character).

Starting in Palm OS 3.5, drawing the command stroke displays the command toolbar (see [Figure 4.17](#)). This toolbar is the width of the screen. (Previous versions of Palm OS simply display the string "Command:" in the lower-left portion of the screen.) The command toolbar displays a status message on the left and buttons on the right. After drawing the command stroke, the user has the choice of writing a character or of tapping one of the buttons on the command toolbar. Both of these actions cause the status message to be briefly displayed and (in most cases) a `menuEvent` to be added to the event queue.

Figure 4.17 Command toolbar



The buttons displayed on the toolbar depend on the user context. If the focus is in an editable field, the Field Manager displays buttons for cut, copy, and paste on the command toolbar. If there is an action to undo, the field manager also displays a button for undo.

The active application may also add its own buttons to the toolbar. To do so, respond to the [menuCmdBarOpenEvent](#) and use [MenuCmdBarAddButton](#) to add the button. [Listing 4.1](#) shows some code from the Memo application that adds to the command toolbar a button that displays the security dialog and then prevents the field manager from adding other buttons.

Listing 4.1 Responding to menuCmdBarOpenEvent

```
else if (event->eType == menuCmdBarOpenEvent) {  
  
    MenuCmdBarAddButton(menuCmdBarOnLeft,  
        BarSecureBitmap, menuCmdBarResultMenuItem,  
        ListOptionsSecurityCmd, 0);  
  
    // Tell the field package to not add buttons  
    // automatically; we've done it all ourselves.  
    event->data.menuCmdBarOpen.preventFieldButtons =  
        true;  
  
    // Don't set handled to true; this event must  
    // fall through to the system.  
}
```

The system contains bitmaps that represent such commands as beaming and deleting records. If your application performs any of these actions, it should use the system bitmap. [Table 4.15](#) shows the system bitmaps and the commands they represent. If you use any of these, you should use them in the order shown, from right to left. That is, `BarDeleteBitmap` should always be the rightmost of these bitmaps, and `BarInfoBitmap` should always be the leftmost.

Table 4.15 System command toolbar bitmaps

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

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

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

Tables

Tables support multi-column displays. Examples are:

- the List view of the ToDo application
- the Day view in the Datebook

The table object is used to organize several types of UI objects. The number of rows and the number of columns must be specified for each table object. A UI object can be placed inside a cell of a table.

Tables often consist of rows or columns of the same object. For example, a table might have one column of labels and another column of fields. Tables can only be scrolled vertically. Tables can't include bitmaps.

A problem may arise if non-text elements are used in the table. For example, assume you have a table with two columns. In the first column is an icon that displays information, the second column is a text column. The table only allows users to select elements in the first column that are as high as one row of text. If the icon is larger, only a narrow strip at the top of the column can be selected.

Table Event

The table object generates the event [tblSelectEvent](#). This event contains:

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

When [tblSelectEvent](#) is sent to a table, the table generates an event to handle any possible events within the item's UI object.

Lists

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

Figure 4.18 List



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

- the time list in the time edit window of the datebook
- the Category pop-up list (see “[Categories](#)” in this chapter)

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

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

When the pen is released over an item, that item becomes the current selection. When the pen is dragged outside the list, the item that was highlighted before the [penDownEvent](#) is highlighted again if it’s visible. If it’s not, no item is highlighted.

An application can use a list in two ways:

- Initialize a structure with all data for all entries in the list and let the list manage its own data.
- Provide list drawing functions but don’t keep any data in memory. The list picks up the data as it’s drawing.

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

The [LstHandleEvent](#) function handles list events. [Table 4.16](#) provides an overview of how `LstHandleEvent` deals with the different events.

User Interface

Lists

Table 4.16 Event flow for lists

User Action	System Response	LstHandleEvent Response
Pen goes down on pop-up trigger button.	winEnterEvent identifying list's window.	Adds the <code>lstEnterEvent</code> to the event queue.
	lstEnterEvent with list's ID number and selected item.	Tracks the pen.
Pen goes down on a list box.	penDownEvent with the x and y coordinates stored in <code>EventType</code> .	Highlights the selection underneath the pen.
Pen is lifted from the list box.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>lstSelectEvent</code> to the event queue.
	lstSelectEvent with list's ID number and number of selected item.	Stores the new selection. If the list is associated with a pop-up trigger, adds a popSelectEvent to the event queue. with the pop-up trigger ID, the pop-up list ID, and the item number selected in <code>EventType</code> . Control passes to <code>FrmHandleEvent</code> .
Pen is lifted outside the list box.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds winExitEvent to event queue.

Using Lists in Place of Tables

Lists really consist of single-column rows of text, but it is possible to imitate a multi-column display if you provide a custom list drawing function. Many programmers choose to use list objects instead of tables for multi-column displays because lists are generally easier to program than tables are. Doing so is acceptable, but it is somewhat problematic because the list object always displays a rectangular border around the list. If you choose to use lists to display multi-

column data that would normally be displayed in a table, you must suppress the drawing of the list border. The safest way to do so is to set the draw window's clipping rectangle to the bounds of the list before drawing the list, as shown in [Listing 4.2](#). See the *Palm OS User Interface Guidelines* for more information.

Listing 4.2 Suppressing the list border

```
void DrawFormWithNoListBorder(FormType *frmP,
UInt16 listIndex)
{
    RectangleType *clip;
    RectangleType *newClip;
    ListType *listP = FrmGetObjectPtr(frmP, listIndex);

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

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

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

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

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

```
        case frmUpdateEvent:
            frmP = FrmGetActiveForm();
            listIndex = FrmGetObjectIndex(frmP, MyListRscID);
            DrawFormWithNoListBorder(frmP, listIndex);
            handled = true;
            break;
        ...
    }
}
```

Categories

Categories allow you to group records logically into manageable lists. In the user interface, categories typically appear in a pop-up list in a form's titlebar and in dialogs that allow you to edit a single database record.

You create a category pop-up list the same way you create any other pop-up list: create the list resource, create the pop-up trigger control resource with a width of 0, and set the trigger's list ID to be the ID of the list. You manage the category pop-up list using the category API described in the chapter "[Categories](#)" on page 131 of the *Palm OS Programmer's API Reference*.

For the most part, you can handle a category pop-up list using only these calls:

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

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

This section focuses on the user interface aspects of categories. For information on how categories are stored and how to manage categories in a database, read [Chapter 6](#), “[Files and Databases](#).”

Initializing Categories in a Database

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

Category information is stored in the [AppInfoType](#) structure within the database’s application info block. As described in the chapter titled “[Files and Databases](#)” in this book, the application info block may contain any information that your database needs. If you want to use the category API, the first field in the application info block must be an `AppInfoType` structure.

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

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

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

- Place any categories that you don’t want the user to be able to change at the beginning of the list. For example, it’s common

User Interface

Categories

to have at least one uneditable category named Unfiled, so it should be the first item in the list.

- The string list must have 16 entries. Typically, you don't want to predefine 16 categories. You might define one or two and leave the remaining entries blank. The unused slots should have 0 length.
- Keep in mind that there is a limit of 16 categories. That includes both the predefined categories and the categories your users will create.
- Each category name has a maximum length defined by the `dmCategoryLength` constant (currently, 16 bytes).
- Don't include strings for "All" or "Edit Categories." While these two items often appear in category lists, they are not categories, and they are treated differently by the category functions.

[Listing 4.3](#) shows an example function that creates and initializes a database with an application info block. Notice that because the application info block is stored with the database, you allocate memory for it using `DmNewHandle`, not with `MemHandleNew`.

Listing 4.3 Creating a database with an app info block

```
typedef struct {
    AppInfoType appInfo;
    UInt16 myCustomAppInfo;
} MyAppInfoType;

Err CreateAndOpenDatabase(DmOpenRef *dbPP, UInt16 mode)
{
    Err error = errNone;
    DmOpenRef dbP;
    UInt16 cardNo;
    MemHandle h;
    LocalID dbID;
    LocalID appInfoID;
    MyAppInfoType *appInfoP;

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

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

// Get database local ID and card number. We need these to
// initialize app info block.
if (DmOpenDatabaseInfo(dbP, &dbID, NULL, NULL, &cardNo, NULL))
    return dmErrInvalidParam;

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

// Associate app info with database.
appInfoID = MemHandleToLocalID (h);
DmSetDatabaseInfo(cardNo, dbID, NULL, NULL, NULL, NULL, NULL,
    NULL, NULL, &appInfoID, NULL, NULL, NULL);

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

// Initialize the categories.
CategoryInitialize ((AppInfoPtr) appInfoP,
    MyLocalizedAppInfoStr);

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

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

Initializing the Category Pop-up Trigger

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

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

Forms that display information from a single record should show that record's category in the pop-up list. Each database record stores the index of its category in its attribute word. You can retrieve the record attribute using [DmRecordInfo](#) and then AND it with the mask `dmRecAttrCategoryMask` to obtain the category index.

[Listing 4.4](#) shows how to set the trigger label to match the category for a particular database record.

Listing 4.4 Setting the category trigger label

```
UInt16 attr, category;
Char categoryName [dmCategoryLength];
ControlType *ctl;

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

Managing a Category Pop-up List

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

Calling CategorySelect

The following is a typical call to CategorySelect:

Listing 4.5 Calling CategorySelect

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

This example uses the following as parameters:

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

Because uneditable categories are assumed to be at the beginning of the category list, passing 1 for this parameter means that CategorySelect does not allow the user to edit the category at index 0.

- categoryDefaultEditCategoryString is a constant that means include an Edit Categories item in the list and use the

User Interface

Categories

default string for its name (“Edit Categories” on US English ROMs).

To use a different name (for example, if you don’t have enough room for the default name), pass the ID of a string resource containing the desired name.

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

NOTE: The `categoryDefaultEditCategoryString` and `categoryHideEditCategory` constants are only defined if [3.5 New Feature Set](#) is present. See the [CategorySelect](#) function description in the *Palm OS Programmer’s API Reference* for further compatibility information.

Interpreting the Return Value

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

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

Listing 4.6 CategorySelect return value

```
Int16 category;  
Boolean categoryEdited;  
  
category = CurrentCategory;  
  
categoryEdited = CategorySelect (AddrDB, frm,  
    ListCategoryTrigger, ListCategoryList, true, &category,
```



```
CategoryName, 1, categoryDefaultEditCategoryString);  
  
if ( categoryEdited || (category != CurrentCategory)) {  
    /* user changed category selection or edited category list.  
       Do something. */  
}
```

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

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

Note that the `CategorySelect` function handles the results of the Edit Categories dialog for you. It adds, deletes, and renames items in the database's `AppInfoType` structure. If the user deletes a category that contains records, it moves those records to the Unfiled category. If the user changes the name of an existing category to the name of another existing category, it prompts the user and, if confirmed, moves the records from the old category to the new category. Therefore, you never have to worry about managing the category list after a call to `CategorySelect`.

Bitmaps

A bitmap is a graphic displayed by Palm OS. There are several ways to create a bitmap resource in Constructor:

- If you simply want to display a bitmap at a fixed location on a form, drag a Form Bitmap object to the form. Assign a resource ID in the Bitmap ID field, and you can then create a bitmap resource. The bitmap resource is a 'Tbmp' resource, and the Form Bitmap object that contains it is a 'tFBM' resource.
- If you want to create a bitmap for some other purpose (for example, to use in animation or to display a gadget), create either a Bitmap resource or a Bitmap Family resource in the

main project window. In this case, Constructor creates a 'tbfm' resource, and the PalmRez post linker converts it and its associated PICTs to a 'Tbmp' resource. (Constructor creates PICT format images on both the Macintosh and Microsoft Windows operating systems.)

Versions of Bitmap Support

There are four different bitmap encodings:

- Version 0 encoding is supported by all Palm OS releases.
- Version 1 encoding is supported on Palm OS 3.0 and later. PalmRez creates version 1 bitmaps unless you've explicitly specified a transparency index or a compression type when creating the bitmap in Constructor.
- Version 2 encoding is supported on Palm OS 3.5 and later. This encoding supports transparency indices and RLE compression.

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

When a bitmap with a transparency index is rendered at a depth other than the one at which it was created, the transparent color is first translated to the corresponding depth color, and the resulting color is named transparent. This may result in a group of colors becoming transparent.

- Version 3 encoding is supported on Palm OS 5 and handhelds running the [High-Density Display Feature Set](#), and adds support for displays of varying densities.

High-Density Bitmaps

The [BitmapTypeV3](#) data structure contains a 16-bit density field. For the screen bitmap, this field represents the screen density. An enumerated list representing density is defined in `Bitmap.h`:

```
typedef enum {  
    kDensityLow = 72,  
    kDensityOneAndAHalf = 108,  
    kDensityDouble = 144,  
};
```

```
kDensityTriple = 216,  
kDensityQuadruple = 288  
} DensityType;
```

The `kDensityLow` value of 72 is arbitrary. Although this value doesn't necessarily represent pixels per inch, it is useful to think of it that way.

IMPORTANT: Not all densities listed in the `DensityType` enum are supported by this version of the High-Density Display feature set. For this release, only `kDensityLow` and `kDensityDouble` are supported.

Palm OS 4.0 was released with a version 2 `BitmapType` structure. The density field is defined only on `BitmapType` structures with a version greater than 2. If a given bitmap structure is version 2 or less, the operating system assumes that the bitmap contains low-density data.

The blitter uses the density field in the source and destination bitmaps to determine an appropriate scaling factor. Because default density bitmaps must be scaled for high-density displays, some handhelds with high-density screens may use graphic accelerators. Nevertheless, the software blitter incorporates pixel-scaling logic for when the destination is an offscreen window.

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

The following examples demonstrate the above concepts.

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

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

```
WinDrawBitmap(bitmapP, 31, 23);
```

with the intention of placing the bitmap on the screen beginning at screen coordinate (31, 23), assuming the standard 160 by 160 coordinate system.

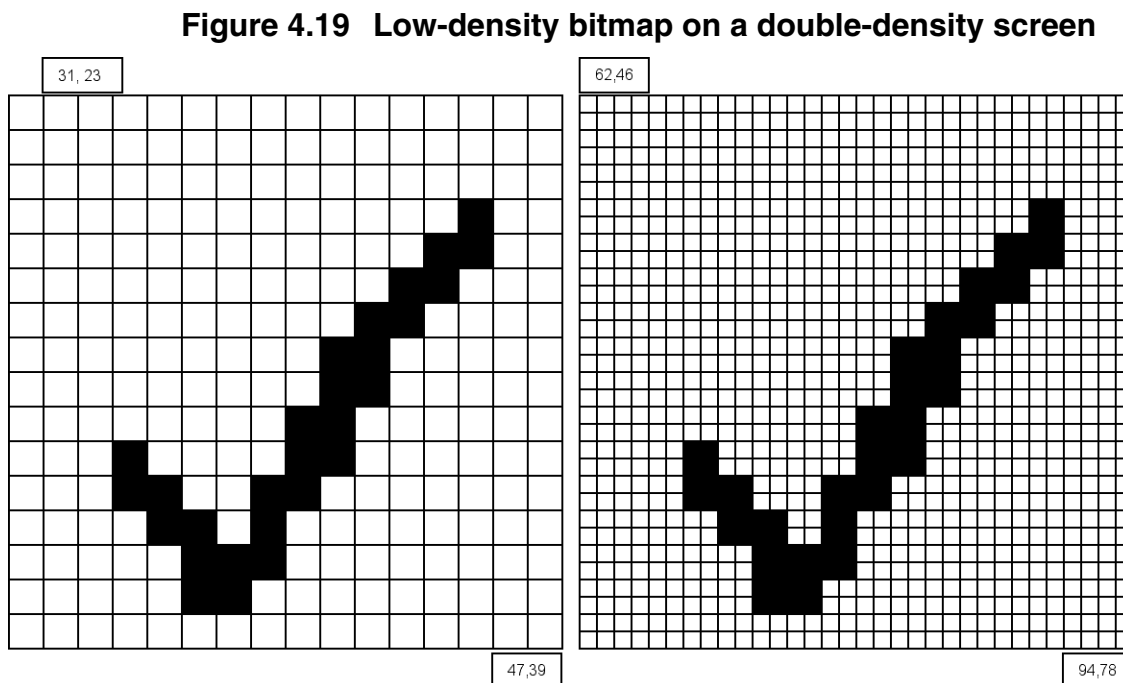
User Interface

Bitmaps

Since Palm OS by default uses the standard coordinate system and since the handheld has a double-density screen, the draw window's draw state contains a scale field value of 2.0. `WinDrawBitmap` transforms (31, 23) to high-density coordinates by multiplying (31, 23) by the scale field, and then calls the blitter with coordinates (62, 46).

The blitter receives the screen coordinates (62, 46) along with the low-density bitmap. The blitter recognizes the bitmap as low density, based upon the version of its [BitmapType](#) structure, and pixel-doubles the source data when blitting to the double-density screen.

The following illustration shows the source data on the left, with low-density window coordinates for the top-left and bottom-right corners. The illustration on the right shows the result as displayed on the screen, with top-left coordinates scaled by the Window Manager and bitmap data pixel-doubled by the blitter.



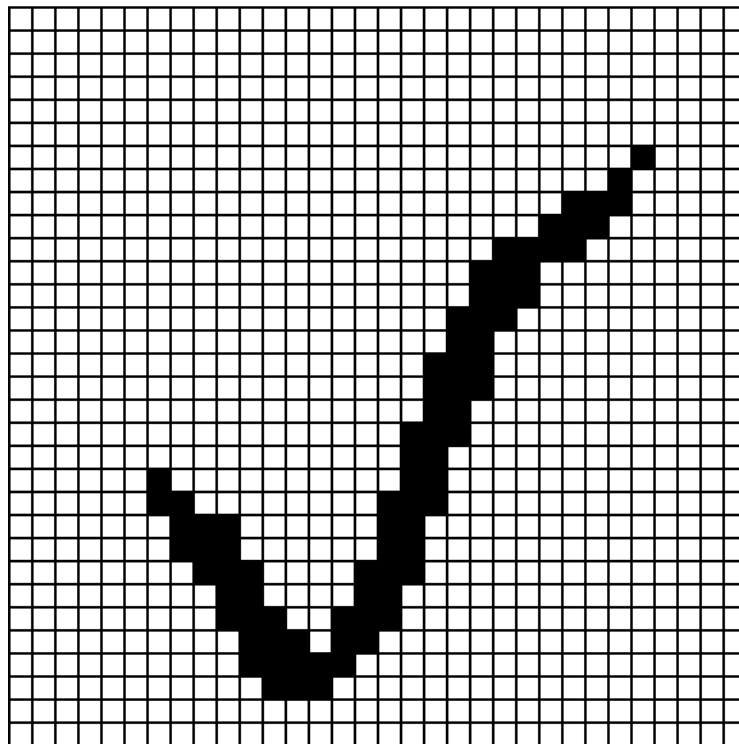
- **A new application draws a double-density bitmap to a double-density screen.**

The source data is a 32 by 32 double-density bitmap. Recognizing that the bitmap is being drawn to a double-density screen, the application uses the new functions to establish the double-density coordinate system, and calls `WinDrawBitmap` with high-density coordinates:

```
WinPushDrawState();  
oldScale =  
WinSetCoordinateSystem(kCoordinatesNative);  
WinDrawBitmap(bitmapP, 61, 45);  
WinPopDrawState();
```

Figure 4.20 Double-density bitmap on a double-density screen

61,45



The double-density coordinates (61, 45) allow the application to position the bitmap more precisely on the screen; these

coordinates are equivalent to coordinates (30.5, 22.5) in the standard coordinate system.

Since the window's native coordinate system is active, the Window Manager leaves the double-density coordinates (61, 45) unchanged. The blitter receives these coordinates along with the double-density source bitmap. Because the screen bitmap has the same density, the blitter copies the source data to the screen unchanged.

Note that the point of calling [WinSetCoordinateSystem](#) is not to have the OS draw the double-density bitmap, but to place the top-left corner of the bitmap at a double-density coordinate. If the application does not need the precision of double-density coordinates, the application can simply call `WinDrawBitmap`:

```
WinDrawBitmap(bitmapP, x, y);
```

and pass standard coordinates for `x` and `y`. The Window Manager transforms `(x, y)` to the screen coordinate system, and the blitter draws the double-density bitmap at that location.

If standard coordinates are acceptable, and if the application's bitmap family contains both low-density and double-density bitmaps, `WinDrawBitmap` selects the appropriate bitmap from the bitmap family based on the destination window's density; no additional logic is needed in the application. By providing both double-density and low-density bitmaps in a bitmap family, applications can display images properly on handhelds with various screen densities without separate code paths.

See "[High-Density Bitmap Families](#)" on page 131 for a more complete description of bitmap families.

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

If an application includes only high-density bitmaps, the blitter needs to shrink them when drawing them to the

screen. The application can determine the screen density like this:

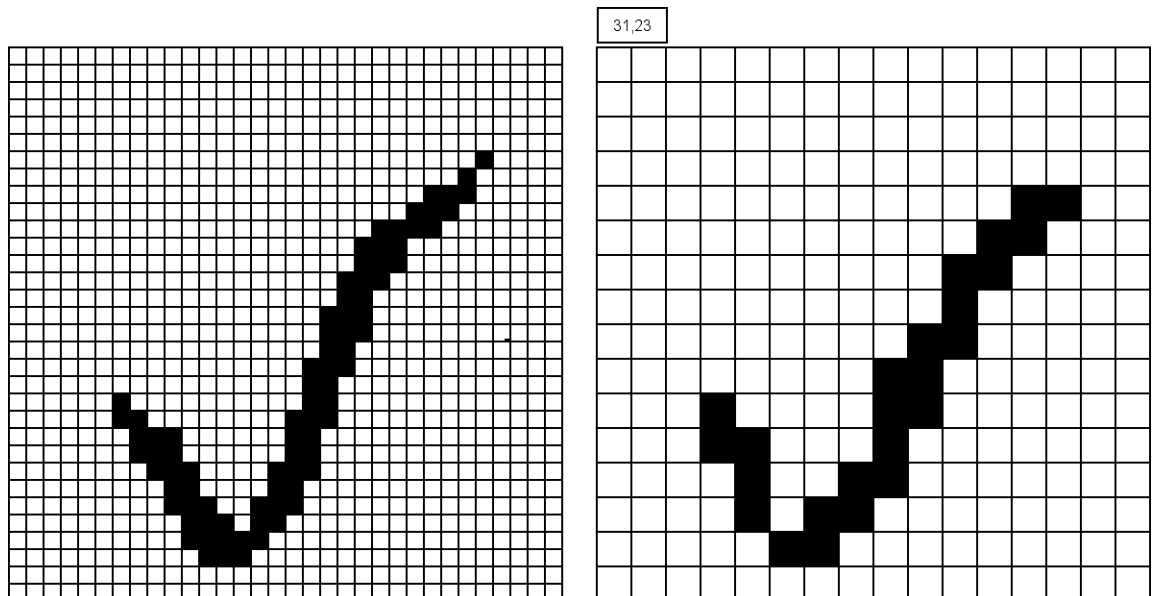
```
UInt32 density;  
  
err = WinScreenGetAttribute(winScreenDensity, &density);
```

Understanding that the destination is low density, the application calls `WinDrawBitmap` using the standard coordinate system:

```
WinDrawBitmap(bitmapP, 31, 23);
```

Because the destination window is low density, and because the passed coordinates are standard coordinates, the Window Manager does not scale the passed coordinates. The blitter, however, recognizes that the source bitmap has a density of `kDensityDouble` and shrinks the data to one-half the original size when blitting it to the low-density screen.

Figure 4.21 Double-density bitmap on a low-density screen



The result, shown above on the right, is poor. Because of this, for an application to look good on both low and high-density screens it should include both low and high-density bitmaps.

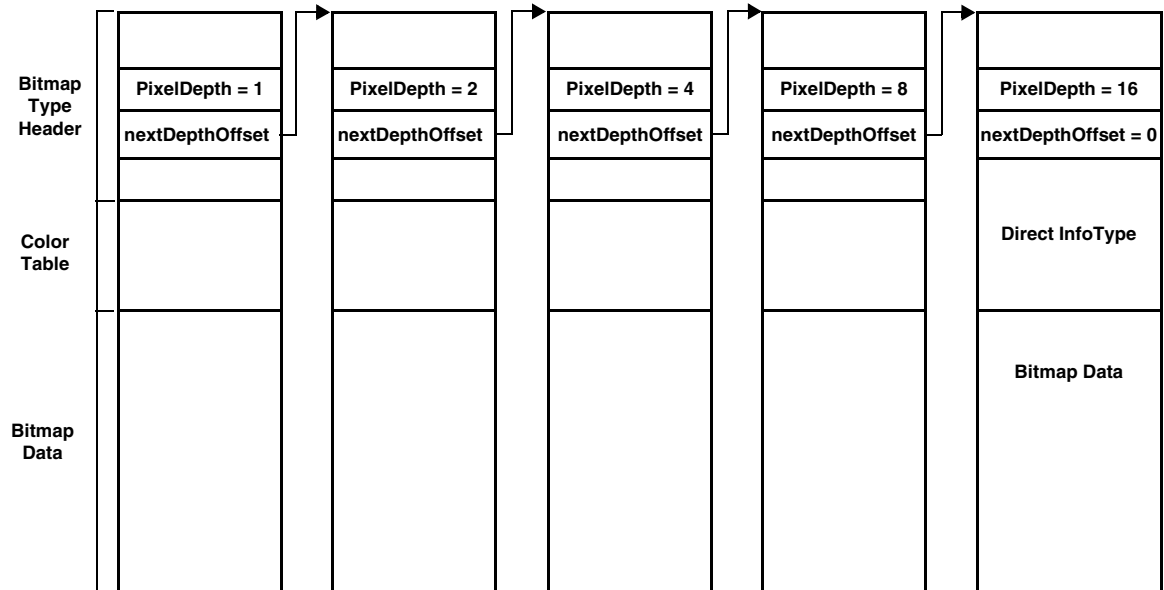
Note that although the blitter included with the High-Density Display feature set can expand or shrink a bitmap as necessary, on handhelds without this feature set if an application contains a bitmap family with only high-density bitmaps, nothing is drawn.

Bitmap Families

A 'Tbmp' resource defines either a single bitmap or a bitmap family. A **bitmap family** is a group of bitmaps, each containing the same drawing but at a different pixel depth (see [Figure 4.22](#)). When requested to draw a bitmap family, the operating system chooses the version of the bitmap with the pixel depth equal to the display. If such a bitmap doesn't exist, the bitmap with the pixel depth closest to but less than the display depth is chosen. If there are no bitmaps less than the display depth, then the bitmap with the pixel depth closest to the display depth is used.

Programmatically, a bitmap or bitmap family is represented by a [BitmapType](#) structure. This structure is simply a header. It is followed by the bitmap data in the same memory block. Bitmaps in Palm OS 3.0 and higher are also allowed to have their own color tables. When a bitmap has its own color table, it is stored between the bitmap header and the bitmap data.

Figure 4.22 Single-density bitmap family



High-Density Bitmap Families

Bitmap families represent a single image across a variety of Palm handhelds with screens of different bit depths. Prior to the [High-Density Display Feature Set](#), a bitmap family is a null-terminated linked list of bitmaps ordered from low to high bit depth. [WinDrawBitmap](#) iterated through the linked list and selected the bitmap with the greatest bit depth less than or equal to the draw window's bit depth.

Although bitmap families are still represented using a null-terminated linked list of bitmaps, the algorithm used to select the appropriate bitmap for a given situation changes with the High-Density Display feature set. There are two reasons for this change. First, when the draw window is 8-bit, it is better to select a 16-bit image over a grayscale image. Second, density must now be taken into account when selecting a bitmap.

The algorithm that is used in the High-Density Display feature set depends upon the density of the draw window. If the draw window is low density, low-density bitmaps are always favored over double-density bitmaps, regardless of source bitmap depth. If the draw window is double density, however, the color domain match (color

vs. grayscale) is favored over a double-density bitmap with a color domain mismatch. The following algorithm is used on a handheld with a double-density screen:

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

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

Table 4.17 Double-density algorithm results

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

Table 4.17 Double-density algorithm results (*continued*)

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

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

IMPORTANT: A bitmap family used for a graphic button, slider, or form bitmap must include at least one low-density version of the image in the bitmap family. This restriction doesn't apply to bitmaps used for custom gadgets: if your application will only run on handhelds with high-density displays, you don't need to have any low-density images in your custom gadget bitmap families.

Handhelds that don't have the High-Density Display feature set don't display high-density bitmaps. They do, however, display any low-density bitmaps in applications that contain both low and high-density bitmaps. This is because the older versions of the OS don't attempt to follow the linked list of bitmaps in a bitmap family as it crosses over from low-density to high-density bitmaps. This is accomplished by inserting a dummy version 1 bitmap structure between the low-density and high-density bitmaps within a bitmap family. The dummy bitmap contains no bitmap data, no color table, and an invalid bit depth. By setting the bit depth of the dummy bitmap to 0xFF, the logic in older versions of the OS that traverse the linked list of bitmaps stops at the appropriate place in the list.

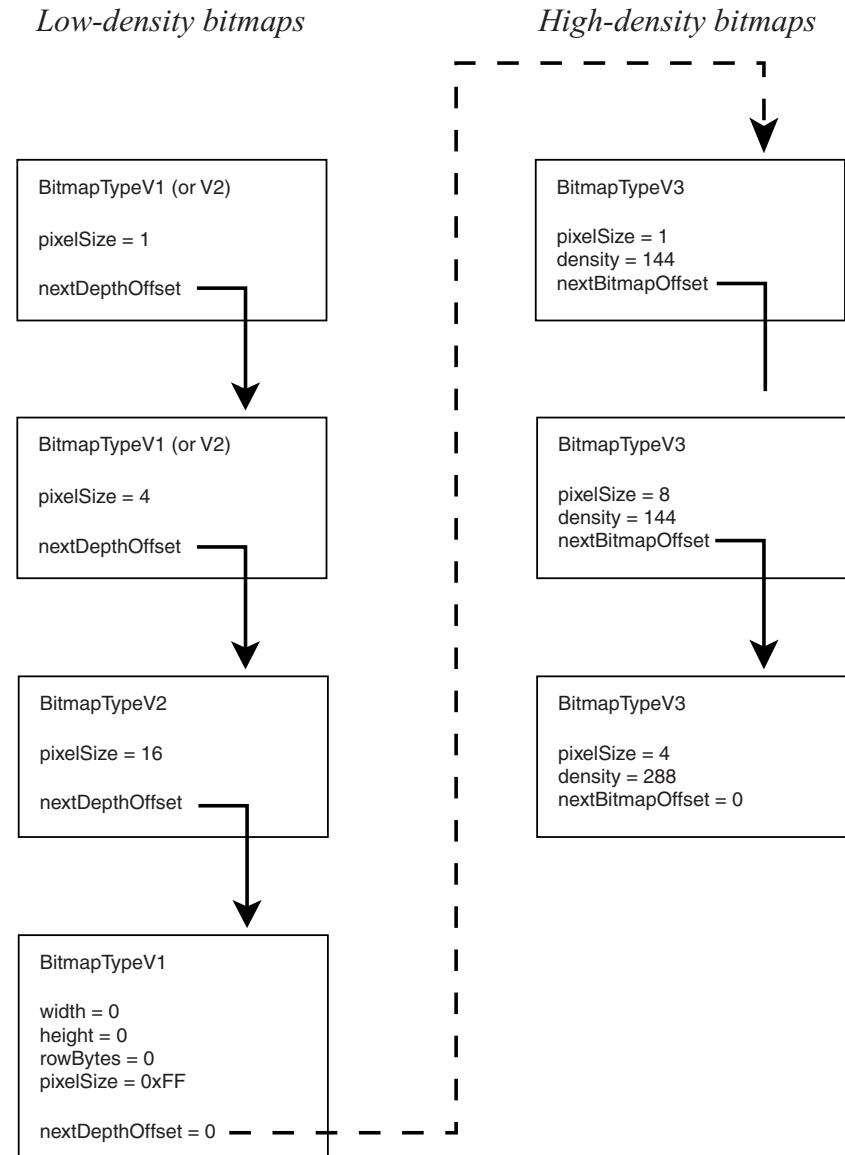
User Interface

Bitmaps

The High-Density Display feature set recognizes that the dummy bitmap is followed by high-density bitmaps and continues the traversal, skipping over the dummy bitmap. Note that the dummy bitmap is only present if there are one or more high-density bitmaps, and is always present if there are any high-density bitmaps. The dummy bitmap is the first bitmap if there are high-density bitmaps but no low-density bitmaps.

[Figure 4.23](#) illustrates the process of traversing the bitmaps in a bitmap family. The dotted line indicates a step that is taken only in handhelds running the High-Density Display feature set.

Figure 4.23 Linked list of bitmaps in a bitmap family



Drawing a Bitmap

If you use a Form Bitmap object, your bitmap is drawn when the form is drawn. No extra coding is required on your part.

If you're not using a Form Bitmap object, to draw the bitmap you obtain it from the resource database and then call either

User Interface

Bitmaps

[WinDrawBitmap](#) or [WinPaintBitmap](#). (The form manager code uses [WinDrawBitmap](#) to draw Form Bitmap objects.) If passed a bitmap family, these two functions draw the bitmap that has the depth equal to the current draw window depth or the closest depth that is less than the current draw window depth if available, or the closest depth greater than the current draw depth if not.

Listing 4.7 Drawing a bitmap

```
MemHandle resH = DmGetResource (bitmapRsc, rscID);
BitmapType *bitmap = MemHandleLock (resH);
WinPaintBitmap(bitmap, 0, 0);
```

If you want to modify a bitmap, starting in Palm OS 3.5 you can create the bitmap programmatically with [BmpCreate](#), create an offscreen window wrapper around the bitmap using [WinCreateBitmapWindow](#), set the active window to the new bitmap window, and use the window drawing functions to draw to the bitmap:

Listing 4.8 Programmatically creating a bitmap

```
BitmapType *bmpP;
WinHandle win;
Err error;

bmpP = BmpCreate(10, 10, 8, NULL, &error);
if (bmpP) {
    win = WinCreateBitmapWindow(bmpP, &error);
    if (win) {
        WinSetDrawWindow(win);
        WinDrawLines(win, ...);
        /* etc */
    }
}
```

Note that [BmpCreate](#) always creates a version 2 bitmap

To learn how to modify a bitmap in releases prior to Palm OS 3.5, download the Signatures example application from the Knowledge Base on the Palm OS Developer website.

Color Tables and Bitmaps

As mentioned previously, bitmaps can have their own color tables attached to them. A bitmap might have a custom color table if it requires a palette that differs from the default system palettes. If a bitmap has its own color table, the system must create a conversion table to convert the color table of the current draw window before it can draw the bitmap. This conversion is a drain on performance, so using custom color tables with bitmaps is not recommended if performance is critical.

As an alternative, if your bitmap needs a custom palette, use the [WinPalette](#) function to change the system palette that is currently in use, then draw your bitmap. After the bitmap is no longer visible, use `WinPalette` again to set the system palette back to its previous state.

Labels

You can create a label in a form by creating a label resource.

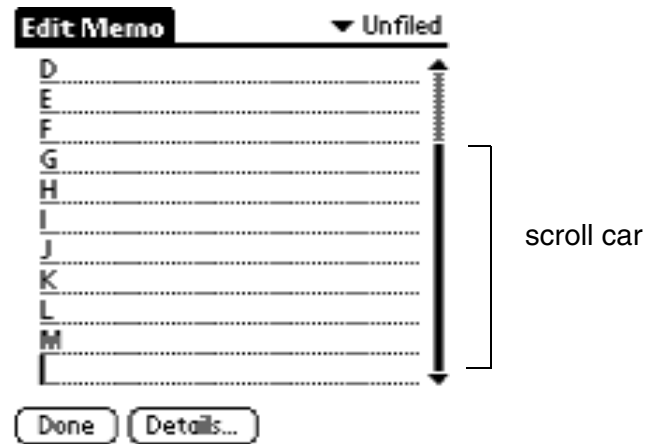
The label resource displays noneditable text or labels on a form (dialog or full-screen). It's used, for example, to have text appear to the left of a checkbox instead of the right.

You don't interact with a label as a programmatic entity; however, you can use Form API to create new labels or to change labels dynamically. See the "[Summary of User Interface API](#)" at the end of this chapter.

Scroll Bars

Palm OS 2.0 and later provides vertical scroll bar support. As a result, you can attach scroll bars to fields, tables, or lists, and the system sends the appropriate events when the end user interacts with the scroll bar (see [Figure 4.24](#)).

Figure 4.24 Scroll bar



Here's what you have to do to include a scroll bar in your user interface:

1. Create a scroll bar (tSCL) UI resource.
Provide the ID and the bounds for the scroll bar rectangle. The height has to match the object you want to attach it to. The width should be 7.
2. Provide a minimum and maximum value as well as a page size.
 - Minimum is usually 0.
 - Maximum is usually 0 and set programmatically.
 - The page size determines how many lines the scroll bar moves when the text scrolls.
3. Make the scroll bar part of the form.
When you compile your application, the system creates the appropriate scroll bar UI object. (See the chapter "[Scroll Bars](#)" in the *Palm OS Programmer's API Reference* for more information on the scroll bar UI object.)

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

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

To get this functionality, set the `hasScrollbar` attribute of the field, table, or list. (For tables, you must set this programmatically with the function [TblHasScrollbar](#).)

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

If `hasScrollbar` is set for a table, you should keep track of when the table's size changes. Whenever it does, you should compute new values for the scroll bar's minimum, maximum, and current position and then use `SclSetScrollbar` to update it.

Lists are intended for static data, so you typically don't have to worry about the size of a list changing.

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

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

The system sends the following scroll bar events:

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

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

User Interface

Custom UI Objects (Gadgets)

object ([FldScrollField](#), [LstScrollList](#), or your own scrolling function in the case of tables).

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

Custom UI Objects (Gadgets)

A gadget resource lets you implement a custom UI object. The gadget resource contains basic information about the custom gadget, which is useful to the gadget writer for drawing and processing user input.

You interact with gadgets programmatically using the Form API. See the “[Summary of User Interface API](#)” at the end of this chapter.

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

Starting in Palm OS 3.5, you can create an extended gadget. An extended gadget is simply a gadget with a callback routine ([FormGadgetHandlerType](#)) that provides drawing and event handling code for the gadget. Use [FrmSetGadgetHandler](#) to set the callback function. (A pointer to the gadget is passed to the callback, so you can use the same function for multiple gadgets.) When the form receives certain requests to draw itself, delete itself, or to hide or show a gadget object, the form manager calls the gadget handler function you provide. When the form receives events intended for the gadget, it passes those to the gadget handler function as well.

In versions prior to 3.5, gadgets do not have a callback function. Instead, you must write code to draw the gadget and respond to pen down events in the form's event handler. [Listing 4.9](#) shows the event handler for the main form in the Rock Music sample application. This code makes calls to draw the gadget in response to a `frmOpenEvent` or `frmUpdateEvent`, and if there is a `penDownEvent` within the bounds of the gadget, it calls a function to handle that event as well. [Listing 4.10](#) shows how a gadget handler function might be written for Rock Music.

Listing 4.9 Pre-Palm OS 3.5 gadget example

```
Boolean MainViewHandleEvent(EventPtr event)
{
    Boolean handled = false;
    Word objIndex;
    FormPtr frm;
    RectangleType r;

    switch (event->eType) {
        case frmOpenEvent:
            MainViewInit();
            frm = FrmGetActiveForm ();
            FrmDrawForm (frm);
            DrawGadget();
            handled = true;
            break;

        case frmUpdateEvent:
            frm = FrmGetActiveForm ();
            FrmDrawForm (frm);
            DrawGadget();
            handled = true;
            break;

        case penDownEvent:
            frm = FrmGetActiveForm ();
            objIndex = FrmGetObjectIndex (frm,
                RockMusicMainInputGadget);
            FrmGetObjectBounds (frm, objIndex, &r);
            if (RctPtInRectangle (event->screenX,
                event->screenY, &r)) {
                GadgetTapped ();
                handled=true;
            }
            break;

        ...
    }
}
```

Listing 4.10 Palm OS 3.5 gadget example

```
Boolean GadgetHandler (struct FormGadgetType *gadgetP,
    UInt16 cmd, void *paramP)
{
    Boolean handled = false;
```

User Interface

Dynamic UI

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

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

Dynamic UI

Palm OS 3.0 and higher provide functions that can be used to create forms and form elements at runtime. Most applications will never need to change any user interface elements at runtime—the built-in applications don't do so, and the Palm user interface guidelines discourage it. The preferred method of having UI objects appear as needed is to create the objects in Constructor and set their usable attributes to false. Then use [FrmShowObject](#) and [FrmHideObject](#) to make the object appear and disappear as needed.

Some applications, such as forms packages, must create their displays at runtime—it is for applications such as these that the Dynamic UI API is provided. If you're not absolutely sure that you need to change your UI dynamically, don't do it—unexpected changes to an application's interface are likely to confuse or frustrate the end user.

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

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

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

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

When creating forms dynamically, or just to make your application more robust, use the [FrmValidatePtr](#) function to ensure that your form pointer is valid and the form it points to is valid. This routine can catch lots of bugs for you—use it!

Dynamic User Interface Functions

The following API can be used to create forms dynamically:

- [CtlNewControl](#)
- [CtlValidatePointer](#)
- [FldNewField](#)
- [FrmNewBitmap](#)
- [FrmNewForm](#)
- [FrmNewGadget](#)
- [FrmNewLabel](#)
- [FrmRemoveObject](#)
- [FrmValidatePtr](#)
- [LstNewList](#)
- [WinValidateHandle](#)
- [FrmNewGsi](#) (available only if [3.5 New Feature Set](#) is present)

Color and Grayscale Support

Starting in Palm OS version 3.5, the operating system supports system palettes of 1, 2, 4, or 8 bits-per-pixel, as follows:

- 1-bit: white (0) and black (1)
- 2-bit: white (0), light gray (1), dark gray (2), and black (3)
- 4-bit: 16 shades of gray, from white (0) to black (0xF)
- 8-bit: 216 color “Web-safe” palette, which includes all combinations of red, green, and blue at these levels: 0x00, 0x33, 0x66, 0x99, 0xCC, and 0xFF. Also, it includes all 16 gray shades at these levels: 0x00, 0x11, 0x22, ... 0xFF. Finally, it includes these extra named HTML colors: 0xC0C0C0 (silver), 0x808080 (gray), 0x800000 (maroon), 0x800080 (purple), 0x008000 (green), and 0x008080 (teal). The remaining 24 entries (indexes 0xE7 through 0xFE) are unspecified and filled with black. These entries may be defined by an application.

Generalized support for color tables in all bit depths is included, with performance degrading if the color tables are not standard.

Starting in Palm OS version 4.0, the operating system supports 16-bit color. However, support is not provided to allow the UI layer of the OS to utilize 16-bit color mode. Buttons, controls, and other gadgets continue to be displayed with a color bit depth of no more than 8-bits.

Indexed Versus Direct Color Display

Displays that support 1, 2, 4, or 8 bits per pixel rely on a color lookup table in the display hardware in order to map pixel values into colors. The only colors that can be displayed on the screen at any given time are those that are found in the display's color lookup table.

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

The color indexed mode for setting the foreground, background, and text colors used previous to Palm OS release 4.0 continues to work even with direct color displays because the system uses a translation table for mapping color index values into direct colors.

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

Color Table

The system color table is stored in a 'tc1t' resource (symbolically named `colorTableRsc`). The color table is a count of the number of entries, followed by an array of [RGBColorType](#) colors. An `RGBColorType` struct holds 8 bits each of red, green, and blue plus an "extra" byte to hold an index value.

User Interface

Color and Grayscale Support

A color's index is used in different ways by different software layers. When querying for a color or doing color fitting, the index holds the index of the closest match to the RGB value in the reference color table. When setting a color in a color table, the index can specify which slot the color should occupy. In some routines, the index is ignored.

Generally, the drawing routines and the operating system use indexed colors rather than RGB. Indexed colors are used for performance reasons; it allows the RGB-to-index translation to be skipped for most drawing operations.

Care should be taken not to confuse a full color table (which includes the count) with an array of RGB color values. Some routines operate on entire color tables, others operate on lists of color entries.

Color Translation Table

When rendering requires a translation from one depth to another, a color translation table is used. For example, suppose you are trying to display an 8-bit color bitmap image on a 2-bit display. Palm OS must translate the color bitmap to a grayscale bitmap in order to display it. To do so, it creates the translation table by stepping through each element of the source color table (the 8-bit bitmap) and finding the best fit for the RGB value in the destination color table (which has exactly 4 values). This table is generated once and is reused for all drawing operations until it is no longer valid.

Palm OS uses one of two algorithms to build the translation table:

- Luminosity fitting if the destination color table is grayscale.
- Shortest distance in the RGB space if the destination color table is color.

Although shortest distance RGB fitting does not always produce the best perceptual match, it is fast, and it works well for the available palettes on Palm OS.

Color Table Management

If you want to change the color table used by the current draw window, you can do so with the [WinPalette](#) function. If the current draw window is onscreen, the palette for the display

hardware is also changed. For more information see the `WinPalette` function description in the *Palm OS Programmer's API Reference*.

If your application needs to know which RGB color corresponds to which index color in the current palette, it can do so with the function calls [WinRGBToIndex](#) and [WinIndexToRGB](#). When calling `WinRGBToIndex`, an exact match may not be available. That is, you may be calling `WinRGBToIndex` with an RGB value that is not in the palette and thus does not have an index. If there is no exact RGB match, the best-fit algorithm currently in place is used to determine the index value. For `WinIndexToRGB`, the RGB value returned is always the exact match. (An error is displayed on debug ROMs if the index is out of range.)

UI Color List

The system builds a UI color list in addition to the system color table. The UI color list contains the colors used by the various user interface elements. Each UI color is represented by a symbolic color constant. See [Table 4.18](#) for a list of colors used.

Each bit depth has its own list of UI colors, allowing for a different color scheme in monochrome, grayscale, and color modes. This is important because even with a default monochrome look and feel, highlighted field text is black-on-yellow in color and white-on-black in other modes.

To obtain the color list, the system first tries to load it from the synchronized preferences database using the value `sysResIDPrefUIColorTableBase` plus the current screen depth. The use of a preference allows for the possibility that individual users could customize the look using a third party “personality” or “themes” editor. If the preference is not defined, it loads the default color table from the system color table resource plus the current screen depth.

Using a list allows easy variation of the colors of UI elements to either personalize the overall color scheme of a given Palm Powered handheld or to adjust it within an application. Defining these as color classes ensures that the user interface elements are consistent with each other.

Table 4.18 UI objects and colors

UI Object	Symbolic Colors Used
Forms	UIFormFrame, UIFormFill
Modal dialogs	UIDialogFrame, UIDialogFill
Alert dialogs	UIAlertFrame, UIAlertFill
Buttons (push button, repeating button, check boxes, and selector triggers)	UIObjectFrame, UIObjectFill, UIObjectForeground, UIObjectSelectedFill, UIObjectSelectedForeground
Fields	UIFieldBackground, UIFieldText, UIFieldTextLines, UIFieldTextHighlightBackground, UIFieldTextHighlightForeground
Menus	UIMenuFrame, UIMenuFill, UIMenuForeground, UIMenuSelectedFill, UIMenuSelectedForeground
Tables	Uses UIFieldBackground for the background, other colors controlled by the object in the table cell.
Lists and pop-up triggers	UIObjectFrame, UIObjectFill, UIObjectForeground, UIObjectSelectedFill, UIObjectSelectedForeground
Labels	Labels on a control and noneditable fields use UIObjectForeground, and text written to a form using WinDrawChars or WinPaintChars use the current text setting in the draw state.
Scroll bars	UIObjectFill, UIObjectForeground, UIObjectSelectedFill, UIObjectSelectedForeground

Table 4.18 UI objects and colors (*continued*)

UI Object	Symbolic Colors Used
Insertion point	UIFieldCaret
Front-end processor (currently only used on Japanese systems)	UIFieldFepRawText, UIFieldFepRawBackground, UIFieldFepConvertedText, UIFieldFepConvertedBackground, UIFieldFepUnderline

Should your application need to change the colors used by the UI color list, it can do so with [UIColorSetTableEntry](#). If you need to retrieve a color used, it can do so with [UIColorGetTableEntryIndex](#) or [UIColorGetTableEntryRGB](#).

If you change the UI color list, your changes are in effect only while your application is active. The UI color list is reset as soon as control switches to another application. When control switches back to your application, you'll have to call [UIColorSetTableEntry](#) again.

Direct Color Functions

The direct color function calls are more generic than their indexed forms and can be used with both indexed (1, 2, 4, or 8 bit) or direct 16-bit color displays. The system automatically looks up the color index value of the closest color if necessary.

The direct color functions are: [WinSetForeColorRGB](#), [WinSetBackColorRGB](#), [WinSetTextColorRGB](#), and [WinGetPixelRGB](#)

Because these calls are only available on systems with the direct color enhancements present, applications should generally stick to using the indexed form of these calls: [WinSetForeColor](#), [WinSetBackColor](#), [WinSetTextColor](#), and [WinGetPixel](#) unless they need finer control over the choice and dynamic range of colors.

Pixel Reading and Writing

The Palm OS 3.5 API call for reading a pixel value, [WinGetPixel](#), is designed to return a color index value. When this call is performed

User Interface

Color and Grayscale Support

on a direct color display, it must first get the actual pixel value (a 16 or 24 bit direct color value). The system then looks up the closest color from the system's virtual 8-bit color lookup table, and returns the index of the closest color from that table. This mode of operation ensures compatibility for applications that take the return value from `WinGetPixel` and use it as an indexed color to [WinSetForeColor](#), [WinSetBackColor](#), and [WinSetTextColor](#).

Applications that need to copy pixels exactly from one location to another on direct color displays should use [WinGetPixelRGB](#) instead of [WinGetPixel](#). If you use [WinGetPixel](#) on a direct color display, it can result in a loss of color because of the closest-match color table lookup operation that [WinGetPixel](#) performs.

[WinGetPixelRGB](#) returns the pixel as an [RGBColorType](#) with a full 8 bits each of red, green, and blue, assuring no loss of color resolution. This call is more generic than the [WinGetPixel](#) call and can be used with both indexed (1, 2, 4, or 8 bit) or direct color modes. The system automatically looks up the RGB components of indexed color pixels as necessary.

The pixel setting API calls ([WinPaintPixel](#), [WinDrawPixel](#), and so on) all rely on using the current foreground and background colors and do not require new forms for the direct color mode. An application can simply pass in the return [RGBColorType](#) from [WinGetPixelRGB](#) to [WinSetForeColorRGB](#) and then call [WinDrawPixel](#) in order to copy a direct color pixel.

Direct Color Bitmaps

In Palm OS release 4.0 the Window Manager supports 16 bits per pixel direct color bitmaps, as well as the previously supported 1, 2, 4, and 8 bit indexed color bitmaps. A direct color bitmap is indicated by the new `directColor` bit in the [BitmapFlagsType](#) bit-field of the [BitmapType](#) data structure. In addition to this flag, a direct color bitmap must also include the [BitmapDirectInfoType](#) fields: `redBits`, `greenBits`, `blueBits`, `reserved`, and `transparentColor`.

The `redBits`, `greenBits`, and `blueBits` fields indicate the number of bits in each pixel for each color component. The current implementation only supports 16 bits per pixel, with 5 bits of red, 6 bits of green, and 5 bits of blue:

R R R R R G G G G G G B B B B B
MSB LSB

The `transparentColor` field contains the red, green, and blue components of the transparent color of the bitmap. For direct color bitmaps, this field is used instead of the `transparentIndex` field to designate the transparent color value of the bitmap, because the `transparentIndex` field is only 8 bits wide and can only represent an indexed color. The `transparentColor` field, like the `transparentIndex` field, is ignored unless the `hasTransparency` bit is set in the bitmap's flags field.

With Palm OS 4.0, a 16-bit direct color bitmap can always be rendered, regardless of the actual screen depth. The 16-bit color functions automatically perform the necessary bit depth conversion to render the bitmap into whatever depth the destination is in.

Bitmap resources can be built to contain multiple depth images in the same bitmap resource, one image for each possible depth. A potential incompatibility could arise if an application includes only a direct color version of a bitmap. Therefore, applications need to either check that version 4.0 of Palm OS is present before drawing a direct color bitmap, or they must always include a 1, 2, 4, or 8 bit per pixel image of the bitmap in the bitmap resource along with the direct color version.

Special Drawing Modes

The special drawing modes of `winErase`, `winMask`, `winInvert`, and `winOverlay` introduce a complication when it comes to direct color models. These drawing modes were originally designed for use with monochrome bitmaps where black is designated by 1 bits and white is designated by 0 bits. With these color assignments, these various modes can be described as:

- WinErase becomes an AND operation (black pixels in the source leave the destination alone whereas white pixels in the source make the destination white).
- WinMask becomes an AND NOT operation (black pixels in the source make the destination white whereas white pixels leave the destination alone)

User Interface

Color and Grayscale Support

- `WinInvert` becomes an XOR operation (black pixels in the source invert the destination whereas white pixels leave the destination alone)
- `WinOverlay` becomes an OR operation (black pixels in the source make the destination black, white pixels in the source leave the destination alone)

In a direct color bitmap, black is designated by all 0s and white is designated by all 1s. Because of this, if all the drawing modes were implemented as logical operations in the same way as they are for indexed color modes, the desired effect would not be achieved.

The assumption made by direct color functions is that the desired effect is more important to the caller than the actual logical operation that is performed. Thus, the various drawing modes, when drawing to a direct color bitmap, become:

- `WinErase` becomes an OR operation (black pixels in the source leave the destination alone whereas white pixels in the source make the destination white).
- `WinMask` becomes an OR NOT operation (black pixels in the source make the destination white whereas white pixels leave the destination alone)
- `WinInvert` becomes an XOR NOT operation (black pixels in the source invert the destination whereas white pixels leave the destination alone)
- `WinOverlay` becomes an AND operation (black pixels in the source make the destination black, white pixels in the source leave the destination alone)

As long as the source and destination bitmaps contain only black and white colors, the new interpretations of the drawing modes in direct color modes produce the same effects as they would have with an indexed color mode.

With non-black and white pixels however, an application may get unexpected results from these drawing modes if they assume that the direct color function calls perform the same logical operation in direct color mode as they do in indexed color mode.

Insertion Point

The insertion point is a blinking indicator that shows where text is inserted when users write characters in the input area or paste clipboard text.

In general, an application doesn't need to be concerned with the insertion point; the Palm OS UI manages the insertion point.

Application Launcher

The Application Launcher is the screen from which most applications are launched. Users navigate to the Launcher by tapping the Applications icon in the input area. They then launch a specific application by tapping its icon.

To integrate well with the Application Launcher, you must provide application icons and a version string as described in the following sections. In rare cases, you may need to provide a default application category as well.

Icons in the Launcher

Applications installed on the Palm Powered handheld (resource databases of type 'appl') appear in the Application Launcher automatically. Specifically, the Launcher displays an application icon and an application name.

Your application needs to have two icons:

- A large icon of type `tAIB`, with an ID of 1000. For compatibility with Palm OS 2.0, this icon should be 22 x 32 pixels; for all other Palm OS versions, you can make this icon 22 x 22 pixels.
- A smaller icon, also of type `tAIB`, with an ID of 1001. This icon should be 15 x 9 pixels.

NOTE: The Constructor program supplied with Palm OS SDK versions 3.5 and later allows you to create an Application Icon Family. You should not use the App Icon or Multi-bit Icon resources if the Application Icon Family is available.

The application name is defined in two ways:

- The application name (required) is specified in the PalmRez panel of your CodeWarrior project and used by HotSync application, the About box, the Memory display, and the database header.
- The application icon name (optional) is a string resource in the application's resource file. It is used by the Launcher screen and in the Button Assignment preferences panel (available in OS versions 2.0 and later). You assign the name using Constructor.

The application icon name is technically optional, but if you want the name to appear with the icon in the Launcher's main view, you must supply it.

Note: If you use an application icon name, make it short!

- Together with the application name, each application displays a application icon in the launcher.

Application Version String

The Launcher displays a version string from each application's `tver` resource, ID 1000. This short string (usually 3 to 7 characters) is displayed in the Info dialog.

A version string should have the format:

major.minor.[stage.change]

where *major* is the major version number, *minor* is a minor version number, *stage* is a letter denoting a development stage (a for alpha b for beta or d for developer release) and *change* is the build number. Remove the *stage* and *change* numbers for the final release.

The Default Application Category

Launcher divides applications into categories starting in Palm OS 3.5. You can store an application's category in a 'taic' resource (symbolically named `defaultCategoryRscType`) with the ID 1000 in the PRC file. Starting in Palm OS 3.5, the Launcher application installs your application into the specified category. In Constructor, you can specify the 'taic' resource by providing a value for the Default App Category field in the main window.

Most applications should **not** specify a 'taic' resource. By default, Launcher installs applications in the Unfiled category, and each user chooses where to file the application.

Only specify a 'taic' resource in these instances:

- Your application is intended for consumers and clearly belongs to one of the Launcher predefined categories (see [Table 4.19](#)).

Always specify the Launcher predefined categories in US English in ASCII characters. Launcher provides the appropriate translations for localized ROMs.

- Your application is intended for a vertical market or you've created a suite of custom applications that work together to provide a complete custom solution.

In this case, you might define a 'taic' resource with a custom category name. Launcher creates the category if it doesn't already exist in the Launcher database. When you're not identifying one of Launcher's predefined categories, you may identify the category in any language.

Table 4.19 Launcher predefined categories

Default Launcher Category	Description
Games	Any game.
Main	Applications that would be used on a daily basis, such as Date Book or Address Book.
System	Applications that control how the system behaves, such as the Preferences, HotSync, and Security.
Utilities	Applications that help the user with system management.
Unfiled	The default category.

Do not treat the default application category as something analogous to the Microsoft Windows Start menu category. On a

Palm Powered handheld, the user is limited to 16 categories including Unfiled. Obviously, that limit would be quickly reached if each application defines its own category. Only assign a default category where it is a clear benefit to your users.

Opening the Launcher Programmatically

Situations in which you need to open the Application Launcher programmatically are rare, but the system does provide an API for doing so. To activate the Launcher from within your application, enqueue a [keyDownEvent](#) that contains a `launchChr`, as shown in [Listing 4.11](#).

WARNING! Do not use the [SysUIAppSwitch](#) or [SysAppLaunch](#) functions to open the Application Launcher application.

Listing 4.11 Opening the Launcher

```
EventType newEvent;  
  
MemSet(&newEvent, sizeof(newEvent), 0);  
newEvent.eType = keyDownEvent;  
newEvent.data.keyDown.chr = launchChr;  
newEvent.data.keyDown.modifiers = commandKeyMask;  
EvtAddEventToQueue (&newEvent);
```

Note that this technique will run whatever is run whenever you tap on the Applications icon. For information on launching other applications programmatically, see [“Launching Applications Programmatically”](#) in the chapter [“Application Startup and Stop.”](#)

NOTE: Versions of Palm OS prior to 3.0 implemented the Launcher as a pop-up. The [SysAppLauncherDialog](#) function, which provides the API to the old pop-up launcher, is still present in Palm OS for compatibility purposes, but it has not been updated and, in most cases, should not be used.

Summary of User Interface API

Progress Manager Functions

PrgHandleEvent	PrgStartDialog
PrgStopDialog	PrgUpdateDialog
PrgUserCancel	

Form Functions

Initialization

[FrmInitForm](#)

Event Handling

FrmSetEventHandler	FrmDispatchEvent
FrmHandleEvent	

Displaying a Form

FrmGotoForm	FrmPopupForm
FrmDrawForm	FrmNewForm
FrmSetActiveForm	

Displaying a Modal Dialog

FrmCustomAlert	FrmHelp
FrmCustomResponseAlert	FrmSaveActiveState
FrmAlert	FrmRestoreActiveState
FrmDoDialog	FrmNewGsi

Updating the Display

FrmUpdateForm	FrmReturnToForm
FrmShowObject	FrmHideObject
FrmRemoveObject	FrmUpdateScrollers

Form Attributes

FrmVisible	FrmSaveAllForms
----------------------------	---------------------------------

User Interface

Summary of User Interface API

Form Functions

Accessing a Form Programmatically

[FrmGetActiveForm](#)

[FrmGetFirstForm](#)

[FrmGetFormPtr](#)

[FrmValidatePtr](#)

[FrmGetActiveFormID](#)

[FrmGetFormId](#)

[FrmGetWindowHandle](#)

Accessing Objects Within a Form

[FrmGetFocus](#)

[FrmGetObjectId](#)

[FrmGetObjectType](#)

[FrmGetObjectPtr](#)

[FrmSetFocus](#)

[FrmGetObjectIndex](#)

[FrmGetObjectPosition](#)

[FrmGetNumberOfObjects](#)

Title and Menu

[FrmCopyTitle](#)

[FrmPointInTitle](#)

[FrmSetMenu](#)

[FrmGetTitle](#)

[FrmSetTitle](#)

Labels

[FrmCopyLabel](#)

[FrmGetLabel](#)

[FrmSetCategoryLabel](#)

[FrmNewLabel](#)

Controls

[FrmGetControlValue](#)

[FrmGetControlGroupSelection](#)

[FrmSetControlValue](#)

[FrmSetControlGroupSelection](#)

Gadgets

[FrmGetGadgetData](#)

[FrmNewGadget](#)

[FrmSetGadgetData](#)

[FrmSetGadgetHandler](#)

Bitmaps

[FrmNewBitmap](#)

Coordinates and Boundaries

[FrmGetObjectBounds](#)

[FrmSetObjectPosition](#)

[FrmSetObjectBounds](#)

[FrmGetFormBounds](#)

Form Functions

Removing a Form From the Display

[FrmCloseAllForms](#)

[FrmEraseForm](#)

Releasing a Form's Memory

[FrmDeleteForm](#)

Window Functions

Initialization

[WinCreateWindow](#)

Making a Window Active

[WinSetActiveWindow](#)

[WinSetDrawWindow](#)

Accessing a Window Programmatically

[WinGetActiveWindow](#)

[WinGetDrawWindow](#)

[WinGetDisplayWindow](#)

[WinGetFirstWindow](#)

[WinValidateHandle](#)

Offscreen Windows

[WinRestoreBits](#)

[WinSaveBits](#)

[WinCreateOffscreenWindow](#)

[WinCreateBitmapWindow](#)

Displaying Characters

[WinDrawChar](#)

[WinDrawChars](#)

[WinInvertChars](#)

[WinDrawInvertedChars](#)

[WinDrawTruncChars](#)

[WinEraseChars](#)

[WinPaintChar](#)

[WinPaintChars](#)

Bitmaps

[WinDrawBitmap](#)

[WinGetBitmap](#)

[WinPaintBitmap](#)

User Interface

Summary of User Interface API

Window Functions

Lines

[WinDrawLine](#)

[WinFillLine](#)

[WinEraseLine](#)

[WinPaintLines](#)

[WinDrawGrayLine](#)

[WinInvertLine](#)

[WinPaintLine](#)

Rectangles

[WinDrawRectangle](#)

[WinInvertRectangle](#)

[WinFillRectangle](#)

[WinEraseRectangle](#)

[WinDrawGrayRectangleFrame](#)

[WinPaintRectangle](#)

[WinCopyRectangle](#)

[WinDrawRectangleFrame](#)

[WinInvertRectangleFrame](#)

[WinScrollRectangle](#)

[WinEraseRectangleFrame](#)

[WinPaintRectangleFrame](#)

Pixels

[WinDrawPixel](#)

[WinErasePixel](#)

[WinGetPixel](#)

[WinInvertPixel](#)

[WinPaintPixel](#)

[WinPaintPixels](#)

Clipping Rectangle

[WinGetClip](#)

[WinResetClip](#)

[WinSetClip](#)

[WinClipRectangle](#)

Setting the Drawing State

[WinPopDrawState](#)

[WinModal](#)

[WinSetPattern](#)

[WinGetPatternType](#)

[WinSetBackColor](#)

[WinSetPatternType](#)

[WinPushDrawState](#)

[WinGetPattern](#)

[WinSetUnderlineMode](#)

[WinSetDrawMode](#)

[WinSetForeColor](#)

[WinSetTextColor](#)

Coordinates and Boundaries

[WinDisplayToWindowPt](#)

[WinGetDisplayExtent](#)

[WinSetBounds](#)

[WinGetFramesRectangle](#)

[WinWindowToDisplayPt](#)

[WinGetWindowExtent](#)

[WinGetBounds](#)

[WinGetWindowFrameRect](#)

Window Functions

Working with the Screen

[WinScreenMode](#)
[WinScreenUnlock](#)

[WinScreenLock](#)

Removing a Window From the Display

[WinEraseWindow](#)

Releasing a Window's Memory

[WinDeleteWindow](#)

Working with Colors

[WinIndexToRGB](#)
[WinRGBToIndex](#)

[WinPalette](#)

[WinGetPixelRGB](#)
[WinSetBackColor](#)
[WinSetForeColor](#)
[WinSetTextColor](#)

[WinSetForeColorRGB](#)
[WinSetBackColorRGB](#)
[WinSetTextColorRGB](#)

High-Density Displays

[WinGetCoordinateSystem](#)
[WinGetSupportedDensity](#)
[WinPaintRoundedRectangleFrame](#)
[WinPaintTiledBitmap](#)
[WinScaleCoord](#)
[WinScalePoint](#)

[WinScaleRectangle](#)
[WinScreenGetAttribute](#)
[WinSetCoordinateSystem](#)
[WinUnscaleCoord](#)
[WinUnscalePoint](#)
[WinUnscaleRectangle](#)

Control Functions

Displaying a Control

[CtlShowControl](#)
[CtlSetUsable](#)
[CtlNewGraphicControl](#)

[CtlDrawControl](#)
[CtlNewControl](#)
[CtlNewSliderControl](#)

User Interface

Summary of User Interface API

Control Functions

Control's Value

[CtlGetValue](#)
[CtlGetSliderValues](#)

[CtlSetValue](#)

Label

[CtlSetLabel](#)

[CtlGetLabel](#)

Enabling/Disabling

[CtlSetEnabled](#)
[CtlHideControl](#)

[CtlEnabled](#)
[CtlEraseControl](#)

Event Handling

[CtlHandleEvent](#)

Setting up controls

[CtlGetSliderValues](#)
[CtlSetGraphics](#)

[CtlSetSliderValues](#)

Debugging

[CtlHitControl](#)

[CtlValidatePointer](#)

Field Functions

Obtaining User Input

[FldGetTextPtr](#)
[FldSetDirty](#)
[FldGetSelection](#)

[FldGetTextHandle](#)
[FldDirty](#)

Updating the Display

[FldDrawField](#)
[FldSetSelection](#)
[FldRecalculateField](#)

[FldMakeFullyVisible](#)
[FldSetBounds](#)

Displaying Text

[FldSetTextPtr](#)

Field Functions

Editing Text

[FldSetText](#)

[FldInsert](#)

[FldEraseField](#)

[FldSetTextHandle](#)

[FldDelete](#)

Cut/Copy/Paste

[FldCopy](#)

[FldPaste](#)

[FldCut](#)

[FldUndo](#)

Scrolling

[FldScrollField](#)

[FldSetScrollPosition](#)

[FldGetVisibleLines](#)

[FldGetNumberOfBlankLines](#)

[FldScrollable](#)

[FldGetScrollPosition](#)

[FldGetScrollValues](#)

Field Attributes

[FldGetAttributes](#)

[FldGetFont](#)

[FldGetMaxChars](#)

[FldSetAttributes](#)

[FldSetFont](#)

[FldSetMaxChars](#)

[FldSetMaxVisibleLines](#)

[FldGetBounds](#)

Text Attributes

[FldCalcFieldHeight](#)

[FldGetTextAllocatedSize](#)

[FldSetTextAllocatedSize](#)

[FldGetTextHeight](#)

[FldGetTextLength](#)

[FldWordWrap](#)

Working With the Insertion Point

[FldGetInsPtPosition](#)

[FldSetInsertionPoint](#)

[FldSetInsPtPosition](#)

Releasing Memory

[FldCompactText](#)

[FldFreeMemory](#)

Event Handling

[FldHandleEvent](#)

[FldSendHeightChangeNotification](#)

[FldSendChangeNotification](#)

User Interface

Summary of User Interface API

Field Functions

Dynamic UI

[FldNewField](#)

Menu Functions

[MenuDispose](#)

[MenuEraseStatus](#)

[MenuHandleEvent](#)

[MenuSetActiveMenu](#)

[MenuAddItem](#)

[MenuCmdBarDisplay](#)

[MenuHideItem](#)

[MenuDrawMenu](#)

[MenuInit](#)

[MenuGetActiveMenu](#)

[MenuSetActiveMenuRscID](#)

[MenuCmdBarAddButton](#)

[MenuCmdBarGetButtonData](#)

[MenuShowItem](#)

Table Functions

Drawing Tables

[TblDrawTable](#)

[TblSetLoadDataProcedure](#)

[TblSetCustomDrawProcedure](#)

Updating the Display

[TblRedrawTable](#)

[TblReleaseFocus](#)

[TblRemoveRow](#)

[TblMarkTableInvalid](#)

[TblUnhighlightSelection](#)

[TblGrabFocus](#)

[TblUnhighlightSelection](#)

[TblMarkRowInvalid](#)

[TblSelectItem](#)

Retrieving Data

[TblGetItemPtr](#)

[TblFindRowData](#)

[TblGetSelection](#)

[TblSetSaveDataProcedure](#)

[TblGetRowData](#)

[TblGetItemInt](#)

[TblGetCurrentField](#)

Table Functions

Displaying Data

[TblSetItemInt](#)

[TblSetItemPtr](#)

[TblSetRowData](#)

[TblSetItemStyle](#)

[TblSetRowID](#)

Retrieving a Row

[TblFindRowID](#)

[TblGetRowID](#)

Table Information

[TblEditing](#)

[TblGetItemBounds](#)

[TblGetNumberOfRows](#)

[TblHasScrollBar](#)

[TblGetBounds](#)

[TblGetLastUsableRow](#)

[TblSetBounds](#)

Row Information

[TblGetRowHeight](#)

[TblRowSelectable](#)

[TblRowUsable](#)

[TblSetRowStaticHeight](#)

[TblSetRowHeight](#)

[TblSetRowSelectable](#)

[TblSetRowUsable](#)

[TblRowInvalid](#)

Masked Records

[TblRowMasked](#)

[TblSetColumnMasked](#)

[TblSetRowMasked](#)

Column Information

[TblGetColumnSpacing](#)

[TblGetColumnWidth](#)

[TblSetColumnUsable](#)

[TblSetColumnSpacing](#)

[TblSetColumnWidth](#)

[TblSetColumnEditIndicator](#)

Removing a Table From the Display

[TblEraseTable](#)

Event Handling

[TblHandleEvent](#)

User Interface

Summary of User Interface API

Private Record Functions

[SecSelectViewStatus](#)

[SecVerifyPW](#)

List Functions

Displaying a List

[LstDrawList](#)

[LstSetDrawFunction](#)

[LstPopupList](#)

[LstNewList](#)

Updating the Display

[LstMakeItemVisible](#)

[LstSetHeight](#)

[LstSetListChoices](#)

[LstSetTopItem](#)

[LstSetSelection](#)

[LstSetPosition](#)

[LstScrollList](#)

List Data and Attributes

[LstGetNumberOfItems](#)

[LstGetTopItem](#)

[LstGetSelection](#)

[LstGetVisibleItems](#)

[LstGetSelectionText](#)

Removing a List From the Display

[LstEraseList](#)

Event Handling

[LstHandleEvent](#)

Category Functions

[CategoryCreateList](#)

[CategoryInitialize](#)

[CategoryEdit](#)

[CategorySelect](#)

[CategoryFind](#)

[CategorySetName](#)

[CategoryFreeList](#)

[CategorySetTriggerLabel](#)

[CategoryGetName](#)

[CategorySelect](#)

[CategoryGetNext](#)

[CategoryTruncateName](#)

Bitmap Functions

BmpBitsSize	BmpGetDensity
BmpColortableSize	BmpGetNextBitmapAnyDensity
BmpCompress	BmpGetTransparentValue
BmpCreate	BmpGetVersion
BmpCreateBitmapV3	BmpSetDensity
BmpDelete	BmpSetTransparentValue
BmpGetBits	BmpSize
BmpGetColortable	ColorTableEntries
BmpGetCompressionType	

Scroll Bar Functions

SclSetScrollBar	SclGetScrollBar
SclHandleEvent	SclDrawScrollBar

UI Color List Functions

UIColorGetTableEntryIndex	UIColorGetTableEntryRGB
UIColorSetTableEntry	

UI Controls

UIBrightnessAdjust	UIContrastAdjust
UIPickColor	

Insertion Point Functions

InsPtEnable	InsPtEnabled
InsPtGetHeight	InsPtSetHeight
InsPtGetLocation	InsPtSetLocation

User Interface

Summary of User Interface API

Keyboard Dialog Functions

[SysKeyboardDialog](#)

[SysKeyboardDialogV10](#)

Memory

This chapter helps you understand memory use on Palm OS®.

- [Introduction to Palm OS Memory Use](#) provides information about Palm OS hardware relevant to memory management.
- [Memory Architecture](#) discusses in detail how memory is structured on Palm OS. It also examines the structure of the basic building blocks of Palm OS memory: heaps, chunks, and records.
- [The Memory Manager](#) discusses how to use the Palm OS Memory Manager in your applications. The Memory Manager maintains the location and size of each memory chunk in nonvolatile storage, volatile storage, and ROM. It provides functions for allocating chunks, disposing of chunks, resizing chunks, locking and unlocking chunks, and compacting the heap when it becomes fragmented.

Introduction to Palm OS Memory Use

The Palm OS system software supports applications on low-cost, low-power, handhelds. Given these constraints, Palm OS is efficient in its use of both memory and processing resources. This section presents two aspects of Palm Powered™ handheld that contribute to this efficiency: [Hardware Architecture](#) and [PC Connectivity](#).

Hardware Architecture

The first implementation of Palm OS provided nearly instantaneous response to user input while running on a 16 MHz Motorola® 68000 type processor with a minimum of 128 KB of nonvolatile storage memory and 512 KB of ROM. Subsequent Palm Powered handhelds provide additional RAM and ROM in varying amounts.

The ROM and RAM for each Palm Powered handheld resides on a memory module known as a **card**. Each memory card can contain ROM, RAM, or both. A “card” is really just a logical construct used

Memory

Introduction to Palm OS Memory Use

by the operating system—Palm Powered handhelds can have one card, multiple cards, or no cards. For example, the Simulator provided by the Palm OS SDK on Macintosh can simulate a handheld that has two cards.

IMPORTANT: Do not confuse memory cards with expansion cards, such as SD cards or MemoryStick cards. You access expansion cards through a different API. See [Chapter 7](#), “[Expansion](#),” on page 207 for more information.

The main suite of applications provided with each Palm Powered handheld is built into ROM. This design permits the user to replace the operating system and the entire applications suite simply by installing a single replacement module. Additional or replacement applications and system extensions can be loaded into RAM, but doing so is not always practical in this RAM-constrained environment.

PC Connectivity

PC connectivity is an integral component of Palm Powered handhelds. The handheld comes with a cradle that connects to a desktop PC and with software for the PC that provides “one-button” backup and synchronization of all data on the handheld with the user’s PC.

Because all user data can be backed up on the PC, replacement of the nonvolatile storage area of the Palm Powered handheld becomes a simple matter of installing the new module in place of the old one and resynchronizing with the PC. The format of the user’s data in storage RAM can change with a new version of the ROM; the connectivity software on the PC is responsible for translating the data into the correct format when downloading it onto a handheld with a new ROM.

Memory Architecture

IMPORTANT: This section describes the current implementation of Palm OS memory architecture. This implementation may change as the Palm OS evolves. Do not rely on implementation-specific information described here; instead, always use the API provided to manipulate memory.

The Palm OS system software is designed around a 32-bit architecture. The system uses 32-bit addresses, and its basic data types are 8, 16, and 32 bits long.

The 32-bit addresses available to software provide a total of 4 GB of address space for storing code and data. This address space affords a large growth potential for future revisions of both the hardware and software without affecting the execution model. Although a large memory space is available, Palm OS was designed to work efficiently with small amounts of RAM. For example, the first commercial Palm Powered handheld has less than 1 MB of memory, or 0.025% of this address space.

The Motorola 68328 processor's 32-bit registers and 32 internal address lines support a 32-bit execution model as well, although the external data bus is only 16 bits wide. This design reduces cost without impacting the software model. The processor's bus controller automatically breaks down 32-bit reads and writes into multiple 16-bit reads and writes externally.

Each memory card in the Palm Powered handheld has 256 MB of address space reserved for it. Memory card 0 starts at address \$1000000, memory card 1 starts at address \$2000000, and so on.

The Palm OS divides the total available RAM store into two logical areas: **dynamic** RAM and **storage** RAM. Dynamic RAM is used as working space for temporary allocations, and is analogous to the RAM installed in a typical desktop system. The remainder of the available RAM on the card is designated as storage RAM and is analogous to disk storage on a typical desktop system.

Because power is always applied to the memory system, both areas of RAM preserve their contents when the handheld is turned "off" (i.e., is in low-power sleep mode). See "[Palm OS Power Modes](#)" in

Memory

Memory Architecture

the chapter “[Palm System Support](#)” in this book. All of storage memory is preserved even when the handheld is reset explicitly. As part of the boot sequence, the system software reinitializes the dynamic area, and leaves the storage area intact.

The entire dynamic area of RAM is used to implement a single heap that provides memory for dynamic allocations. From this **dynamic heap**, the system provides memory for dynamic data such as global variables, system dynamic allocations (TCP/IP, IrDA, and so on, as applicable), application stacks, temporary memory allocations, and application dynamic allocations (such as those performed when the application calls the [MemHandleNew](#) function).

The entire amount of RAM reserved for the dynamic heap is always dedicated to this use, regardless of whether it is actually used for allocations. The size of the dynamic area of RAM on a particular handheld varies according to the OS version running, the amount of physical RAM available, and the requirements of pre-installed software such as the TCP/IP stack or IrDA stack. [Table 5.1](#) provides more information about the dynamic heap space that currently available combinations of OS and hardware provide.

Table 5.1 Dynamic Heap Space

RAM Usage	≥ OS 3.5 ≤ 4 MB TCP/IP & IrDA	≥ OS 3.5 ≤ 2 MB TCP/IP & IrDA	OS 3.0 > 3.3 > 1 MB TCP/IP & IrDA (Palm III™)	OS 2.0 1 MB TCP/IP only (Professional)	OS 2.0/1.0 512 KB no TCP/IP or IrDA (Personal)
Total dynamic area	256 KB	128 KB	96 KB	64 KB	32 KB
System Globals (screen buffer, UI globals, database references, etc.)	40 KB (OS)	40 KB (OS)	~2.5 KB	~2.5 KB	~2.5 KB
TCP/IP stack	32 KB	32 KB	32 KB	32 KB	0 KB
System dynamic allocation (IrDA, “Find” window, temporary allocations)	variable	variable	variable amount	~15 KB (no IrDA in this OS)	~15 KB

Table 5.1 Dynamic Heap Space (*continued*)

RAM Usage	≥ OS 3.5 ≤ 4 MB TCP/IP & IrDA	≥ OS 3.5 ≤ 2 MB TCP/IP & IrDA	OS 3.0 > 3.3 > 1 MB TCP/IP & IrDA (Palm III™)	OS 2.0 1 MB TCP/IP only (Professional)	OS 2.0/1.0 512 KB no TCP/IP or IrDA (Personal)
Application stack (call stack and local vars)	N/A (see note)	N/A (see note)	4 KB (default)	2.5 KB	2.5 KB
Remaining dynamic space (dynamic allocations, application global variables, and static variables)	184 KB	56 KB	≤ 36 KB	≤ 12 KB	≤ 12 KB

NOTE: Starting with Palm OS 3.5, the dynamic heap is sized based on the amount of memory available to the system.

The remaining portion of RAM not dedicated to the dynamic heap is configured as one or more **storage heaps** used to hold nonvolatile user data such as appointments, to do lists, memos, address lists, and so on. An application accesses a storage heap by calling the Data Manager or Resource Manager, according to whether it needs to manipulate user data or resources.

The size and number of storage heaps available on a particular handheld varies according to the OS version that is running; the amount of physical RAM that is available; and the storage requirements of end-user application software such as the Address Book, Date Book, or third-party applications.

Versions 1.0 and 2.0 of Palm OS subdivide storage RAM into multiple storage heaps of 64 KB each. Palm OS 3.0 and later configure all storage RAM on a card as a single storage heap. Under all versions of Palm OS, system overhead limits the maximum usable data storage available in a single chunk to slightly less than 64 KB.

Memory

Memory Architecture

In the Palm OS environment, all data are stored in Memory Manager chunks. A **chunk** is an area of contiguous memory between 1 byte and slightly less than 64 KB in size that has been allocated by the Palm OS Memory Manager. (Because system overhead requirements may vary, an exact figure for the maximum amount of usable data storage for all chunks cannot be specified.) Currently, all Palm OS implementations limit the maximum size of any chunk to slightly less than 64 KB; however, the API does not have this constraint, and it may be relaxed in the future.

Each chunk resides in a heap. Some heaps are ROM-based and contain only nonmovable chunks; some are RAM-based and may contain movable or nonmovable chunks. A RAM-based heap may be a dynamic heap or a storage heap. The Palm OS Memory Manager allocates memory in the dynamic heap (for dynamic allocations, stacks, global variables, and so on). The Palm OS Data Manager allocates memory in one or more storage heaps (for nonvolatile user data).

Every memory chunk used to hold storage data (as opposed to memory chunks that store dynamic data) is a **record** in a database implemented by the Palm OS Data Manager. In the Palm OS environment, a **database** is simply a list of memory chunks and associated database header information. Normally, the items in a database share some logical association; for example, a database may hold a collection of all address book entries, all datebook entries, and so on.

A database is analogous to a file in a desktop system. Just as a traditional file system can create, delete, open, and close files, Palm OS applications can create, delete, open, and close databases as necessary. There is no restriction on where the records for a particular database reside as long as they are all on the same memory card. The records from one database can be interspersed with the records from one or more other databases in memory.

Storing data by database fits nicely with the Palm OS Memory Manager design. Each record in a database is in fact a Memory Manager chunk. The Data Manager can use Memory Manager calls to allocate, delete, and resize database records. All heaps except for the dynamic heap are nonvolatile, so database records can be stored in any heap except the dynamic heap. Because records can be stored

anywhere on the memory card, databases can be distributed over multiple discontinuous areas of physical RAM, but all records belonging to a particular database must reside on the same card.

To understand how database records are manipulated, it helps to know something about the way the Memory Manager allocates and tracks memory chunks, as the next section describes.

Heap Overview

IMPORTANT: This section describes the current implementation of Palm OS memory architecture. This implementation may change as the Palm OS evolves. Do not rely on implementation-specific information described here; instead, always use the API provided to manipulate memory.

Recall that a **heap** is a contiguous area of memory used to contain and manage one or more smaller chunks of memory. When applications work with memory (allocate, resize, lock, etc.) they usually work with chunks of memory. An application can specify whether to allocate a new chunk of memory in the storage heap or the dynamic heap. The Memory Manager manages each heap independently and rearranges chunks as necessary to defragment heaps and merge free space.

Heaps in the Palm OS environment are referenced through heap IDs. A **heap ID** is a unique 16-bit value that the Memory Manager uses to identify a heap within the Palm OS address space. Heap IDs start at 0 and increment sequentially by units of 1. Values are assigned beginning with the RAM heaps on card 0, continuing with the ROM heaps on card 0, and then continuing through RAM and ROM heaps on subsequent cards. The sequence of heap IDs is continuous; that is, no values in the sequence are skipped.

The first heap (heap 0) on card 0 is the dynamic heap. This heap is reinitialized every time the Palm Powered handheld is reset. When an application quits, the system frees any chunks allocated by that application in the dynamic heap. All other heaps are nonvolatile storage heaps that retain their contents through soft reset cycles.

Memory

Memory Architecture

When a Palm Powered handheld is presented with multiple dynamic heaps, the first heap (heap 0) on card 0 is the active dynamic heap. All other potential dynamic heaps are ignored. For example, it is possible that a future Palm Powered handheld supporting multiple cards might be presented with two cards, each having its own dynamic heap; if so, only the dynamic heap residing on card 0 would be active—the system would not treat any heaps on other cards as dynamic heaps, nor would heap IDs be assigned to these heaps. Subsequent storage heaps would be assigned IDs in sequential order, as always beginning with RAM heaps, followed by ROM heaps.

NOTE: In Palm OS 3.5, the dynamic heap is sized based on the amount of memory available to the system.

Overview of Memory Chunk Structure

Memory chunks can be movable or nonmovable. Applications need to store data in movable chunks whenever feasible, thereby enabling the Memory Manager to move chunks as necessary to create contiguous free space in memory for allocation requests.

When the Memory Manager allocates a nonmovable chunk it returns a pointer to that chunk. The pointer is simply that chunk's address in memory. Because the chunk cannot move, its pointer remains valid for the chunk's lifetime; thus, the pointer can be passed "as is" to the caller that requested the allocation.

When the Memory Manager allocates a moveable chunk, it generates a pointer to that chunk, just as it did for the nonmovable chunk, but it does not return the pointer to the caller. Instead, it stores the pointer to the chunk, called the **master chunk pointer**, in a **master pointer table** that is used to track all of the moveable chunks in the heap, and returns a reference to the master chunk pointer. This reference to the master chunk pointer is known as a **handle**. It is this handle that the Memory Manager returns to the caller that requested the allocation of a moveable chunk.

Using handles imposes a slight performance penalty over direct pointer access but permits the Memory Manager to move chunks around in the heap without invalidating any chunk references that

an application might have stored away. As long as an application uses handles to reference data, only the master pointer to a chunk needs to be updated by the Memory Manager when it moves a chunk during defragmentation.

An application typically locks a chunk handle for a short time while it has to read or manipulate the contents of the chunk. The process of locking a chunk tells the Memory Manager to mark that data chunk as immobile. When an application no longer needs the data chunk, it should unlock the handle immediately to keep heap fragmentation to a minimum.

Note that any handle is good only until the system is reset. When the system resets, it reinitializes all dynamic memory areas and relaunches applications. Therefore, you must not store a handle in a database record or a resource.

Each chunk on a memory card is actually located by means of a card-relative reference called a **local ID**. A local ID is a reference to a data chunk that the system computes from the base address of the card. The local ID of a nonmovable chunk is simply the offset of the chunk from the base address of the card. The local ID of a movable chunk is the offset of the master pointer to the chunk from the base address of the card, but with the low-order bit set. Since chunks are always aligned on word boundaries, only local IDs of movable chunks have the low-order bit set. Once the base address of the card is determined at runtime, a local ID can be converted quickly to a pointer or handle.

For example, when an application needs the handle to a particular data record, it calls the Data Manager to retrieve the record by index from the appropriate database. The Data Manager fetches the local ID of the record out of the database header and uses it to compute the handle to the record. The handle to the record is never actually stored in the database itself.

Although currently available Palm Powered handhelds do not provide hardware support for multiple cards, the use of local IDs provides support in software for future handhelds that may allow the user to remove or insert memory cards. If the user moves a memory card to a slot having a different base address, the handle to a memory chunk on that card is likely to change, but the local ID associated with that chunk does not change.

IMPORTANT: Do not confuse memory cards with expansion cards, such as SD cards or MemoryStick cards. You access expansion cards through a different API. See [Chapter 7](#), “[Expansion](#),” on page 207 for more information.

The Memory Manager

The Palm OS Memory Manager is responsible for maintaining the location and size of every memory chunk in nonvolatile storage, volatile storage, and ROM. It provides an API for allocating new chunks, disposing of chunks, resizing chunks, locking and unlocking chunks, and compacting heaps when they become fragmented. Because of the limited RAM and processor resources of the Palm Powered handheld, the Memory Manager is efficient in its use of processing power and memory.

This section provides background information on the organization of memory in Palm OS and provides an overview of the Memory Manager API, discussing these topics:

- [Memory Manager Structures](#)
- [Using the Memory Manager](#)

Memory Manager Structures

This section discusses the different structures the Memory Manager uses:

- [Heap Structures](#)
- [Chunk Structures](#)
- [Local ID Structures](#)

Heap Structures

IMPORTANT: Expect the heap structure to change in the future. Use the API to work with heaps.

A heap consists of the heap header, master pointer table, and the heap chunks.

- **Heap header.** The heap header is located at the beginning of the heap. It holds the size of the heap and contains flags for the heap that provide certain information to the Memory Manager; for example, whether the heap is ROM-based.
- **Master pointer table.** Following the heap header is a master pointer table. It is used to store 32-bit pointers to movable chunks in the heap.
 - When the Memory Manager moves a chunk to compact the heap, the pointer for that chunk in the master pointer table is updated to the chunk's new location. The handles an application uses to track movable chunks reference the address of the master pointer to the chunk, not the chunk itself. In this way, handles remain valid even after a chunk is moved. The OS compacts the heap automatically when available contiguous space is not sufficient to fulfill an allocation request.
 - If the master pointer table becomes full, another is allocated and its offset is stored in the `nextMstrPtrTable` field of the previous master pointer table. Any number of master pointer tables can be linked in this way. Because additional master pointer chunks are nonmovable, they are allocated at the end of the heap, according to the guidelines described in the "Heap chunks" section following immediately.
- **Heap chunks.** Following the master pointer table are the actual chunks in the heap.
 - Movable chunks are generally allocated at the beginning of the heap, and nonmovable chunks at the end of the heap.
 - Nonmovable chunks do not need an entry in the master pointer table since they are never relocated by the Memory Manager.
 - Applications can easily walk the heap by hopping from chunk to chunk because each chunk header contains the size of the chunk. All free and nonmovable chunks can be found in this manner by checking the flags in each chunk header.

Because heaps can be ROM-based, there is no information in the header that must be changed when using a heap. Also, ROM-based heaps contain only nonmovable chunks and have a master pointer table with 0 entries.

Chunk Structures

IMPORTANT: Expect the chunk structure to change in the future. Use the API to work with chunks.

Each chunk begins with an 8-byte header followed by that chunk's data. The chunk header consists of a `Flags: size` adjustment byte, 3 bytes of size information, a `lock: owner` byte, and 3 bytes of `hOffset` information.

- **Flags: sizeAdj byte.** This byte contains flags in the high nibble and a size adjustment in the low nibble.
 - The flags nibble has 1 bit currently defined, which is set for free chunks.
 - The size adjustment nibble can be used to calculate the requested size of the chunk, given the actual size. The requested size is computed by taking the size as stored in the chunk header and subtracting the size of the header and the size adjustment field. The actual size of a chunk is always a multiple of two so that chunks always start on a word boundary.
- **size field (3 bytes).** This three-byte value describes the size of the chunk, which is **larger** than the size requested by the application and includes the size of the chunk header itself. The maximum data size for a chunk is slightly less than 64 KB.
- **Lock: owner byte.** Following the size information is a byte that holds the lock count in the high nibble and the owner ID in the low nibble.
 - The lock count is incremented every time a chunk is locked and decremented every time a chunk is unlocked. A movable chunk can be locked a maximum of 14 times before being unlocked. Nonmovable chunks always have 15 in the lock field.

- The owner ID determines the owner of a memory chunk and is set by the Memory Manager when allocating a new chunk. Owner ID information is useful for debugging and for garbage collection when an application terminates abnormally.
- **hoffset field (3 bytes).** The last three bytes in the chunk header is the distance from the master pointer for the chunk to the chunk's header, divided by two. Note that this offset could be a negative value if the master pointer table is at a higher address than the chunk itself. For nonmovable chunks that do not need an entry in the master pointer table, this field is 0.

Local ID Structures

IMPORTANT: Expect the local ID structure to change in the future. Use the API to work with chunks.

Chunks that contain database records or other database information are tracked by the Data Manager through local IDs. A local ID is card relative and is always valid no matter what memory slot the card resides in. A local ID can be easily converted to a pointer or the handle to a chunk once the base address of the card is known.

The upper 31 bits of a local ID contain the offset of the chunk or master pointer to the chunk from the beginning of the card. The low-order bit is set for local IDs of handles and clear for local IDs of pointers.

The [MemLocalIDToGlobal](#) function converts a local ID and card number (either 0 or 1) to a pointer or handle. It looks at the card number and adds the appropriate card base address to convert the local ID to a pointer or handle for that card.

Using the Memory Manager

Use the Memory Manager API to allocate memory in the dynamic heap (for dynamic allocations, stacks, global variables, and so on) and use the Data Manager API to allocate memory in one or more storage heaps (for user data). The Data Manager calls the Memory

Memory

The Memory Manager

Manager as appropriate to perform low-level allocations. (See [The Data Manager](#) for more information.)

Overview of the Memory Manager API

To allocate a movable chunk, call [MemHandleNew](#) and pass the desired chunk size. Before you can read or write data to this chunk, you must call [MemHandleLock](#) to lock it and get a pointer to it. Every time you lock a chunk, its lock count is incremented. You can lock a chunk a maximum of 14 times before an error is returned. (Recall that unmovable chunks hold the value 15 in the lock field.) [MemHandleUnlock](#) reverses the effect of [MemHandleLock](#)—it decrements the value of the lock field by 1. When the lock count is reduced to 0, the chunk is free to be moved by the Memory Manager.

When an application allocates memory in the dynamic heap, the Memory Manager uses an owner ID to associate that chunk with the application. The system further distinguishes chunks belonging to the currently active allocation by setting a special bit in the owner ID information. When the application quits, all chunks in the dynamic heap having this bit set are freed automatically.

If the system needs to allocate a chunk that is not disposed of when an application quits, it changes the chunk's owner ID to 0 by calling the system functions [MemHandleSetOwner](#) or [MemPtrSetOwner](#). These functions are not generally used by applications, except in special circumstances. For example, when the current application is passing a parameter block to a new application that it is launching with [SysUIAppSwitch](#), the owner of the block must be set to the system; otherwise, when the current application exits, the system deletes the block when it frees all memory allocated by the current application.

To determine the size of a movable chunk, pass its handle to [MemHandleSize](#). To resize it, call [MemHandleResize](#). You generally cannot increase the size of a chunk if it's locked unless there happens to be free space in the heap immediately following the chunk. If the chunk is unlocked, the Memory Manager is allowed to move it to another area of the heap to increase its size. When you no longer need the chunk, call [MemHandleFree](#), which releases the chunk even if it is locked.

If you have a pointer to a locked, movable chunk, you can recover the handle by calling [MemPtrRecoverHandle](#). In fact, all of the `MemPtrXxx` calls, including [MemPtrSize](#), also work on pointers to locked, movable chunks.

To allocate a nonmovable chunk, call [MemPtrNew](#) and pass the desired size of the chunk. This call returns a pointer to the chunk, which can be used directly to read or write to it.

NOTE: You cannot allocate a zero-size chunk.

To determine the size of a nonmovable chunk, call `MemPtrSize`. To resize it, call [MemPtrResize](#). You generally can't increase the size of a nonmovable chunk unless there is free space in the heap immediately following the chunk. When you no longer need the chunk, call [MemPtrFree](#), which releases the chunk even if it's locked.

Use the Memory Manager utility routines [MemMove](#) and [MemSet](#) to move memory from one place to another or to fill memory with a specific value.

In most situations, the proper way to free memory is by calling one of the [MemPtrFree](#) or [MemHandleFree](#) functions.

NOTE: For important cautions and practical advice regarding the proper use of memory on Palm Powered handhelds, be sure to read "[Writing Robust Code](#)" on page 9 in [Chapter 1](#), "[Programming Palm OS in a Nutshell](#)," in this book.

Storage Heap Sizes and Memory Management Schemes

In Palm OS version 1.0, individual storage heaps were limited to a maximum size of 64 KB each and the Memory Manager moved objects automatically among multiple storage heaps to prevent any of them from becoming too full. This strategy tended to decrease the availability of contiguous space for large objects. The version 2.0 Memory Manager abandoned this approach, increasing the availability of contiguous heap space; however, it still limited the maximum size of individual heaps to 64 KB each. Palm OS version 3.0 removed the 64 KB maximum size restriction on individual

Memory

The Memory Manager

heaps and creates just two heaps: one 96 KB dynamic heap and one storage heap that is the size of all remaining RAM on the card.

Starting with Palm OS 3.5, the dynamic heap is sized based on the amount of memory available to the system, as follows:

Device RAM size	Heap size
< 2 MB of ram	64 KB
>= 2 MB	128 KB
>= 4 MB	256 KB

Achieving Optimum Performance

Because the Palm Powered handheld has limited heap space and storage, optimization is critical. To make your application as fast and efficient as possible, optimize for heap space first, speed second, code size third.

Follow these guidelines to optimize memory use:

- Allocate handles for your memory to avoid heap fragmentation. That is, use [MemHandleNew](#) to allocate memory rather than [MemPtrNew](#) as much as possible.
- Sort on demand; don't keep different sort lists around. This makes your program simpler and requires less storage.
- Dynamic memory is a potential bottleneck. Don't put large structures on the stack.
- Arrange subroutines within the application to avoid 32K jumps.

Because Palm OS applications must perform well in a RAM-constrained environment, proper code segmentation is critical to achieving optimum performance.

If your application segments are too large, your application may not perform well (or to run at all) when large contiguous blocks of memory are not available. Conversely, if your application segments are too small, performance may be

hindered by the overhead required to find and load resources too frequently.

Unfortunately, it's impossible to specify a single size for memory chunks that will perform optimally for all applications. You will need to experiment with segmenting your code in different ways while measuring your application's performance in order to discover the size and arrangement of resource chunks that will optimize your particular application's responsiveness and overall performance. The Metrowerks CodeWarrior Debugger, Palm OS Debugger, and the Simulator provide tools for examining the internal structure of heaps, viewing the amount of free space available, manipulating blocks, and so on.

- To have your application run well within the constraints of the limited dynamic heap, follow these guidelines:
 - Allocate memory chunks instead of using global variables where possible.
 - Switch from one UI form to another instead of stacking up dialogs. To accomplish this, use [FrmGotoForm](#) to switch to forms and [FrmDoDialog](#) to switch to modal dialogs. Avoid [FrmPopupForm](#).
 - Edit database records in place; don't make extra copies on the dynamic heap.
- Avoid placing large amounts of data on the stack. Heap corruption is hard to debug. Global variables are preferable to local variables; however, chunks are preferable to global variables. Your application has a limited amount of stack space depending on the system software version.

Memory

Summary of Memory Management

Summary of Memory Management

Memory Manager Functions

Allocating and Freeing Memory

MemHandleNew	MemPtrNew
MemHandleLock	MemHandleUnlock
MemLocalIDToLockedPtr	MemPtrUnlock
MemHandleFree	MemPtrFree

Resizing Chunks

MemHandleResize	MemHandleSize
MemPtrResize	MemPtrSize
MemHeapFreeBytes	MemHeapSize

Working With Memory

MemMove	MemSet
MemCmp	MemHeapCompact

Converting Pointers

MemPtrRecoverHandle	MemHandleToLocalID
MemLocalIDKind	MemLocalIDToGlobal
MemPtrToLocalID	MemLocalIDToPtr

Chunk Information

MemHandleCardNo	MemHandleDataStorage
MemHandleHeapID	MemHandleSetOwner
MemPtrCardNo	MemPtrDataStorage
MemPtrSetOwner	

Heap Information

MemPtrHeapID	MemHeapID
MemHeapDynamic	MemHeapCheck
MemHeapFlags	

Memory Manager Functions

Card Information[MemCardInfo](#)[MemNumHeaps](#)[MemStoreInfo](#)[MemNumCards](#)[MemNumRAMHeaps](#)**Debugging**[MemDebugMode](#)[MemSetDebugMode](#)[MemHeapScramble](#)

Files and Databases

This chapter describes how to work with databases using Palm OS® managers.

- [The Data Manager](#) manages user data, which is stored in databases for convenient access.
- [The Resource Manager](#) can be used by applications to conveniently retrieve and save chunks of data. It's similar to the Data Manager, but has the added capability of tagging each chunk with a unique resource type and ID. These tagged data chunks, called resources, are stored in resource databases. Resources are typically used to store the application's user interface elements, such as images, fonts, or dialog layouts.
- [File Streaming Application Program Interface](#) can be used by applications to handle large blocks of data.

IMPORTANT: To access data or resources on secondary storage (such as expansion cards), you use a different set of APIs. See [Chapter 7](#), “[Expansion](#),” on page 207 for more information.

The Data Manager

A traditional file system first reads all or a portion of a file into a memory buffer from disk, using and/or updating the information in the memory buffer, and then writes the updated memory buffer back to disk. Because Palm Powered™ handhelds have limited amounts of dynamic RAM and use nonvolatile RAM instead of disk storage, a traditional file system is not optimal for storing and retrieving Palm OS user data.

Palm OS accesses and updates all information in place. This works well because it reduces dynamic memory requirements and

eliminates the overhead of transferring the data to and from another memory buffer involved in a file system.

As a further enhancement, data in the Palm Powered handheld is broken down into multiple, finite-size **records** that can be left scattered throughout the memory space; thus, adding, deleting, or resizing a record does not require moving other records around in memory. Each record in a database is in fact a Memory Manager chunk. The Data Manager uses Memory Manager functions to allocate, delete, and resize database records.

This section explains how to use the Data Manager by discussing these topics:

- [Records and Databases](#)
- [Structure of a Database Header](#)
- [Using the Data Manager](#)

Records and Databases

Databases organize related records; every record belongs to one and only one database. A database may be a collection of all address book entries, all datebook entries, and so on. A Palm OS application can create, delete, open, and close databases as necessary, just as a traditional file system can create, delete, open, and close a traditional file. There is no restriction on where the records for a particular database reside as long as they all reside on the same memory card. The records from one database can be interspersed with the records from one or more other databases in memory.

Storing data by database fits nicely with the Palm OS Memory Manager design. All heaps except for the dynamic heap(s) are nonvolatile, so database records can be stored in any heap except the dynamic heap(s) (see “[Heap Overview](#)” in the “[Memory](#)” chapter). Because records can be stored anywhere on the memory card, databases can be distributed over multiple discontinuous areas of physical RAM.

Accessing Data With Local IDs

A database maintains a list of all records that belong to it by storing the local ID of each record in the database header. Because local IDs are used, the memory card can be placed into any memory slot of a

Palm Powered handheld. An application finds a particular record in a database by index. When an application requests a particular record, the Data Manager fetches the local ID of the record from the database header by index, converts the local ID to a handle using the card number that contains the database header, and returns the handle to the record.

Structure of a Database Header

A database header consists of some basic database information and a list of records in the database. Each record entry in the header has the local ID of the record, 8 attribute bits, and a 3-byte unique ID for the record.

This section provides information about database headers, discussing these topics:

- [Database Header Fields](#)
- [Structure of a Record Entry in a Database Header](#)

IMPORTANT: Expect the database header structure to change in the future. Use the API to work with database structures.

Database Header Fields

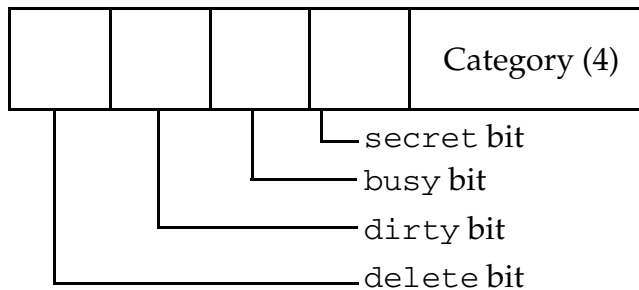
The database header has the following fields:

- The `name` field holds the name of the database.
- The `attributes` field has flags for the database.
- The `version` field holds an application-specific version number for that database.
- The `modificationNumber` is incremented every time a record in the database is deleted, added, or modified. Thus applications can quickly determine if a shared database has been modified by another process.
- The `appInfoID` is an optional field that an application can use to store application-specific information about the database. For example, it might be used to store user display preferences for a particular database.

- The `sortInfoID` is another optional field an application can use for storing the local ID of a sort table for the database.
- The `type` and `creator` fields are each 4 bytes and hold the database type and creator. The system uses these fields to distinguish application databases from data databases and to associate data databases with the appropriate application.
- The `numRecords` field holds the number of record entries stored in the database header itself. If all the record entries cannot fit in the header, then `nextRecordList` has the local ID of a `recordList` that contains the next set of records.

Each record entry stored in a record list has three fields and is 8 bytes in length. Each entry has the local ID of the record which takes up 4 bytes: 1 byte of attributes and a 3-byte unique ID for the record. The `attribute` field, shown in [Figure 6.1](#), is 8 bits long and contains 4 flags and a 4-bit category number. The category number is used to place records into user-defined categories like “business” or “personal.”

Figure 6.1 Record Attributes



Structure of a Record Entry in a Database Header

Each record entry has the local ID of the record, 8 attribute bits, and a 3-byte unique ID for the record.

- Local IDs make the database slot-independent. Since all records for a database reside on the same memory card as the header, the handle of any record in the database can be quickly calculated. When an application requests a specific

record from a database, the Data Manager returns a handle to the record that it determines from the stored local ID.

A special situation occurs with ROM-based databases. Because ROM-based heaps use nonmovable chunks exclusively, the local IDs to records in a ROM-based database are local IDs of pointers, not handles. So, when an application opens a ROM-based database, the Data Manager allocates and initializes a fake handle for each record and returns the appropriate fake handle when the application requests a record. Because of this, applications can use handles to access both RAM- and ROM-based database records.

- The unique ID must be unique for each record within a database. It remains the same for a particular record no matter how many times the record is modified. It is used during synchronization with the desktop to track records on the Palm Powered handheld with the same records on the desktop system.

When the user deletes or archives a record on Palm OS:

- The `delete` bit is set in the `attributes` flags, but its entry in the database header remains until the next synchronization with the PC.
- The `dirty` bit is set whenever a record is updated.
- The `busy` bit is set when an application currently has a record locked for reading or writing.
- The `secret` bit is set for records that should not be displayed before the user password has been entered on the handheld.

When a user “deletes” a record on the Palm Powered handheld, the record’s data chunk is freed, the local ID stored in the record entry is set to 0, and the `delete` bit is set in the `attributes`. When the user archives a record, the `deleted` bit is also set but the chunk is not freed and the local ID is preserved. This way, the next time the user synchronizes with the desktop system, the desktop can quickly determine which records to delete (since their record entries are still around on the Palm Powered handheld). In the case of archived records, the desktop can save the record data on the PC before it permanently removes the record entry and data from the Palm

Powered handheld. For deleted records, the PC just has to delete the same record from the PC before permanently removing the record entry from the Palm Powered handheld.

Using the Data Manager

Using the Data Manager is similar to using a traditional file manager, except that the data is broken down into multiple records instead of being stored in one contiguous chunk. To create or delete a database, call [DmCreateDatabase](#) and [DmDeleteDatabase](#).

Each memory card is akin to a disk drive and can contain multiple databases. To open a database for reading or writing, you must first get the database ID, which is simply the local ID of the database header. Calling [DmFindDatabase](#) searches a particular memory card for a database by name and returns the local ID of the database header. Alternatively, calling [DmGetDatabase](#) returns the database ID for each database on a card by index.

After determining the database ID, you can open the database for read-only or read/write access. When you open a database, the system locks down the database header and returns a reference to a database access structure, which tracks information about the open database and caches certain information for optimum performance. The database access structure is a relatively small structure (less than 100 bytes) allocated in the dynamic heap that is disposed of when the database is closed.

Call [DmDatabaseInfo](#), [DmSetDatabaseInfo](#), and [DmDatabaseSize](#) to query or set information about a database, such as its name, size, creation and modification dates, attributes, type, and creator.

Call [DmGetRecord](#), [DmQueryRecord](#), and [DmReleaseRecord](#) when viewing or updating a database.

- [DmGetRecord](#) takes a record index as a parameter, marks the record busy, and returns a handle to the record. If a record is already busy when [DmGetRecord](#) is called, an error is returned.
- [DmQueryRecord](#) is faster if the application only needs to view the record; it doesn't check or set the busy bit, so it's not

necessary to call `DmReleaseRecord` when finished viewing the record.

- [`DmReleaseRecord`](#) clears the busy bit, and updates the modification number of the database and marks the record dirty if the `dirty` parameter is true.

To resize a record to grow or shrink its contents, call [`DmResizeRecord`](#). This routine automatically reallocates the record in another heap of the same card if the current heap does not have enough space for it. Note that if the Data Manager needs to move the record into another heap to resize it, the handle to the record changes. [`DmResizeRecord`](#) returns the new handle to the record.

To add a new record to a database, call [`DmNewRecord`](#). This routine can insert the new record at any index position, append it to the end, or replace an existing record by index. It returns a handle to the new record.

There are three methods for removing a record: `DmRemoveRecord`, `DmDeleteRecord`, and `DmArchiveRecord`.

- [`DmRemoveRecord`](#) removes the record's entry from the database header and disposes of the record data.
- [`DmDeleteRecord`](#) also disposes of the record data, but instead of removing the record's entry from the database header, it sets the deleted bit in the record entry attributes field and clears the local chunk ID.
- [`DmArchiveRecord`](#) does not dispose of the record's data; it just sets the deleted bit in the record entry.

Both `DmDeleteRecord` and `DmArchiveRecord` are useful for synchronizing information with a desktop PC. Since the unique ID of the deleted or archived record is still kept in the database header, the desktop PC can perform the necessary operations on its own copy of the database before permanently removing the record from the Palm OS database.

Call [`DmRecordInfo`](#) and [`DmSetRecordInfo`](#) to retrieve or set the record information stored in the database header, such as the attributes, unique ID, and local ID of the record. Typically, these routines are used to set or retrieve the category of a record, which is

stored in the lower four bits of the record's attribute field (see [Listing 6.1](#)).

Listing 6.1 Determining the category for a record

```
UInt16 category;

DmRecordInfo (MyDB, CurrentRecord, &attr, NULL, NULL);
category = attr & dmRecAttrCategoryMask;
//category now contains the index of the category to which
// CurrentRecord belongs.
```

To move records from one index to another or from one database to another, call [DmMoveRecord](#), [DmAttachRecord](#), and [DmDetachRecord](#). `DmDetachRecord` removes a record entry from the database header and returns the record handle. Given the handle of a new record, `DmAttachRecord` inserts or appends that new record to a database or replaces an existing record with the new record. `DmMoveRecord` is an optimized way to move a record from one index to another in the same database.

Data Manager Tips

Working properly with databases makes your application run faster and synchronize without problems. Follow these suggestions:

- Database names can be up to 31 characters in length, and on the handheld can be composed of any valid ASCII characters. Conduits—in particular, the backup conduit—impose additional limitations, however. The following characters are replaced with an underscore (“_”) when the database is transferred to the desktop by the backup conduit:

* + , . / : ; < = > ? [] | \ ^ “

As well, the backup conduit stores databases in case-insensitive format, so you should avoid filenames that depend on case for distinction.

By convention, filename extensions are not used on the handheld. Instead, database types are used to identify databases as members of a certain type or class. Note that when the backup conduit transfers a file to the desktop, it automatically appends a .pdb or .prc extension, as

appropriate, to the database filename. This extension is removed when the file is transferred back to the handheld.

- When the user deletes a record, call [DmDeleteRecord](#) to remove all data from the record, not [DmRemoveRecord](#) to remove the record itself. That way, the desktop application can retrieve the information that the record is deleted the next time there is a HotSync.

Note: If your application doesn't have an associated conduit, call `DmRemoveRecord` to completely remove the record.

- Keep data in database records compact. To avoid performance problems, Palm OS databases are not compressed, but all data are tightly packed. This pays off for storage and during HotSync operations.
- All records in a database should be of the same type and format. This is not a requirement, but is highly recommended to avoid processing overhead.
- Be sure your application modifies the flags in the database header appropriately when the user deletes or otherwise modifies information. This flag modification is only required if you're synchronizing with the Palm PIM applications.
- Don't display deleted records.
- Call [DmSetDatabaseInfo](#) when creating a database to assign a version number to your application. Databases default to version 0 if the version isn't explicitly set.
- Call [DmDatabaseInfo](#) to check the database version at application start-up.

The Resource Manager

Applications can use the Resource Manager much like the Data Manager to retrieve and save chunks of data conveniently. The Resource Manager has the added capability of tagging each chunk of data with a unique resource type and resource ID. These tagged data chunks, called **resources**, are stored in resource databases. Resource databases are almost identical in structure to normal databases except for a slight amount of increased storage overhead per resource record (two extra bytes). In fact, the Resource Manager

is nothing more than a subset of routines in the Data Manager that are broken out here for conceptual reasons only.

Resources are typically used to store the user interface elements of an application, such as images, fonts, dialog layouts, and so forth. Part of building an application involves creating these resources and merging them with the actual executable code. In the Palm OS environment, an application is, in fact, simply a resource database with the executable code stored as one or more code resources and the graphics elements and other miscellaneous data stored in the same database as other resource types.

Applications may also find the Resource Manager useful for storing and retrieving application preferences, saved window positions, state information, and so forth. These preferences settings can be stored in a separate resource database.

This section explains how to work with the Resource Manager and discusses these topics:

- [Structure of a Resource Database Header](#)
- [Using the Resource Manager](#)
- [Resource Manager Functions](#)

Structure of a Resource Database Header

A resource database header consists of some general database information followed by a list of resources in the database. The first portion of the header is identical in structure to a normal database header. Resource database headers are distinguished from normal database headers by the `dmHdrAttrResDB` bit in the `attributes` field.

IMPORTANT: Expect the resource database header structure to change in the future. Use the API to work with resource database structures.

- The `name` field holds the name of the resource database.
- The `attributes` field has flags for the database and always has the `dmHdrAttrResDB` bit set.

- The `modificationNumber` is incremented every time a resource in the database is deleted, added, or modified. Thus, applications can quickly determine if a shared resource database has been modified by another process.
- The `appInfoID` and `sortInfoID` fields are not normally needed for a resource database but are included to match the structure of a regular database. An application may optionally use these fields for its own purposes.
- The `type` and `creator` fields hold 4-byte signatures of the database type and creator as defined by the application that created the database.
- The `numResources` field holds the number of resource info entries that are stored in the header itself. In most cases, this is the total number of resources. If all the resource info entries cannot fit in the header, however, then `nextResourceList` has the `chunkID` of a `resourceList` that contains the next set of resource info entries.

Each 10-byte resource info entry in the header has the resource type, the resource ID, and the local ID of the Memory Manager chunk that contains the resource data.

Using the Resource Manager

You can create, delete, open, and close resource databases with the routines used to create normal record-based databases (see [Using the Data Manager](#)). This includes all database-level (not record-level) routines in the Data Manager such as [DmCreateDatabase](#), [DmDeleteDatabase](#), [DmDatabaseInfo](#), and so on.

When you create a new database using [DmCreateDatabase](#), the type of database created (record or resource) depends on the value of the `resDB` parameter. If set, a resource database is created and the `dmHdrAttrResDB` bit is set in the `attributes` field of the database header. Given a database header ID, an application can determine which type of database it is by calling [DmDatabaseInfo](#) and examining the `dmHdrAttrResDB` bit in the returned `attributes` field.

Once a resource database has been opened, an application can read and manipulate its resources by using the resource-based access

Files and Databases

The Resource Manager

routines of the Resource Manager. Generally, applications use the [DmGetResource](#) and [DmReleaseResource](#) routines.

[DmGetResource](#) returns a handle to a resource, given the type and ID. This routine searches all open resource databases for a resource of the given type and ID, and returns a handle to it. The search starts with the most recently opened database. To search only the most recently opened resource database for a resource instead of all open resource databases, call [DmGet1Resource](#).

[DmReleaseResource](#) should be called as soon as an application finishes reading or writing the resource data. To resize a resource, call [DmResizeResource](#), which accepts a handle to a resource and reallocates the resource in another heap of the same card if necessary. It returns the handle of the resource, which might have been changed if the resource had to be moved to another heap to be resized.

The remaining Resource Manager routines are usually not required for most applications. These include functions to get and set resource attributes, move resources from one database to another, get resources by index, and create new resources. Most of these functions reference resources by index to optimize performance. When referencing a resource by index, the `DmOpenRef` of the open resource database that the resource belongs to must also be specified. Call [DmSearchResource](#) to find a resource by type and ID or by pointer by searching in all open resource databases.

To get the `DmOpenRef` of the topmost open resource database, call [DmNextOpenResDatabase](#) and pass `NULL` as the current `DmOpenRef`. To find out the `DmOpenRef` of each successive database, call [DmNextOpenResDatabase](#) repeatedly with each successive `DmOpenRef`.

Given the access pointer of a specific open resource database, [DmFindResource](#) can be used to return the index of a resource, given its type and ID. [DmFindResourceType](#) can be used to get the index of every resource of a given type. To get a resource handle by index, call [DmGetResourceIndex](#).

To determine how many resources are in a given database, call [DmNumResources](#). To get and set attributes of a resource including its type and ID, call [DmResourceInfo](#) and [DmSetResourceInfo](#). To attach an existing data chunk to a resource database as a new

resource, call [DmAttachResource](#). To detach a resource from a database, call [DmDetachResource](#).

To create a new resource, call [DmNewResource](#) and pass the desired size, type, and ID of the new resource. To delete a resource, call [DmRemoveResource](#). Removing a resource disposes of its data chunk and removes its entry from the database header.

File Streaming Application Program Interface

The file streaming functions in Palm OS 3.0 and later let you work with large blocks of data. File streams can be arbitrarily large—they are not subject to the 64 KB maximum size limit imposed by the Memory Manager on allocated objects. File streams can be used for permanent data storage; in Palm OS 3.0, their underlying implementation is a Palm OS database. You can read, write, seek to a specified offset, truncate, and do everything else you'd expect to do with a desktop-style file.

Other than backup/restore, Palm OS does not provide direct HotSync support for file streams, and none is planned at this time.

The use of double-buffering imposes a performance penalty on file streams that may make them unsuitable for certain applications. Record-intensive applications tend to obtain better performance from the Data Manager.

Using the File Streaming API

The File Streaming API is derived from the C programming language's `<stdio.h>` interface. Any C book that explains the `<stdio.h>` interface should serve as a suitable introduction to the concepts underlying the Palm OS File Streaming API. This section provides only a brief overview of the most commonly used file streaming functions.

The [FileOpen](#) function opens a file, and the [FileRead](#) function reads it. The semantics of [FileRead](#) and [FileWrite](#) are just like their `<stdio.h>` equivalents, the `fread` and `fwrite` functions. The other `<stdio.h>` routines have obvious analogs in the File Streaming API as well.

Files and Databases

File Streaming Application Program Interface

For example,

```
theStream =  
FileOpen(cardId, "KillerAppDataFile",  
          'KILR', 'KILD', fileModeReadOnly,  
          &err);
```

As on a desktop, the filename is the unique item. The creator ID and file type are for informational purposes and your code may require that an opened file have the correct type and creator.

Normally, the [FileOpen](#) function returns an error when it attempts to open or replace an existing stream having a type and creator that do not match those specified. To suppress this error, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the [FileOpen](#) function.

To read data, use the [FileRead](#) function as in the following example:

```
FileRead(theStream, &buf, objSize, numObjs,  
         &err);
```

To free the memory used to store stream data as the data is read, you can use the [FileControl](#) function to switch the stream to destructive read mode. This mode is useful for manipulating temporary data; for example, destructive read mode would be ideal for adding the objects in a large data stream to a database when sufficient memory for duplicating the entire file stream is not available. You can switch a stream to destructive read mode by passing the `fileOpDestructiveReadMode` selector as the value of the `op` parameter to the [FileControl](#) function.

The [FileDmRead](#) function can read data directly into a Data Manager chunk for immediate addition to a Palm OS database.

Summary of Files and Databases

Data Manager Functions

Creating Databases

[DmCreateDatabase](#)

[DmCreateDatabaseFromImage](#)

Opening and Closing Databases

[DmOpenDatabase](#)

[DmCloseDatabase](#)

[DmDatabaseProtect](#)

[DmOpenDatabaseByTypeCreator](#)

Creating Records

[DmNewHandle](#)

[DmNewRecord](#)

Accessing Records

[DmGetRecord](#)

[DmQueryRecord](#)

[DmFindRecordById](#)

[DmSearchRecord](#)

Adding Records

[DmAttachRecord](#)

Unlocking Records

[DmReleaseRecord](#)

Changing Records

[DmMoveRecord](#)

[DmResizeRecord](#)

[DmSet](#)

[DmStrCopy](#)

[DmWrite](#)

[DmWriteCheck](#)

Deleting Records

[DmArchiveRecord](#)

[DmDeleteDatabase](#)

[DmDeleteRecord](#)

[DmDetachRecord](#)

[DmRemoveRecord](#)

[DmRemoveSecretRecords](#)

Sorting

[DmInsertionSort](#)

[DmFindSortPositionV10](#)

[DmFindSortPosition](#)

[DmQuickSort](#)

Files and Databases

Summary of Files and Databases

Data Manager Functions

Categories

[DmMoveCategory](#)

[DmDeleteCategory](#)

[DmQueryNextInCategory](#)

[DmNumRecordsInCategory](#)

[DmPositionInCategory](#)

[DmSeekRecordInCategory](#)

Locating Databases

[DmFindDatabase](#)

[DmGetDatabase](#)

[DmNextOpenDatabase](#)

[DmGetNextDatabaseByTypeCreator](#)

Database Information

[DmDatabaseInfo](#)

[DmRecordInfo](#)

[DmOpenDatabaseInfo](#)

[DmNumDatabases](#)

[DmSetDatabaseInfo](#)

[DmSetRecordInfo](#)

[DmDatabaseSize](#)

[DmNumRecords](#)

Application Information

[DmGetAppInfoID](#)

Error Handling

[DmGetLastErr](#)

Resource Manager Functions

[DmOpenDBNoOverlay](#)

[DmNewResource](#)

[DmReleaseResource](#)

[DmDetachResource](#)

[DmSearchResource](#)

[DmFindResourceType](#)

[DmGetResource](#)

[DmNumResources](#)

[DmResourceInfo](#)

[DmAttachResource](#)

[DmRemoveResource](#)

[DmGetResourceIndex](#)

[DmFindResource](#)

[DmGet1Resource](#)

[DmNextOpenResDatabase](#)

[DmResizeResource](#)

[DmSetResourceInfo](#)

File Streaming Function Summary

Opening and Closing

[FileOpen](#)
[FileSeek](#)

[FileClose](#)

Reading Files

[FileRead](#)
[FileRewind](#)

[FileDmRead](#)
[FileControl](#)

Writing to Files

[FileWrite](#)

[FileTruncate](#)

File Information

[FileEOF](#)

[FileTell](#)

Deleting Files

[FileDelete](#)

[FileFlush](#)

Error Handling

[FileError](#)
[FileClearerr](#)

[FileGetLastError](#)

Expansion

This chapter describes how to work with expansion cards and add-on devices using the Palm OS® Expansion and Virtual File System (VFS) Managers.

- [Expansion Support](#) introduces basic terminology and discusses the hardware and file systems supported by the Expansion and VFS Managers.
- [Architectural Overview](#) illustrates the Palm OS expansion architecture and discusses the differences between primary and secondary storage.
- [Standard Directories](#) lists directories that are treated specially by the Palm OS and describes their use.
- [Applications on Cards](#) covers the various implications of running Palm OS applications from an expansion card.
- [Card Insertion and Removal](#) covers, in detail, the sequence of events that occur when an expansion card is inserted into or removed from an expansion slot.
- [Checking for Expansion Cards](#) shows you how to verify that the handheld supports expansion, how to check each of the handheld's slots for expansion cards, and how to determine the capabilities of a card in a given slot.
- [Volume Operations](#) discusses the various ways in which you can work with volumes on an expansion card.
- [File Operations](#) discusses the various ways in which you can work with files on an expansion card.
- [Directory Operations](#) discusses the various ways in which you can work with directories on an expansion card.
- [Custom Calls](#) briefly discusses how you can go beyond the functions provided by the Expansion and VFS Managers and interact with specialized I/O devices.
- [Debugging](#) briefly introduces the process of debugging an application that relies on the presence of an expansion card.

Expansion Support

Beginning with Palm OS 4.0¹, a set of optional system extensions provide a standard mechanism by which Palm OS applications can take advantage of the expansion capabilities of various Palm Powered™ handhelds. This capability not only augments the memory and I/O of the handheld, but facilitates data interchange with other Palm Powered handhelds and with devices that aren't running the Palm OS. These other devices include digital cameras, digital audio players, desktop or laptop computers, and the like.

Primary vs. Secondary Storage

All Palm Powered handhelds contain **primary storage**—directly addressable memory that is used for both long-term and temporary storage. This includes storage RAM, used to hold nonvolatile user data and applications; and dynamic RAM, which is used as working space for temporary allocations.

On most handhelds, primary storage is contained entirely within the device itself. The Palm OS memory architecture doesn't limit devices to this, however; devices can be designed to accept additional storage RAM. The products developed by Handspring™ work this way; memory modules plugged into the Springboard slot are fully-addressable and appear to a Palm OS application as additional storage RAM.

Secondary storage, by contrast, is designed primarily to be add-on nonvolatile storage. Although not limited to any particular implementation, most secondary storage media:

- can be inserted and removed from the expansion slot at will
- are based upon a third-party standard, such as Secure Digital (SD) memory cards, MultiMedia (MMC) cards, CompactFlash, Sony's Memory Stick™, and others
- present a serial interface, accessing data one bit, byte, or block at a time

1. The Sony CLIE™ handheld running Palm OS 3.5 runs a binary-compatible version of these extensions.

Applications access primary storage either directly, in the case of most dynamic RAM, or through the Database and Resource Managers. To access secondary storage, however, applications use the Expansion and VFS Managers. These have been designed to support as broad a range of serial expansion architectures as possible.

Expansion Slot

The expansion slots found on many Palm Powered handhelds vary depending on the manufacturer. While some may accept SD and MMC cards, others may accept Memory Stick or CompactFlash. Note that there is no restriction on the number of expansion slots that a given handheld can have.

Depending on the expansion technology used, there can be a wide variety of expansion cards usable with a given handheld:

- Storage cards provide secondary storage and can either be used to hold additional applications and data, or can be used for a specific purpose, for instance as a backup mechanism.
- ROM cards hold dedicated applications and data.
- I/O cards extend the handheld's I/O capabilities. A modem, for instance, could provide wired access, while a Bluetooth™ transceiver could add wireless capability.
- “Combo” cards provide both additional storage or ROM along with some I/O capability.

Universal Connector

Certain newer Palm Powered handhelds may be equipped with a universal connector that connects the handheld to a HotSync® cradle. This connector can be used to connect the handheld to snap-on I/O devices as well. A special slot driver dedicated to this connector allows handheld-to-accessory communication using the serial portion of the connector. This “plug and play” slot driver presents the peripheral as a card in a slot, even to the extent of providing the card insertion notification when the peripheral is attached.

Because the universal connector's slot driver makes a snap-on peripheral appear to be a card in a slot, such peripherals can be

Expansion

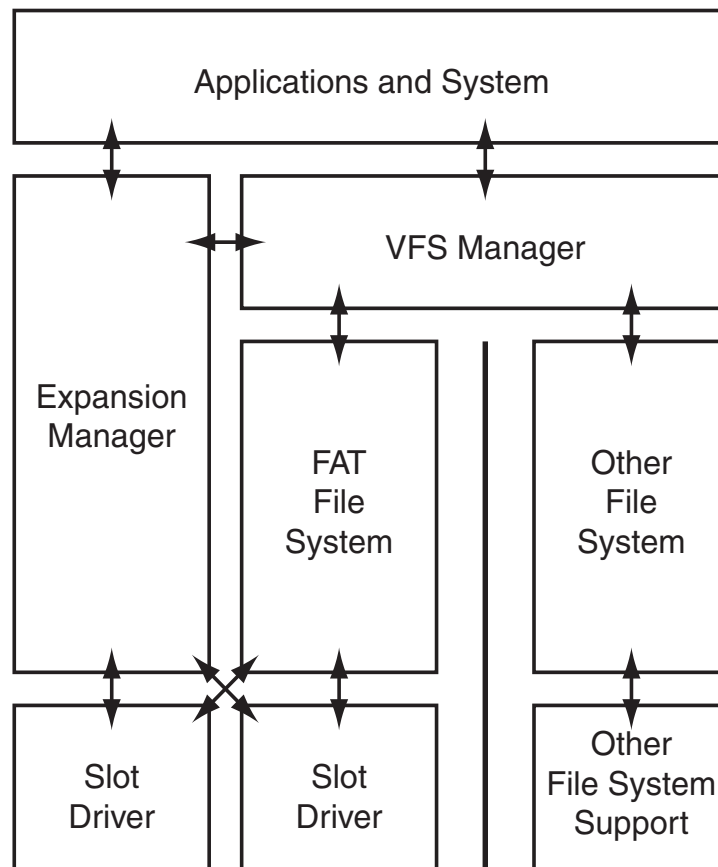
Architectural Overview

treated as expansion cards, at least from an application developer's perspective. For the remainder of this chapter, wherever an I/O card could be used, the phrase "expansion card" can be taken to mean both "expansion card" and "plug and play peripheral."

Architectural Overview

[Figure 7.1](#) illustrates the Palm OS expansion architecture. It is designed to be flexible enough to support multiple file systems and diverse physical expansion mechanisms while still presenting a consistent set of APIs to applications and to other parts of the Palm OS. The following sections describe the major components of the Palm OS expansion architecture. Working from the bottom up, those components are: slot drivers, file systems, the VFS Manager, and the Expansion Manager.

Figure 7.1 Palm OS expansion architecture



Slot Drivers

A slot driver is a standard Palm OS shared library of type `sysFileTSlotDriver('libs')`. It is a special library that encapsulates direct access to the hardware and provides a standard set of services to the Expansion Manager and, optionally, to file system libraries. Adding support for a new type of hardware expansion is usually simply a matter of writing a slot driver for it. As illustrated in [Figure 7.1](#), applications don't normally interact directly with slot drivers.

Each expansion slot has a slot driver associated with it. Slots are identified by a unique **slot reference number**, which is assigned by the Expansion Manager. Expansion cards themselves are not numbered individually; applications typically reference the slot into which a card is inserted. Note, however, that a slot may or may not have a card in it at any given time, and that a card can be inserted and removed while an application is running.

NOTE: “Card number” is a Palm OS Memory Manager term, and is not to be confused with “slot reference number.”

The current implementation only supports one volume per slot.

File Systems

The Palm OS expansion architecture defines a common interface for all file system implementations on the Palm OS. This interface consists of a complete set of APIs for interacting with the file system, including the ability to open, close, read, write, and delete both files and directories on named volumes.

File system implementations are packaged as shared libraries of type `sysFileTFileSystem('libf')`. They are modular plug-ins that add support for a particular type of file system, such as VFAT, HFS, or NFS. The Palm OS expansion architecture allows multiple file system libraries to be installed at any given time. Typically, an implementation of the VFAT file system is present.

VFAT is the industry standard for flash memory cards of all types. It enables easy transfer of data and or applications to desktops and other devices. The VFAT file system library included with Palm OS

Expansion

Architectural Overview

4.0 natively supports VFAT file systems on secondary storage media. It is able to recognize and mount FAT and VFAT file systems, and offers to reformat unrecognizable or corrupted media.

Because the VFAT file system requires long filenames to be stored in Unicode/UCS2 format, the standard VFAT file system library supports conversion between UCS2 and Shift-JIS (the standard Palm OS multi-byte character encoding), and the Palm/Latin encoding.

VFS Manager

The VFS (Virtual File System) Manager provides a unified API that gives applications access to many different file systems on many different media types. It abstracts the underlying file systems so that applications can be written without regard to the actual file system in use. The VFS Manager includes APIs for manipulating files, directories, and volumes.

NOTE: Although the great majority of the functions in the VFS Manager can be used by any application, some are intended only for use by slot drivers and file systems. Others are not intended for use by third-party applications but are designed primarily for system use.

The VFS Manager, the Data Manager, and File Streaming APIs

With the addition of the VFS Manager to the Palm OS, there are now three distinct ways applications can store and retrieve Palm OS user data:

- The Data Manager manages user data in the storage heap. It was specifically designed to make the most of the limited dynamic RAM and the nonvolatile RAM used instead of disk storage on most handhelds. Use the Data Manager to store and retrieve Palm OS user data when storage on the handheld is all that is needed, or when efficient access to data is paramount.
- The File Streaming API is a layer on top of the Data Manager that provides file functionality with all data being read from or written to a database in the storage heap. Most

applications have no need for the File Streaming APIs; they are primarily used by applications that need to work with large blocks of data.

- The VFS and Expansion Managers were designed specifically to support many types of expansion memory as secondary storage. The VFS Manager APIs present a consistent interface to many different types of file systems on many types of external media. Applications that use the VFS APIs can support the widest variety of file systems. Use the VFS Manager when your application needs to read and write data stored on external media.

Palm OS applications should use the appropriate APIs for each given situation. The Data Manager, being an efficient manager of storage in the storage heap, should be used whenever access to external media is not absolutely needed. Use the VFS API when interoperability and file system access is needed. Note, however, that the VFS Manager adds the extra overhead of buffering all reads and writes in memory when accessing data, so only applications that specifically need this functionality should use the VFS Manager.

For more information on the Data and Resource Managers, as well as on the File Streaming APIs, see [Chapter 6, “Files and Databases.”](#) For details of the APIs presented by the VFS Manager, see [Chapter 53, “Virtual File System Manager,”](#) in the *Palm OS Programmer’s API Reference*.

Expansion Manager

The Expansion Manager is a software layer that manages slot drivers on Palm OS handhelds. Supported expansion card types include, but are not limited to, Memory Stick and SD cards. The Expansion Manager does not support these expansion cards directly; rather, it provides an architecture and higher level set of APIs that, with the help of low level slot drivers and file system libraries, support these types of media.

The Expansion Manager:

- broadcasts notification of card insertion and removal
- plays sounds to signify card insertion and removal
- mounts and unmounts card-resident volumes

NOTE: Some of the other functions provided by the Expansion Manager are for use by slot drivers and file systems and are not generally used by their-party applications.

For details of the APIs presented by the VFS Manager, see [Chapter 29, “Expansion Manager,”](#) in the *Palm OS Programmer’s API Reference*.

Standard Directories

The user experience presented by the Palm OS is simpler and more intuitive than that of a typical desktop computer. Part of this simplicity arises from the fact that the Palm OS doesn’t present a file system to the user. Users don’t have to understand the complexities of a typical file system; applications are readily available with one or two taps of a button or icon, and data associated with those applications is accessible only through each application. Maintaining this simplicity of user operation while supporting a file system on an expansion card is made possible through a standard set of directories on the expansion card.

The following table lists the standard directory layout for all “standards compliant” Palm OS secondary storage. All Palm OS relevant data should be in the /PALM directory (or in a subdirectory of the /PALM directory), effectively partitioning off a private name space.

Directory	Description
/	Root of the secondary storage.
/PALM	Most data written by Palm™ applications lives in a subdirectory of this directory. <code>start.prc</code> lives directly in /PALM. This optional file is automatically run when the secondary storage volume is mounted. Other applications may also reside in this directory.

Directory	Description
/PALM/Backup	Reserved by the Palm OS for backup purposes.
/PALM/Programs	Catch-all for other applications and data.
/PALM/Launcher	Home of Launcher-visible applications.

The Palm OS Launcher has been enhanced to be expansion card aware. When an expansion card containing a file system is inserted, all applications listed in the card's /PALM/Launcher directory are automatically added to a new Launcher category. This new category takes the name of the expansion card volume. Note that the name displayed in the Launcher for a given application is the name in the application's `tAIN` (application icon name) resource or, if this resource is empty, the database name, which may or may not match the name of the file.

NOTE: Whenever possible give the same name to the `.prc` file and to the database. If the `.prc` filename differs from the database name, and users copy your application from the card to the handheld and then to another card, the filename may change. This is because the database name is used when an application is copied from the handheld to the card.

When a writable volume is mounted, the Launcher automatically creates the /PALM and /PALM/Launcher directories if they don't already exist. If they do, and if there are applications present in the /PALM/Launcher directory, the Launcher automatically switches to the card's list of applications unless it runs `start.prc`.

In addition to these standard directories, the VFS Manager supports the concept of a **default directory**; a directory in which data of a particular type is typically stored. See "[Determining the Default Directory for a Particular File Type](#)" on page 242 for more information.

Applications on Cards

Palm OS applications located in the `/PALM/Launcher` directory of an expansion card volume appear in a separate Launcher category when the card is inserted into the handheld's expansion slot. If you tap the icon for one of these applications, it is copied to main memory and then launched.

Applications launched from a card ("card-launched" applications) are first sent a [`sysAppLaunchCmdCardLaunch`](#) launch code, along with a parameter block that includes the reference number of the volume on which the application resides and the complete path to the application. When processing this launch code, the application shouldn't interact with the user or access globals. Unless the application sets the `sysAppLaunchStartFlagNoUISwitch` bit in the `start` flags (which are part of the parameter block), the application is then sent a `sysAppLaunchCmdNormalLaunch` launch code. This is when the application should, if it needs to, interact with user. Applications may want to save some state when `sysAppLaunchCmdCardLaunch` is received, then act upon that state information when `sysAppLaunchCmdNormalLaunch` is received.

When the user switches to a new application, the card-launched application is removed from main memory. Note, however, that any databases created by the card-launched application remain.

There are certain implications to this "copy and run" process.

- There must be sufficient memory for the application. If the handheld doesn't have enough memory to receive the application, it isn't copied from the expansion card and it isn't launched.
- The copying process takes time. For large applications, this can cause a noticeable delay before the application is actually launched.
- If some version of the application on the card is already present in main memory, the Launcher puts up a dialog that requires the user to choose whether or not to overwrite the in-memory version.

- Card-launched applications have a limited lifetime: applications reside in main memory only while they are running. When the user switches to a different application, the card-launched application that was just running is removed from main memory. If the card-launched application is then re-launched, it is once again copied into the handheld's memory.
- "Legacy" applications—those that are unaware that they are being launched from a card—only work with databases in main memory. Associated databases aren't copied to main memory along with the application unless the database is bundled with the application. Databases created by card-launched applications are not removed along with the application, however, so this data is available to the application when it is subsequently run. Applications that are written to take advantage of the VFS Manager can read and write data on the expansion card, so this limitation generally only applies to legacy applications.

Bundled databases, although copied to main memory along with their associated application, are meant for static data that doesn't change, such as a game level database. Bundled databases are not copied back to the card; they are simply deleted from memory when the user chooses another application. To bundle a database with an application, give it the same creator ID as the owning application, set the `dmHdrAttrBundle` bit, and place it in the `/PALM/Launcher` directory along with the application.

- Unless a card-launched application is running, it doesn't receive notifications or launch codes since it isn't present on the handheld. In particular, these applications don't receive notifications and aren't informed when an alarm is triggered.

Card Insertion and Removal

The Expansion Manager supports the insertion and removal of expansion media at any time. The handheld continues to run as before, though an application switch may occur upon card insertion. The handheld need not be reset or otherwise explicitly informed that a card has been inserted or removed.

Expansion

Card Insertion and Removal

WARNING! Due to the way certain expansion cards are constructed, if the user removes an expansion card while it is being written to, in certain rare circumstances it is possible for the card to become damaged to the point where either it can no longer be used or it must be reformatted. To the greatest extent possible, applications should only write to the card at well-defined points, and the application should warn the user—perhaps with a “Please Wait” or progress dialog—at that time not to remove the expansion card. The card can be removed while an application is reading from it without fear of damage.

The Palm OS uses a series of notifications to indicate that a card has been inserted or removed, or that a volume has been mounted or unmounted. The following table lists these notifications, and the priority for which they have been registered by the Expansion and VFS Managers. Note that the priorities may change in a future release, so applications shouldn’t depend on these precise values. Applications that register for these using normal priority get the correct behavior.

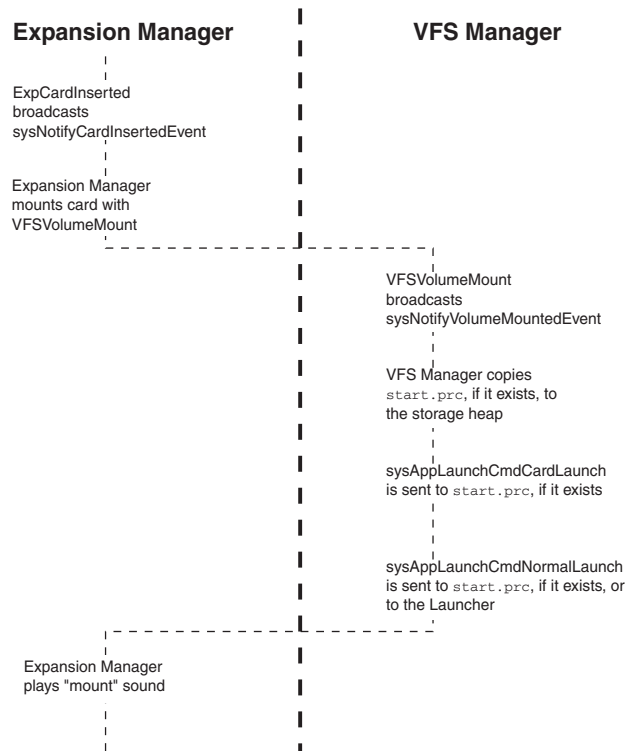
Table 7.1 Expansion card notifications

Notification	Registered by	Priority
<code>sysNotifyCardInsertedEvent</code>	Exp. Manager	20
<code>sysNotifyCardRemovedEvent</code>	Exp. Manager	-20
<code>sysNotifyVolumeMountedEvent</code>	Exp. Manager	-20
<code>sysNotifyVolumeMountedEvent</code>	VFS Manager	10
<code>sysNotifyVolumeUnmountedEvent</code>	Exp. Manager	-20

The following diagram shows the sequence of events that occur when an expansion card is inserted into a Palm Powered handheld’s expansion slot. For clarity, it assumes that no errors occur. If the card doesn’t contain a mountable volume, and if the card cannot be formatted and then mounted, this sequence is aborted and the card

remains unmounted, although the card insertion notification is still broadcast.

Figure 7.2 Sequence of events upon card insertion



The Expansion Manager registers for `sysNotifyCardInsertedEvent` with a priority of 20, ensuring that it is notified after other handlers that may have registered with normal priority. To override the Expansion Manager's default handler, register your handler to receive `sysNotifyCardInsertedEvent` with normal priority, and have it set the appropriate bits in the `handled` member of the `SysNotifyParamType` structure:

- `expHandledVolume` indicates that any volumes associated with the card have been dealt with, and prevents the

Expansion

Card Insertion and Removal

Expansion Manager from mounting or unmounting the card's volumes.

- `expHandledSound` indicates that your application has handled the playing of an appropriate sound, and prevents the Expansion Manager from playing a sound when the card is inserted or removed.

Note that the number of the slot into which the card was inserted is passed to your handler using the `notifyDetailsP` member—which is a `UInt16`, cast to a `void *`—of the `SysNotifyParamType` structure.

Although most applications only register for volume mount and unmount notifications, if you need to receive notifications when the user removes a card from a slot managed by the Expansion Manager, have your application register to receive `sysNotifyCardRemovedEvent`. Unlike with `sysNotifyCardInsertedEvent`, the Expansion Manager registers for `sysNotifyCardRemovedEvent` with a priority of -20, ensuring that it receives the notification before other handlers that are registered for it with normal priority. This notification, too, passes the number of the slot from which the card was removed to your handler using the `notifyDetailsP` member—which is a `UInt16`, cast to a `void *`—of the `SysNotifyParamType` structure.

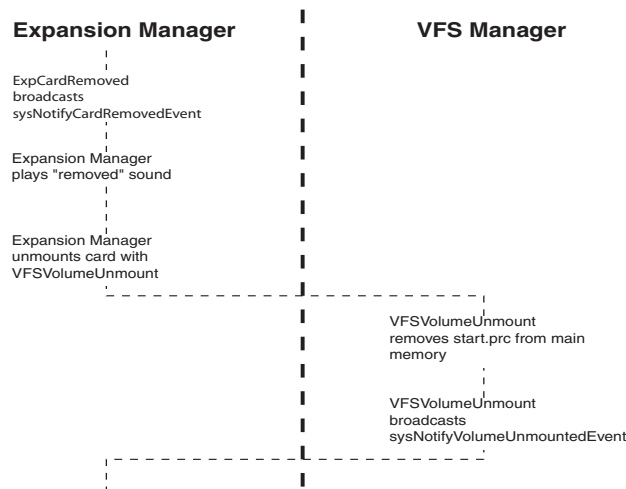
The VFS Manager registers for `sysNotifyVolumeMountedEvent` with a priority of 10. To override the VFS Manager's default handler, register your handler to receive `sysNotifyVolumeMountedEvent` with normal priority, and have it set the appropriate bits in the `handled` member of the `SysNotifyParamType` structure:

- `vfsHandledUIAppSwitch` indicates that your application has handled `SysUIAppSwitch` to `start.prc`. This bit prevents the VFS Manager from performing its own `SysUIAppSwitch` to `start.prc` (although `start.prc` is still loaded and a `SysAppLaunch` is performed), and also prevents the launcher from switching to itself.
- `vfsHandledStartPrc` indicates that your handler has dealt with the automatic running of `start.prc`. The VFS Manager won't load it and won't call either `SysAppLaunch` or `SysUIAppSwitch`.

Note that if your application handles the running of `start.prc`, you need to keep security in mind. If the handheld is locked when an expansion card is inserted, the VFS Manager's own handler defers the execution of `start.prc` until the user unlocks the handheld.

Card removal follows a similar sequence, although there is no equivalent to `start.prc` that is automatically run. This sequence is illustrated in the following diagram.

Figure 7.3 Sequence of events upon card removal



Upon card removal, the Expansion Manager broadcasts a notification to all applications that have registered to receive card removal notifications and unmounts any mounted volumes on the card. This causes the VFS Manager to issue a card unmounted notification. Each application must register for the card unmounted notification and provide the necessary error handling code if card removal at any time will cause a problem for the application.

Note that the card insertion and removal notifications are intended primarily for system use, although they can be registered for by applications that need them. Applications that deal only with file systems and the VFS Manager should confine themselves to the volume mounted and unmounted notifications.

Expansion

Card Insertion and Removal

Start.prc

Upon receipt of a [sysNotifyVolumeMountedEvent](#) that hasn't already been handled (as indicated by the state of the `vfsHandledStartPrc` bit, as described in the previous section), the VFS Manager copies `/Palm/start.prc`—and its overlay, if there is one—to the storage heap and launches it. This process enables “application cards”—single-function cards that execute automatically upon card insertion. It also allows for combo cards that automatically load any necessary drivers and applications to support card I/O.

To launch `start.prc`, the VFS Manager first sends it a special launch code, [sysAppLaunchCmdCardLaunch](#). If the application only needs to do a bit of work and return, it should do it here and then set the `sysAppLaunchStartFlagNoUISwitch` bit in the start flags, which are part of the `sysAppLaunchCmdCardLaunch` parameter block. Note that the application doesn't have access to globals and it shouldn't interact with the user here. If the `sysAppLaunchStartFlagNoUISwitch` bit is not set, as it isn't if the application ignores the `sysAppLaunchCmdCardLaunch` launch code, the VFS Manager then sends it a `sysAppLaunchCmdNormalLaunch` launch code to run the application normally. This ensures backwards compatibility with applications that do not understand the `sysAppLaunchCmdCardLaunch` launch code. This is where the application can interact with the user; an application may want to save state when it receives `sysAppLaunchCmdCardLaunch`, and then act upon that state when it receives `sysAppLaunchCmdNormalLaunch`.

To avoid running out of stack space, the VFS Manager sets the “new stack” bit when launching `start.prc`. The `start.prc` application remains in system memory until the volume from which it was copied is removed. `start.prc` is deleted before `VFSVolumeUnmount` broadcasts `sysNotifyVolumeUnmountedEvent` but after the Expansion Manager broadcasts `sysNotifyCardRemovedEvent`. By registering for `sysNotifyCardRemovedEvent`, `start.prc` can react to the volume being removed before it is deleted.

NOTE: If an expansion card is inserted while the handheld is locked, `start.prc` is not executed until the user unlocks the handheld.

Checking for Expansion Cards

Before looking for an expansion card, your program should first make sure that the handheld supports expansion by verifying the presence of the Expansion and VFS Managers. It can then query for mounted volumes. Finally, your program may want to ascertain the capabilities of the card; whether it has memory, whether it does I/O, and so on. The following sections describe each of these steps.

Verifying Handheld Compatibility

There are many different Palm OS handhelds, and in the future there will be many more. Some will have expansion slots to support secondary storage, and some will not. Hardware to support secondary storage is optional, and may or may not be present on a given handheld. Since the Expansion and VFS Managers are of no use on a handheld that has no physical expansion capability, they are optional system extensions that are not present on every Palm Powered handheld.

Due to the great variability both in handheld configuration and in the modules which can be plugged into or snapped onto the handheld, applications shouldn't attempt to detect the manufacturer or model of a specific handheld when determining if it supports secondary storage. Instead, check for the presence and capabilities of the underlying operating system.

The VFS Manager and the Expansion Manager are individual system extensions that are both optional. They both make use of other parts of the operating system that were introduced in Palm OS 4.0. Thus, in order to be fully capable of running an application that

Expansion

Checking for Expansion Cards

relies on the Expansion and VFS Managers, the following all have to be true for a given handheld:

- The handheld must be running Palm OS 4.0².
- The Expansion Manager must be present.
- The VFS Manager must be present.

[Appendix B, “Compatibility Guide,”](#) details how to verify the presence of each:

- [4.0 New Feature Set](#) begins by illustrating how to verify that the handheld is running Palm OS 4.0.
- [Expansion Manager Feature Set](#) shows how to check for the presence of the Expansion Manager.
- [VFS Manager Feature Set](#) shows how to check for the presence of the VFS Manager.

Although your program could check for the presence of all of the above, it can take advantage of the fact that the VFS Manager relies on the Expansion Manager and won't be present without it. Thus, if the VFS Manager is present, you can safely assume that the Expansion Manager is present as well.

Checking for Mounted Volumes

Many applications rely on the handheld's expansion capabilities for additional storage. Applications that don't care about the physical characteristics of the secondary storage module, and that don't need to know the slot into which the module is inserted, can rely on the fact that the Palm OS automatically mounts any recognized volumes inserted into or snapped onto the handheld. Thus, many applications can simply enumerate the mounted volumes and select one as appropriate. The following code illustrates how to do this:

2. The Sony CLIE™ handheld running Palm OS 3.5 uses a version of the Expansion and VFS Managers. Sony's version of these managers is binary compatible with those included with Palm OS 4.0.

Listing 7.1 Enumerating mounted volumes

```
UInt16 volRefNum;
UInt32 volIterator = vfsIteratorStart;

while (volIterator != vfsIteratorStop) {
    err = VFSVolumeEnumerate(&volRefNum, &volIterator);
    if (err == errNone) {
        // Do something with the volRefNum
    } else {
        // handle error... possibly by
        // breaking out of the loop
    }
}
```

The volume reference number obtained from [VFSVolumeEnumerate](#) can then be used with many of the volume, directory, and file operations that are described later in this chapter.

Occasionally an application needs to know more than that there is secondary storage available for use. Those applications likely need to take a few extra steps, beginning with checking each of the handheld's slots.

Enumerating Slots

Before you can determine which expansion modules are attached to a Palm OS handheld, you must first determine how those modules could be attached. Expansion cards and some I/O devices could be plugged into physical slots, and snap-on modules could be connected through the handheld's universal connector. Irrespective of how they're physically connected, the Expansion Manager presents these to the developer as slots. Enumerating these slots is made simple due to the presence of the [ExpSlotEnumerate](#) function. The use of this function is illustrated here:

Listing 7.2 Iterating through a handheld's expansion slots

```
UInt16 slotRefNum;
UInt32 slotIterator = expIteratorStart;

while (slotIterator != expIteratorStop) {
    // Get the slotRefNum for the next slot
    err = ExpSlotEnumerate(&slotRefNum, &slotIterator);
}
```

Expansion

Checking for Expansion Cards

```
if(err == errNone) {  
    // perform slot-specific processing here  
} else {  
    // handle error... possibly by  
    // breaking out of the loop  
}
```

The slot reference number returned by `ExpSlotEnumerate` uniquely identifies a given slot. This can be supplied to various Expansion Manager functions to obtain information about the slot, such as whether there is a card or other expansion module present in the slot.

Checking a Slot for the Presence of a Card

Use the [ExpCardPresent](#) function to determine if a card is present in a given slot. Given the slot reference number, this function returns `errNone` if there is a card in the slot, or an error if either there is no card in the slot or there is a problem with the specified slot.

Determining a Card's Capabilities

Just knowing that an expansion card is inserted into a slot or connected to the handheld isn't enough; your application needs to know something about the card to ensure that the operations it needs to perform are compatible with the card. For instance, if your application needs to write data to the card, its important to know if writing is permitted.

The capabilities available to your application depend not only on the card but on the slot driver as well. Handheld manufacturers will provide one or more slot drivers that define standard interfaces to certain classes of expansion hardware. Card and device manufacturers may also choose to provide card-specific slot drivers, or they may require that applications use the slot custom control function and a registered creator code to access and control certain cards.

The slot driver is responsible for querying expansion cards for a standard set of capabilities. When a slot driver is present for a given

expansion card, you can use the [ExpCardInfo](#) function to determine the following:

- the name of the expansion card's manufacturer
- the name of the expansion card
- the "device class," or type of expansion card. Values returned here might include "Ethernet" or "Backup"
- a unique identifier for the device, such as a serial number
- whether the card supports both reading and writing, or whether it is read-only
- whether the card supports a simple serial interface

Note that the existence of the `ExpCardInfo` function does not imply that all expansion cards support these capabilities. It only means that the slot driver is able to assess a card and report its findings up to the Expansion Manager.

Volume Operations

If an expansion card supports a file system, the VFS Manager allows you to perform a number of standard volume operations. To determine which volumes are currently mounted and available, use [VFSVolumeEnumerate](#). This function, the use of which is illustrated in "[Checking for Mounted Volumes](#)" on page 224, returns a volume reference number that you then to supply to the remainder of the volume operations.

When the user inserts a card containing a mountable volume into a slot (note that the current implementation only supports one volume per slot), the VFS Manager attempts to mount the volume automatically. You should rarely, if ever, have to mount volumes directly. You can attempt to mount a volume using a different file system, however, perhaps after installing a new file system driver on the handheld. To explicitly mount or unmount a volume, use [VFSVolumeMount](#) and [VFSVolumeUnmount](#). When mounting a volume, you can either specify an explicit file system with which to mount the volume, or you can request that the VFS Manager try to determine the appropriate file system. If the VFS Manager cannot mount the volume using any of the available file systems, it attempts to format the volume using a file system deemed

appropriate for the slot, and then mount it. See the description of `VFSVolumeMount` in the *Palm OS Programmer's API Reference* for the precise arguments you must supply when explicitly mounting a volume.

Use [VFSVolumeFormat](#) to format a volume. This function can be used to change the file system on the expansion card; you can explicitly indicate a file system to use when formatting it. Once the card has been formatted, the VFS Manager automatically mounts it; a new volume reference number is returned from `VFSVolumeFormat`.

The [VFSVolumeGetLabel](#) and [VFSVolumeSetLabel](#) functions get and set the volume label, respectively. Since the file system is responsible for verifying the validity of strings, you can try to set the volume label to any desired value. If the file system doesn't natively support the name given, the VFS Manager creates the `/VOLUME.NAM` file used to support long volume names (see "[Naming Volumes](#)" on page 230 for more information) or you get an error back if the file system doesn't support the supplied string.

Additional information about the volume can be obtained through the use of [VFSVolumeSize](#) and [VFSVolumeInfo](#). As the name implies, `VFSVolumeSize` returns size information about the volume. In particular, it returns both the total amount of space on the volume, in bytes, and the amount of that volume's space that is currently in use, again in bytes. `VFSVolumeInfo` returns various pieces of information about the volume, including:

- whether the volume is hidden
- whether the volume is read-only
- whether the volume is supported by a slot driver, or is being simulated by the Palm OS Emulator
- the type and creator of the underlying file system
- the slot with which the volume is associated, and the reference number of the slot driver controlling the slot
- the type of media on which this volume is located, such as SD, CompactFlash, or Memory Stick

All of the above information is returned encapsulated within a [VolumeInfoType](#) structure. Whether the volume is hidden or read-only is further encoded into a single field within this structure;

see [Volume Attributes](#) in the *Palm OS Programmer's API Reference* for the bits that make up this field.

Hidden Volumes

Included among the volume attributes is a “hidden” bit, `vfsVolumeAttrHidden`, that indicates whether the volume on the card is to be visible or hidden. Hidden volumes are typically not meant to be directly available to the user; the Launcher and the CardInfo application both ignore all hidden volumes.

To make a volume hidden, simply create an empty file named `HIDDEN.VOL` in the `/PALM` directory. The [VFSVolumeInfo](#) function looks for this file and, if found, returns the `vfsVolumeAttrHidden` bit along with the volume's other attributes.

Matching Volumes to Slots

Many applications don't need to know the specifics of an expansion card as provided by the [ExpCardInfo](#) function. Often, the information provided by the [VFSVolumeInfo](#) function is enough. Some applications need to know more about a particular volume, however. The name of the manufacturer or the type of card, for instance, may be important.

The [VolumeInfoType](#) structure returned from `VFSVolumeInfo` contains a `slotRefNum` field that can be passed to `ExpCardInfo`. This allows you to obtain specific information about the card on which a particular volume is located.

Although slot drivers currently only support one volume per slot, obtaining volume information that corresponds to a given slot reference number isn't quite so simple, since there isn't a function that returns the volume reference number given a slot reference number. You can, however, iterate through the mounted volumes and check each volume's slot reference number. This is the technique that the CardInfo application uses.

Naming Volumes

Different file system libraries support volume names of different maximum lengths and have different restrictions on character sets. The file system library is responsible for verifying whether or not a given volume name is valid, and returns an error if it is not. From a Palm OS developer's standpoint, volume names can be up to 255 characters long, and can include any printable character.

The file system library is responsible for translating the volume name into a format that is acceptable to the underlying file system. For example, in a file system where the 8.3 naming convention is used for filenames, to translate a long volume name the first eleven valid, non-space characters are used. Valid characters in this instance are A-Z, 0-9, \$, %, ', -, _, @, ~, !, (,), ^, #, and &.

When the underlying file system doesn't support a long volume name, [VFSVolumeSetLabel](#) creates the file /VOLUME.NAM in an effort to preserve the long volume name. This file contains the following, in order:

Field	Description
Char cookie[4]	4 byte cookie that identifies this file. The value of this cookie is vfsVolumeNameFileCookie.
UInt16 cacheLen	Big-endian length, in bytes, of the cached file-system-level volume label.

Field	Description
Char cacheLabel[cacheLen]	Unicode UCS-2 format string containing the volume label as it is stored in the file system layer. This is compared with the file system volume label to see if the user has changed the volume label on a device that doesn't support the /VOLUME.NAM file. In this event, the file system volume label is used; the contents of /VOLUME.NAM are ignored.
UInt16 length	Big-endian length, in bytes, of the long volume label.
Char label[length]	Unicode UCS-2 format string containing the long volume label.

File Operations

All of the familiar operations you'd use to operate on files in a desktop application are supported by the VFS Manager; these are listed in "[Common Operations](#)," below. In addition, the VFS Manager includes a set of functions that simplify the way you work with files that represent Palm databases (.pdb) or Palm resource databases (.prc). These are covered in "[Working with Palm Databases](#)" on page 234.

Common Operations

The VFS Manager provides all of the standard file operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use isn't detailed here. See the descriptions of each individual function

in the *Palm OS Programmer's API Reference* for the arguments, return values, and side effects of each.

Note that some of these functions can be applied to both files and directories, while others work only with files.

Table 7.2 Common file operations

Function	Description
VFSFileOpen	Open a file, given a volume reference number and a file path.
VFSFileClose	Close an open file.
VFSFileRead	Read data from a file into the dynamic heap or any writable memory.
VFSFileReadData	Read data from a file into a chunk of memory in the storage heap.
VFSFileWrite	Write data to an open file.
VFSFileSeek	Set the position within an open file from which to read or write.
VFSFileTell	Get the current position of the file pointer within an open file.
VFSFileEOF	Get the end-of-file status for an open file.
VFSFileCreate	Create a file, given a volume reference number and a file path.
VFSFileDelete	Delete a closed file.
VFSFileRename	Rename a closed file.
VFSFileSize	Obtain the size of an open file.
VFSFileResize	Change the size of an open file.

Table 7.2 Common file operations (*continued*)

Function	Description
VFSFileGetAttributes	Obtain the attributes of an open file, including hidden, read-only, system, and archive bits. See “ File and Directory Attributes ” in the <i>Palm OS Programmer’s API Reference</i> for the bits that make up the attributes field.
VFSFileSetAttributes	Set the attributes of an open file, including hidden, read-only, system, and archive bits.
VFSFileGetDate	Get the created, modified, and last accessed dates for an open file.
VFSFileSetDate	Set the created, modified, and last accessed dates for an open file.

Once a file has been opened, it is identified by a unique reference number: a [FileRef](#). Functions that work with open files take a file reference. Others, such as `VFSFileOpen`, require a volume reference and a path that identifies the file within the volume. Note that all paths are volume relative, **and absolute within that volume**: the VFS Manager has no concept of a “current working directory,” so relative path names are not supported. The directory separator character is the forward slash: “/”. The root directory for the specified volume is specified by a path of “/”.

Naming Files

Different file systems support filenames and paths of different maximum lengths. The file system library is responsible for verifying whether or not a given path is valid and returns an error if it is not valid. From an application developer’s standpoint, filenames can be up to 255 characters long and can include any

normal character including spaces and lower case characters in any character set. They can also include the following special characters:

\$ % ' - _ @ ~ \ ! () ^ # & + , ; = []

The file system library is responsible for translating each filename and path into a format that is acceptable to the underlying file system. For example, when the 8.3 naming convention is used to translate a long filename, the following guidelines are used:

- The name is created from the first six valid, non-space characters which appear before the last period. The only valid characters are A-Z, 0-9, \$, %, ', -, _, @, ~, \, !, (,), ^, #, and &.
- The extension is the first three valid characters after the last period.
- The end of the six byte name has “~1” appended to it for the first occurrence of the shortened filename. Each subsequent occurrence uses the next unique number, so the second occurrence would have “~2” appended, and so on.

The standard VFAT file system library provided with all Palm Powered handhelds that support expansion uses the above rules to create FAT-compliant names from long filenames.

Working with Palm Databases

Expansion cards are often used to hold Palm applications and data in .prc and .pdb format. Due to the way that secondary storage media are connected to the Palm Powered handheld, applications cannot be run directly from the expansion card, nor can databases be manipulated using the Data Manager without first transferring them to main memory. Applications written to use the VFS Manager, however, can operate directly on files located on an expansion card.

NOTE: Whenever possible give the same name to the `.prc` file and to the database. If the `.prc` filename differs from the database name, and the user copies your application from the card to the handheld and then to another card, the filename may change. This is because the database name is used when an application is copied from the handheld to the card.

Stand-Alone Applications

To allow the user to run an application that is self-contained—that isn't accompanied by a separate database—you need only do one of two things:

- If the application is to be run whenever the card is inserted into the expansion slot, simply name the application `start.prc` and place it in the `/PALM` directory. The operating system takes care of transferring the application to main memory and starting it automatically.
- If the application is to be run on-demand, place it in the `/PALM/Launcher` directory. All applications located in this directory appear in the launcher when the user selects the category bearing the name of the expansion card.

Both of these mechanisms allow applications that were written without any knowledge of the VFS or Expansion Manager APIs to be run from a card. Because they are transferred to main memory prior to being run, such applications need not know that they are being run from an expansion card. Databases created by these applications are placed in the storage heap, as usual. When the card containing the application is removed, the application disappears from main memory unless it is running, in which case it remains until such time as the application is no longer running. Any databases it created remain. When the card is re-inserted and the application re-run, it is once again copied into main memory and is able to access those databases.

Applications with Static Data

Many applications are accompanied by one or more associated Palm databases when installed. These applications, at least to a limited

degree, need to be cognizant of the fact that they reside on an expansion card.

If there is no specific requirement for the application's data to be stored in Palm database format, you may want to use the VFS Manager's many file I/O operations to read and write the data on the card. Because of the large data storage capabilities of the expansion media relative to the handheld's memory, this latter solution is the one preferred by applications where large capacity data storage is a key feature.

Bundled Databases

When an application is launched from a card using the launcher, any bundled databases present in the /PALM/Launcher directory are also imported. Bundled databases have the same creator as the "owning" application and have the `dmHdrAttrBundle` bit set. Note that bundled databases are intended only for read-only data, such as a game-level database. Bundled databases are removed from main memory along with the application when the user switches to another application and are not copied back to the expansion card.

Transferring Palm Databases to and from Expansion Cards

The [`VFSExportDatabaseToFile`](#) function converts a database from its internal format on the handheld to its equivalent `.prc` or `.pdb` file format and transfers it to an expansion card. The [`VFSImportDatabaseFromFile`](#) function does the reverse; it transfers the `.prc` or `.pdb` file to main memory and converts it to the internal format used by the Palm OS. Use these functions when moving Palm databases between main memory and an expansion card. These two functions rely upon Exchange Manager routines to convert and transfer the data; see [Chapter 1](#), "[Object Exchange](#)" in *Palm OS Programmer's Companion*, vol. II, *Communications* for more information on using the Exchange Manager to send and receive data.

The `VFSExportDatabaseToFile` and `VFSImportDatabaseFromFile` routines are atomic and, depending on the size of the database and the mechanism by which it is being transferred, can take some time. Use [`VFSExportDatabaseToFileCustom`](#) and

[VFSTImportDatabaseFromFileCustom](#) if you want to display a progress dialog or allow the user to cancel the operation. These routines make repeated calls to a callback function that you specify; within this callback function you can update a progress indicator. The return value from your callback determines whether the database transfer should proceed; return `errNone` if it should continue, or return any other value to abort the process. See the documentation for [VFSEExportProcPtr](#) and [VFSTImportProcPtr](#) in the *Palm OS Programmer's API Reference* for the format of each callback function.

The following code excerpt illustrates the use of `VFSTImportDatabaseFromFileCustom` with a progress tracker.

Listing 7.3 Using VFSTImportDatabaseFromFileCustom

```
typedef struct {
    ProgressType *progressP;
    const Char   *nameP;
} CBDataType, *CBDataPtr;

static Boolean ProgressTextCB(PrgCallbackDataPtr cbP) {
    const Char *nameP = ((CBDataPtr) cbP->userDataP)->nameP;

    // Set up the progress text to be displayed
    StrPrintf(cbP->textP, "Importing %s.", nameP);
    cbP->textChanged = true;

    return true; // So what we specify here is used to update the dialog
}

static Err CopyProgressCB(UInt32 size, UInt32 offset, void *userDataP) {
    CBDataPtr CBDataP = (CBDataPtr) userDataP;

    if (offset == 0) { // If we're just starting, we need to set up the dialog
        CBDataP->progressP = PrgStartDialog("Importing Database", ProgressTextCB,
            CBDataP);

        if (!CBDataP->progressP)
            return memErrNotEnoughSpace;
    } else {
        EventType event;
        Boolean    handled;

        do {
```

Expansion

File Operations

```
    EvtGetEvent(&event, 0); // Check for events

    handled = PrgHandleEvent(CBDataP->progressP, &event);

    if (!handled) { // Did the user tap the "Cancel" button?
        if( PrgUserCancel(CBDataP->progressP) )
            return exgErrUserCancel;
    }
} while(event.eType != sysEventNilEvent);
}

return errNone;
}

static Err ImportFile(UInt16 volRefNum, Char *pathP, Char *nameP,
    UInt16 *cardNoP, LocalID *dbIDP)
{
    CBDataType userData;
    Char        fullPathP[256];
    Err         err;

    userData.progressP = NULL;
    userData.nameP = nameP;

    StrPrintf(fullPathP, "%s/%s", pathP, nameP); // rebuild full path to the
file
    err = VFSImportDatabaseFromFileCustom(volRefNum, fullPathP, cardNoP, dbIDP,
        CopyProgressCB, &userData);

    if (userData.progressP) // If the progress dialog was displayed, remove it.
        PrgStopDialog(userData.progressP, (err == exgErrUserCancel) );

    return err;
}
```

Exploring Palm Databases on Expansion Cards

The VFS Manager includes functions specifically designed for exploring the contents of a Palm database located on an expansion card. This access is read-only, however. You can extract individual records and resources from a database, and you can determine information such as the last modification date of a database on an expansion card. But there aren't parallel functions to write records and resources to a database or to update database-specific information for a database that is located on an expansion card. To

do this you need to import the database into main memory, make the necessary changes, and then export it back to the expansion card.

To obtain a single record from a database located on an expansion card without first importing the database into main memory, use [VFSFileDBGetRecord](#). This function is analogous to [DmGetRecord](#) but works with files on an external card rather than with databases in main memory. It transfers the specified record to the storage heap after allocating a handle of the appropriate size. Note that you'll need to free this memory, using [MemHandleFree](#), when the record is no longer needed.

The [VFSFileDBGetResource](#) function operates in a similar fashion, but instead of loading a particular database record it loads a specified resource from a resource database located on an expansion card. This resource is put onto the storage heap. Again, free this memory once the resource is no longer needed.

To obtain more general information about a database on an external card, use [VFSFileDBInfo](#). In addition to the information you could obtain about any file on an external card using the [VFSFileGetAttributes](#) and [VFSFileGetDate](#) functions, [VFSFileDBInfo](#) returns:

- the database name
- the version of the database
- the number of times the database was modified
- the application info block handle
- the sort info block handle
- the database's type
- the database's creator
- the number of records in the database

NOTE: The functions described in this section incur a lot of overhead in order to parse the database file format. Frequent use of these functions is not recommended. Also, if you request either the application info block handle or the sort info block handle, you must free the handle when it is no longer needed.

Directory Operations

All of the familiar operations you'd use to operate on directories are supported by the VFS Manager; these are listed in "[Common Operations](#)", below. One common operation—determining the files that are contained within a given directory—is covered in some detail in "[Enumerating the Files in a Directory](#)" on page 241. To improve data interchange with devices that aren't running the Palm OS, expansion card manufacturers have specified default directories for certain file types. "[Determining the Default Directory for a Particular File Type](#)" on page 242 discusses how you can both determine and set the default directory for a given file type.

Directory Paths

All paths are volume relative, **and absolute within that volume**: the VFS Manager has no concept of a "current working directory," so relative path names are not supported. The directory separator character is the forward slash: `"/"`. The root directory for the specified volume is specified by a path of `"/"`.

Common Operations

The VFS Manager provides all of the standard directory operations that should be familiar from desktop and larger computer systems. Because these functions work largely as you would expect, their use isn't detailed here. See the descriptions of each individual function in the *Palm OS Programmer's API Reference* for the arguments, return values, and side effects of each.

Note that most of these functions can be applied to files as well as directories.

Table 7.3 Common directory operations

Function	Description
VFSDirCreate	Create a new directory.
VFSFileDelete	Delete a directory, given a path.
VFSFileRename	Rename a directory.
VFSFileOpen	Open the file or directory.

Table 7.3 Common directory operations (*continued*)

Function	Description
VFSFileClose	Close the file or directory.
VFSFileGetAttributes	Obtain the attributes of an open directory, including hidden, read-only, system, and archive bits. See “ File and Directory Attributes ” in the <i>Palm OS Programmer’s API Reference</i> for the bits that make up the attributes field.
VFSFileSetAttributes	Set the attributes of an open directory, including hidden, read-only, system, and archive bits.
VFSFileGetDate	Get the created, modified, and last accessed dates for an open file.
VFSFileSetDate	Set the created, modified, and last accessed dates for an open file.

Enumerating the Files in a Directory

Enumerating the files within a directory is made simple due to the presence of the [VFSDirEntryEnumerate](#) function. The use of this function is illustrated below. Note that `volRefNum` and `dirPathStr` must be declared and initialized prior to the following code.

Listing 7.4 Enumerating a directory’s contents

```
// Open the directory and iterate through the files in it.
// volRefNum must have already been defined.
err = VFSFileOpen(volRefNum, "/", vfsModeRead, &dirRef);
if(err == errNone) {
    // Iterate through all the files in the open directory
    UInt32 fileIterator;
    FileInfoType fileInfo;
    FileRef dirRef;
    Char *fileName = MemPtrNew(256);    // should check for err
```

Expansion

Directory Operations

```
fileInfo.nameP = fileName;    // point to local buffer
fileInfo.nameBufLen = sizeof(fileName);
fileIterator = expIteratorStart;
while (fileIterator != expIteratorStop) {
    // Get the next file
    err = VFSDirEntryEnumerate(dirRef, &fileIterator,
                               &fileInfo);
    if(err == errNone) {
        // Process the file here.
    }
} else {
    // handle directory open error here
}
MemPtrFree(fileName);
}
```

Each time through the while loop, `VFSDirEntryEnumerate` sets the [FileInfoType](#) structure as appropriate for the file currently being enumerated. Note that if you want the file name it isn't enough to simply allocate space for the `FileInfoType` structure; you must also allocate a buffer for the filename, set the appropriate pointer to it in the `FileInfoType` structure, and specify your buffer's length. Since the only other information encapsulated within `FileInfoType` is the file's attributes, most applications will want to also know the file's name.

Determining the Default Directory for a Particular File Type

As explained in "[Standard Directories](#)" on page 214, the expansion capabilities of Palm OS 4.0 include a mechanism to map MIME types or file extensions to specific directory names. This mechanism is specific to the slot driver: where an image might be stored in the `/Images` directory on a Memory Stick, on an MMC card it may be stored in the `/DCIM` directory. The VFS Manager includes a function that enables you to get the default directory on a particular volume for a given file extension or MIME type, along with functions that allow you to register and un-register your own default directories.

The [VFSGetDefaultDirectory](#) function takes a volume reference and a string containing the file extension or MIME type and returns a string containing the full path to the corresponding

default directory. When specifying the file type, either supply a MIME media type/subtype pair, such as “image/jpeg”, “text/plain”, or “audio/basic”; or a file extension, such as “.jpeg”. As with most other Palm OS functions, you’ll need to pre-allocate a buffer to contain the returned path. Supply a pointer to this buffer along with the buffer’s length. The length is updated upon return to indicate the actual length of the path, which won’t exceed the originally-specified buffer length.

The default directory registered for a given file type is intended to be the “root” default directory. If a given default directory has one or more subdirectories, applications should also search those subdirectories for files of the appropriate type.

VFSGetDefaultDirectory allows you to determine the directory associated with a particular file suffix. However, there’s no way to get the entire list of file suffixes that are mapped to default directories. For this reason, CardInfo keeps its own list of possible file suffixes. It iterates through this list, calling VFSGetDefaultDirectory for each file suffix to get the full path to the corresponding default directory. It then looks into each default directory for files that match the expected suffix or suffixes for that directory.

Registering New Default Directories

In addition to the default directories that the underlying slot driver is already aware of, you can create your own mappings between files of a given type and a specific directory on a particular kind of external storage card. Most applications don’t need this functionality; it is generally used by a slot driver to register those files and media types that are supported by that slot driver. However, [VFSRegisterDefaultDirectory](#) and its opposite, [VFSUnregisterDefaultDirectory](#), are available to those applications that need them. Such applications should generally register the desired file types for `expMediaType_Any`. This is a wildcard which works for all media types; it can be overridden by a registration that specifies a real media type.

If a default directory has already been registered for a given file/media type combination, applications should use the pre-existing registration instead of establishing a new one. Existing registrations should generally not be removed.

Default Directories Registered at Initialization

The VFS Manager registers the following under the `expMediaType_Any` media type, which [VFSGetDefaultDirectory](#) reverts to when there is no default registered by the slot driver for a given media type.

Table 7.4 Default registrations

File Type	Path
<code>.prc</code>	<code>/PALM/Launcher/</code>
<code>.pdb</code>	<code>/PALM/Launcher/</code>
<code>.pqa</code>	<code>/PALM/Launcher/</code>
<code>application/vnd.palm</code>	<code>/PALM/Launcher/</code>
<code>.jpg</code>	<code>/DCIM/</code>
<code>.jpeg</code>	<code>/DCIM/</code>
<code>image/jpeg</code>	<code>/DCIM/</code>
<code>.gif</code>	<code>/DCIM/</code>
<code>image/gif</code>	<code>/DCIM/</code>
<code>.qt</code>	<code>/DCIM/</code>
<code>.mov</code>	<code>/DCIM/</code>
<code>video/quicktime</code>	<code>/DCIM/</code>
<code>.avi</code>	<code>/DCIM/</code>
<code>video/x-msvideo</code>	<code>/DCIM/</code>
<code>.mpg</code>	<code>/DCIM/</code>
<code>.mpeg</code>	<code>/DCIM/</code>
<code>video/mpeg</code>	<code>/DCIM/</code>
<code>.mp3</code>	<code>/AUDIO/</code>
<code>.wav</code>	<code>/AUDIO/</code>
<code>audio/x-wav</code>	<code>/AUDIO/</code>

The SD slot driver provided by PalmSource, Inc. registers the following, since it has an appropriate specification for these file types:

Table 7.5 Directories registered by the SD slot driver

File Type	Path
.jpg	/DCIM/
.jpeg	/DCIM/
image/jpeg	/DCIM/
.qt	/DCIM/
.mov	/DCIM/
video/quicktime	/DCIM/
.avi	/DCIM/
video/x-msvideo	/DCIM/

Although the directories registered by Palm's SD slot driver all happen to be duplicates of the default registrations made by the VFS Manager, they are also registered under the SD media type since the SD specification explicitly includes them.

Slot drivers written by other Palm Powered handheld manufacturers that support different media types, such as Memory Stick, will register default directories appropriate to their media's specifications. In some cases these registrations will override the `expMediaType_Any` media type registration, or in some cases augment the `expMediaType_Any` registrations with file types not previously registered.

These registrations are intended to aid applications developers, but you aren't required to follow them. Although you can choose to ignore these registrations, by following them you'll improve interoperability between applications and other devices. For example, a digital camera which conforms to the media specifications will put its pictures into the registered directory (or a subdirectory of it) appropriate for the image format and media type. By looking up the registered directory for that format, an image

viewer application on the handheld can easily find the images without having to search the entire card. These registrations also help prevent different developers from hard-coding different paths for specific file types. Thus, if a user has two different image viewer applications, both will look in the same location and find the same set of images.

Registering these file types at initialization allows you to use HotSync to transfer files of these types to an expansion card. During the HotSync process, files of the registered types are placed directly in the specified directories on the card.

Custom Calls

Recognizing that some file systems may implement functionality not covered by the APIs included in the VFS and Expansion Managers, the VFS Manager includes a single function that exists solely to give developers access to the underlying file system. This function, [VFSCustomControl](#), takes a registered creator code and a selector that together identify the operation that is to be performed. `VFSCustomControl` can either request that a specific file system perform the specified operation, or it can iterate through all of the currently-registered file systems in an effort to locate one that responds to the desired operation.

Parameters are passed to the file system's custom function through a single `VFSCustomControl` parameter. This parameter, `valueP`, is declared as a `void *` so you can pass a pointer to a structure of any type. A second parameter, `valueLenP`, allows you to specify the length of `valueP`. Note that these values are simply passed to the file system and are in reality dependent upon the underlying file system. See the description of [VFSCustomControl](#) in the *Palm OS Programmer's API Reference* for more information.

Because `VFSCustomControl` is designed to allow access to non-standard functionality provided by a particular file system, see the documentation provided with that file system for a list of any custom functions that it provides.

Custom I/O

While the Expansion and VFS Managers provide higher-level OS support for secondary storage applications, they don't attempt to present anything more than a raw interface to custom I/O applications. Since it isn't really possible to envision all uses of an expansion mechanism, the Expansion and VFS Managers simply try to get out of the way of custom hardware.

The Expansion Manager provides insertion and removal notification and can load and unload drivers. Everything else is the responsibility of the application developer. Palm has defined a common expansion slot driver API which is extensible by licensees. This API is designed to support all of the needs of the Expansion Manager, the VFS Manager, and the file system libraries. Applications that need to communicate with an I/O device, however, may need to go beyond the provided APIs. Such applications should wherever possible use the slot custom call, which provides direct access to the expansion slot driver. See the developer documentation provided to licensees for more information on slot drivers and the slot custom call. For documentation on functions made available by a particular I/O device, along with how you access those functions, contact the I/O device manufacturer.

Debugging

The Palm OS Emulator has been extended to support the expansion capabilities of the VFS Manager. It can be configured to present a directory on the host file system as a volume to the virtual file system. You can populate this directory on your host system and then simulate a volume mount. Changes made to the emulated expansion card's contents can be verified simply by examining the directory on the host.

For more information on configuring and operating the Palm OS Emulator, see the *Palm OS Programming Development Tools Guide*.

Expansion

Summary of Expansion and VFS Managers

Summary of Expansion and VFS Managers

Expansion Manager Functions

ExpCardGetSerialPort	ExpSlotDriverRemove
ExpCardInfo	ExpSlotEnumerate
ExpCardPresent	ExpSlotLibFind
ExpSlotDriverInstall	

VFS Manager Functions

Working with Files

VFSFileClose	VFSFileRename
VFSFileCreate	VFSFileResize
VFSFileDelete	VFSFileSeek
VFSFileEOF	VFSFileSetAttributes
VFSFileGetAttributes	VFSFileSetDate
VFSFileGetDate	VFSFileSize
VFSFileOpen	VFSFileTell
VFSFileRead	VFSFileWrite
VFSFileReadData	

Working with Directories

VFSDirCreate	VFSFileSetAttributes
VFSDirEntryEnumerate	VFSFileSetDate
VFSFileClose	VFSGetDefaultDirectory
VFSFileDelete	VFSRegisterDefaultDirectory
VFSFileGetAttributes	VFSUnregisterDefaultDirectory
VFSFileGetDate	
VFSFileOpen	
VFSFileRename	

Working with Volumes

VFSVolumeEnumerate	VFSVolumeMount
VFSVolumeFormat	VFSVolumeSetLabel
VFSVolumeGetLabel	VFSVolumeSize
VFSVolumeInfo	VFSVolumeUnmount

VFS Manager Functions

Miscellaneous Functions[VFSCustomControl](#)[VFSExportDatabaseToFile](#)[VFSExportDatabaseToFileCustom](#)[VFSFileDBInfo](#)[VFSFileDBGetRecord](#)[VFSFileDBGetResource](#)[VFSImportDatabaseFromFile](#)[VFSImportDatabaseFromFileCustom](#)[VFSInstallFSLib](#)[VFSRemoveFSLib](#)

Text

This chapter describes how to work with text in the user interface—whether it’s text the user has entered or text that your application has created to display on the screen. When you work with text, you must take special care to do so in a way that makes your application easily localizable. This chapter describes how to write code that manipulates characters and strings in such a way that it works properly for any language that is supported by Palm OS®. It covers:

- [Text Manager and International Manager](#)
- [Characters](#)
- [Strings](#)
- [Fonts](#)

When you work with text, you work mainly with the Text Manager and the String Manager. Text Manager support begins in Palm OS 3.1. If you want to support releases earlier than Palm OS 3.1, use the PalmOSGlue library described in “[Backward Compatibility with PalmOSGlue](#)” on page 14. This chapter notes all functions that have a glue equivalent in parentheses and shows code examples using the PalmOSGlue equivalents.

IMPORTANT: Palm OS version 3.1 introduced some changes to the Latin character encoding and to some of the String Manager functions. If you are updating a legacy application written before the release of Palm OS 3.1, these changes may affect your application. See “[3.1 New Feature Set](#)” on page 2262 of the *Palm OS Programmer’s API Reference* for details about these changes.

Text Manager and International Manager

The Palm OS provides two managers that help you work with localizable strings and characters. These managers are called the Text Manager and the International Manager.

Computers represent the characters in an alphabet with a numeric code. The set of numeric codes for a given alphabet is called a **character encoding**. Of course, a character encoding contains more than codes for the letters of an alphabet. It also encodes punctuation, numbers, control characters, and any other characters deemed necessary. The set of characters that a character encoding represents is called, appropriately enough, a **character set**.

Different languages use different alphabets. Most European languages use the Latin alphabet. The Latin alphabet is relatively small, so its characters can be represented using a single-byte encoding ranging from 32 to 255. On the other hand, Asian languages such as Chinese, Korean, and Japanese require their own alphabets, which are much larger. These larger character sets are represented by a combination of single-byte and double-byte numeric codes ranging from 32 to 65,535.

A given Palm Powered™ handheld supports one character encoding. Although Palm OS supports multiple character encodings, a given handheld uses only one of those encodings. For example, a French handheld uses the Palm™ Latin encoding, which is identical to the Microsoft® Windows® code page 1252 character encoding (an extension of ISO Latin 1) but includes Palm-specific characters in the control range. A Japanese handheld, on the other hand would use the Palm Shift JIS character encoding, which is identical to Microsoft Windows code page 932 (an extension of Shift JIS) but includes Palm-specific characters in the control range. Code page 932 is not supported on the French handheld, and code page 1252 is not supported on the Japanese handheld even though they both use the same version of Palm OS. No matter what the encoding is on a handheld, Palm guarantees that the low ASCII characters (0 to 0x7F) are the same. The exception to this rule is 0x5C, which is a yen symbol on Japanese handhelds and a backslash on all others.

The Text Manager allows you to work with text, strings, and characters independent of the character encoding. If you use Text Manager routines and don't work directly with string data, your

code should work on any system, regardless of which language and character encoding the handheld supports (as long as it supports the Text Manager).

The International Manager's job is to detect which character encoding a handheld uses and initialize the corresponding version of the Text Manager. The International Manager also sets system features that identify which encoding and fonts are used. For the most part, you don't work with the International Manager directly.

Characters

Depending on the handheld's supported languages, Palm OS may encode characters using either a single-byte encoding or a multi-byte encoding. Because you do not know which character encoding is used until runtime, **you should never make an assumption about the size of a character.**

For the most part, your application does not need to know which character encoding is used, and in fact, it should make no assumptions about the encoding or about the size of characters. Instead, your code should use Text Manager functions to manipulate characters. This section describes how to work with characters correctly. It covers:

- [Declaring Character Variables](#)
- [Using Character Constants](#)
- [Missing and Invalid Characters](#)
- [Retrieving a Character's Attributes](#)
- [Virtual Characters](#)
- [Retrieving the Character Encoding](#)

Declaring Character Variables

Declare all character variables to be of type `WChar`. `WChar` is a 16-bit unsigned type that can accommodate characters of any encoding. Don't use `Char`. `Char` is an 8-bit variable that cannot accommodate larger character encodings. The only time you should ever use `Char` is to pass a parameter to an older Palm OS function.

```
WChar ch; // Right. 16-bit character.  
Char ch; // Wrong. 8-bit character.
```

When you receive input characters through the `keyDownEvent`, you'll receive a `WChar` value. (That is, the `data.keyDown.chr` field is a `WChar`.)

Even though character variables are now declared as `WChar`, string variables are still declared as `Char *`, even though they may contain multi-byte characters. See the section "[Strings](#)" for more information on strings.

Using Character Constants

Character constants are defined in several header files. The header file `Chars.h` contains characters that are guaranteed to be supported on all systems regardless of the encoding. Other header files exist for each supported character encoding and contain characters specific to that encoding. The character encoding-specific header files are not included in the `PalmOS.h` header by default because they define characters that are not available on every system.

To make it easier for the compiler to find character encoding problems with your project, make a practice of using the character constants defined in these header files rather than directly assigning a character variable to a value. For example, suppose your code contained this statement:

```
WChar ch = 'å'; // WRONG! Don't use.
```

This statement may work on a Latin system, but it would cause problems on an Asian-language system because the `å` character does not exist. If you instead assign the value this way:

```
WChar ch = chrSmall_A_RingAbove;
```

you'll find the problem at compile time because the `chrSmall_A_RingAbove` constant is defined in `CharLatin.h`, which is not included by default.

Missing and Invalid Characters

If during application testing, you see an open rectangle, a shaded rectangle, or a gray square displayed on the screen, you have a missing character.

A **missing character** is one that is valid within the character encoding but the current font is not able to display it. In this case, nothing is wrong with your code other than you have chosen the wrong font. The system displays an open rectangle in place of a missing single-byte rectangle (see [Figure 8.1](#)).

Figure 8.1 Missing characters



Missing single-byte character

In multi-byte character encodings, a character may be missing as described above, or it may be invalid. In single-byte character encodings, there's a one-to-one correspondence between numeric values and characters to represent. This is not the case with multi-byte character encodings. In multi-byte character encodings, there are more possible values than there are characters to represent. Thus, a character variable could end up containing an **invalid character**—a value that doesn't actually represent a character.

If the system is asked to display an invalid character, it prints an open rectangle for the first invalid byte. Then it starts over at the next byte. Thus, the next character displayed and possibly even the remaining text displayed is probably not what you want. Check your code for the following:

- Truncating strings. You might have truncated a string in the middle of a multi-byte character.
- Appending characters from one encoding set to a string in a different encoding.
- Arithmetic on character variables that could result in an invalid character value.
- Arithmetic on a string pointer that could result in pointing to an intra-character boundary. See "[Performing String Pointer Manipulation](#)" for more information.
- Assumptions that a character is always a single byte long.

Use the Text Manager function [TxtCharIsValid](#) (`TxtGlueCharIsValid`) to determine whether a character is valid or not.

Retrieving a Character's Attributes

The Text Manager defines certain functions that retrieve a character's attributes, such as whether the character is alphanumeric, and so on. You can use these functions on any character, regardless of its size and encoding.

A character also has attributes unique to its encoding. Functions to retrieve those attributes are defined in the header files specific to the encoding.

WARNING! In previous versions of the Palm OS, the header file `CharAttr.h` defined character attribute macros such as `IsAscii`. Using these macros on double-byte characters produces incorrect results. Use the Text Manager macros instead of the `CharAttr.h` macros.

Virtual Characters

Virtual characters are nondisplayable characters that trigger special events in the operating system, such as displaying low battery warnings or displaying the keyboard dialog. Virtual characters should never occur in any data and should never appear on the screen.

The Palm OS uses character codes 256 decimal and greater for virtual characters. The range for these characters may actually overlap the range for “real” characters (characters that should appear on the screen). The `keyDownEvent` distinguishes a virtual character from a displayable character by setting the command bit in the event record.

The best way to check for virtual characters, including virtual characters that represent the hard keys, is to use the [TxtGlueCharIsVirtual](#) function defined in the `PalmOSGlue` library. See [Listing 8.1](#).

Listing 8.1 Checking for virtual characters

```
if (TxtGlueCharIsVirtual (eventP->data.keyDown.modifiers,  
    eventP->data.keyDown.chr)) {  
    if (TxtCharIsHardKey (event->data.keyDown.modifiers,  
        event->data.keyDown.chr)) {  
        // Handle hard key virtual character.  
    } else {  
        // Handle standard virtual character.  
    }  
} else {  
    // Handle regular character.  
}
```

Retrieving the Character Encoding

Occasionally, you may need to determine which character encoding is being used. For example, your application may need to do some unique text manipulation if it is being run on a European handheld. You can retrieve the character encoding from the system feature set using the `FtrGet` function as shown in [Listing 8.2](#).

Listing 8.2 Retrieving the character encoding

```
UInt32 encoding;  
Char* encodingName;  
if (FtrGet(sysFtrCreator, sysFtrNumEncoding, &encoding) != 0)  
    encoding = charEncodingPalmLatin;  
    //default encoding  
if (encoding == charEncodingPalmSJIS) {  
    // encoding for Palm Shift-JIS  
} else if (encoding == charEncodingPalmLatin) {  
    // extension of ISO Latin 1  
}  
  
// The following Text Manager function returns the  
// official name of the encoding as required by  
// Internet applications.  
encodingName = TxtGlueEncodingName(encoding);
```

Strings

When working with text as strings, you use the String Manager and the Text Manager.

The String Manager is supported in all releases of Palm OS. It is closely modeled after the standard C string-manipulation functions like `strcpy`, `strcat`, and so on. Note that the standard C functions are not built in to Palm OS. Use String Manager calls instead of standard C calls to make your application smaller.

The Text Manager was added in Palm OS 3.1 to provide support for multi-byte strings. On systems that support the Text Manager, strings are made up of characters that are either a single-byte long or multiple bytes long, up to four bytes. As stated previously, character variables are always two bytes long. However, when you add a character to a string, the operating system may shrink it down to a single byte if it's a low ASCII character. Thus, any string that you work with may contain a mix of single-byte and multi-byte characters.

Applications can use both the Text Manager and the String Manager to work with strings. The String Manager functions in Palm OS 3.1 and later can work with strings containing multi-byte characters. Use the Text Manager functions when:

- A String Manager equivalent is not available.
- The length of the matching strings are important. For example, to compare two strings, you can use either [StrCompare](#) or [TxtCompare](#). The difference between the two is that `StrCompare` does not return the length of the characters that matched. `TxtCompare` does.

This section discusses the following topics:

- [Manipulating Strings](#)
- [Performing String Pointer Manipulation](#)
- [Truncating Displayed Text](#)
- [Comparing Strings](#)
- [Global Find](#)
- [Dynamically Creating String Content](#)
- [Using the StrVPrintf Function](#)

TIP: Many of the pre-3.1 Palm OS functions have been modified to work with strings containing multi-byte characters. All Palm OS functions that return the length of a string, such as `FldGetTextLength` and `StrLen`, always return the size of the string in bytes, not the number of characters in the string. Similarly, functions that work with string offsets always use the offset in bytes, not characters.

Manipulating Strings

Any time that you want to work with character pointers, you need to be careful not to point to an intra-character boundary (a middle or end byte of a multi-byte character). For example, any time that you want to set the insertion point position in a text field or set the text field's selection, you must make sure that you use byte offsets that point to inter-character boundaries. (The **inter-character boundary** is both the start of one character and the end of the previous character, except when the offset points to the very beginning or very end of a string.)

Suppose you want to iterate through a string character by character. Traditionally, C code uses a character pointer or byte counter to iterate through a string a character at a time. Such code will not work properly on systems with multi-byte characters. Instead, if you want to iterate through a string a character at a time, use Text Manager functions:

- [`TxtGetNextChar`](#) (`TxtGlueGetNextChar`) retrieves the next character in a string.
- [`TxtGetPreviousChar`](#) (`TxtGlueGetPreviousChar`) retrieves the previous character in a string.
- [`TxtSetNextChar`](#) (`TxtGlueSetNextChar`) changes the next character in a string and can be used to fill a string buffer.

Each of these three functions returns the size of the character in question, so you can use it to determine the offset to use for the next character. For example, [Listing 8.3](#) shows how to iterate through a string character by character until a particular character is found.

Listing 8.3 Iterating through a string or text

```
Char* buffer; // assume this exists
UInt16 bufLen = StrLen(buffer);
// Length of the input text.
WChar ch = 0;
UInt16 i = 0;
while ((i < bufLen) && (ch != chrAsterisk))
    i+= TxtGlueGetNextChar(buffer, i, &ch);
```

The Text Manager also contains functions that let you determine the size of a character in bytes without iterating through the string:

- [TxtCharSize](#) (TxtGlueCharSize) returns how much space a given character will take up inside of a string.
- [TxtCharBounds](#) (TxtGlueCharBounds) determines the boundaries of a given character within a given string.

Listing 8.4 Working with arbitrary limits

```
UInt32* charStart, charEnd;
Char* fldTextP = FldGetTextPtr(fld);
TxtGlueCharBounds(fldTextP,
    min(kMaxBytesToProcess, FldGetTextLength(fld)),
    &charStart, &charEnd);
// process only the first charStart bytes of text.
```

Performing String Pointer Manipulation

Never perform any pointer manipulation on strings you pass to the Text Manager unless you use Text Manager calls to do the manipulation. For Text Manager functions to work properly, the string pointer must point to the first byte of a character. If you use Text Manager functions when manipulating a string pointer, you can be certain that your pointer always points to the beginning of a character. Otherwise, you run the risk of pointing to an inter-character boundary.

Listing 8.5 String pointer manipulation

```
// WRONG! buffer + kMaxStrLength is not
// guaranteed to point to start of character.
buffer[kMaxStrLength] = '\0';

// Right. Truncate at a character boundary.
UInt32 charStart, charEnd;
TxtCharBounds(buffer, kMaxStrLength,
    &charStart, &charEnd);
TxtGlueSetNextChar(buffer, charStart, chrNull);
```

Truncating Displayed Text

If you're performing drawing operations, you often have to determine where to truncate a string if it's too long to fit in the available space. Two functions help you perform this task on strings with multi-byte characters:

- [WinDrawTruncChars](#) (WinGlueDrawTruncChars) — This function draws a string within a specified width, determining automatically where to truncate the string. If it can, it draws the entire string. If the string doesn't fit in the space, it draws one less than the number of characters that fit and then ends the string with an ellipsis (...).
- [FntWidthToOffset](#) (FntGlueWidthToOffset) — This function returns the byte offset of the character displayed at a given pixel position. It can also return the width of the text up to that offset.

[Listing 8.6](#) shows how you can use `FntWidthToOffset` to determine how many lines are necessary to write a string to the screen. This example passes 160 as the pixel position so that upon return, `widthToOffset` contains the byte offset of the last character in the string that can be displayed on a single line. The characters up to and including the one at `widthToOffset` are drawn, then the `msg` pointer is advanced in the string by `widthToOffset` characters, and `FntWidthToOffset` is called again to find out how many characters fit on the next line of text. The process is repeated until all of the characters in the string have been drawn.

Listing 8.6 Drawing multiple lines of text

```
Coord y;
Char *msg;
Int16 msgWidth;
Int16 widthToOffset = 0;
Int16 pixelWidth = 160;
Int16 msgLength = StrLen(msg);

while (msg && *msg) {
    widthToOffset = FntGlueWidthToOffset(msg, msgLength,
        pixelWidth, NULL, &msgWidth);
    WinDrawChars(msg, widthToOffset, 0, y);
    y += FntLineHeight();
    msg += widthToOffset;
    msgLength = StrLen(msg);
}
```

Comparing Strings

Use the Text Manager functions [TxtCompare](#) (TxtGlueCompare) and [TxtCaselessCompare](#) (TxtGlueCaselessCompare) to perform comparisons of strings.

In character encodings that use multi-byte characters, some characters are accurately represented as either single-byte characters or multi-byte characters. That is, a character might have both a single-byte representation and a double-byte representation. One string might use the single-byte representation and another might use the multi-byte representation. Users expect the characters to match regardless of how many bytes a string uses to store that character. `TxtCompare` and `TxtCaselessCompare` can accurately match single-byte characters with their multi-byte equivalents.

Because a single-byte character might be matched with a multi-byte character, two strings might be considered equal even though they have different lengths. For this reason, `TxtCompare` and `TxtCaselessCompare` take two parameters in which they pass back the length of matching text in each of the two strings. See the function descriptions in the *Palm OS Programmer's API Reference* for more information.

Note that the String Manager functions `StrCompare` and `StrCaselessCompare` are equivalent, but they do not pass back the length of the matching text.

Global Find

A special case of performing string comparison is implementing the global system find facility. To implement this facility, you should call [TxtFindString](#) (`TxtGlueFindString`). As with `TxtCompare` and `TxtCaselessCompare`, `TxtFindString` accurately matches single-byte characters with their corresponding multi-byte characters. Plus, it passes back the length of the matched text. You'll need this value to highlight the matching text when the system requests that you display the matching record.

Older versions of Palm OS use the function [FindStrInStr](#). `FindStrInStr` is not able to return the length of the matching text. Instead, it assumes that characters within the string are always one byte long.

When the user taps the Find icon, the system sends the launch code [sysAppLaunchCmdFind](#) to each application. [Listing 8.7](#) shows an example of a function that should be called in response to that launch code. This function implements a global find that works on all systems whether the Text Manager exists or not. When the user taps one of the results displayed in the Find Results dialog, the system sends a [sysAppLaunchCmdGoto](#) launch code to the application containing the matching record. [Listing 8.8](#) shows how to respond to the `sysAppLaunchCmdGoto` launch code.

These two listings are only code excerpts. For the complete implementation of these two functions, see the example code in the Palm OS SDK.

Note that if you want to use `TxtFindString` to implement a search within your application (as opposed to the global find facility), you need to call [TxtGluePrepFindString](#) before you call `TxtFindString` to ensure that the string is in the proper format. (In the global find facility, the system has already prepared the string before your code is executed.)

Listing 8.7 Implementing global find

```
static void Search (FindParamsPtr findParams)
{
    UInt16 recordIndex = 0;
    DmOpenRef dbP;
    UInt16 cardNo = 0;
    LocalID dbID;
    MemoDBRecordPtr memoRecP;

    // Open the database to be searched.
    dbP = DmOpenDatabaseByTypeCreator(memoDBType,
        sysFileCMemo, findParams->dbAccessMode);
    DmOpenDatabaseInfo(dbP, &dbID, 0, 0, &cardNo,
        0);

    // Get first record to search.
    memoRecP = GetRecordPtr(dbP, recordIndex);
    while (memoRecP != NULL) {
        Boolean done;
        Boolean match;
        UInt32 matchPos, matchLength;

        // TxtGlueFindString calls TxtFindString if it
        // exists, or else it implements the Latin
        // equivalent of it.
        match = TxtGlueFindString (&(memoRecP->note),
            findParams->strToFind, &matchPos,
            &matchLength);

        if (match) {
            done = FindSaveMatch (findParams,
                recordIndex, matchPos, 0, matchLength,
                cardNo, dbIDP);
        }
        MemPtrUnlock (memoRecP);

        if (done) break;
        recordIndex += 1;
    }
    DmCloseDatabase (dbP);
}
```

Listing 8.8 Displaying the matching record

```
static void GoToRecord (GoToParamsPtr goToParams, Boolean
launchingApp)
{
    UInt16 recordNum;
    EventType event;

    recordNum = goToParams->recordNum;
    ...

    // Send an event to goto a form and select the
    // matching text.
    MemSet (&event, sizeof(EventType), 0);

    event.eType = frmLoadEvent;
    event.data.frmLoad.formID = EditView;
    EvtAddEventToQueue (&event);

    MemSet (&event, sizeof(EventType), 0);
    event.eType = frmGotoEvent;
    event.data.frmGoto.recordNum = recordNum;
    event.data.frmGoto.matchPos =
        goToParams->matchPos;
    event.data.frmGoto.matchLen =
        goToParams->matchCustom;
    event.data.frmGoto.matchFieldNum =
        goToParams->matchFieldNum;
    event.data.frmGoto.formID = EditView;
    EvtAddEventToQueue (&event);
    ...
}
```

Dynamically Creating String Content

When working with strings in a localized application, you never hard code them. Instead, you store strings in a resource and use the resource to display the text. If you need to create the contents of the string at runtime, store a template for the string as a resource and then substitute values as needed.

For example, consider the Edit view of the Memo application. Its title bar contains a string such as “Memo 3 of 10.” The number of the memo being displayed and the total number of memos cannot be determined until runtime.

To create such a string, use a template resource and the Text Manager function [TxtParamString](#) (TxtGlueParamString). TxtParamString allows you to search for the sequence ^0, ^1, up to ^3 and replace each of these with a different string. If you need more parameters, you can use [TxtReplaceStr](#) (TxtGlueReplaceStr), which allows you to replace up to ^9; however, TxtReplaceStr only allows you to replace one of these sequences at a time.

In the Memo title bar example, you'd create a string resource that looks like this:

```
Memo ^0 of ^1
```

And your code might look like this:

Listing 8.9 Using string templates

```
static void EditViewSetTitle (void)
{
    Char* titleTemplateP;
    FormPtr frm;
    Char posStr [maxStrIToALen];
    Char totalStr [maxStrIToALen];
    UInt16 pos;
    UInt16 length;

    // Format as strings, the memo's position within
    // its category, and the total number of memos
    // in the category.
    pos = DmPositionInCategory (MemoPadDB,
        CurrentRecord, RecordCategory);
    StrIToA (posStr, pos+1);

    if (MemosInCategory == memosInCategoryUnknown)
        MemosInCategory = DmNumRecordsInCategory
            (MemoPadDB, RecordCategory);
    StrIToA (totalStr, MemosInCategory);

    // Get the title template string. It contains
    // '^0' and '^1' chars which we replace with the
    // position of CurrentRecord within
    // CurrentCategory and with the total count of
    // records in CurrentCategory ().
    titleTemplateP = MemHandleLock (DmGetResource
        (strRsc, EditViewTitleTemplateStringString));
```



```
    EditViewTitlePtr =  
        TxtGlueParamString(titleTemplateP, posStr,  
            totalStr, NULL, NULL);  
  
    // Now set the title to use the new title  
    // string.  
    frm = FrmGetFormPtr (MemoPadEditForm);  
    FrmSetTitle (frm, EditViewTitlePtr);  
    MemPtrUnlock(titleTemplateP);  
}
```

Using the StrVPrintf Function

Like the C `vsprintf` function, the [StrVPrintf](#) function is designed to be called by your own function that takes a variable number of arguments and passes them to `StrVPrintf` for formatting. This section gives a brief overview of how to use `StrVPrintf`. For more details, refer to `vsprintf` and the use of the `stdarg.h` macros in a standard C reference book.

When you call `StrVPrintf`, you must use the special macros from `stdarg.h` to access the optional arguments (those specified after the fixed arguments) passed to your function. This is necessary, because when you declare your function that takes an optional number of arguments, you declare it using an ellipsis at the end of the argument list:

```
MyPrintf(CharPtr s, CharPtr formatStr, ...);
```

The ellipsis indicates that zero or more optional arguments may be passed to the function following the `formatStr` argument. Since these optional arguments don't have names, the `stdarg.h` macros must be used to access them before they can be passed to `StrVPrintf`.

To use these macros in your function, first declare an `args` variable as type `va_list`:

```
va_list args;
```

Next, initialize the `args` variable to point to the optional argument list by using `va_start`:

```
va_start(args, formatStr);
```

Note that the second argument to the `va_start` macro is the last required argument to your function (last before the optional arguments begin). Now you can pass the `args` variable as the last parameter to the `StrVPrintf` function:

```
StrVPrintf(text, formatStr, args);
```

When you are finished, invoke the macro `va_end` before returning from your function:

```
va_end(args);
```

Note that the [StrPrintf](#) and `StrVPrintf` functions implement only a subset of the conversion specifications allowed by the ANSI C function `vsprintf`. See the [StrVPrintf](#) function reference for details.

Fonts

All fonts in Palm OS are bitmapped fonts. A **bitmapped font** is one that provides a separate bitmap for each glyph in each size and style. Scalable fonts such as TrueType or PostScript fonts are not supported.

Each font is associated with a particular character encoding. The font contains **glyphs** that define how to draw each character in the encoding.

Palm OS provides built-in fonts, and in Palm OS 3.0 and later, allows you to create your own fonts. If the [High-Density Display Feature Set](#) is present, high-density fonts are supported and used on high-density displays. This section describes the font support in Palm OS 3.0 and later. It covers:

- [Built-in Fonts](#)
- [Selecting Which Font to Use](#)
- [Fonts for High-Density Displays](#)
- [Setting the Font Programmatically](#)

- [Obtaining Font Information](#)
- [Creating Custom Fonts](#)

Built-in Fonts

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

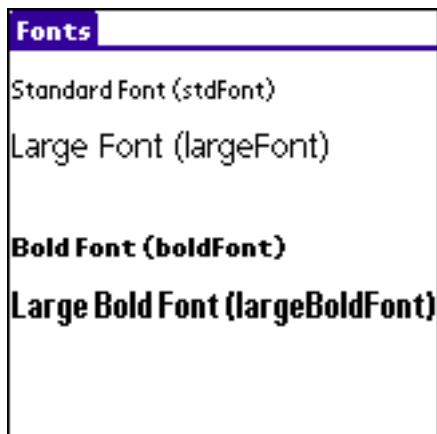
[Table 8.1](#) lists and describes the built-in fonts that may be used to display text.

Table 8.1 Built-in text fonts

Constant	Description
<code>stdFont</code>	A small standard font used to display user input. This font is small to display as much text as possible.
<code>largeFont</code>	A larger font provided as an alternative for users who find the standard font too small to read.
<code>boldFont</code>	Same size as <code>stdFont</code> but bold for easier reading. Used for text labels in the user interface.
<code>largeBoldFont</code>	In Palm OS 3.0 and later only. Same size as <code>largeFont</code> but bold.

[Figure 8.2](#) shows what each of the fonts in [Table 8.1](#) looks like.

Figure 8.2 Built-in text fonts



Palm OS also defines the fonts listed in [Table 8.2](#). These fonts do not contain most letters of the alphabet. They are used only for special purposes.

Table 8.2 Built-in symbol fonts

Constant	Description
<code>symbolFont</code>	Contains many special characters such as arrows, shift indicators, and so on.
<code>symbol11Font</code>	Contains the check boxes, the large left arrow, and the large right arrow.
<code>symbol7Font</code>	Contains the up and down arrows used for the repeating button scroll arrows and the dimmed version of the same arrows.
<code>ledFont</code>	Contains the numbers 0 through 9, -, ., and the comma (.). Used by the Calculator application for its numeric display.

Selecting Which Font to Use

The default fonts used to display normal text and bold text vary based on the handheld's character encoding. Handhelds with the Palm Latin encoding typically use `stdFont` and `boldFont`, while Japanese handhelds use `largeFont` and `largeBoldFont` as the default. When your application starts up for the first time, it should

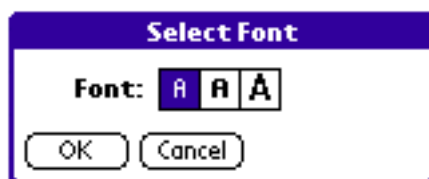
respect the system defaults. Use the [FntGlueGetDefaultFontID](#) function to determine what the default fonts are (see [Listing 8.10](#)).

Listing 8.10 Determining the default system fonts

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

In general, where users can enter text, you should allow them to select the font through the Select Font dialog (see [Figure 8.3](#)).

Figure 8.3 Select Font dialog



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

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

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

Fonts for High-Density Displays

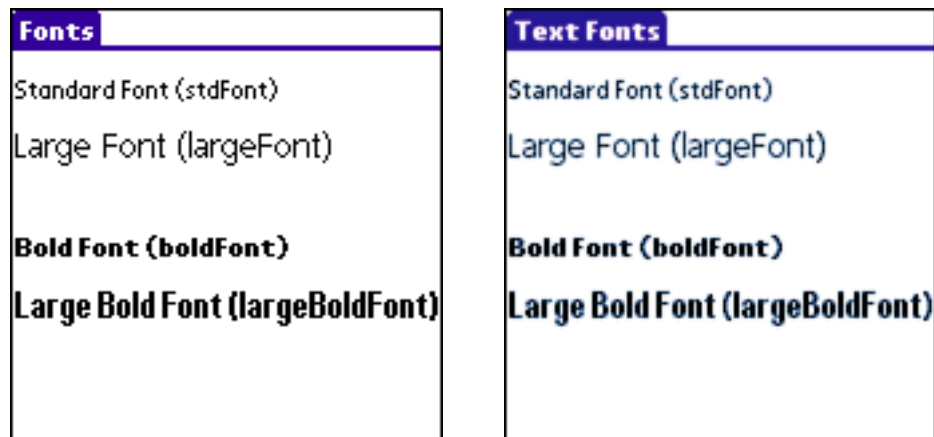
If the [High-Density Display Feature Set](#) is defined, Palm OS supports both low-density (160 X 160) and double-density (320 X 320) displays. Double-density displays pack more pixels into the same space to create a finer resolution.

To support multiple display densities, Palm OS uses an **extended font resource**. An extended font resource contains a separate set of

glyphs for each supported density. At runtime, Palm OS determines the current display density and then draws text using the glyphs that match the display density. On a double-density display, text is drawn using the double-density glyphs. On a low-density display, text is drawn using the low-density glyphs.

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

Figure 8.4 Low-density and high-density fonts



You only need to be concerned about the font density if you use custom fonts.

- When creating a custom font, you'll want to create an extended font resource. See "[Creating Custom Fonts](#)" on page 275. If an extended font resource is not available or does not contain a double-density glyphs, Palm OS pixel doubles the low-density glyphs when drawing to a double-density display.
- When drawing text in a custom font to an off-screen window, you must take care. Off-screen windows also have a display density. If you create a low-density off-screen window and draw text to it, you *must* use a low-density font. If you use an extended font, it must contain low-density glyphs. If the resource contains only double-density glyphs, Palm OS does

not scale the glyphs. The result is undefined and may cause a system crash.

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

Setting the Font Programmatically

To set the font that a user interface element uses for its label or for its textual contents, you use different functions depending on the element. [Table 8.3](#) shows which functions set fonts for which user interface elements.

Table 8.3 Setting the font

UI object	Function
Field	FldSetFont
Field within a table	TblSetItemFont
Command button, push button, pop-up trigger, selector trigger, or check box	CtlGlueSetFont
Label resource	FrmGlueSetLabelFont
List items	LstGlueSetFont
All other text (text drawn directly on the screen)	FntSetFont

The [FntSetFont](#) function changes the font that the system uses by default. It returns the previously used font. You should save the font returned by `FntSetFont` and restore it once you are done. [Listing 8.11](#) shows an example of setting the font to draw items in the custom list drawing function.

Listing 8.11 Setting the font programmatically

```
void DrawOneRowInList(Int16 itemNum, RectangleType *bounds,
    Char **itemsText)
{
    Boolean didSetFont = false;
    FontID oldFont;

    if ((itemNum % 5) == 0) {
        oldFont = FntSetFont(boldFont);
        didSetFont = true;
    }
    WinDrawChars(itemsText[itemNum],
        StrLen(itemsText[itemNum]), bounds.topLeft.x,
        bounds.topLeft.y);
    if (didSetFont)
        FntSetFont(oldFont);
}
```

Obtaining Font Information

Use functions in the Font Manager to obtain information about a font and how it is drawn to the screen. [Figure 8.5](#) shows graphically the characteristics of a font. [Table 8.4](#) describes the types of information that can be retrieved with the Font Manager.

Figure 8.5 Font characteristics

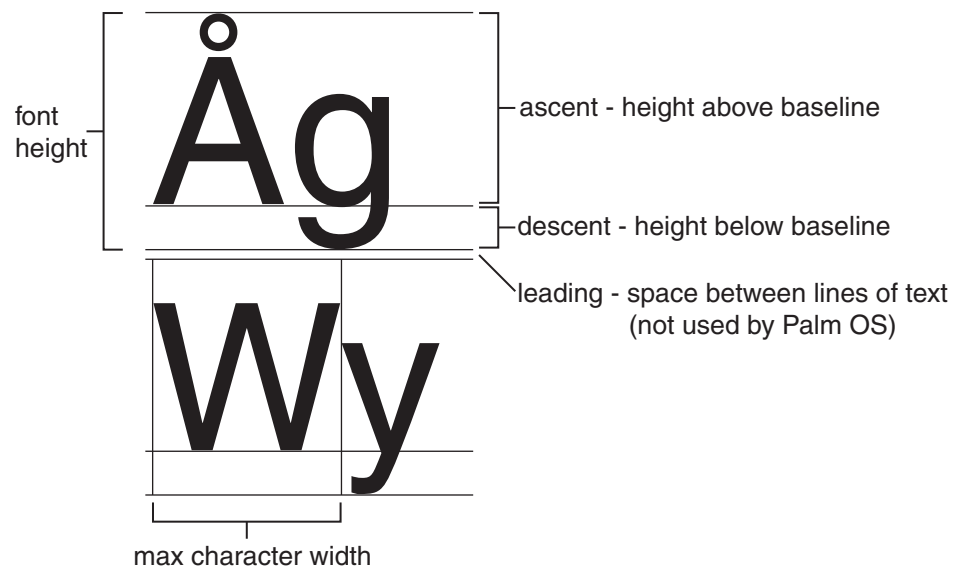


Table 8.4 Obtaining font information

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

The functions listed in [Table 8.4](#) all work on the current font, that is, the font listed in the draw state. To change the current font, use [FntSetFont](#). For example, to determine the line height of the font used for text in a field, do the following:

Listing 8.12 Obtaining characteristics of a field font

```
FieldType *fieldP;  
FontID oldFont;  
Int16 lineHeight;  
  
oldFont = FntSetFont(FldGetFont(fieldP));  
lineHeight = FntLineHeight();  
FntSetFont(oldFont);
```

Creating Custom Fonts

Palm OS 3.0 and later supports the use of custom fonts. You can create your own font resource ('NFNT') and use it within your application.

If the [High-Density Display Feature Set](#) is present, you can use an extended font resource ('nfnt') instead. The extended font

resource contains a separate set of glyphs for each of the supported display densities.

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

Creating a Font Resource

The Palm OS SDK does not provide tools to create a custom font; however, several third party applications, such as xFont and PilRC, are available that support the creation of custom fonts.

TIP: Leave a vertical column of blank space on the right side of each glyph in a font. Palm OS draws characters side by side, in contrast to the Mac OS, which draws a character, clears the pixels to its right and then draws the next character. If your font requires leading, you should leave blank space at the bottom of each glyph as well because Palm OS does not support leading.

To create an extended font resource, you use Constructor for Palm OS:

1. Create two font resources using a third party tool (such as xFont). Every aspect of the second font must be exactly double that of the first font. That is, the second font must be twice as many pixels high, twice as many pixels wide, and so on.
2. Create a font family using Constructor for Palm OS as described in the book *Constructor for Palm OS*.
3. Use the first 'NFNT' resource as the Normal density font. Use the second 'NFNT' resource as the Double density font.

The PalmRez post linker uses the font family resource to create the extended font resource that you use on the device.

Using a Custom Font in Your Application

Once you have defined a custom font in a resource file, you must assign it a font ID before you can use the font in your application. The font ID is different from the font's resource ID. It's a number between 0 and 255. The font ID you use must be greater than or

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

[Listing 8.13](#) shows code that loads a font resource, assigns a font ID to that resource, and then draws characters to the screen using the new font.

Listing 8.13 Loading and using a custom font

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

MemHandle customFontH;
FontType *customFontP;

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

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

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

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

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

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

The 'NFNT' resources you created to build the extended font are discarded after the extended font is created. For a backward compatible application, you need to define a separate 'NFNT' resource containing the low-density glyphs in your font.

It's possible to create an extended font resource that contains only double-density glyphs and not low-density glyphs. You could define an 'NFNT' resource for the low-density glyphs and an extended font resource with just the double-density glyphs. Then you could load and use the 'NFNT' resource on all handhelds with low-density screens (including those that don't support the high-density feature set) and load and use the extended font resource when the display is not low-density. If you do this, you must carefully check the display density before deciding which resource to load (see [Listing 8.14](#)).

WARNING! If you write text to a low-density off-screen window using a double-density glyph, the result is undefined and may cause Palm OS to crash. If you load your fonts as shown in [Listing 8.14](#), load and use the old font resource when drawing text to a low-density off-screen window. See “[Fonts for High-Density Displays](#)” on page 271 for more information.

Listing 8.14 Conditionally loading a font resource

```
#define customFontID ((FontID) fntAppFontCustomBase)
MemHandle fontH = NULL;
FontType *fontP;
UInt16 winVersion;
Err error;
```

```
error = FtrGet(sysFtrCreator, sysFtrNumWinVersion,
    &winVersion);
// If winVersion is >= 4, the high-density feature set
// is present. Check what type of display we are on
// and load the appropriate font resource.
if (!error && (winVersion >= 4)) {
    UInt32 density;
    error = WinScreenGetAttribute(winScreenDensity, &density);
    if (!error && (density != kDensityLow)) {
        // load and use the extended font
        // resource.
        fontH = DmGetResource(fontExtRscType,
            MyNewFontRscID);
        fontP = MemHandleLock(fontH);
    }
}

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

FntDefineFont(customFontID, fontP);
```

Summary of Text API

Text Manager

Accessing Text

TxtCharBounds	TxtGetPreviousChar
TxtPreviousCharSize	TxtCharSize
TxtGetNextChar	TxtNextCharSize
TxtGetChar	

Changing Text

TxtReplaceStr	TxtSetNextChar
TxtConvertEncoding	TxtTransliterate

Segmenting Text

Text

Summary of Text API

Text Manager

[TxtGetTruncationOffset](#)

[TxtWordBounds](#)

[TxtGetWordWrapOffset](#)

Searching/Comparing Text

[TxtCaselessCompare](#)

[TxtCompare](#)

[TxtFindString](#)

[TxtGluePrepFindString](#)

Obtaining a Character's Attributes

[TxtCharIsAlNum](#)

[TxtCharIsAlpha](#)

[TxtCharIsDigit](#)

[TxtCharIsGraph](#)

[TxtCharIsLower](#)

[TxtCharIsPrint](#)

[TxtCharIsSpace](#)

[TxtCharIsUpper](#)

[TxtCharIsValid](#)

[TxtCharXAttr](#)

[TxtCharIsCntrl](#)

[TxtCharIsHex](#)

[TxtCharIsPunct](#)

[TxtCharAttr](#)

Obtaining Character Encoding information

[TxtStrEncoding](#)

[TxtEncodingName](#)

[TxtMaxEncoding](#)

[TxtCharEncoding](#)

[TxtNameToEncoding](#)

Working With Multi-Byte Characters

[TxtByteAttr](#)

String Manager Functions

Length of a String

[StrLen](#)

Comparing Strings

[StrCompare](#)

[StrNCompare](#)

[StrCaselessCompare](#)

[StrNCaselessCompare](#)

String Manager Functions

Changing Strings

[StrPrintf](#)

[StrCat](#)

[StrCopy](#)

[StrToLower](#)

[StrVPrintf](#)

[StrNCat](#)

[StrNCopy](#)

Searching Strings

[StrStr](#)

[StrChr](#)

Converting

[StrAToI](#)

[StrIToH](#)

[StrIToA](#)

Localized Numbers

[StrDelocalizeNumber](#)

[StrLocalizeNumber](#)

Font Functions

Changing the Font

[FontSelect](#)

[FldSetFont](#)

[CtlGlueSetFont](#)

[LstGlueSetFont](#)

[FntSetFont](#)

[TblSetItemFont](#)

[ErmGlueSetLabelFont](#)

Accessing the Font Programmatically

[FntGetFont](#)

[FntGetFontPtr](#)

Wrapping Text

[FntWordWrap](#)

[FntWordWrapReverseNLines](#)

String Width

[FntCharsInWidth](#)

[FntLineWidth](#)

[FntCharsWidth](#)

[FntWidthToOffset](#)

Text

Summary of Text API

Font Functions

Character Width

[FntAverageCharWidth](#)
[FntWCharWidth](#)

[FntCharWidth](#)

Height

[FntCharHeight](#)
[FntBaseLine](#)

[FntLineHeight](#)
[FntDescenderHeight](#)

Scrolling

[FntGetScrollValues](#)

Creating a Font

[FntDefineFont](#)

[FntIsAppDefined](#)

Attentions and Alarms

In this chapter you learn how to get the user's attention and how to set real-time alarms that can be used to either perform some periodic activity or display a reminder to the user.

This chapter is divided into the following broad topics:

- [Getting the User's Attention](#) begins with an introduction to the Attention Manager. This is followed by a detailed description of the Attention Manager from a user's perspective. Finally, it details what developers need to do in order to use the Attention Manager in their applications.
- [Alarms](#) covers the Alarm Manager, which can notify your programs when a specified point in time is reached.

Getting the User's Attention

Palm OS® 4.0 introduces a standard mechanism that manages competing demands for the user's attention by both applications and drivers. This mechanism is known as the Attention Manager.

The Role of the Attention Manager

This section provides a brief introduction to the Attention Manager. It covers the relationship between the Attention, Alarm and Notification Managers, and then discusses when it is appropriate to make use of the Attention Manager.

The Attention Manager provides a standard mechanism by which applications can tell the user that something of significance has occurred. It is designed to support communications devices which can receive data without explicit user interaction. The Attention Manager is responsible only for interacting with the user; it is not responsible for generating those events. In particular, the Alarm

Attentions and Alarms

Getting the User's Attention

Manager can be used in conjunction with the Attention Manager to inform the user that a particular point in time has been reached.

By maintaining a single list of all “alarm-like” things, the Attention Manager also improves the user’s experience when returning to the handheld after being gone for a while: he no longer has to click through a series of old alarm dialogs. Often the user doesn’t care about most of the missed appointments—although he might care about a few of them. Without the Attention Manager, the user cannot selectively dismiss or follow up on dialogs.

Applications have complete control over the types of attention they can ask for. They can query the handheld for the set of special effects available—possibly including sound, vibration, and an LED—and then act on that set. The default option is to beep. All other options are either on or off; different vibrating patterns or multicolored LEDs are currently not supported. Note that the set of special effects is extensible; manufacturers may choose to add other means to get the user’s attention beyond the anticipated LED and vibration.

IMPORTANT: The Attention Manager was introduced in Palm OS 4.0. Applications running on earlier versions of the Palm OS need to use the techniques described under “[Alarms](#)” on page 306.

In Palm OS 4.0, the Datebook, SMS, and Clock applications use the Attention Manager. Refer to the Datebook application’s source code for real-world examples of how you might use the Attention and Alarm Managers.

Attentions, Alarms and Notifications

The Attention, Alarm, and Notification Managers are distinct subsystems that are often used in combination.

- The Attention Manager is designed solely to interact with the user when an event must be brought to the user’s attention.
- The Alarm Manager simply sends an event to an application when a particular point in time is reached. The application can then use the Attention Manager or some other mechanism to bring the alarm to the user’s attention, if appropriate.

- The Notification Manager informs those applications that have registered their interest whenever certain system-level or application-level events occur. If the user is to be informed of the event, the executable can use the Attention Manager. The Attention Manager itself uses the Notification Manager to broadcast notifications when getting the user's attention or nagging him about an existing attention item.

When the Attention Manager Isn't Appropriate

The Attention Manager is only designed for attempts to get attention that can be effectively suspended. It is not suitable for anything requiring an immediate response, such as a request to connect to another user or the "put away" dialog that is used during beaming. The Attention Manager also doesn't attempt to replace error messages. Applications must use modal dialogs and other existing OS facilities to handle these cases.

The Attention Manager is also not intended to replace the ToDo application, or to act as a universal inbox. Applications must make it clear that an item appearing in the Attention Manager is simply a reminder, and that dismissing it does not delete the item itself. That is, saying "OK" to an alarm does not delete the appointment, and dismissing an SMS reminder does not delete the SMS message from the SMS inbox.

Attention Manager Operation

This section provides a detailed introduction to the Attention Manager from a user's point of view, introducing some of the terminology used throughout the rest of the chapter and pointing out operational subtleties that you should be aware of when developing applications that use the Attention Manager.

Attention-getting attempts can either be **insistent** or **subtle**. They differ only in the lengths to which each goes to get your attention. Insistent attempts get "in your face" by popping up a dialog and triggering other visible and audible special effects in an effort to bring important events to your attention. A meeting reminder or incoming high-priority email message might warrant interrupting your work in this fashion. Subtle attentions substitute a small on-screen **attention indicator** for the dialog, allowing you to be made aware of less-critical events without interrupting your current work

Attentions and Alarms

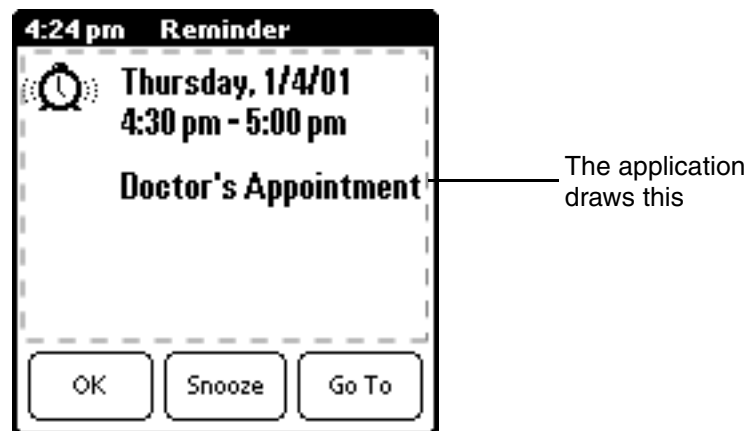
Getting the User's Attention

flow. Although they can also trigger various special effects, subtle attentions don't typically do so. Examples of subtle events might include a reminder of an upcoming birthday or holiday, or an incoming SMS message.

Insistent Attentions

When an application makes an insistent attempt to get the user's attention, the **detail dialog** opens:

Figure 9.1 Detail Dialog

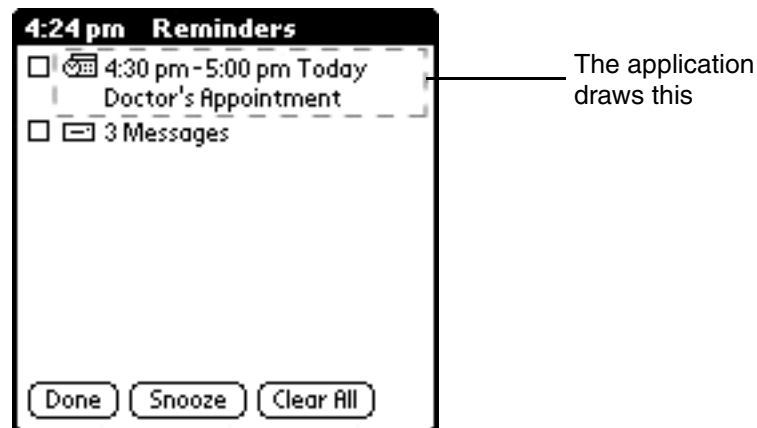


The Attention Manager draws the title and the buttons. The application is responsible for drawing the rest. Most applications draw text and an appropriate icon, as shown in [Figure 9.1](#).

When a second application attempts to get attention, or when the first application makes a second attempt, and the first has not yet

been dismissed or snoozed, the window changes to the **list dialog**, presenting a list of things that require the user's attention:

Figure 9.2 List Dialog



In this dialog, the Attention Manager draws the title and the buttons, and manages the list of items including the checkbox in the left-hand column. Items are listed in order of occurrence, with the newest at the top. The application is responsible for drawing some part of each line, giving it some flexibility over what to display. Applications have space to draw an icon and two lines of text in the standard font on the right-hand side of the list area.

In the detail dialog the OK button dismisses the item. In the list dialog, tapping the checkbox to the left of the item dismisses it. The Clear All button can be used to dismiss all items in the list view. Dismissing an item removes it from the list or closes the detail dialog. Note that although it is gone from the Attention Manager, the item itself remains in the application.

Unique to the list view is a "Done" button which simply closes the list view. It makes no changes to items in the Attention Manager list, nor to any snooze timer.

In either dialog, the "Snooze" button temporarily dismisses the Attention Manager dialog. The attention indicator remains visible and the user can redisplay the dialog at any time. After an interval of five minutes, if any attempts to get attention are still pending the Attention Manager redisplay the dialog. Snooze does not remove attempts to get attention.

Attentions and Alarms

Getting the User's Attention

There is just one “Snooze” timer, and the snooze operation applies to the Attention Manager as a whole. This can lead to seemingly odd behavior when new attention items are coming in while there is currently a snooze in progress. This situation should be rare, however.

To “go to” an individual item, tap the text or icon of the item in the list dialog or tap the “Go To” button in the detail dialog. This temporarily dismisses the Attention Manager and launches the appropriate application to display details about the item. For an SMS message, this could take you to the detail dialog showing the message, or, if there are more than one, it could take you to the list of pending SMS messages. For an alarm, this could take you to the Datebook view for the current day, with the current meeting scrolled into view. A successful “go to” also removes the attention item from the list.

Note that while the Attention Manager dialogs are displayed, hard and soft buttons are ignored. This is to prevent you from missing an attention item when you turn on the handheld by pressing one of the hard keys.

Subtle Attentions

When an application makes a subtle attempt to get the users attention, no dialog appears. Instead, the title bar of all applications that use the standard form title object show a blinking indicator.

Figure 9.3 Attention indicator



When the list contains one or more items, all of which have been seen by the user, the “star” indicator blinks on and off until the list is empty. When the list contains one or more unseen items, the attention indicator performs an “exploding star” animation.

Tapping this indicator opens the Attention Manager in the list mode, even if there is only one item. Tapping to the right of the indicator, or tapping in the indicator’s area when there are no pending attention attempts opens the menu bar as expected.

The attention indicator only functions with applications which use a standard form title object. The indicator doesn’t appear when:

- there are no items in the Attention Manager’s queue.
- the current application uses a custom title.
- the current application draws in the title area.
- the current form uses the Dialog title style.
- the current application’s form title is too narrow to include the attention indicator.

Special Effects

When a new attention item is added, the Attention Manager performs some combination of special effects, which can include playing sounds, flashing a LED, and triggering vibration. The exact combination of effects depends on user settings and on the application itself.

The Attention Manager attempts to open the dialog before performing any special effects so you know immediately why it is trying to get your attention. However, it may not be possible to open the Attention Manager dialog. If this is the case, the Attention Manager performs the special effects as soon as possible. It’s better for the user to be made aware that something is happening, even if the handheld cannot say exactly what it is.

System-wide user preferences control the special effects: the volume at which to play alarms, whether or not to flash the LED (if any), whether or not to vibrate (if equipped). Applications can override these system-wide settings in either a positive or a negative way. For instance, an application could always blink the LED, even if the user said not to, or never blink the LED, even if the user desires it in general.

Attentions and Alarms

Getting the User's Attention

Nagging

As with Datebook alarms in Palm OS 3.5 and earlier, if you don't explicitly dismiss or snooze an attention item it continues to "nag" you at predefined intervals, using the item's specified special effects. Applications control how frequently the user should be reminded, and how many times before the Attention Manager gives up.

When there are multiple attention items competing for nagging, the Attention Manager respects the nag settings for the most recent insistent item, or if there are none then for the most recent subtle item. Each special effect is handled separately; if one reminder wants sound but no vibration, and another wants vibration but no sound, the combination results in the sound from the first one and the vibration from the second one.

Attention Manager and Existing Applications

The Attention Manager makes no attempt to override existing application behavior. If an application written for Palm OS 3.5 or earlier puts up a dialog to get the user's attention, the Attention Manager doesn't get involved. Applications must be specifically written to use the Attention Manager in order to take advantage of its features and seamless integration with the Palm[™] user experience.

Some existing third-party applications put up modal alarm-like dialogs. These dialogs can potentially interfere with the Attention Manager. However, issuing of the UI launch code by the Alarm Manager is deferred until after the Attention Manager is closed. This prevents existing applications from putting up their dialogs while the Attention Manager is being displayed. If the reverse happens, and the Attention Manager pops up while an existing application is displaying an alarm-like dialog, only a "go to" becomes problematic: the third-party dialog may consume events required to perform the "go to," preventing it from taking place. This is acceptable, however, since the attention item remains in the Attention Manager's queue. Once the third-party dialog has been dismissed, you can then re-open the Attention Manager and re-initiate the "go to."

Effectively, this means the Attention Manager always shows up on top of any existing application's alarm dialogs that use the Alarm Manager. This ensures that you'll most likely be greeted by the Attention Manager's list after a prolonged period of inactivity.

Getting the User's Attention

This section shows how your applications request the user's attention through the Attention Manager.

Getting the user's attention is simply a matter of calling [AttnGetAttention](#) with the appropriate parameters and then handling various callbacks made by the Attention Manager. These callbacks allow your application to control what is displayed in the Attention Manager dialogs, to play sounds or perform other special effects, and to do any necessary processing when the user takes action on an existing attention item.

The `AttnGetAttention` prototype looks like this:

```
Err AttnGetAttention (UInt16 cardNo,  
    LocalID dbID, UInt32 userData,  
    AttnCallbackProc *callbackFnP,  
    AttnLevelType level, AttnFlagsType flags,  
    UInt16 nagRateInSeconds, UInt16 nagRepeatLimit)
```

Specify the application requesting the user's attention in the `cardNo` and `dbID` arguments. You can use the [DmGetNextDatabaseByTypeCreator](#) function to obtain these values.

`userData` is used to distinguish a given attention attempt from others made by the same application; most applications pass the unique ID or other key for the record which caused the attention request. This value is passed to your code through the callback function, and can be an integer, a pointer, or any other 32-bit value as needed by your application.

The `callbackFnP` argument controls whether the Attention Manager invokes a callback function or issues a launch code to request services from your application. Applications typically supply `NULL` for this parameter, causing launch codes to be sent to the application specified by the `cardNo` and `dbID` arguments. See

Attentions and Alarms

Getting the User's Attention

[“Callback or Launch Code?”](#) on page 293 for a discussion of callback functions and launch codes.

For the `level` argument, supply `kAttnLevelInsistent` or `kAttnLevelSubtle` depending on whether the given attention attempt is to be insistent or subtle.

Regardless of the level of the attention attempt, set the appropriate bits in the `flags` argument to cause sounds to play, LEDs to blink, or other physical effects to be performed. Depending on which flags you specify, the effect can always occur or can be suppressed by the user. For instance, to trigger a sound while honoring the user's preferences, you need only supply `kAttnFlagsSoundBit`. Or, to blink the LED but suppress any sounds, regardless of any preferences the user may have set, supply a value of `kAttnFlagsAlwaysLED | kAttnFlagsNoSound`. Finally, to choose only vibrate, do something like:

```
flags = kAttnFlagsNothing ^ kAttnFlagsNoVibrate
      | kAttnFlagsAlwaysVibrate;
```

While the above is somewhat complex, it does ensure that you override all defaults in the negative except vibration, which is overridden in the positive. See the definition of [AttnFlagsType](#) in the *Palm OS Programmer's API Reference* for a complete set of constants that can be used in combination for the `flags` argument.

NOTE: Applications may want to verify that the handheld is properly equipped to perform the desired effect. See [“Detecting Device Capabilities”](#) on page 305 for information on how to do this. If the handheld isn't properly equipped to handle a given special effect, the effect isn't performed. For example, if you set the `kAttnFlagsLEDBit` flag and the Palm Powered™ handheld doesn't have an LED, the attention attempt is processed as if the `kAttnFlagsLEDBit` had never been set.

Assuming that the handheld is capable of getting the user's attention using the requested special effect, the `nagRateInSeconds` and `nagRepeatLimit` arguments control how often and how many times the special effect is triggered in an attempt to get the user's attention. As the name implies, specify the

amount of time, in seconds, the Attention Manager should wait between special effect triggers with `nagRateInSeconds`. Indicate the desired number of times the effect should be triggered using `nagRepeatLimit`. Applications typically supply a value of 300 for `nagRateInSeconds` and a value of 3 for `nagRepeatLimit`, indicating that the special effect should be triggered three times after the initial attempt, at five minute intervals.

The following line of code shows how the SMS application calls the Attention Manager upon receiving a new SMS message. Note that a subtle attention is generated so that the user isn't interrupted every time an SMS message is received. Also note that the application doesn't override the user's settings when getting his attention. Finally, if the user doesn't respond to the first attention-getting attempt, the special effects are repeated three additional times in five-minute intervals.

```
err = AttnGetAttention(cardNo, dbID, NULL,  
    NULL, kAttnLevelSubtle,  
    kAttnFlagsUseUserSettings, 300, 3);
```

Callback or Launch Code?

For a given attention item, the Attention Manager calls back to the code resource that created that item whenever the Attention Manager needs the resource to draw the attention dialog contents or whenever it needs to inform the code resource of activity relating to the attention item. The Attention Manager calls back using one of two mechanisms:

- If a callback routine has been specified for a given attention item, the Attention Manager invokes it. This callback routine doesn't have application globals available to it, so it is important that anything necessary to draw or otherwise display be available through `commandArgsP`. A callback routine is typically used by libraries and system extensions. See the description of the [AttnCallbackProc](#) in the *Palm OS Programmer's API Reference* for the specifics of the callback routine.
- If a callback routine has not been specified for a given attention item, the Attention Manager instead sends a [sysAppLaunchCmdAttention](#) launch code to the application that registered the attention item. Accompanying

Attentions and Alarms

Getting the User's Attention

that launch code is an [AttnLaunchCodeArgsType](#) structure containing the three parameters documented above. Applications typically use the launch-code mechanism due to the restrictions that are placed on callback routines.

IMPORTANT: It is your responsibility to ensure that the callback procedure is still in the same place when it gets called, dealing with the possibility that the code resource might be unlocked and moved in memory, and with the possibility that the database containing the code resource might be deleted. For the most part, these problems don't exist when using launch codes.

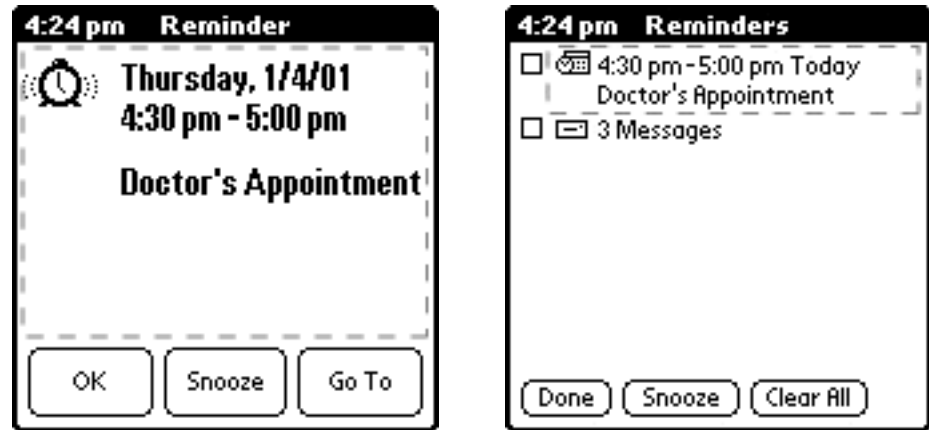
Attention Manager Commands

In addition to calling [AttnGetAttention](#), your code must also respond to commands from the Attention Manager. The Attention Manager issues these commands either by invoking a callback function or issuing a launch code, depending on what you supplied for the `callbackFnP` argument in the `AttnGetAttention` call. Among the possible commands are requests for your application to draw the contents of the detail and list dialogs, play application-specific sounds or perform other special effects, navigate to the item in your application that caused the attention item to be posted, and so forth.

Draw Detail and List Dialogs

The Attention Manager's detail and list dialogs are drawn as a joint effort between the Attention Manager and applications requesting the user's attention. The shell of each dialog, the title, and the buttons and checkboxes are drawn by the Attention Manager, while the remainder—text specific to each attention attempt, and frequently an accompanying icon—is drawn by the application itself. This gives each application full control over what sort of information to display.

Figure 9.4 Attention Manager dialogs



Although your application has full control over the display of its attention items, it may only draw in the designated area; active UI elements, such as scroll bars, custom buttons, or other widgets cannot be included. Users should be encouraged to launch the application with the “Go To” button and use the richer UI provided there. The clipping region is appropriately set so that your application cannot accidentally draw outside the designated area.

The `kAttnCommandDrawDetail` command indicates that your application is to draw the contents of the detail dialog. Along with the command, the Attention Manager passes a structure of type [drawDetail](#). This structure contains the window-relative boundaries of the screen rectangle in which your application is to draw, a flag indicating whether this is the first time the user has seen this attention item, and a set of flags indicating which special effects will also be triggered.

NOTE: Unless the `firstTime` bit is set, the portion of the detail dialog in which your application draws is not guaranteed to be blank. If `firstTime` is not set, erase the area indicated by the `bounds` rectangle prior to drawing the contents of the detail dialog.

The following code excerpt shows how a simple application might render the contents of the detail dialog in response to

Attentions and Alarms

Getting the User's Attention

[sysAppLaunchCmdAttention](#) launch code accompanied by a `kAttnCommandDrawDetail` command:

Listing 9.1 Drawing the contents of the detail dialog

```
// Draw the icon
resH = DmGetResource(bitmapRsc, MyIconBitmap);
WinDrawBitmap(MemHandleLock(resH),
    paramsPtr->drawDetail.bounds.topLeft.x,
    paramsPtr->drawDetail.bounds.topLeft.y + 4);
MemHandleUnlock(resH);
DmReleaseResource(resH);

// Draw the text. The content of the string depends on the
// uniqueID that accompanies the kAttnCommandDrawDetail
// command
curFont = FntSetFont (largeBoldFont);
x = paramsPtr->drawDetail.bounds.topLeft.x + 37;
y = paramsPtr->drawDetail.bounds.topLeft.y + 4;
WinDrawChars(alertString, StrLen(alertString), x, y);
FntSetFont (curFont);
```

For a more complex, real-world example, see the `DrawDetailAlarm` function in the Datebook application. In particular, note how that application adjusts the displayed text so that it all fits in the allocated space.

Note that because subtle attention items are only shown using the list view, your application doesn't need to respond to `kAttnCommandDrawDetail` if it doesn't produce insistent attention attempts.

The `kAttnCommandDrawList` command is similar; it indicates that your application is to draw a portion of the contents of the list dialog. Along with this command, the Attention Manager passes a structure of type [drawList](#), which contains a selected flag in addition to the bounds, `firstTime`, and `flags` fields described above. The selected flag indicates whether the item has been selected. Before sending your application the `kAttnCommandDrawList` command, the Attention Manager sets

the background, foreground, and text colors as follows, depending on whether or not the item is selected:

Affected Color	Not Selected	Selected
Background Color	UIFieldBackground	UIObjectSelectedFill
Foreground Color	UIObjectForeground	UIObjectSelectedForeground
Text Color	UIObjectForeground	UIObjectSelectedForeground

Particularly if your attention item icon is in color, you may need to draw the attention item differently when it is selected. The selected flag exists to allow you to do this.

The code to draw an attention item in the list dialog is very similar to the code you use to draw the same item in the detail dialog. Here is an excerpt:

Listing 9.2 Drawing the contents of the list dialog

```
// Draw the icon. Ignore the 'selected' flag for this example
resH = DmGetResource(bitmapRsc, MySmallIconBitmap);
iconP = (BitmapPtr) (MemHandleLock(resH));
// center it in the space allotted
iconOffset = (kAttnListMaxIconWidth - iconP->width)/2;
x = paramsPtr->drawList.bounds.topLeft.x;
y = paramsPtr->drawList.bounds.topLeft.y;
WinDrawBitmap(iconP, x + iconOffset, y);
MemHandleUnlock(resH);
DmReleaseResource(resH);

// Draw the text
curFont = FntSetFont(stdFont);
WinDrawChars(alertString, StrLen(alertString),
             x + kAttnListTextOffset, y);
FntSetFont(curFont);
```

The primary differences arise from the fact that the list dialog provides a smaller, more structured area in which to draw. Also, the area in which you draw will always have been erased before you are asked to draw in it, so you don't ever have to erase it beforehand.

Attentions and Alarms

Getting the User's Attention

The icon should be no wider than `kAttnListMaxIconWidth` pixels, and should be centered within this width if it is smaller than `kAttnListMaxIconWidth`. The text should honor a left-hand margin of `kAttnListTextOffset` pixels. In the above example, the alert string is assumed to fit within the available space; see the `DrawListAlarm` function in the Datebook application for an example of how that application adjusts the displayed text in the event that it doesn't all fit in the allocated space.

NOTE: Applications may, in certain rare circumstances, receive a `kAttnCommandDrawDetail` or `kAttnCommandDrawList` command for an item which is no longer valid. Respond by either calling [AttnForgetIt](#), by drawing nothing, or by drawing an error message.

Play Sound or Perform a Custom Effect

Most applications play a sound when attempting to get the user's attention. If the `kAttnFlagsAlwaysSound` is set when your application calls [AttnGetAttention](#), the Attention Manager sends a `kAttnCommandPlaySound` command to your application when attempting to get the user's attention. Your application should simply play the appropriate sound in response to this command. Both the Datebook and SMS applications play sounds based upon the user's preferences when getting the user's attention; see the Datebook application's source code for an example of how to respond to the `kAttnCommandPlaySound` command.

Because the Attention Manager can potentially play a sound, blink an LED, and vibrate in addition to displaying a dialog or blinking the attention indicator, most applications don't request that some other application-specific custom effect be performed. If your application needs to do something different, you can specify `kAttnFlagsAlwaysCustomEffect` when calling `AttnGetAttention`. This causes a `kAttnCommandCustomEffect` command to be sent to your application, at which time it should perform the desired effect. If your application is like most, however, it won't ever receive a `kAttnCommandCustomEffect` command, so you needn't worry about responding to it.

Neither `kAttnCommandPlaySound` nor `kAttnCommandCustomEffect` are accompanied by any kind of data structure indicating the sound or effect to be performed. If your application doesn't hard-wire this information, you may want to store it in the application's preferences database.

Go There

When the user taps on the "Go To" button in the detail view or on the item text or icon (not the checkbox) in the list view, your application receives a `kAttnCommandGoThere` command. It then needs to switch to your application and display the information relating to the chosen attention item. The `kAttnCommandGoThere` command is similar to the [sysAppLaunchCmdGoto](#) launch code, but you don't have globals when your application receives the `kAttnCommandGoThere` command and your application is called using [SysAppLaunch](#), rather than [SysUIAppSwitch](#). Because of this, most applications perform a `SysUIAppSwitch` upon receiving `kAttnCommandGoThere`.

Note that your application should verify that the data that triggered the attention attempt is still relevant. In the Datebook, for instance, the user could:

1. Be alerted to an appointment by the Attention Manager.
2. Tap "Snooze."
3. Press the Datebook button and delete the appointment that was just brought to the user's attention.
4. Tap the blinking attention indicator.
5. Tap the attention item corresponding to the now-deleted appointment.

In this scenario, the Datebook application could theoretically receive a `kAttnCommandGoThere` command along with a unique ID referencing a Datebook record that has been deleted. Whenever the Datebook application receives `kAttnCommandGoThere` along with a unique ID for a deleted Datebook database record, it calls [AttnForgetIt](#) for the now defunct attention item and then returns.

In reality, the Datebook calls `AttnForgetIt` whenever the user deletes an alarm, and whenever an alarm is determined to be no longer valid. It can do this without even checking to see if the alarm

Attentions and Alarms

Getting the User's Attention

is among those that the Attention Manager is currently tracking; if you pass a combination of card number, database ID, and unique ID to `AttnForgetIt` that doesn't correspond to an item in the Attention Manager's queue, `AttnForgetIt` does nothing and returns a value of `false`.

Most applications will choose to call `AttnForgetIt` once the user has viewed the data corresponding to the attention item. This is how the SMS application operates: incoming messages are brought to the user's attention via the Attention Manager, and are removed from the Attention Manager's queue once they've been read.

Got It

When the user dismisses a particular attention item, a `kAttnCommandGotIt` command is sent to the application. Along with this command is a boolean that indicates if the item was explicitly dismissed by the user, since `kAttnCommandGotIt` is also issued as the result of a successful [AttnForgetIt](#) call. Upon receipt of this command, you may want to clean up memory, delete an alarm, or do other application-specific processing.

Iterate

When something happens that may potentially cause an application's attention items to become invalid, that application should call [AttnIterate](#). `AttnIterate` generates a series of `kAttnCommandIterate` commands, one for each of the application's pending attention items. Thus, when your application receives a `kAttnCommandIterate` command it should validate the indicated attention item and take appropriate action if the item is no longer valid—up to and including calling [AttnForgetIt](#) for the item.

The SMS application does not respond to `kAttnCommandIterate`. The Datebook does; this command is generated whenever the user updates his preferences. Many applications call `AttnIterate` after receiving a [sysAppLaunchCmdSyncNotify](#) launch code so that they can update (with [AttnUpdate](#)) or remove (with `AttnForgetIt`) items which were affected by the HotSync® operation.

Note that you can safely call `AttnForgetIt` from within the iteration since `AttnForgetIt` only marks the record for deletion and thus doesn't confuse the iteration.

Snooze

Most applications—including the Datebook and SMS applications—ignore `kAttnCommandSnooze`, which indicates that the user has tapped the Snooze button. Beyond the unique ID that identifies the attention item, nothing accompanies the Snooze command.

`kAttnCommandSnooze` is passed to each and every item currently pending, insistent or subtle. This means that applications with more than one attention item pending receive this command more than once.

Triggering Special Effects

The Attention Manager activates any requested special effects for each attention item, but your application might want to activate those special effects without also posting an attention item to the queue. You can do this through a call to [AttnDoSpecialEffects](#). Supply the appropriate combination of flags to trigger the desired effects. See [AttnFlagsType](#) for a complete list of flags.

Attentions and Alarms

The Attention Manager is often used in conjunction with the Alarm Manager to get the user's attention at a particular time. The basic use of the Alarm Manager is covered in "[Alarms](#)" on page 306, but because the Attention Manager handles UI synchronization, do the following when using the Alarm Manager with the Attention Manager:

- Call [AttnGetAttention](#) when your application receives the `sysAppLaunchCmdAlarmTriggered` launch code.
- In your `sysAppLaunchCmdAlarmTriggered` handling code, set the `purgeAlarm` field in the launch code's parameter block to `true` before returning. This removes the alarm from the queue, so your application won't receive the [sysAppLaunchCmdDisplayAlarm](#) launch code.

Attentions and Alarms

Getting the User's Attention

Don't wait until the `sysAppLaunchCmdDisplayAlarm` launch code is received to call `AttnGetAttention`: `sysAppLaunchCmdDisplayAlarm` is not issued while another Alarm Manager dialog is open, so the Attention Manager is prevented from doing anything if a "legacy" application is displaying an Alarm Manager dialog.

NOTE: If you want to use the Alarm Manager to force periodic updates of your application's attention items, don't call [AttnUpdate](#) after receiving an Alarm Manager launch code since this keeps the handheld from sleeping. Instead, use a procedure alarm as described under "[Setting a Procedure Alarm](#)" on page 311 and call `AttnUpdate` from within your procedure alarm callback function.

Detecting and Updating Pending Attentions

Once your application has requested that the Attention Manager get the user's attention, it may later need to update that attention request. For instance, if your application uses a single attention item to indicate that a number of unread messages have been received, it should update that item as additional items are received and read. The Attention Manager provides a handful of functions that allow applications to examine the Attention Manager's queue and update the items within that queue.

The [AttnGetCounts](#) function allows you to determine how many items are currently competing for the user's attention. It always returns the total number of items, but can return the number of subtle and/or insistent items as well. It can also return the counts for all applications. For instance, the following would set `numItems` to the total number of pending attention items from all sources:

```
numItems = AttnGetCounts(0, 0, NULL, NULL);
```

Or, to get the number of subtle and insistent attention items in addition to the total requested by a single application, use something like:

```
numItems = AttnGetCounts(cardNo, dbID,  
    &insistentItems, &subtleItems);
```

To verify each pending attention item for a given application, use the Attention Manager's [AttnIterate](#) function as described under "[Iterate](#)" on page 300.

After a HotSync is a popular time to invoke [AttnIterate](#); the HotSync operation may have altered the application's underlying data in such a way as to render pending attention items obsolete or invalid.

Deleting Pending Attention Items

In many cases you'll need to delete an attention item. For instance, a HotSync may alter the underlying application data in such a way so as to invalidate the attention attempt. Or, the user might switch to your application and manually update the data in a similar way, for example by deleting an appointment for which there is a pending alarm. The [AttnForgetIt](#) function exists for this purpose. Simply invoke this function and supply the card number, database ID, and user data that uniquely identify the attention attempt. You don't even have to verify that the attention attempt is still in the Attention Manager's queue: [AttnForgetIt](#) doesn't complain if the attention attempt doesn't exist, merely returning a value of `false` to indicate this condition.

Updating Pending Attention Items

To update an existing attention item, use [AttnUpdate](#). This function is very similar to [AttnGetAttention](#), though instead of actual values you pass pointers to those values you want to update. Supply `NULL` pointers for `flagsP`, `nagRateInSecondsP`, and/or `nagRepeatLimitP` to leave them untouched. For instance, to

Attentions and Alarms

Getting the User's Attention

change the flags that control the special effects used to get the user's attention, do the following:

Listing 9.3 Updating an existing attention item

```
// This assumes that cardNo, dbID, and myUserData are
// declared and set elsewhere to values that identify the
// attention item we're trying to update
Boolean updated;
AttnFlagsType newFlags;

// set newFlags appropriately
updated = AttnUpdate(cardNo, dbID, myUserData, NULL,
    &newFlags, NULL, NULL);
if (updated){
    // update succeeded
} else {
    // update failed - attention item may no longer exist
}
```

NOTE: Although `AttnUpdate` may cause a given attention item to redraw, it does not rerun the special effects (if any) that occurred when that attention item was added. If you want to trigger Attention Manager effects for a particular item, call [`AttnForgetIt`](#) followed by [`AttnGetAttention`](#).

When calling `AttnUpdate`, note that you must not only supply the card number, database ID, and user data that uniquely identifies the attention attempt; you must also supply a pointer to the callback procedure if one is used.

While updating the attention item, if the handheld is on and the Attention Manager dialog is currently showing then `AttnUpdate` forces the item to be redrawn. This in turn calls back to the client application so that it can update its portion of the dialog. `AttnUpdate` causes the specified item to be redrawn if it is visible, regardless of the `flags`, `nagRateInSeconds`, and `nagRepeatLimit` parameters. Thus, `AttnUpdate` isn't limited to updating one or more aspects of an attention item; it also allows an application to update the text of an attention attempt without having to destroy and then rebuild the Attention Manager dialog.

Note that if the handheld is off, the update is delayed until the handheld is next turned on; `AttnUpdate` doesn't itself turn the screen on.

Detecting Device Capabilities

Although you can blindly request that a given special effect, such as vibration, be used to get the user's attention without checking to see if that special effect is supported on the Palm Powered handheld, you may want your application to behave differently in the absence of a particular device feature. The Attention Manager defines a feature that you use with the `FtrGet` function to determine the handheld's physical capabilities. You can also use this feature to determine the user's attention-getting preferences. For example:

Listing 9.4 Checking for vibrate capability

```
// See if the device supports vibration
FtrGet(kAttnFtrCreator, kAttnFtrCapabilities, &capabilities);
if (capabilities & kAttnFlagsHasVibrate){
    // Vibrate-specific processing goes here
    ...
}
```

See “[Attention Manager Feature Constants](#)” in the *Palm OS Programmer's API Reference* for descriptions of all of the relevant flags.

Controlling the Attention Indicator

For the most part, applications can ignore the attention indicator, which consumes a small portion of a form's title bar. If the currently-displayed form doesn't have a title bar, or if the form is modal, the attention indicator isn't drawn. However, if an application takes over the entire screen or does something special with the form's title bar, it should explicitly disable the attention indicator while the form is displayed.

Attentions and Alarms

Alarms

As an example, the Datebook application disables the attention indicator when:

- a note associated with a Datebook entry is displayed.
- an entry's description is being displayed in the week view.
- the time is being displayed in the title bar in place of the date.

To disable the attention indicator, simply call [AttnIndicatorEnable](#) and supply a value of `false` for the `enableIt` argument. To re-enable it, simply call [AttnIndicatorEnable](#) again, this time supplying an argument value of `true`.

If your application disables the attention indicator, it may want to provide some means for the user to open the Attention Manager's dialog in list mode. The [AttnListOpen](#) function can be used to do this.

Alarms

The Palm OS Alarm Manager provides support for setting real-time alarms, for performing some periodic activity, or for displaying a reminder. The Alarm Manager:

- Works closely with the Time Manager to handle real-time alarms.
- Sends launch codes to applications that set a specific time alarm to inform the application the alarm is due.
- Allows only one alarm to be set per application.
- Handles alarms by application in a two cycle operation:
 - First, it notifies each application that the alarm has occurred; the application verifies that the alarm is still valid at this point. Applications that don't use the Attention Manager typically play a sound here.
 - Second, after all pending alarms have received their first notification, it sends another notification to each application, allowing it to display some UI.

The Alarm Manager doesn't have any UI of its own; applications that need to bring an alarm to the user's attention must do this themselves. It doesn't provide reminder dialog boxes, and it doesn't

play the alarm sound. Applications running on Palm OS 4.0 should use the Attention Manager to interact with the user; see “[Attentions and Alarms](#)” on page 301 for tips on doing this. Applications running on earlier versions of the Palm OS need to provide their own UI, as explained in the following sections.

IMPORTANT: When the handheld is in sleep mode, alarms can occur almost a minute late. This is particularly important to note when utilizing procedure alarms (discussed in “[Setting a Procedure Alarm](#)” on page 311); although procedure alarms will continue to fire when the handheld is in sleep mode, they may fire less frequently.

Setting an Alarm

The most common use of the Alarm Manager is to set a real-time alarm within an application. Often, you set this type of alarm because you want to inform the user of an event. For example, the Datebook application sets alarms to inform users of their appointments.

Implementing such an alarm is a two step process. First, use the function [AlmSetAlarm](#) to set the alarm. Specify when the alarm should trigger and which application should be informed at that time.

[Listing 9.5](#) shows how the Datebook application sets an alarm.

Listing 9.5 Setting an alarm

```
static void SetTimeOfNextAlarm (UInt32 alarmTime, UInt32 ref)
{
    UInt16 cardNo;
    LocalID dbID;
    DmSearchStateType searchInfo;

    DmGetNextDatabaseByTypeCreator (true, &searchInfo,
                                     sysFileTApplication, sysFileCDatebook, true, &cardNo,
                                     &dbID);

    AlmSetAlarm (cardNo, dbID, ref, alarmTime, true);
}
```

Attentions and Alarms

Alarms

Second, have your [PilotMain](#) function respond to the launch codes [sysAppLaunchCmdAlarmTriggered](#) and [sysAppLaunchCmdDisplayAlarm](#).

When an alarm is triggered, the Alarm Manager notifies each application that set an alarm for that time via the [sysAppLaunchCmdAlarmTriggered](#) launch code. After each application has processed this launch code, the Alarm Manager sends each application [sysAppLaunchCmdDisplayAlarm](#) so that the application can display the alarm. The section “[Alarm Scenario](#)” gives more information about when these launch codes are received and what actions your application might take. For a specific example of responding to these launch codes, see the Datebook sample code included with Palm OS 3.5.

It’s important to note the following:

- An application can have only one alarm pending at a time. If you call `AlmSetAlarm` and then call it again before the first alarm has triggered, the Alarm Manager replaces the first alarm with the second alarm. You can use the [AlmGetAlarm](#) function to find out if the application has any alarms pending.
- You do not have access to global variables or code outside segment 0 (in a multi-segment application) when you respond to the launch codes. `AlmSetAlarm` takes a `UInt32` parameter that you can use to pass a specific value so that you have access to it when the alarm triggers. (This is the `ref` parameter shown in [Listing 9.5](#).) The parameter blocks for both launch codes provide access to this reference parameter. If the reference parameter isn’t sufficient, you can define an application feature. See the “[Features](#)” section in the “[Palm System Support](#)” chapter.
- The database ID that you pass to `AlmSetAlarm` is the local ID of the **application** (the `.prc` file), not of the record database that the application accesses. You use record database’s local ID more frequently than you do the application’s local ID, so this is a common mistake to make.
- In `AlmSetAlarm`, the alarm time is given as the number of seconds since 1/1/1904. If you need to convert a conventional date and time value to the number of seconds since 1/1/1904, use [TimDateTimeToSeconds](#).

If you want to clear a pending alarm, call `AlmSetAlarm` with 0 specified for the alarm seconds parameter.

Alarm Scenario

Here's how an application and the Alarm Manager typically interact when processing an alarm:

1. The application sets an alarm using [AlmSetAlarm](#).
The Alarm Manager adds the new alarm to its alarm queue. The alarm queue contains all alarm requests. Triggered alarms are queued up until the Alarm Manager can send the launch code to the application that created the alarm. However, if the alarm queue becomes full, the oldest entry that has been both triggered and notified is deleted to make room for a new alarm.
2. When the alarm time is reached, the Alarm Manager searches the alarm queue for the first application that set an alarm for this alarm time.
3. The Alarm Manager sends this application the [sysAppLaunchCmdAlarmTriggered](#) launch code.
4. The application can now:
 - Set the next alarm.
 - Play a short sound.
 - Perform some quick maintenance activity.

The application should not perform any lengthy tasks in response to `sysAppLaunchCmdAlarmTriggered` because doing so delays other applications from receiving alarms that are set to trigger at the same time.

If the application is using the Attention Manager to bring this alarm to the user's attention, call [AttnGetAttention](#) here and set the `purgeAlarm` field in the launch codes' parameter block to `true` before returning.

If this alarm requires no further processing, the application should set the `purgeAlarm` field in the launch code's parameter block to `true` before returning. Doing so removes the alarm from the queue, which means it won't receive the [sysAppLaunchCmdDisplayAlarm](#) launch code.

Attentions and Alarms

Alarms

5. The Alarm Manager finds in the alarm queue the next application that set an alarm and repeats steps 2 and 3.
6. This process is repeated until no more applications are found with this alarm time.
7. The Alarm Manager then finds once again the first application in the alarm queue who set an alarm for this alarm time and sends this application the launch code [`sysAppLaunchCmdDisplayAlarm`](#). Note that alarms that had their `purgeAlarm` field set to `true` during the processing of `sysAppLaunchCmdAlarmTriggered`—including all alarms that are being brought to the user's attention through the Attention Manager—are no longer in the queue at this point.
8. The application can now:
 - Display a dialog box.
 - Display some other type of reminder.

Applications typically create a nested event loop at this point to handle the dialog's events. This nested event loop ignores virtually all events that would cause the dialog to go away, causing the dialog to be fixed on the screen; it can't be dismissed until one of the embedded buttons is tapped. Note that the currently active application remains active. You may not be able to see it (if the alarm dialog is full screen) and you cannot interact with it because the dialog's nested event loop is processing all events. It is effectively suspended, waiting for an event to occur.

9. The Alarm Manager processes the alarm queue for the next application that set an alarm for the alarm being triggered and steps 7 and 8 are repeated.
10. This process is repeated until no more applications are found with this alarm time.

If a new alarm time is triggered while an older alarm is still being displayed, all applications with alarms scheduled for this second alarm time are sent the `sysAppLaunchCmdAlarmTriggered` launch code, but the display cycle for the second set of alarms is postponed until all earlier alarms have finished displaying.

Note that when a second alarm goes off before the first has been dismissed, the alarm manager sends the

[sysAppLaunchCmdAlarmTriggered](#) launch code for the second alarm but waits to send the [sysAppLaunchCmdDisplayAlarm](#) launch code until after the first alarm's dialog has been dismissed. For applications that put up dialogs, this typically means that only one dialog at a time will appear on the screen. The Alarm Manager doesn't return to the event loop between the issuing of launch codes, so when the first alarm's dialog has been dismissed, the second alarm's dialog is immediately displayed. The net result for the user is that each alarm dialog in turn must be dismissed before the handheld can be used.

Setting a Procedure Alarm

Beginning with Palm OS version 3.2, the system supports setting procedure alarms in addition to the application-based alarms described in the previous sections. The differences between a procedure alarm and an application-based alarm are:

- When you set a procedure alarm, you specify a pointer to a function that should be called when the alarm triggers instead of an application that should be notified.
- When the alarm triggers, the Alarm Manager calls the specified procedure directly instead of using launch codes.
- If the system is in sleep mode, the alarm triggers without causing the LCD display to light up.

You might use procedure alarms if:

- You want to perform a background task that is completely hidden from the user.
- You are writing a shared library and want to implement an alarm within that library.
- You want to use [AttnUpdate](#) to update an attention item, but you don't want the display to turn on if the handheld is currently sleeping.

To set a procedure alarm, you call [AlmSetProcAlarm](#) instead of [AlmSetAlarm](#). (Similarly, you use the [AlmGetProcAlarm](#) function instead of [AlmGetAlarm](#) to see if any alarms are pending for this procedure.)

Attentions and Alarms

Alarms

`AlmSetProcAlarm` is currently implemented as a macro that calls `AlmSetAlarm` using a special value for the card number parameter to notify the Alarm Manager that this is a procedure alarm. Instead of specifying the application's local ID and card number, you specify a function pointer. The other rules for `AlmSetAlarm` still apply. Notably, a given function can only have one alarm pending at a time, and you can clear any pending alarm by passing 0 for the alarm time.

When the alarm triggers, the Alarm Manager calls the function you specified. The function should have the prototype:

```
void myAlarmFunc (UInt16 almProcCmd,  
SysAlarmTriggeredParamType *paramP)
```

IMPORTANT: The function pointer must remain valid from the time `AlmSetProcAlarm` is called to the time the alarm is triggered. If the procedure is in a shared library, you must keep the library open. If the procedure is in a separately-loaded code resource, the resource must remain locked until the alarm fires. When you close a library or unlock a resource, you must remove any pending alarms. If you don't, the system crashes when the alarm is triggered.

The first parameter to your function specifies why the Alarm Manager has called the function. Currently, the Alarm Manager calls the function in two instances:

- The alarm has triggered.
- The user has changed the system time, so the alarm time should be adjusted.

The second parameter is the same structure that is passed with the [sysAppLaunchCmdAlarmTriggered](#) launch code. It provides access to the reference parameter specified when the alarm was set, the time specified when the alarm was set, and the `purgeAlarm` field, which specifies if the alarm should be removed from the queue. In the case of procedure alarms, the alarm should always be removed from the queue. The system sets the `purgeAlarm` value to `true` after calling your function.

It is important to note that your procedure alarm function should not access global variables if the alarm could be triggered after the application that contains the procedure alarm function has terminated (even if the code remains locked in memory), since the globals no longer exist at this point.

Procedure Alarms and Menus

Procedure alarms are often used to update the handheld's display on a regular basis. Menus aren't automatically informed when an alarm is triggered, so if a menu is open when your procedure alarm code updates the display, your code may overwrite the open menu. To detect an open menu, watch for [winExitEvent](#) and [winEnterEvent](#) as described in "[Checking Menu Visibility](#)" on page 107 of the *Palm OS Programmer's Companion*, vol. I.

Summary of Attentions and Alarms

Attention Manager Functions

AttnDoSpecialEffects	AttnIndicatorEnabled
AttnForgetIt	AttnIterate
AttnGetAttention	AttnListOpen
AttnGetCounts	AttnUpdate
AttnIndicatorEnable	

Alarm Manager Functions

AlmSetAlarm	AlmGetAlarm
AlmSetProcAlarm	AlmGetProcAlarm

Palm System Support

In this chapter, you learn how to work with the miscellaneous supporting functionality that the Palm OS[®] system provides, such as sound, time, and floating-point operations. Most parts of the Palm OS are controlled by a manager, which is a group of functions that work together to implement a certain functionality. As a rule, all functions that belong to one manager use the same prefix and work together to implement a certain aspect of functionality.

This chapter discusses these topics:

- [Features](#)
- [Preferences](#)
- [Sound](#)
- [System Boot and Reset](#)
- [ARM-Native Functions](#)
- [Hardware Interaction](#)
- [The Microkernel](#)
- [Retrieving the ROM Serial Number](#)
- [Time](#)
- [Floating-Point](#)
- [Summary of System Features](#)

Features

A *feature* is a 32-bit value that has special meaning to both the feature publisher and to users of that feature. Features can be published by the system or by applications.

Each feature is identified by a feature creator and a feature number:

- The feature creator is a unique creator registered with PalmSource, Inc. You usually use the creator type of the application that publishes the feature.
- The feature number is any 16-bit value used to distinguish between different features of a particular creator.

Once a feature is published, it remains present until it is explicitly unregistered or the handheld is reset. A feature published by an application sticks around even after the application quits.

This section introduces the feature manager by discussing these topics:

- [The System Version Feature](#)
- [Application-Defined Features](#)
- [Using the Feature Manager](#)
- [Feature Memory](#)

The System Version Feature

An example for a feature is the system version. This feature is published by the system and contains a 32-bit representation of the system version. The system version has a feature creator of `sysFtrCreator` and a feature number of `sysFtrNumROMVersion`). Currently, the different versions of the system software have the following numbers:

0x01003001	Palm OS 1.0
0x02003000	Palm OS 2.0
0x03003000	Palm OS 3.0
0x03103000	Palm OS 3.1
0x03203000	Palm OS 3.2
0x03503000	Palm OS 3.5
0x04003000	Palm OS 4.0

Rather than hard wiring an obscure constant like one of the above into your code, however, you can use the `sysMakeROMVersion`

macro (defined in `SystemMgr.h`) to construct a version number for comparison purposes. It takes five parameters:

- Major version number
- Minor version number
- Fix level
- Build stage (either `sysROMStageDevelopment`, `sysROMStageAlpha`, `sysROMStageBeta`, or `sysROMStageRelease`)
- Build number

The fix level and build number parameters are normally set to zero, while build stage is usually set to `sysROMStageRelease`. Simply check to see whether `sysFtrNumROMVersion` is greater than or equal to the version number constructed with `sysMakeROMVersion`, as shown here:

```
// See if we're on ROM version 3.1 or later.
FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
if (romVersion >= sysMakeROMVersion(3, 1, 0,
    sysROMStageRelease, 0)) {
    ....
}
```

Other system features are defined in `SystemMgr.h`. System features are stored in a feature table in the ROM. (In Palm OS 3.1 and higher, the contents of this table are copied into the RAM feature table at system startup.) Checking for the presence of system features allows an application to be compatible with multiple versions of the system by refining its behavior depending on which capabilities are present or not. Future hardware platforms may lack some capabilities present in the first platform, so checking the system version feature is important.

IMPORTANT: For best results, we recommend that you check for specific features rather than relying on the system version number to determine if a specific API is available. For more details on checking for features, see the appendix [Compatibility Guide](#) in *Palm OS Programmer's API Reference*.

Application-Defined Features

Applications may find the feature manager useful for their own private use. For example, an application may want to publish a feature that contains a pointer to some private data it needs for processing launch codes. Because an application's global data is not generally available while it processes launch codes, using the feature manager is usually the easiest way for an application to get to its data.

The feature manager maintains one feature table in the RAM as well as the feature table in the ROM. Application-defined features are stored in the RAM feature table.

Using the Feature Manager

To check whether a particular feature is present, call [FtrGet](#) and pass it the feature creator and feature number. If the feature exists, [FtrGet](#) returns the 32-bit value of the feature. If the feature doesn't exist, an error code is returned.

To publish a new feature or change the value of an existing one, call [FtrSet](#) and pass the feature creator, number, and the 32-bit value of the feature. A published feature remains available until it is explicitly removed by a call to [FtrUnregister](#) or until the system resets; simply quitting an application doesn't remove a feature published by that application.

Call [FtrUnregister](#) to remove features that were created by calling [FtrSet](#).

You can get a complete list of all published features by calling [FtrGetByIndex](#) repeatedly. Passing an index value starting at 0 to [FtrGetByIndex](#) and incrementing repeatedly by 1 eventually returns all available features. [FtrGetByIndex](#) accepts a parameter that specifies whether to search the ROM feature table or RAM feature table. Note that in Palm OS version 3.1 and higher, the contents of the ROM table are copied into the RAM table at system startup; thus the RAM table serves the entire system.

Feature Memory

Palm OS 3.1 adds support for **feature memory**. Feature memory provides quick, efficient access to data that persists between invocations of an application. The values stored in feature memory persist until the handheld is reset or until you explicitly free the memory. Feature memory is memory allocated from the storage heap. Thus, you write to feature memory using [DmWrite](#), which means that writing to feature memory is no faster than writing to a database. However, feature memory can provide more efficient access to that data in certain circumstances.

To allocate a chunk of feature memory, call [FtrPtrNew](#), specifying a feature creator, a feature number, the number of bytes to allocate, and a location where the feature manager can return a pointer to the newly allocated memory chunk. For example:

```
FtrPtrNew(appCreator,  
          myFtrMemFtr, 32, &ftrMem);
```

Elsewhere in your application, you can obtain the pointer to the feature memory chunk using [FtrGet](#).

NOTE: Starting with Palm OS 3.5 [FtrPtrNew](#) allows allocating chunks larger than 64KB. Do keep in mind standard issues with allocating large chunks of memory: there might not be enough contiguous space, and it can impact system performance.

Feature memory is considered a performance optimization. The conditions under which you'd use it are not common, and you probably won't find them in a typical application. You use feature memory in code that:

- Is executed infrequently
- Does not have access to global variables
- Needs access to data whose contents change infrequently and that cannot be stored in a 32-bit feature value

For example, suppose you've written a function that is called in response to a launch code, and you expect to receive this launch code frequently. Suppose that function needs access to the application's preferences database. At the start of the function,

you'd need to open the database and read the data from it. If the function is called frequently, opening the database each time can be a drain on performance. Instead, you can allocate a chunk of feature memory and write the values you need to that chunk. Because the chunk persists until the handheld is reset, you only need to open the database once. [Listing 10.1](#) illustrates this example.

Listing 10.1 Using feature memory

```
MyAppPreferencesType prefs;

if (FtrGet(appCreator, myPrefFtr, (UInt32*)&prefs) != 0) {

    // Feature memory doesn't exist, so allocate it.
    FtrPtrNew(appCreator, myPrefFtr, 32, &thePref);

    // Load the preferences database.
    PrefGetAppPreferences (appCreator, prefID, &prefs,
        sizeof(prefs), true);

    // Write it to feature memory.
    DmWrite(thePref, 0, &prefs, sizeof(prefs));
}
// Now prefs is guaranteed to be defined.
```

Another potential use of feature memory is to “publish” data from your application or library to other applications when that data doesn’t fit in a normal 32-bit feature value. For example, suppose you are writing a communications library and you want to publish an icon that client applications can use to draw the current connection state. The library can use `FtrPtrNew` to allocate a feature memory chunk and store an icon representing the current state in that location. Applications can then use `FtrGet` to access the icon and pass the result to `WinDrawBitmap` to display the connection state on the screen.

Preferences

The Preferences Manager handles both system-wide preferences and application-specific preferences. The Preferences Manager maintains preferences in two separate databases:

- The “saved” preferences database contains preferences that are backed up during a HotSync operation. There is one “saved” preferences database that all applications use. This database contains all system-wide preferences as well as application-specific preferences.
- The “unsaved” preferences database contains application-specific preferences that are not to be backed up during a HotSync operation. There is one “unsaved” preferences database that all application use.

This section describes how to obtain and set values for each of these preferences databases. It covers:

- [Accessing System Preferences](#)
- [Setting System Preferences](#)
- [Setting Application-Specific Preferences](#)

Accessing System Preferences

The system preferences specify how users want their Palm Powered™ handhelds to behave. For example, system preferences specify how dates and times are displayed and whether the system plays a sound when an alarm fires. These values are typically set using the built-in Preferences or Security application. Applications should, as a rule, respect the values stored in the system preferences.

To obtain the value of a system preference, use the [PrefGetPreference](#) function and pass one of the [SystemPreferencesChoice](#) enum constants. For example, if an application’s user interface displays the current date and time, it could do the following to find out how the user wants the date and time displayed:

```
TimeFormatType timeFormat = (TimeFormatType)
    PrefGetPreference(prefTimeFormat);
DateFormatType dateFormat = (DateFormatType)
    PrefGetPreference(prefDateFormat);
```

WARNING! Do not confuse `PrefGetPreference` with `PrefGetPreferences`. The latter function is obsolete and retrieves the 1.0 version of the system preferences structure.

Note that the `PrefGetPreference` function by default returns a `UInt32` value. This return value must be cast to the appropriate type for the preference being returned.

Also note that the system preferences structure has been updated many times and maintains its own version information. Each Palm OS release that modifies the system preferences structure adds its new values to the end and increments the structure's version number. See [Table 10.1](#).

Table 10.1 System preference version numbers

Palm OS Version	System Preference Version
2.0	2
3.0	3
3.0	4
3.1	5
3.2	6
3.3	7
3.5	8
4.0	9

To learn which preferences were added in which version, as well as the return type expected for each preference, see [Table 43.1](#) on page 798 in the *Palm OS Programmer's API Reference*.

To maintain backward compatibility, check the preference version number before checking the value of any preference added after Palm OS 2.0. For example, Palm OS 4.0 added a preference that allows you to access the handheld's locale information (the country and language) as an `LmLocaleType` structure. Before you try to

access that preference, you should check the preference version, as shown in [Listing 10.2](#).

Listing 10.2 Checking the system preference version

```
LmLocaleType currentLocale;
CountryType currentCountry;
if (PrefGetPreference(prefVersion) >= preferenceDataVer9) {
    currentLocale = (LmLocaleType)
        PrefGetPreference(prefLocale);
} else { /* make do with the country */
    currentCountry = (CountryType)
        PrefGetPreference(prefCountry);
}
```

In some cases, a newer preference is intended to replace an existing preference. For example, the `prefAutoOffDuration` preference is replaced by `prefAutoOffDurationSecs` in version 8 of the preference structure. The older preference stored the auto-off time in minutes, and the newer one stores the time in seconds. If you use `prefAutoOffDuration` in Palm OS 4.0, the system still returns the current auto-off time in minutes; however, to obtain this value, the system converts the value in seconds to minutes and rounds the result if necessary. You'll receive a more precise value if you use `prefAutoOffDurationSecs`.

Setting System Preferences

Occasionally, an application may need to set the value of a system-wide preference. It is strongly recommended that you not override the system preferences without user input.

For example, suppose you are writing a replacement for the built-in Address Book application. The Preferences application contains a panel where the user can remap the Address Book hard key to open any application they choose. However, 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 Yes, then you should call

[PrefSetPreference](#) with the new value. The code might look like the following:

Listing 10.3 Setting a system preference

```
if (PrefGetPreference(prefHard2CharAppCreator !=
    myAppCreatorId)) {
    if (FrmAlert(MakeMeTheDefaultAlert) == 0) {
        /* user pressed Yes */
        PrefSetPreference(prefHard2CharAppCreator ,
            myAppCreatorId);
    }
}
```

WARNING! Do not confuse `PrefSetPreference` with `PrefSetPreferences`. The latter function is obsolete and sets the entire system preferences structure for version 1.0.

Setting Application-Specific Preferences

You can use the Preferences Manager to set and retrieve preferences specific to your application. You do this by storing the preferences in one of two databases: the “saved” preferences database or the “unsaved” preferences database.

To write application preferences, you use [PrefSetAppPreferences](#). To read them back in, you use [PrefGetAppPreferences](#). Typically, you write the preferences in response to the `appStopEvent` when control is about to pass to another application. You read the preferences in response to a normal launch.

`PrefSetAppPreferences` and `PrefGetAppPreferences` take roughly the same parameters: the application creator ID, a preference ID that uniquely identifies this preference resource, a pointer to a structure that holds the preference values, the size of the preferences structure, and a Boolean that indicates whether the “saved” or the “unsaved” preferences database is to be used. `PrefSetAppPreferences` also takes a version number for the

preference structure. This value is the return value for `PrefGetAppPreferences`.

The following sections discuss the issues involved in using application-specific preferences:

- [When to Use Application Preferences](#)
- [How to Store Preferences](#)
- [Which Preferences Database to Use](#)
- [Updating Preferences Upon a New Release](#)

When to Use Application Preferences

You use application preferences to store state specific to your application that should persist across invocations of your application. For example, the built-in applications store information about the last form and the last record or records displayed before control switched to another application. This way, the user can be returned to the same view when he or she goes back to that application.

You can also use preferences for other values. You might allow the user to customize the way the application behaves and store such information in the preferences database. You might also use the preferences database as a way to share information with other applications.

Make sure that the preference values you choose are as concise as possible. In games, for example, it is often tempting to store a bitmap for the current state of the screen. Such a bitmap is over 25KB on a color handheld, and it is therefore best avoided. Instead, it is better to store items that let you recreate the current state, such as the player's position in pixels and the current level.

There are other ways to store values pertinent to your application. For example, you can store a single value as a feature using the Feature Manager. The differences between storing application value as preferences and storing application values as features are:

- Preferences (including those stored in the "unsaved" database) survive a soft reset because they reside in the storage heap. Features are deleted upon a soft reset. For this reason, preferences are more appropriate for storing application state than feature are.

- An application preference is a database record, so it has a size limit of 64KB. Multiple application preferences are allowed. The features that are supported by all releases of Palm OS have a maximum size of 4KB.

In Palm OS 3.1 and higher, you can create features greater than 4KB by using feature memory. Feature memory has a maximum size of 64KB before Palm OS 3.5. Palm OS 3.5 and higher allows allocating a chunk of feature memory that is larger than 64KB. However, feature memory is intended to be used in different situations than an application preference is. See the section “[Feature Memory](#)” on page 319 for more information.

Instead of storing application state values as preferences or features, you could also use a database that your application creates and maintains itself. If you choose this method of storing application preference values, you must write your own functions to read the preferences from the database and write the preferences to the database. If you want the preferences backed up, you need to set the backup bit. However, there may be cases where using your own database has advantages. See “[Which Preferences Database to Use](#)” on page 327.

How to Store Preferences

Most applications store a single preference structure under a single preference resource ID. When the application receives an `appStopEvent`, it writes the entire structure to the resource using [PrefSetAppPreferences](#). When it receives a `sysAppLaunchCmdNormalLaunch`, it reads the structure back in using [PrefGetAppPreferences](#).

Storing a single preference structure in the database is a convention that most applications follow because it is convenient to access the all preferences at once. The Preferences Manager does allow an application to store more than one preference resource. This requires more calls to `PrefSetAppPreferences` and `PrefGetAppPreferences`, but you may find it more convenient to use several preference resources if you have several variable-length preferences.

Which Preferences Database to Use

Both [PrefGetAppPreferences](#) and [PrefSetAppPreferences](#) take a Boolean value that indicates whether the value is to be read from and written to the “saved” or the “unsaved” preferences database. To write the preference to the “saved” preferences database, use `true`. To write to the “unsaved” preferences database, use `false`.

The only difference between the two databases is that the “saved” preferences database is backed up when a user performs the HotSync operation, and the “unsaved” preferences database is not backed up by default. (The user can use a third-party tool to set the backup bit in the “unsaved” preferences database, which would cause it to be backed up.) Both the “saved” and the “unsaved” preferences reside in the storage heap and thus persist across soft resets. The only way that preferences are lost is if a hard reset is performed.

Use the “saved” preferences only for items that must be restored after a hard reset, and use the “unsaved” preferences for the current state of the application. For example, if your application has a registration code, you might write that to the “saved” preferences database so that the user does not have to look up the registration code and re-enter it after a hard reset. However, such items as the current form being displayed and the current database record being displayed can be lost, so they are written to the “unsaved” preferences database. For games, you might write the high score to the “saved” preferences database and any information about the current game to the “unsaved” preferences database.

It is important to use the “saved” preferences database sparingly. Any time that any application stores or changes a preference in the “saved” preferences database, the **entire** database is backed up during the next HotSync operation. For users with a large number of applications, this practice can seriously impact the time that it takes to perform a HotSync operation.

[Listing 10.4](#) shows the preferences structures and the `StopApplication` function from the `HardBall` application. The `HardBallPreferenceType`, which is written to the “saved” preferences database, only stores the high score information and accumulated time. All other preferences are stored in

GameStatusType, which is written to the “unsaved” preferences database.

Listing 10.4 Saving application-specific preferences

```
typedef struct {
    SavedScore highScore[highScoreMax];
    UInt8 lastHighScore;
    UInt8 startLevel;
    UInt32 accumulatedTime;
} HardBallPreferenceType;

typedef struct {
    enum gameProgress status;
    UInt8 periodLength;
    UInt32 nextPeriodTime;
    UInt32 periodsToWait;
    Boolean paused;
    UInt32 pausedTime;
    BrickType brick[rowsOfBricks][columnsOfBricks];
    UInt8 bricksRemaining;
    UInt8 level;
    WorldState last;
    WorldState next;
    RemovedBrick brokenBricks[brokenBricksMax];
    Int16 brokenBricksCount;
    UInt8 ballsRemaining;
    Boolean movePaddleLeft;
    Boolean movePaddleRight;
    SoundType soundToMake;
    Int8 soundPeriodsRemaining;
    Int32 scoreToAwardBonusBall;
    Boolean lowestHighScorePassed;
    Boolean highestHighScorePassed;
    Boolean gameSpedUp;
    Boolean cheatMode;
    UInt32 startTime;
} GameStatusType;

HardBallPreferenceType Prefs;
static GameStatusType GameStatus;

static void StopApplication (void)
{
    ...
    // Update the time accounting.
    Prefs.accumulatedTime += (TimGetTicks() -
```

```
        GameStatus.startTime);

    // If we are saving a game resuming (it hasn't started
    // playing yet) then preserve the game status.
    if (GameStatus.status == gameResuming) {
        GameStatus.status = SavedGameStatus;
    }

    // Save state/prefs.
    PrefSetAppPreferences (appFileCreator, appPrefID,
        appPrefVersion, &Prefs, sizeof (Prefs), true);

    PrefSetAppPreferences (appFileCreator, appSavedGameID,
        appSavedGameVersion, &GameStatus, sizeof (GameStatus),
        false);

    // Close all the open forms.
    FrmCloseAllForms ();
}
```

If you have a large amount of preference data that must be backed up during a HotSync operation and is frequently changed, you could, as a performance optimization, store the preferences in a database that your own application creates and maintains rather than in the “saved” preferences database. This saves the user from having to have the entire “saved” preferences database backed up on every HotSync operation. The disadvantage of this method of saving application preferences is that you must write all code to maintain the database and to retrieve information from it.

Updating Preferences Upon a New Release

When you update your application, you may have new items that you want to store in the preferences database. You may choose to write a separate preference record to the database. However, it is better to update the current preference structure, size permitting.

The [PrefSetAppPreferences](#) and [PrefGetAppPreferences](#) functions use a versioning system that allows you to update an existing preference structure. To use it, keep track of the version number that you pass to `PrefSetAppPreferences`. Add any new preferences to the end of the preferences structure, and then

increment the version number. You might use a macro for this purpose:

```
#define CurrentPrefsVersion 2
```

When a user launches the new version of the application, `PrefGetAppPreferences` is called before `PrefSetAppPreferences`. The `PrefGetAppPreferences` function returns the version number of the preference structure that it retrieved from the database. For example, if the new version is version 2, `PrefGetAppPreferences` returns 1 the first time that version 2 is run. If the returned version does not match the current version, you know that the user does not have values for the new preferences introduced in version 2. You can then decide to provide default values for those new preferences.

The first time any version of your application is run, `PrefGetAppPreferences` returns `noPreferenceFound`. This indicates that the user does not have any preferences for the current application and the application must supply default values for the entire preferences structure. [Listing 10.5](#) shows how the Datebook handles retrieving the version number from `PrefGetAppPreferences`.

Listing 10.5 Checking the preference version number

```
#define datebookPrefsVersionNum 4

Int16 DatebookLoadPrefs (DatebookPreferenceType* prefsP)
{
    UInt16 prefsSize;
    Int16 prefsVersion = noPreferenceFound;
    Boolean haveDefaultFont = false;
    UInt32 defaultFont;

    ErrNonFatalDisplayIf(!prefsP, "null prefP arg");

    // Read the preferences / saved-state information.  Fix-up if no prefs or
    // older/newer version
    prefsSize = sizeof (DatebookPreferenceType);
    prefsVersion = PrefGetAppPreferences (sysFileCDatebook, datebookPrefID,
        prefsP, &prefsSize, true);

    // If the preferences version is from a future release (as can happen when
```



```
// going back and syncing to an older version of the device), treat it the
// same as "not found" because it could be significantly different
if ( prefsVersion > datebookPrefsVersionNum )
    prefsVersion = noPreferenceFound;

if ( prefsVersion == noPreferenceFound ) {
    // Version 1 and 2 preferences
    prefsP->dayStartHour = defaultDayStartHour;
    prefsP->dayEndHour = defaultDayEndHour;
    prefsP->alarmPreset.advance = defaultAlarmPresetAdvance;
    prefsP->alarmPreset.advanceUnit = defaultAlarmPresetUnit;
    prefsP->saveBackup = defaultSaveBackup;
    prefsP->showTimeBars = defaultShowTimeBars;
    prefsP->compressDayView = defaultCompressDayView;
    prefsP->showTimedAppts = defaultShowTimedAppts;
    prefsP->showUntimedAppts = defaultShowUntimedAppts;
    prefsP->showDailyRepeatingAppts =
        defaultShowDailyRepeatingAppts;

    // We need to set up the note font with a default value for the system.
    FtrGet(sysFtrCreator, sysFtrDefaultFont, &defaultFont);
    haveDefaultFont = true;

    prefsP->v20NoteFont = (FontID)defaultFont;
}

if ((prefsVersion == noPreferenceFound) || (prefsVersion <
    datebookPrefsVersionNum)) {
    // Version 3 preferences
    prefsP->alarmSoundRepeatCount = defaultAlarmSoundRepeatCount;
    prefsP->alarmSoundRepeatInterval = defaultAlarmSoundRepeatInterval;
    prefsP->alarmSoundUniqueRecID = defaultAlarmSoundUniqueRecID;
    prefsP->noteFont = prefsP->v20NoteFont;

    // Fix up the note font if we copied from older preferences.
    if ((prefsVersion != noPreferenceFound) && (prefsP->noteFont ==
        largeFont))
        prefsP->noteFont = largeBoldFont;

    if (!haveDefaultFont)
        FtrGet(sysFtrCreator, sysFtrDefaultFont, &defaultFont);

    prefsP->apptDescFont = (FontID)defaultFont;
}

if ((prefsVersion == noPreferenceFound) || (prefsVersion <
    datebookPrefsVersionNum)) {
    // Version 4 preferences
```

```
    prefsP->alarmSnooze = defaultAlarmSnooze;
}

return prefsVersion;
}
```

Sound

The Palm OS 5 Sound Manager controls two independent sound facilities:

- **Simple sound:** Single voice, monophonic, square-wave sound synthesis, useful for system beeps. This is the traditional (pre-OS 5) PalmSource sound.
- **Sampled sound:** Stereo, multi-format, sampled data recording and playback (new in Palm OS 5). Sampled sounds can be generated programmatically or read from a soundfile.

These facilities are independent of each other. Although you can play a simple sound and a sampled sound at the same, their respective APIs have no effect on each other. For example, you can't use the sampled sound volume-setting function ([SndStreamSetVolume](#)) to change the volume of a simple sound.

The following sections take a look at the concepts introduced by the Sound Manager. For detailed API descriptions, and for more guidance with regard to the sampled data concepts presented here, see [Chapter 45, "Sound Manager."](#)

Simple Sound

There are three ways to play a simple sound:

- You can play a single tone of a given pitch, amplitude, and duration by calling [SndDoCmd](#).
- You can play a pre-defined system sound ("Information," "Warning," "Error," and so on) through [SndPlaySystemSound](#).
- You can play a tune by passing in a Level 0 Standard MIDI File (SMF) through the [SndPlaySmf](#) function. For example, the alarm sounds used in the built-in Date Book application are MIDI records stored in the System MIDI database. MIDI

support is included with Palm OS 3.0 and later. For information on MIDI and the SMF format, go to the official MIDI website, <http://www.midi.org>.

Sampled Sound

Over in the sampled sound facilities, there are two fundamental functions:

- [SndStreamCreate](#) opens a new sampled sound “stream” from/into which you record/playback buffers of “raw” data. The trick is that you first have to configure the stream to tell it how to interpret the data. (An alternate function, [SndStreamCreateExtended](#), lets you declare an encoding format.)
- [SndPlayResource](#) is used to playback sound data that’s read from a (formatted) soundfile. The function configures the playback stream for you, based on the format information in the soundfile header. Currently, only uncompressed WAV and IMA ADPCM WAV formats are recognized.

The Sound Manager also provides functions that let you set the volume and stereo panning for individual recording and playback streams. See [SndStreamSetVolume](#) and [SndStreamSetPan](#).

Simple vs Sampled

Comparing the two facilities, simple sound is easy to understand and requires very little programming: In most cases, you load up a structure, call a function, and out pops a beep. Unfortunately, the sound itself is primitive. (An example of simple sound programming is given in “[Sound Preferences](#),” below.)

Sampled sound, on the other hand, is (or can be) much more satisfying, but requires more planning than simple sound. How much more depends on what you’re doing. Playing samples from a soundfile isn’t much more difficult than playing a simple sound, but you have to supply a soundfile. Generating samples programmatically—and recording sound—requires more work: You have to implement a callback function that knows something about sound data.

IMPORTANT: One significant difference between simple sounds and sampled sounds is that they use different volume scales: Simple sound volumes are in the range [0, 64]; sampled sound volumes are [0, 1024].

Sound Preferences

If you're adding short, "informative" sounds to your application, such as system beeps, alarms, and the like, you should first consider using the (simple) system sounds that are defined by the Palm OS, as listed in the reference documentation for [SndPlaySystemSound/](#)

If you want to create your own system-like sounds, you should at least respect the user's preferences settings with regard to sound volume. In Palm OS 3.0 and later, there are three sound preference constants:

- `prefSysSoundVolume` is the default system volume.
- `prefGameSoundVolume` is used for game sounds.
- `prefAlarmSoundVolume` is used for alarms.

To apply a sound preference setting to a simple sound volume, you have to retrieve the setting and apply it yourself. For example, here we retrieve the alarm sound and use it to set the volume of a simple sound:

```
/* Create a 'sound command' structure. This will encode the parameters of the
tone we want to generate.
*/
SndCommandType sndCommand;

/* Ask for the 'play a tone' command. */
sndCommand.cmd = sndCmdFreqDurationAmp;

/* Set the frequency and duration. */
sndCommand.param1 = 1760;
sndCommand.param2 = 500;

/* Now get the alarm volume and set it in the struct. */
```

```
sndCommand.param3 = PrefGetPreference (prefAlarmSoundVolume);

/* Play the tone. */
SndDoCmd( 0, &sndCommand, true);
```

The sampled sound API, on the other hand, provides volume constants (sndSystemVolume, sndGameVolume, and sndSysVolume) that look up a preference setting for you:

```
/* Point our sound data pointer to a record that contains WAV data (record
retrieval isn't shown).
*/
SndPtr soundData = MemHandleLock(...);

/* Play the data using the default alarm volume setting. */
SndPlayResource(soundData, sndAlarmVolume, sndFlagNormal);

/* Unlock the data. */
MemPtrUnlock(soundData);
```

For greatest compatibility with multiple versions of the sound preferences mechanism, your application should check the version of Palm OS on which it is running. See [“The System Version Feature”](#) for more information.

Standard MIDI Files

Although you can use a Level 0 Standard MIDI File to control simple sound generation, this doesn't imply broad support for MIDI messages: Only key down, key up, and tempo change messages are recognized.

You can store your MIDI data in a MIDI database:

- The database type `sysFileTMidi` identifies MIDI record databases.
- The system MIDI database is further identified by the creator `sysFileCSysm`. The database holds a number of system alarm sounds.

You can add MIDI records to the system MIDI database, or you can store them in your own.

Each record in a MIDI database is concatenation of a PalmSource-defined MIDI record header, the human-readable name of the MIDI data, and then the MIDI data itself.

The following code creates a new MIDI record and adds it to the system MIDI database.

```
/* We need three things: A header, a name, and some data. We'll get the name
and data from somewhere, and create the header ourselves.
*/
char *midiName = ...;
MemHandle midiData = ...;
SndMidiRecHdrType midiHeader;

/* Database and record gadgetry. */
DmOpenRef database;
MemHandler record;
UInt16 *recordIndex = dmMaxRecordIndex;
UInt8* recordPtr;
UInt8* midiPtr;

/* MIDI header values: Always set the signature to sndMidiRecSignature, and
reserved to 0. bDataOffset is an offset from the beginning of the header to the
first byte of actual MIDI data. The name includes a null-terminator, hence the
'+ 1'.
*/
midiHeader.signature = sndMidiRecSignature;
midiHeader.reserved = 0;
midiHeader.bDataOffset = sizeof(SndMidiRecHdrType) + StrLen(midiName) + 1;

/* Open the database and allocate a record. */
database = DmOpenDatabaseByTypeCreator( sysFileTMidi, sysFileCSystem,
                                         dmModeReadWrite | dmModeExclusive);
record = DmNewRecord database, &recordIndex,
          midiHeader.bDataOffset + MemHandleSize(midiData));

/* Lock the data and the record. */
midiDataPtr = MemHandleLock(midiData);
recordPtr = MemHandleLock(record);

/* Write the MIDI header. */
DmWrite( recordPtr, 0, &smidiHeader, sizeof(midiHeader));

/* Write the track name. */
DmStrCopy( recordPtr, ((UInt32)(&((SndMidiRecType *)0)->field, midiName));

/* Write the MIDI data. */
```

```
DmWrite( recordPtr, midiHeader.bDataOffset, midiDataPtr,  
        MemHandleSize(midiData));  
  
/* Unlock the handles, release the record, close the database. */  
MemHandleUnlock( midiData);  
MemHandleUnlock( record);  
DmReleaseRecord( database, recordIndex, 1);  
DmCloseDatabase( database);
```

To retrieve a MIDI record, you can use the [SndCreateMidiList](#) function if you know the record's creator, or you can use the Data Manager functions to iterate through all MIDI records.

Creating a Sound Stream

The sound stream API, part of the sampled sound facility, is the most flexible part of the Sound Manager. A sound stream sends sampled data to or reads sampled data from the sound hardware. There are 15 sound output streams and one input stream, all running (or potentially running) concurrently.

To use a sound stream, you have to tell it what sort of data you're going to give it or that you expect to get from it. All of the sound format information that you need to supply to set up the stream—data quantization, sampling rate, channel count, and so on—is passed in the [SndStreamCreate](#) or [SndStreamCreateExtended](#) function.

You also have to pass the function a pointer to a callback function (see [SndStreamBufferCallback](#) or [SndStreamVariableBufferCallback](#)); implementing this function is where you'll be doing most of your work. When you tell your stream to start running ([SndStreamStart](#)), the callback function is called automatically, once per buffer of data. If you're operating on an input stream (in other words, if you're recording), your callback function is expected to empty the buffer, do something with the data, and then return before the next buffer shows up. Output stream callbacks do the opposite—they fill the buffer with data.

Because of the amount of data involved, the callbacks must operate as quickly as possible. This is particularly important for output stream callbacks: Not only can there be more than one stream

competing for attention, but all output callbacks run in the same task (which is created and managed by the Sound Manager). If the callback for (output) stream **A** takes too long to fill the stream with data, the callbacks for stream **B**, **C**, and so on, will starve. Starving threads make glitchy sounds.

The formats that are supported by the sampled sound functions are described in the functions themselves.

System Boot and Reset

Any reset is normally performed by sticking a bent-open paper clip into the small hole in the back of the handheld. This hole, known as the “reset switch” is above and to the right of the serial number sticker (on Palm III handhelds). Depending on additional keys held down, the reset behavior varies, as follows:

Soft Reset

A soft reset clears all of the dynamic heap (Heap 0, Card 0). The storage heaps remain untouched. The operating system restarts from scratch with a new stack, new global variables, restarted drivers, and a reset communication port. All applications on the handheld receive a [`sysAppLaunchCmdSystemReset`](#) launch code.

Soft Reset + Up Arrow

Holding the up-arrow down while pressing the reset switch with a paper clip causes the same soft reset logic with the following two exceptions:

- The `sysAppLaunchCmdSystemReset` launch code is not sent to applications. This is useful if there is an application on the handheld that crashes upon receiving this launch code (not uncommon) and therefore prevents the system from booting.
- The OS won't load any system patches during startup. This is useful if you have to delete or replace a system patch database. If the system patches are loaded and therefore open, they cannot be replaced or deleted from the system.

Hard Reset

A hard reset is performed by pressing the reset switch with a paper clip while holding down the power key. This has all the effects of the soft reset. In addition, the storage heaps are erased. As a result, all programs, data, patches, user information, etc. are lost. A confirmation message is displayed asking the user to confirm the deletion of all data.

The `sysAppLaunchCmdSystemReset` launch code is sent to the applications at this time. If the user selected the “Delete all data” option, the digitizer calibration screen comes up first. The default databases for the four main applications is copied out of the ROM.

If you hold down the up arrow key when the “Delete all data” message is displayed, and then press the other four application buttons while still holding the up arrow key, the system is booted without reading the default databases for the four main applications out of ROM.

System Reset Calls

The system manager provides support for rebooting the Palm Powered handheld. It calls [SysReset](#) to reset the handheld. This call does a soft reset and has the same effect as pressing the reset switch on the unit. *Normally applications should not use this call.*

`SysReset` is used, for example, by the Sync application. When the user copies an extension onto the Palm Powered handheld, the Sync application automatically resets the handheld after the sync is completed to allow the extension to install itself.

ARM-Native Functions

Palm OS 5 includes the Palm Application Compatibility Environment (PACE), within which all Palm OS applications run. PACE emulates the 68K-family processor traditionally used in Palm Powered handhelds, enabling both new and existing applications to run on Palm Powered handhelds that employ an ARM processor.

Palm OS 5 itself is entirely ARM-native, so all operating system functions run at the full speed of the underlying processor. Because

most applications spend the bulk of their time executing operating system functions, they get the performance benefit of the ARM processor with no effort. Occasionally, however, you'll write a processor-intensive function that could benefit from being recompiled as ARM code, and Palm OS 5 has a mechanism to allow this. These ARM functions can call back into the operating system and can call user-defined 68K functions as well, but because it is difficult to debug both the "68K side" and the "ARM side" of an application, and because of the overhead involved in byte-swapping the parameters, functions that make many calls into the operating system typically aren't good candidates to make ARM-native.

Palm OS 5 includes [PceNativeCall](#) which, when given a pointer to an ARM function and a pointer to a parameter block, calls the ARM function. Due to endianness differences between the 68K and ARM processors, [PceNativeCall](#) byte-swaps the parameter pointer and return value. Because the operating system has no knowledge of the structure of the parameter block, however, it performs no byte-swapping in this block. Your ARM code must do this as necessary for your application (see "[Accessing 68K Data From an ARM Function](#)" on page 342 for more information). Also note that the context in which the code is called is undefined other than to include enough stack space for "reasonable" algorithms.

You typically store the ARM code in a resource in the calling 68K application, and locate it using the [DmGetResource](#) and [MemHandleLock](#) functions.

Calling an ARM Function

Calling an ARM function is usually just a matter of passing the proper parameters to [PceNativeCall](#), as defined in the *Palm OS Programmer's API Reference*. However, before calling [PceNativeCall](#) you must test the processor type:

- If the processor is ARM, the 68K application should call the ARM function.
- If the processor is an x86 family processor (that is, the application is running in Palm OS Simulator on Windows), the 68K application should call the Windows DLL that represents the ARM function.

- Otherwise, the 68K application should either call a 68K version of the function or fail gracefully if the functionality cannot be reasonably incorporated into a 68K application. Note that in this instance your application may be running on a version of Palm OS earlier than Palm OS 5.

[Listing 10.6](#) illustrates this process. For simplicity, in this example no parameters are passed to the ARM function.

Listing 10.6 Calling an ARM function

```
static UInt32 PceNativeResourceCall(DmResType resType, DmResID resID,
char *DLEntryPointP, void *userCPB) {
    UInt32    processorType;
    MemHandle armH;
    MemPtr    armP;
    UInt32    result;

    // get the processor type
    FtrGet(sysFileCSystem, sysFtrNumProcessorID, &processorType);

    if (sysFtrNumProcessorIsARM(processorType)){
        // running on ARM; call the actual ARM resource
        armH = DmGetResource(resType, resID);
        armP = MemHandleLock(armH);

        result = PceNativeCall(armP, userCPB);

        MemHandleUnlock(armH);
        DmReleaseResource(armH);
    } else if (processorType == sysFtrNumProcessorx86) {
        // running on Simulator; call the DLL
        result = PceNativeCall((NativeFuncType *)DLEntryPointP, userCPB);
    } else {
        // some other processor; fail gracefully
        ErrNonFatalDisplay("Unsupported processor type");
        result = -1;
    }

    return result;
}
```

Note that the #defines for the various processor types (all of which are named `sysFtrNumProcessor...`) can be found in `SystemMgr.h`.

ARM Function Definition

ARM functions that are to be called from the 68K side—that is, functions that serve as entry points into your ARM code—must use the following function prototype (defined in `PceNativeCall.h`):

```
typedef unsigned long NativeFuncType (const void *emulStateP,  
void *userData68KP, Call68KFuncType *call68KFuncP)
```

The function parameters are defined as follows:

- > `emulStateP` A pointer to the (opaque) PACE emulation state. This pointer is used when calling Palm OS functions and application callbacks; see [“Calling Palm OS Functions From ARM Code”](#) on page 345 for more information.
- <-> `userData68KP` The `userDataP` argument that was passed in to `PceNativeCall`, byte-swapped so it can be dereferenced directly by the ARM code. See [“Accessing 68K Data From an ARM Function,”](#) below, for tips on accessing the data block indicated by this pointer.
- > `call68KFunc` A hook to call back into the PACE emulated environment from ARM code. It is used for both OS function calls and application callbacks. See [“Calling a Function via a Function Pointer”](#) on page 347 for more information.

ARM entry-point functions should return a value that is meaningful to the 68K side, since that value is passed back to the calling code. If no value is meaningful, return 0. Both register A0 and D0 are set to this return value, making it meaningful to code that is expecting either a pointer or an immediate result.

Accessing 68K Data From an ARM Function

When writing an ARM function that accesses data in the parameter block passed in from the 68K side, you need to be aware of differences in endianness, word alignment, and structure packing.

- The 68K processor is big endian, while Palm OS 5 uses the ARM processor in little-endian mode. Because of this, you'll need to byte-swap all 2- and 4-byte integers and pointers (other than `userData68KP`, which is already byte-swapped for you).
- 68K structures generally are aligned to 2-byte boundaries. (This includes stack-based structures.) The ARM processor expects 4-byte values to be aligned on a 4-byte boundary. Copy 4-byte integers into local variables before using them, but note that because of possible alignment problems a simple pointer dereference won't do when copying the values; you must provide your own unaligned read and write functions. Note that [MemPtrNew](#) returns chunks beginning on 4-byte boundaries; careful structure layout can avoid alignment (but not endian) problems.
- Depending on the compiler options specified, a given compiler can add padding bytes to align structure components on a given byte boundary. If the ARM compiler you use expects structures to be aligned other than how they are by the 68K compiler, errors in alignment will result. You'll generally want to make a local copy of any structures that you use.

The macros in [Listing 10.7](#) can prove very useful in your ARM code. `ByteSwap16` and `ByteSwap32` perform byte swapping on 2- and 4-byte quantities, respectively. `Write68KUnaligned32` and `Read68KUnaligned32` copy 4-byte integers to and from a location in memory that is not necessarily aligned on a 4-byte boundary.

Listing 10.7 Byte-swapping macros

```
#define ByteSwap16(n) ( (((unsigned int) n) << 8) & 0xFF00) | \
                      (((unsigned int) n) >> 8) & 0x00FF) )

#define ByteSwap32(n) ( (((unsigned long) n) << 24) & 0xFF000000) | \
                      (((unsigned long) n) << 8) & 0x00FF0000) | \
                      (((unsigned long) n) >> 8) & 0x0000FF00) | \
                      (((unsigned long) n) >> 24) & 0x000000FF) )

#define Read68KUnaligned32(addr) \
    ( (((unsigned char *) (addr))[0]) << 24) | \
      (((unsigned char *) (addr))[1]) << 16) | \
      (((unsigned char *) (addr))[2]) << 8) | \
```

```
((((unsigned char *) (addr)) [3])) )

#define Write68KUnaligned32(addr, value) \
( ((unsigned char *) (addr)) [0]=(unsigned char) ((unsigned long) (value)>>24), \
  ((unsigned char *) (addr)) [1]=(unsigned char) ((unsigned long) (value)>>16), \
  ((unsigned char *) (addr)) [2]=(unsigned char) ((unsigned long) (value)>>8), \
  ((unsigned char *) (addr)) [3]=(unsigned char) ((unsigned long) (value)) )
```

Embedding ARM Code in a 68K Application

Regardless of the mechanism that you use to generate the ARM binary, there are a couple of different ways to get it into a .prc file:

- Use CodeWarrior or a tool such as PiIRC to place the raw ARM binary into a resource file. Then simply include this resource file in your 68K project. This is the easiest of the alternatives by far.
- Copy the resulting binary data into a different resource file as hex data. Use a hex dump utility to process the ARM binary file into a resource.
- Include the ARM code directly in your application's source as integer arrays. Note, however, that the arrays are interpreted as big-endian by the 68K compiler, and as little-endian by the ARM processor. Thus you must byte swap the integer values to get appropriate opcodes. Also, the array itself must be 4-byte aligned in your source, so insure that your compiler settings are appropriate to produce this.

Native-ARM Code and the Palm OS Simulator

Palm OS Simulator doesn't run ARM code. Instead, it provides an implementation of Palm OS 5 running as a native Windows application. Similarly, any ARM function you write must be compiled as a Windows DLL in order to be used with Simulator. Your 68K code must recognize that it is running on Simulator (as described in "[Calling an ARM Function](#)" on page 340) and supply a pointer to the name of the DLL and the function within that DLL when calling your "ARM" function. These two names must be separated by a null character, and the entire sequence must be terminated by a null character. For example, to load the DLL found at C:\TEST_DLL\Debug\Simple.dll and call the function

TestNativeCall within that DLL, you might pass a pointer to the following character string literal:

```
"C:\\TEST_DLL\\Debug\\Simple.dll\\0TestNativeCall"
```

Note that if you don't supply an absolute path, Simulator looks for the DLL in (or relative to) the directory from which PalmSim.exe is running. Thus, if the DLL is located in the same directory as PalmSim.exe, you can call the above function with:

```
"Simple.dll\\0TestNativeCall"
```

Calling Palm OS Functions From ARM Code

In Palm OS 5, native ARM code can call back into the 68K world, either to call Palm OS functions or to call developer-provided callbacks. A single entry point, Call68KFuncType, provides the mechanism for calling both developer-specified 68K functions and OS functions via traps. This function is declared in PceNativeCall.h as follows:

```
typedef unsigned long Call68KFuncType(const void *emulStateP,  
unsigned long trapOrFunction, const void *argsOnStackP,  
unsigned long argsSizeAndwantA0)
```

The function parameters are defined as follows:

- > emulStateP Pointer to the PACE emulation state. Supply the pointer that was passed to your ARM function by PACE.
- > trapOrFunction The trap number AND'ed with kPceNativeTrapNoMask, or a pointer to the function to call. Any value less than kPceNativeTrapNoMask is treated as a trap number.

-> argsOnStackP

Native (little-endian) pointer to a block of memory to be copied to the 68K stack prior to the function call. This memory normally contains the arguments for the 68K function being called. `Call68KFuncType` pops these values from the 68K stack before returning.

-> argsSizeAndwantA0

The number of bytes, in little-endian format, from `argsOnStackP` that are to be copied to the 68K emulator stack. If the function or trap returns its result in 68K register A0 (as when the result is a pointer type), you must OR the byte count with `kPceNativeWantA0`.

The return value from the 68K function (passed either in the 68K register D0 or A0) is returned as the result of this function, based on `argsSizeAndwantA0`. It is returned in native (little-endian) form.

Because of the amount of effort involved in getting parameters byte-swapped and properly aligned, if your ARM code routinely needs to call a series of operating system functions you may find it easier to write a small 68K callback function that calls the operating system functions, and then call this 68K function instead.

Calling a Trap

[Listing 10.8](#) shows how to call an operating system function from ARM native code using the function's trap number. This sample calls `MemPtrNew` to allocate a block of 10 bytes, initializes that block, and returns it as the result of the ARM function.

Listing 10.8 Calling a Palm OS function from ARM code

```
/* This armlet makes a call (through PACE) to MemPtrNew to allocate a buffer.
 * The arguments to the OS function (here just size) must be on the stack, and
 * must be in big-endian format.
 */

#include "PceNativeCall.h"

#include "endianutils.h" // byte-swapping macros
```



```
// from CoreTraps.h
#define sysTrapMemPtrNew 0xA013 // we need this in order to call into MemPtrNew

// prototype for our OS call convenience function
void *PalmOS_MemPtrNew (const void *emulStateP, Call68KFuncType *call68KFuncP,
unsigned long sizeLE);

// This is the main entry point into the armlet.  It's the first function in
// the file so we can calculate its address easily.
unsigned long NativeFunction (const void *emulStateP, void *userData68KP,
Call68KFuncType *call68KFuncP) {
    unsigned char *bufferP;
    int i;

// allocate 10 bytes of memory using a convenience function
    bufferP = (unsigned char*)PalmOS_MemPtrNew(emulStateP, call68KFuncP, 10);

// Do something with the bytes in the buffer
    for (i = 0; i < 9; i++) bufferP[i] = i+'A'; // write in "ABCDEFGH"
    bufferP[9] = 0;                          // terminate the string

    return (unsigned long)bufferP;
}

// Convenience function for calling MemPtrNew within ARM code
void *PalmOS_MemPtrNew(const void *emulStateP, Call68KFuncType *call68KFuncP,
unsigned long sizeLE) {
    // First, declare the argument(s) that will be passed to the OS call.
    // In this case, we're calling MemPtrNew, so we need a size argument.
    // Because this code is compiled by an ARM compiler (little endian),
    // and MemPtrNew expects its argument to be big endian, swap it.
    unsigned long sizeBE = ByteSwap32(sizeLE);

    // Call the trap. Note that because MemPtrNew returns a pointer, the byte
    // count (the last parameter) must be "OR'd" with kPceNativeWantA0.
    return ((void *)((call68KFuncP)(emulStateP,
        PceNativeTrapNo(sysTrapMemPtrNew), &sizeBE, 4 | kPceNativeWantA0)));
}
```

Calling a Function via a Function Pointer

The code excerpt in [Listing 10.9](#) shows how to pass a 68K callback function pointer to ARM native code, and [Listing 10.10](#) shows how to call that 68K function from within the ARM code. The ARM code ultimately accomplishes the same result as in the previous example

(calling `MemPtrNew`), but this example lets the 68K-side callback function do the allocation. Note that the pointer to the callback function is passed to the ARM code as data, embedded in a structure.

The following is implemented on the 68K side:

Listing 10.9 Calling PACE application code from ARM code (68K side)

```
typedef struct MyParamsTag {
    void *myAllocateFunctionP;
    UInt32 anotherValue;
} MyParamsType;

// function to allocate 10 bytes
void *MyAllocateFunction() {
    return MemPtrNew(10);
}

// code to call the native function, defined in the next listing
MyParamsType myParams;
MemHandle armChunkH;
void *myNativeFuncP;
Byte *result;

armChunkH = DmGetResource('armc', 0);
myNativeFuncP = MemHandleLock(armChunkH);

myParams.myAllocateFunctionP = &MyAllocateFunction;

result = (Byte *)PceNativeCall(myNativeFuncP, &myParams);
```

In the ARM file, the following code accepts the callback function pointer and uses the 68K function to allocate the 10-byte memory block:

Listing 10.10 Calling PACE application code from ARM code (ARM side)

```
typedef struct MyParamsTag {
    void *myAllocateFunctionP;
    unsigned long anotherValue;
} MyParamsType;
```

```
unsigned long MyNativeFunc(const void *emulStateP,  
    void *userData68KP, Call68KFuncType *call68KFunc) {  
    unsigned char *buffer68K; // array of Byte  
    unsigned char i; // Byte  
    void *my68KFuncP;  
  
    // get the function pointer out of the passed parameter block  
    my68KfuncP = ByteSwap4(userData68KP->myAllocateFunctionP);  
  
    // invoke the callback function to allocate 10 bytes  
    buffer68K = (void *)((call68KFunc)(emulStateP, my68KFuncP,  
        &size, 4 | kPceNativeWantA0));  
  
    // do something with the bytes in the buffer  
    for (i = 10; i > 0; i--)  
        buffer68K[i] = i;  
  
    return (unsigned long)buffer68K;  
}
```

Hardware Interaction

Palm OS differs from a traditional desktop system in that it's never really turned off. Power is constantly supplied to essential subsystems and the on/off key is merely a way of bringing the handheld in or out of low-power mode. The obvious effect of pressing the on/off key is that the LCD turns on or off. When the user presses the power key to turn the handheld off, the LCD is disabled, which makes it appear as if power to the entire unit is turned off. In fact, the memory system, real-time clock, and the interrupt generation circuitry are still running, though they are consuming little current.

This section looks at Palm OS power management, discussing the following topics:

- [Palm OS Power Modes](#)
- [Guidelines for Application Developers](#)
- [Power Management Calls](#)

Palm OS Power Modes

To minimize power consumption, the operating system dynamically switches between three different modes of operation: sleep mode, doze mode, and running mode. The system manager controls transitions between different power modes and provides an API for controlling some aspects of the power management.

- In **sleep mode**, the handheld looks like it's turned off: the display is blank, the digitizer is inactive, and the main clock is stopped. The only circuits still active are the real-time clock and interrupt generation circuitry.

The handheld enters this mode when there is no user activity for a number of minutes or when the user presses the off button. The handheld comes out of sleep mode only when there is an interrupt, for example, when the user presses a button.

To enter sleep mode, the system puts as many peripherals as possible into low-power mode and sets up the hardware so that an interrupt from any hard key or the real-time clock wakes up the system. When the system gets one of these interrupts while in sleep mode, it quickly checks that the battery is strong enough to complete the wake-up and then takes each of the peripherals, for example, the LCD, serial port, and timers, out of low-power mode.

- In **doze mode**, the main clock is running, the handheld appears to be turned on, the LCD is on, and the processor's clock is running but it's not executing instructions (that is, it's halted). When the processor receives an interrupt, it comes out of halt and starts processing the interrupt.

The handheld enters this mode whenever it's on but has no user input to process.

The system can come out of doze mode much faster than it can come out of sleep mode since none of the peripherals need to be woken up. In fact, it takes no longer to come out of doze mode than to process an interrupt. Usually, when the system appears on, it is actually in doze mode and goes into running mode only for short periods of time to process an interrupt or respond to user input like a pen tap or key press.

- In **running mode**, the processor is actually executing instructions.

The handheld enters this mode when it detects user input (like a tap on the screen) while in doze mode or when it detects an interrupt while in doze or sleep mode. The handheld stays in running mode only as long as it takes to process the user input (most likely less than a second), then it immediately reenters doze mode. A typical application puts the system into running mode only about 5% of the time.

To maximize battery life, the processor on the Palm Powered handheld is kept out of running mode as much as possible. Any interrupt generated on the handheld must therefore be capable of “waking” up the processor. The processor can receive interrupts from the serial port, the hard buttons on the case, the button on the cradle, the programmable timer, the memory module slot, the real-time clock (for alarms), the low-battery detector, and any built-in peripherals such as a pager or modem.

Guidelines for Application Developers

Normally, applications don’t need to be aware of power management except for a few simple guidelines. When an application calls [EvtGetEvent](#) to ask the system for the next event to process, the system automatically puts itself into doze mode until there is an event to process. As long as an application uses [EvtGetEvent](#), power management occurs automatically. If there has been no user input for the amount of time determined by the current setting of the auto-off preference, the system automatically enters sleep mode without intervention from the application.

Applications should avoid providing their own delay loops. Instead, they should use [SysTaskDelay](#), which puts the system into doze mode during the delay to conserve as much power as possible. If an application needs to perform periodic work, it can pass a time out to [EvtGetEvent](#); this forces the unit to wake up out of doze mode and to return to the application when the time out expires, even if there is no event to process. Using these mechanisms provides the longest possible battery life.

Power Management Calls

The system calls `SysSleep` to put itself immediately into low-power sleep mode. Normally, the system puts itself to sleep when there has been no user activity for the minimum auto-off time or when the user presses the power key.

The [SysSetAutoOffTime](#) routine changes the auto-off time value. This routine is normally used by the system only during boot, and by the Preferences application. The Preferences application saves the user preference for the auto-off time in a preferences database, and the system initializes the auto-off time to the value saved in the preferences database during boot. While the auto-off feature can be disabled entirely by calling `SysSetAutoOffTime` with a time-out of 0, doing this depletes the battery.

The current battery level and other information can be obtained through the [SysBatteryInfo](#) routine. This call returns information about the battery, including the current battery voltage in hundredths of a volt, the warning thresholds for the low-battery alerts, the battery type, and whether external power is applied to the unit. This call can also change the battery warning thresholds and battery type.

The Microkernel

Palm OS has a preemptive multitasking kernel that provides basic task management.

Most applications don't need the microkernel services because they are handled automatically by the system. This functionality is provided mainly for internal use by the system software or for certain special purpose applications.

In this version of the Palm OS, there is only one user interface application running at a time. The User Interface Application Shell (UIAS) is responsible for managing the current user-interface application. The UIAS launches the current user-interface application as a subroutine and doesn't get control back until that application quits. When control returns to the UIAS, the UIAS immediately launches the next application as another subroutine. See "[Power Management Calls](#)" for more information.

Usually, the UIAS is the only task running. Occasionally though, an application launches another task as a part of its normal operation. One example of this is the Sync application, which launches a second task to handle the serial communication with the desktop. The Sync application creates a second task dedicated to the serial communication and gives this task a lower priority than the main user-interface task. The result is optimal performance over the serial port without a delay in response to the user-interface controls.

Normally, there is no user interaction during a sync, so that the serial communication task gets all of the processor's time. However, if the user does tap on the screen, for example, to cancel the sync, the user-interface task immediately processes the tap, since it has a higher priority. Alternatively, the Sync application could have been written to use just one task, but then it would have to periodically poll for user input during the serial communication, which would hamper performance and user-interface response time.

NOTE: Only system software can launch a separate task. The multi-tasking API is not available to developer applications.

Retrieving the ROM Serial Number

Some Palm™ handhelds, beginning with the Palm III product, hold a 12-digit serial number that identifies the handheld uniquely. (Earlier handhelds do not have this identifier.) The serial number is held in a displayable text buffer with no null terminator. The user can view the serial number in the [Application Launcher](#) application. (The pop-up version of the Launcher does not display the serial number.) The Application Launcher also displays to the user a checksum digit that you can use to validate user entry of the serial number.

To retrieve the ROM serial number programmatically, pass the `sysROMTokenSnum` selector to the [SysGetROMToken](#) function. If the [SysGetROMToken](#) function returns an error, or if the returned pointer to the buffer is `NULL`, or if the first byte of the text buffer is `0xFF`, then no serial number is available.

The `DrawSerialNumOrMessage` function shown in [Listing 10.11](#) retrieves the ROM serial number, calculates the checksum, and

draws both on the screen at a specified location. If the handheld has no serial number, this function draws a message you specify. This function accepts as its input a pair of coordinates at which it draws output, and a pointer to the message it draws when a serial number is not available.

Listing 10.11 DrawSerialNumOrMessage

```
static void DrawSerialNumOrMessage(Int16 x, Int16 y, Char*
noNumberMessage)
{
    Char* bufP;
    UInt16* bufLen;
    Err retval;
    Int16    count;
    UInt8    checksum;
    Char    checksumStr[2];
        // holds the dash and the checksum digit

    retval = SysGetROMToken (0, sysROMTokenSnum,
                                (UInt8**) &bufP,
&bufLen);
    if ((!retval) && (bufP) && ((UInt8) *bufP != 0xFF)) {
        // there's a valid serial number!
        // Calculate the checksum:  Start with zero, add each
digit,
        // then rotate the result one bit to the left and
repeat.
        checksum = 0;
        for (count=0; count<bufLen; count++) {
            checksum += bufP[count];
            checksum = (checksum<<1) | ((checksum & 0x80) >>
7);
        }
        // Add the two hex digits (nibbles) together, +2
        // (range: 2 - 31 ==> 2-9, A-W)
        // By adding 2 to the result before converting to
ascii,
        // we eliminate the numbers 0 and 1, which can be
        // difficult to distinguish from the letters O and I.
        checksum = ((checksum>>4) & 0x0F) + (checksum & 0x0F) +
2;

        // draw the serial number and find out how wide it was
WinDrawChars(bufP, bufLen, x, y);
x += FntCharsWidth(bufP, bufLen);
}
```



```
// draw the dash and the checksum digit right after it
checksumStr[0] = '-';
checksumStr[1] =
    ((checkSum < 10) ? (checkSum + '0') : (checkSum - 10
+ 'A'));
WinDrawChars (checksumStr, 2, x, y);
}
else // there's no serial number
// draw a status message if the caller provided one
if (noNumberMessage)
    WinDrawChars (noNumberMessage,
StrLen(noNumberMessage), x, y);
}
```

Time

The Palm Powered handheld has a real-time clock and programmable timer as part of the 68328 processor. The real-time clock maintains the current time even when the system is in sleep mode (turned off). It's capable of generating an interrupt to wake the handheld when an alarm is set by the user. The programmable timer is used to generate the system tick count interrupts (100 times/second) while the processor is in doze or running mode. The system tick interrupts are required for periodic activity such as polling the digitizer for user input, key debouncing, etc.

The date and time manager (called time manager in this chapter) provides access to both the 1-second and 0.01-second timing resources on the Palm Powered handheld.

- The 1-second timer keeps track of the real-time clock (date and time), even when the unit is in sleep mode.
- The 0.01-second timer, also referred to as the **system ticks**, can be used for finer timing tasks. This timer is not updated when the unit is in sleep mode and is reset to 0 each time the unit resets.

The basic time-manager API provides support for setting and getting the real-time clock in seconds and for getting the current system ticks value (but not for setting it). The system manager provides more advanced functionality for setting up a timer task that executes periodically or in a given number of system ticks.

This section discusses the following topics:

- [Using Real-Time Clock Functions](#)
- [Using System Ticks Functions](#)

Using Real-Time Clock Functions

The real-time clock functions of the time manager include [TimSetSeconds](#) and [TimGetSeconds](#). Real time on the Palm Powered handheld is measured in seconds from midnight, Jan. 1, 1904. Call [TimSecondsToDateTime](#) and [TimDateTimeToSeconds](#) to convert between seconds and a structure specifying year, month, day, hour, minute, and second.

Using System Ticks Functions

The Palm Powered handheld maintains a tick count that starts at 0 when the handheld is reset. This tick increments

- 100 times per second when running on the Palm Powered handheld
- 60 times per second when running on the Macintosh under the Simulator

For tick-based timing purposes, applications should use the macro [SysTicksPerSecond](#), which is conditionally compiled for different platforms. Use the function [TimGetTicks](#) to read the current tick count.

Although the [TimGetTicks](#) function could be used in a loop to implement a delay, it is recommended that applications use the [SysTaskDelay](#) function instead. The [SysTaskDelay](#) function automatically puts the unit into low-power mode during the delay. Using [TimGetTicks](#) in a loop consumes much more current.

Floating-Point

The Palm OS supports IEEE-754 single and double precision floating-point numbers declared with the C types `float` and `double`. Numbers of type `float` occupy four bytes and have an effective range of 1.17549e-38 to 3.40282e+38. Numbers of type

`double` occupy eight bytes and have an effective range of 2.22507e-308 to 1.79769e+308.

You can use basic arithmetic operations to add, subtract, multiply, and divide numbers of type `float` and `double`. Higher-level functions such as those in the standard C header file `math.h` are not part of the core OS; you must either write them yourself, or you must employ a third-party math library.

The standard IEEE-754 special “non-number” values of NaN (not a number), +INF (positive infinity), and -INF (negative infinity) are generated as appropriate if you perform an operation that produces a result outside the range of numbers that can be represented. For instance, dividing a positive number by 0 returns +INF.

The Float Manager contains functions that convert double-precision floating-point numbers to and from a string using scientific notation. It also contains [FlpBufferCorrectedAdd](#) and [FlpBufferCorrectedSub](#), which perform the indicated operation and correct the result in those situations where the result should be zero but isn’t due to the way that floating-point numbers are represented. All of the Float Manager functions that either accept or return a floating-point number require it to be declared as an `FlpDouble`. The Float Manager defines a union, [FlpCompDouble](#), that you use to declare values that can be interpreted either as a `double` or as an `FlpDouble`. You use this union as shown here:

```
double dblFlpCorrectedAdd (double d1, double d2, Int16 acc) {
    FlpCompDouble fcd1, fcd2, fcdResult;

    fcd1.d = d1;
    fcd2.d = d2;
    FlpBufferCorrectedAdd(&fcdResult.fd, fcd1.fd, fcd2.fd,
        acc);
    return fcdResult.d;
}
```

NOTE: If you are using CodeWarrior, you have the option of using [FlpAToF](#), [FlpCorrectedAdd](#), and [FlpCorrectedSub](#) instead of [FlpBufferAToF](#), [FlpBufferCorrectedAdd](#), and [FlpBufferCorrectedSub](#). These “non-buffer” functions all return their results directly, rather than updating a value pointed to by the first parameter. Because they return an `FlpDouble`—which is a struct—and because the GCC compiler’s convention for returning structures from functions is incompatible with Palm OS, GCC users can only use the `FlpBuffer...` versions.

In the rare event that you need to work with the binary representation of a `double`, the Float Manager also contains a number of functions and macros that allow you to obtain and in some cases alter the sign, mantissa, and exponent of a 64-bit floating-point number. See [Chapter 32, “Float Manager,”](#) on page 665 of the *Palm OS Programmer’s API Reference* for the functions and macros that make up the Float Manager.

The Float Manager, which was introduced in Palm OS 2.0, is sometimes referred to as the New Float Manager to distinguish it from the Float Manager that was part of Palm OS 1.0. The 1.0 Float Manager, which is less accurate and less convenient to use (simple operations such as addition require a call to a Float Manager function), remains in the ROM solely for backward compatibility; its functions are no longer publicly declared in the Palm OS SDK and should no longer be used. The functions in the old Float Manager all begin with “Fpl” rather than the current “Flp”; see [Appendix C, “1.0 Float Manager,”](#) on page 2307 of the *Palm OS Programmer’s API Reference* for the functions that make up the original Float Manager.

Summary of System Features

Feature Manager Functions

FtrGet	FtrGetByIndex
FtrSet	FtrUnregister
FtrPtrNew	FtrPtrFree
FtrPtrResize	

Preferences Functions

PrefGetAppPreferences	PrefGetAppPreferencesV10
PrefSetAppPreferences	PrefSetAppPreferencesV10
PrefGetPreference	PrefSetPreference
PrefOpenPreferenceDB	PrefOpenPreferenceDBV10
PrefGetPreferences	PrefSetPreferences

Sound Manager Functions

SndCreateMidiList	SndDoCmd
SndGetDefaultVolume	SndInterruptSmflrregardless
SndPlaySmf	SndPlaySmflrregardless
SndPlaySmfResource	SndPlaySmfResourceIrregardless
SndPlaySystemSound	

System Manager Functions

System Dialogs

SysGraffitiReferenceDialog	SysKeyboardDialog
SysKeyboardDialogV10	

Power Management

SysBatteryInfo	SysBatteryInfoV20
SysSetAutoOffTime	SysTaskDelay

Palm System Support

Summary of System Features

System Manager Functions

System Management

[SysLibFind](#)

[SysRandom](#)

[SysGremlins](#)

[SysLibLoad](#)

[SysReset](#)

Working With Strings and Resources

[SysBinarySearch](#)

[SysQSort](#)

[SysCreatePanelList](#)

[SysFormPointerArrayToStrings](#)

[SysInsertionSort](#)

[SysCopyStringResource](#)

[SysStringByIndex](#)

Database Support

[SysCreateDataBaseList](#)

[SysCurAppDatabase](#)

Error Handling

[SysErrString](#)

Event Handling

[SysHandleEvent](#)

System Information

[SysGetOSVersionString](#)

[SysGetROMToken](#)

[SysGetStackInfo](#)

[SysTicksPerSecond](#)

Time Manager Functions

Allowing User to Change Date and Time

[DayDrawDays](#)

[DayDrawDaySelector](#)

[DayHandleEvent](#)

[SelectDay](#)

[SelectTimeV33](#)

[SelectDayV10](#)

Changing the Date

[DateAdjust](#)

[TimSetSeconds](#)

[TimAdjust](#)

Time Manager Functions

Converting to Date Format

[DateDaysToDate](#)

[DateSecondsToDate](#)

[TimSecondsToDateTime](#)

Converting Dates to Other Formats

[DateToAscii](#)

[TimeToAscii](#)

[DateToDays](#)

[DateToDOWDMFormat](#)

[TimGetSeconds](#)

[TimDateTimeToSeconds](#)

[TimGetTicks](#)

Date Information

[DayOfMonth](#)

[DayOfWeek](#)

[DaysInMonth](#)

Float Manager Functions

[FlpAToF](#)

[FlpBufferAToF](#)

[FlpBufferCorrectedAdd](#)

[FlpBufferCorrectedSub](#)

[FlpCorrectedAdd](#)

[FlpCorrectedSub](#)

[FlpFToA](#)

[FlpNegate](#)

[FlpSetNegative](#)

[FlpSetPositive](#)

Localized Applications

When you write an application, or any other type of software, you need to take special care when working with characters, strings, numbers, and dates, as different countries represent these items in different ways. This chapter describes how to write code that works properly for any language that is supported by Palm OS®. The chapter covers:

- [Localization Guidelines](#)
- [Using Overlays to Localize Resources](#)
- [Dates](#)
- [Numbers](#)
- [Obtaining Locale Information](#)
- [Notes on the Japanese Implementation](#)
- [Summary of Localization](#)

In addition to this chapter, also see [Chapter 8](#), “[Text](#),” on page 251, which describes how to work with text and characters in a way that makes your application easily localizable.

NOTE: PalmOSGlue provides backward compatibility for many of the functions described in this chapter. When a function has a PalmOSGlue equivalent, that equivalent is shown in parentheses following the function name. See “[Backward Compatibility with PalmOSGlue](#)” on page 14 for more information on PalmOSGlue.

This chapter does not cover how to actually perform localization of resources. For more information on this subject, see the *Palm OS Programming Development Tools Guide*.

Localization Guidelines

If there is a possibility that your application is going to be localized, you should follow these guidelines when you start planning the application. It's a good idea to follow these guidelines even if you don't think your application is going to be localized.

- If you use the English language version of the software as a guide when designing the layout of the screen, try to allow:
 - extra space for strings
 - larger dialogs than the English version requires
- Don't put language-dependent strings in code. If you have to display text directly on the screen, remember that a one-line warning or message in one language may need more than one line in another language. See the section "[Strings](#)" on page 258 in [Chapter 8](#), "[Text](#)," for further discussion.
- Don't depend on the physical characteristics of a string, such as the number of characters, the fact that it contains a particular substring, or any other attribute that might disappear in translation.
- Database names must use only 7-bit ASCII characters (0x20 through 0x7E). If an actual PDB name is displayed to the user, the application should have a way of associating a localizable name (resource based, if possible) with each database.
- Use the functions described in this chapter when working with characters, strings, numbers, and dates.
- Consider using string templates as described in the section "[Dynamically Creating String Content](#)" on page 265 in [Chapter 8](#). Use as many parameters as possible to give localizers greater flexibility. Avoid building sentences by concatenating substrings together, as this often causes translation problems.
- Abbreviations may be the best way to accommodate the particularly scarce screen real estate on the Palm Powered™ handheld.
- Remember that user interface elements such as lists, fields, and tips scroll if you need more space.

The book *Palm OS User Interface Guidelines* provides further user interface guidelines.

Using Overlays to Localize Resources

Palm OS version 3.5 adds support for localizing resource databases through **overlays**. Localization overlays provide a method for localizing a software module without requiring a recompile or modification of the software. Each overlay database is a separate resource database that provides an appropriately localized set of resources for a single software module (the PRC file, or **base database**) and a single target **locale** (language and country).

No requirements are placed on the base database, so for example, third parties can construct localization overlays for existing applications without forcing any modifications by the original application developer. In rare cases, you might want to disable the use of overlays to prevent third parties from creating overlays for your application. To do so, you should include an 'xprf'=0 resource (symbolically named sysResTextPrefs) in the database and set its disableOverlays flag. This resource is defined in `UIResources.r`.

An overlay database has the same creator as the base database, but its type is 'ovly', and a suffix identifying the target locale is appended to its name. For example, `Datebook.prc` might be overlaid with a database named `Datebook_jpJP`, which indicates that this overlay is for Japan. Each overlay database has an 'ovly'=1000 resource specifying the base database's type, the target locale, and information necessary to identify the correct version of the base database for which it was designed.

The Palm OS SDK provides tools that you can use to create overlays. See the "PRC to Overlay Tool" chapter in the *Palm OS Programming Development Tools Guide* for more information on creating overlays.

When a PRC file is opened on a system that supports overlays, the Overlay Manager determines what the current locale is for this handheld, and it looks for an overlay matching the base database and the locale. The overlay database's name must match the base database's name, its suffix must match the locale's suffix, and it must have an 'ovly'=1000 resource that matches the base

Localized Applications

Using Overlays to Localize Resources

database. If the name, suffix, and overlay resource are all correct, the overlay is opened in addition to the PRC file. When the PRC file is closed, its overlay is closed as well.

The overlay is opened in read-only mode and is hidden from the programmer. When you open a database, you'll receive a reference to the base database, not the overlay. You can simply make Resource Manager calls like you normally would, and the Resource Manager accesses the overlay where appropriate.

When accessing a localizable resource, do not use functions that search for a resource only in the database you specify. For example:

```
// WRONG! searches only one database.
DmOpenRef dbP = DmNextOpenResDatabase(NULL);
UInt16 resIndex = DmFindResource(dpP, strRsc,
    strRscID);
MemHandle resH = DmGetResourceIndex(dbP,
    resIndex);
```

In the example above, `dbP` is a reference to the most recently opened database, which is typically the overlay version of the database. Passing this reference to `DmFindDatabase` means that you are searching only the overlay database for the resource. If you're searching for a non-localized resource, `DmFindResource` won't be able to locate it. Instead, you should use `DmGet1Resource`, which searches the most recently opened database and its overlay for a resource, or `DmGetResource`, which searches all open databases and their overlays.

```
// Right. DmGet1Resource searches both
// databases.
MemHandle resH = DmGet1Resource(strRsc,
    strRscID);

// Or use DmGetResource to search all open
// databases.
MemHandle resH = DmGetResource(strRsc,
    strRscID);
```

The Data Manager only opens an overlay if the resource database is opened in read-only mode. If you open a resource database in read-write mode, the associated overlay is not opened. What's more, if you modify the an overlaid resource in the base database, the

checksum in the overlay’s ‘ovly’ resource becomes invalid, which prevents the overlay from being used at all. Thus if you change the resource database, you must also change the overlay database.

You typically don’t work with the Overlay Manager directly although it does provide a few public functions. One potentially useful function is [OmGetCurrentLocale](#) (or [OmGlueGetCurrentLocale](#)), which returns a structure identifying the locale on this handheld.

Dates

If your application deals with dates and times, it should abide by the values the user has set in the system preference for date and time display. The default preferences at startup are vary among locales, and the default values can be overridden by the user.

To check the system preferences call [PrefGetPreference](#) with one of the values listed in the second column of [Table 11.1](#). The third column lists an enumerated type that helps you interpret the value.

Table 11.1 Date and time preferences

Preference	Name	Returns a value of type
Date formats (i.e., month first or day first)	prefDateFormat, prefLongDateFormat	DateFormatType
Time formats (i.e., use a 12-hour clock or use a 24-hour clock)	prefTimeFormat	TimeFormatType
Start day of week (i.e., Sunday or Monday)	prefWeekStartDay	0 (Sunday) or 1 (Monday)

Localized Applications

Numbers

Table 11.1 Date and time preferences (*continued*)

Preference	Name	Returns a value of type
Local time zone	<code>prefMinutesWestOfGMT</code> (before Palm OS 4.0), <code>prefTimeZone</code> (Palm OS 4.0 and higher)	Minutes east of Greenwich Mean Time (GMT), also known as Universal Coordinated Time (UTC).
Daylight savings time adjustment	<code>prefDaylightSavings</code> (before Palm OS 4.0), <code>prefDaylightSavingAdjustment</code> (Palm OS 4.0 and higher)	Before 4.0, the DaylightSavingsTypes described the daylight savings adjustment. In Palm OS 4.0 and higher, the preference is stored as the number of minutes by which to adjust the current time.

IMPORTANT: The `prefMinutesWestOfGMT` preference mentioned above is not the same as the `prefTimeZone` preference. The `prefMinutesWestOfGMT` returns an unsigned value ranging from 0 to 1440. The `prefTimeZone` preference ranges from -720 to 720.

To work with dates in your code, use the Date and Time Manager API. It contains functions such as [DateToAscii](#), [DayOfMonth](#), [DayOfWeek](#), [DaysInMonth](#), and [DateTemplateToAscii](#), which allow you to work with dates independent of the user's preference settings.

Numbers

If your application displays large numbers or floating-point numbers, you must check and make sure you are using the

appropriate thousands separator and decimal separator for the handheld's country by doing the following (see [Listing 11.1](#)):

1. Store numbers using US conventions, which means using a "," as the thousands separator and a decimal point (.) as the decimal separator.
2. Use `PrefGetPreference` and [`LocGetNumberSeparators`](#) to retrieve information about how the number should be displayed.
3. Use [`StrLocalizeNumber`](#) to perform the localization.
4. If a user enters a number that you need to manipulate in some way, convert it to the US conventions using [`StrDelocalizeNumber`](#).

Listing 11.1 Working with numbers

```
// store numbers using US conventions.
Char *jackpot = "20,000,000.00";
Char thou; // thousand separator
Char dp; // decimal separator

// Retrieve user's preferred number format.
LocGetNumberSeparators((NumberFormatType)
    PrefGetPreference(prefNumberFormat), &thou,
    &dp);
// Localize jackpot number. Converts "," to thou
// and "." to dp.
StrLocalizeNumber(jackpot, thou, dp);
// Display string.
// Assume inputString is a number user entered,
// convert it to US conventions this way. Converts
// thou to "," and dp to "."
StrDelocalizeNumber(inputNumber, thou, dp);
```

Obtaining Locale Information

Some applications may require information about the current locale. For example, many applications need to know the format for displaying dates or numbers, which is determined in part by the current locale (and described in more detail in the section "[Dates](#)" and "[Numbers](#)" in this chapter). Other applications may need other information, such as the country name.

Localized Applications

Obtaining Locale Information

The information that most applications require is stored in the system preferences structure and can be obtained using [PrefGetPreference](#). This is the recommended way of obtaining locale-specific settings because the user can override many of these settings. Applications should always honor the user's preferences rather than the locale defaults.

Other locale-specific settings can not be set by the user and are not stored in the system preferences. Instead, these settings are stored in a private resource that contains information about several possible locales, including the locale currently used by the system. For example, the user cannot change the symbol used for the local currency. If your application needs this information, it must use the Locale Manager function [LmGetLocaleSetting](#) to retrieve it. The Locale Manager is new in Palm OS 4.0, but for backwards compatibility you can use the corresponding PalmOSGlue function [LmGlueGetLocaleSetting](#). [Listing 11.2](#) shows how to use [LmGlueGetLocaleSetting](#).

Listing 11.2 Retrieving a locale setting using Locale Manager

```
LmLocaleType locale;
Char currencySymbol[kMaxCurrencySymbolLen+1];
UInt16 index;

// Find out what the current locale is.
OmGlueGetCurrentLocale(&locale);

// Find out which index in the locale resource
// contains info about that locale.
LmGlueLocaleToIndex(&locale, &index);

// Get the currency symbol stored in the locale at
// that index.
LmGlueGetLocaleSetting(index, lmChoiceCurrencySymbol,
    currencySymbol, sizeof(currencySymbol));
```

[Table 11.2](#) shows which types of information about the current locale should be retrieved from the system preferences and which types should be retrieved from the locale resource. Of course, if you want to retrieve information about a different locale or if you want to look up the default used for the current locale, you would always use the Locale Manager instead of the Preferences Manager.

Table 11.2 Obtaining locale information

Value	Function used to retrieve value
Language code	<code>PrefGetPreference(prefLanguage)</code>
Locale description	<code>PrefGetPreference(prefLocale)</code>
Country code	<code>PrefGetPreference(prefCountry)</code>
Country name	<code>LmGlueGetLocaleSetting(..., lmChoiceCountryName, ...)</code>
Currency name	<code>LmGlueGetLocaleSetting(..., lmChoiceCurrencyName, ...)</code>
Currency symbol	<code>LmGlueGetLocaleSetting(..., lmChoiceCurrencySymbol, ...)</code>
Unique currency symbol	<code>LmGlueGetLocaleSetting(..., lmChoiceUniqueCurrencySymbol, ...)</code>
Measurement system (metric or English)	<code>PrefGetPreference(prefMeasurementSystem)</code>
Number formats	<code>PrefGetPreference(prefNumberFormat)</code>
Number of decimal places for monetary values	<code>LmGlueGetLocaleSetting(..., lmChoiceCurrencyDecimalPlaces, ...)</code>
Starting day of the week	<code>PrefGetPreference(prefWeekStartDay)</code>
Date formats	<code>PrefGetPreference(prefDateFormat)</code> <code>PrefGetPreference(prefLongDateFormat)</code>
Time format	<code>PrefGetPreference(prefTimeFormat)</code>

Localized Applications

Notes on the Japanese Implementation

Table 11.2 Obtaining locale information (*continued*)

Value	Function used to retrieve value
Time zone	PrefGetPreference(prefMinutesWestOfGMT) (pre 4.0) PrefGetPreference(prefTimeZone) (4.0 and higher)
Daylight savings time	PrefGetPreference(prefDaylightSavings) (pre 4.0) PrefGetPreference(prefDaylightSavingAdjustment) (4.0 and higher)

Notes on the Japanese Implementation

This section describes programming practices for applications that are to be localized for Japanese use. It covers:

- [Japanese Character Encoding](#)
- [Japanese Character Input](#)
- [The Calculator Button](#)
- [Displaying Japanese Strings on UI Objects](#)
- [Displaying Error Messages](#)

Japanese Character Encoding

The character encoding used on Japanese systems is based on Microsoft code page 932. The complete 932 character set (JIS level 1 and 2) is supported in both the standard and large font sizes. The bold versions of these two fonts contain bolded versions of the glyphs found in the 7-bit ASCII range, but on some handhelds, the single-byte Katakana characters and the multi-byte characters are not bolded.

Japanese Character Input

On current Japanese handhelds, users enter Japanese text using Latin (ASCII) characters, and special software called a front-end processor (FEP) transliterates this text into Hiragana or Katakana characters. The user can then ask the FEP to phonetically convert Hiragana characters into a mixture of Hiragana and Kanji (Kana-Kanji conversion).

Four input area icons added to the Japanese handheld control the FEP transliteration and conversion process. These four FEP buttons are arranged vertically between the current left-most icons and the input area. The top-most FEP button tells the FEP to attempt Kana-Kanji conversion on the inline text. The next button confirms the inline text and terminates the inline conversion session. The third button toggles the transliteration mode between Hiragana and Katakana. The last button toggles the FEP on and off.

Japanese text entry is always inline, which means that transliteration and conversion happen directly inside of a field. The field code passes events to the FEP, which then returns information about the appropriate text to display.

During inline conversion, the Graffiti® or Graffiti 2 space stroke acts as a shortcut for the conversion FEP button and the return stroke acts as a shortcut for the confirm FEP button.

The Calculator Button

On current Japanese handhelds, the Calculator silkscreen button doesn't generate a `calcChr`. Instead, it generates a `keyDown` event with the event's `data.keyDown.chr` field set to `keyboardChr` and its `data.keyDown.modifiers` field set to `commandKeyMask`.

Displaying Japanese Strings on UI Objects

To conserve screen space, you should use half-width Katakana characters on user interface elements (such as buttons, menu items, labels, and pop-up lists) whenever the string contains only Katakana characters. If the string contains a mix of Katakana and either Hiragana, Kanji, or Romaji, then use the full-width Katakana characters instead.

Displaying Error Messages

You may have code that uses the macros [`ErrFatalDisplayIf`](#) and [`ErrNonFatalDisplayIf`](#) to determine error conditions. If the error condition occurs, the system displays the file name and line number at which the error occurred along with the message that you passed to the macro. Often these messages are hard-coded

Localized Applications

Summary of Localization

strings. On Japanese systems, the Palm OS traps the messages passed to these two macros and displays a generic message explaining that an error has occurred.

You should only use `ErrFatalDisplayIf` and `ErrNonFatalDisplayIf` for totally unexpected errors. Do not use them for errors that you believe your end users will see. If you wish to inform your users of an error, use a localizable resource to display the error message instead of `ErrFatalDisplayIf` or `ErrNonFatalDisplayIf`.

Summary of Localization

Localizing Numbers

[StrLocalizeNumber](#)

[StrDelocalizeNumber](#)

[LocGetNumberSeparators](#)

Locale Manager

[LmGetLocaleSetting](#)

[LmGetNumLocales](#)

[LmLocaleToIndex](#)

International Manager

[IntlGetRoutineAddress](#)

[IntlSetRoutineAddress](#)

Overlay Manager

[OmGetCurrentLocale](#)

[OmGetSystemLocale](#)

[OmGetIndexedLocale](#)

[OmLocaleToOverlayDBName](#)

[OmGetRoutineAddress](#)

[OmOverlayDBNameToLocale](#)

[OmSetSystemLocale](#)

[OmGetNextSystemLocale](#)

Debugging Strategies

You can use a Palm OS® system manager called the error manager to display unexpected runtime errors such as those that typically show up during program development. Final versions of applications or system software won't use the error manager.

The error manager API consists of a set of functions for displaying an alert with an error message, file name, and the line number where the error occurred. If a debugger is connected, it is entered when the error occurs.

The error manager also provides a “try and catch” mechanism that applications can use for handling such runtime errors as out of memory conditions, user input errors, etc.

This section helps you understand and use the error manager, discussing the following topics:

- [Displaying Development Errors](#)
- [Using the Error Manager Macros](#)
- [The Try-and-Catch Mechanism](#)
- [Summary of Debugging API](#)

This chapter only describes programmatic debugging strategies; to learn how to use the available tools to debug your application, see the book *Palm OS Programming Development Tools Guide*.

Displaying Development Errors

The error manager provides some compiler macros that can be used in source code. These macros display a fatal alert dialog on the screen and provide buttons to reset the handheld or enter the debugger after the error is displayed. There are three macros: [ErrDisplay](#), [ErrFatalDisplayIf](#), and [ErrNonFatalDisplayIf](#).

- `ErrDisplay` always displays the error message on the screen.
- `ErrFatalDisplayIf` and `ErrNonFatalDisplayIf` display the error message only if their first argument is `TRUE`.

The error manager uses the compiler define `ERROR_CHECK_LEVEL` to control the level of error messages displayed. You can set the value of the compiler define to control which level of error checking and display is compiled into the application. Three levels of error checking are supported: none, partial, and full.

If you set <code>ERROR_CHECK_LEVEL</code> to...	The compiler...
<code>ERROR_CHECK_NONE</code> (0)	Doesn't compile in any error calls.
<code>ERROR_CHECK_PARTIAL</code> (1)	Compiles in only <code>ErrDisplay</code> and <code>ErrFatalDisplayIf</code> calls.
<code>ERROR_CHECK_FULL</code> (2)	Compiles in all three calls.

During development, it makes sense to set full error checking for early development, partial error checking during alpha and beta test periods, and no error checking for the final product. At partial error checking, only fatal errors are displayed; error conditions that are only possible are ignored under the assumption that the application developer is already aware of the condition and designed the software to operate that way.

Using the Error Manager Macros

Calls to the error manager to display errors are actually compiler macros that are conditionally compiled into your program. Most of the calls take a boolean parameter, which should be set to `true` to display the error, and a pointer to a text message to display if the condition is true.

Typically, the boolean parameter is an in-line expression that evaluates to `true` if there is an error condition. As a result, both the expression that evaluates the error condition and the message text are left out of the compiled code when error checking is turned off. You can call [ErrFatalDisplayIf](#), or [ErrDisplay](#), but using [ErrFatalDisplayIf](#) makes your source code look neater.

For example, assume your source code looks like this:

```
result = DoSomething();
ErrFatalDisplayIf (result < 0,
    "unexpected result from DoSomething");
```

With error checking turned on, this code displays an error alert dialog if the result from `DoSomething()` is less than 0. Besides the error message itself, this alert also shows the file name and line number of the source code that called the error manager. With error checking turned off, both the expression evaluation `err < 0` and the error message text are left out of the compiled code.

The same net result can be achieved by the following code:

```
result = DoSomething();
#if ERROR_CHECK_LEVEL != ERROR_CHECK_NONE
if (result < 0)
    ErrDisplay ("unexpected result from
DoSomething");
#endif
```

However, this solution is longer and requires more work than simply calling [ErrFatalDisplayIf](#). It also makes the source code harder to follow.

The Try-and-Catch Mechanism

The error manager is aware of the machine state of the Palm Powered™ handheld and can therefore correctly save and restore this state. The built-in try and catch of the compiler can't be used because it's machine dependent.

Try and catch is basically a neater way of implementing a goto if an error occurs. A typical way of handling errors in the middle of a routine is to go to the end of the routine as soon as an error occurs and have some general-purpose cleanup code at the end of every routine. Errors in nested routines are even trickier because the result code from every subroutine call must be checked before continuing.

When you set up a try/catch, you are providing the compiler with a place to jump to when an error occurs. You can go to that error handling routine at any time by calling [ErrThrow](#). When the compiler sees the ErrThrow call, it performs a goto to your error handling code. The greatest advantage to calling ErrThrow, however, is for handling errors in nested subroutine calls.

Even if ErrThrow is called from a nested subroutine, execution immediately goes to the same error handling code in the higher-level call. The compiler and runtime environment automatically strip off the stack frames that were pushed onto the stack during the nesting process and go to the error handling section of the higher-level call. You no longer have to check for result codes after calling every subroutine; this greatly simplifies your source code and reduces its size.

Using the Try and Catch Mechanism

The following example illustrates the possible layout for a typical routine using the error manager's try and catch mechanism.

Listing 12.1 Try and Catch Mechanism Example

```
ErrTry {  
    p = MemPtrNew(1000);  
    if (!p) ErrThrow(errNoMemory);  
    MemSet(p, 1000, 0);  
    CreateTable(p);  
    PrintTable(p);  
}
```



```
    }

    ErrCatch(err) {
        // Recover or cleanup after a failure in the
        // above Try block. "err" is an int
        // identifying the reason for the failure.

        // You may call ErrThrow() if you want to
        // jump out to the next Catch block.

        // The code in this Catch block doesn't
        // execute if the above Try block completes
        // without a Throw.

        if (err == errNoMemory)
            ErrDisplay("Out of Memory");
        else
            ErrDisplay("Some other error");
    } ErrEndCatch
    // You must structure your code exactly as
    // above. You can't have an ErrTry without an
    // ErrCatch { } ErrEndCatch, or vice versa.
```

Any call to [ErrThrow](#) within the [ErrTry](#) block results in control passing immediately to the [ErrCatch](#) block. Even if the subroutine CreateTable called ErrThrow, control would pass directly to the ErrCatch block. If the ErrTry block completes without calling ErrThrow, the ErrCatch block is not executed.

You can nest multiple ErrTry blocks. For example, if you wanted to perform some cleanup at the end of CreateTable in case of error,

- Put ErrTry/ErrCatch blocks in CreateTable
- Clean up in the ErrCatch block first
- Call [ErrThrow](#) to jump to the top-level ErrCatch

Summary of Debugging API

Error Manager Functions

Displaying Errors

[ErrAlert](#)

[ErrDisplayFileLineMsg](#)

[ErrNonFatalDisplayIf](#)

[ErrDisplay](#)

[ErrFatalDisplayIf](#)

Catching Exceptions

[ErrCatch](#)

[ErrExceptionList](#)

[ErrTry](#)

[ErrEndCatch](#)

[ErrThrow](#)

Standard IO Applications

The Palm OS® supports command line (UNIX style) applications for debugging and special purposes such as communications utilities. This capability is not intended for general users, but for developers. This feature is not implemented in the Palm OS, but rather by additional C modules that you must link with your application.

NOTE: Don't confuse this standard IO functionality with the file streaming API. They are unrelated.

There are two parts necessary for a standard IO application:

- The standard IO application itself.

A standard IO application is not like a normal Palm™ application. It is executed by a command line and has minimal user interface. It can take character input from the stdin device (the keyboard) and write character output to the stdout window.

- The standard IO provider application.

A standard IO provider application is necessary to execute and see output from a standard IO application. The standard IO provider application is a normal Palm application that provides a field in which you can enter commands to execute standard IO applications. The field also serves as a stdout window where output from the executing application is written.

The details of creating these two different applications are described in the following sections.

Creating a Standard IO Application

To create a standard IO application, you must include the header file `StdIOPalm.h`. In addition to including this header, you must link the application with the module `StdIOPalm.c`. This module provides a `PilotMain` routine that extracts the command line arguments from the `cmd` and `cmdPBP` parameters and the glue code necessary for executing the appropriate callbacks supplied by the standard IO provider application.

You build the application normally, but give it a database type of `sioDbType('sdio')` instead of `'appl'`. In addition, it must be named `"Cmd-cmdname"` where `cmdname` is the name of the command used to execute the application. For example, the ping command would be placed in a database named `"Cmd-ping"`.

In the Palm VII™ handheld, the Network panel, whose log window is a standard IO provider application, has two standard IO commands built-in: `info` and `finger`. The ROM has two additional ones: `ping` and `nettrace`.

When compiling for the Palm Powered™ handheld, the entry point must be named `SioMain` and must accept two parameters: `argc` and `argv`. Here's the simplest possible example of a standard IO application.

```
#include <StdIOPalm.h>
Int16 SioMain(UInt16 argc, Char* argv[ ])
{
    printf("Hello World\n");
}
```

Standard IO applications can use several input and output functions that mimic their similarly named UNIX counterparts. These are listed in the [summary table](#) at the end of this chapter.

Your standard IO application can accept input from `stdin` and write output to `stdout`. The `stdin` device corresponds to the text field in the standard IO provider application that is used for input and output. The `stdout` device corresponds to that same text field.

Creating a Standard IO Provider Application

In order for a standard IO application to be invoked and able to provide results, you need a standard IO provider application. This application provides the user interface support; that is, the stdin device support and the stdout window that the standard IO application reads from and writes to.

The standard IO provider launches the standard IO application when the user types in a command line and Return (using Graffiti® or Graffiti 2 writing). The provider application passes a structure pointer that contains the callbacks necessary for performing IO to the standard IO application through the `cmdPBP` parameter of `PilotMain`.

To create a standard IO provider application, you must link the application with the module `StdIOProvider.c`.

To handle input and output, the standard IO provider application must provide a form with a text field and a scroll bar. The standard IO provider application must do the following:

1. Call [SioInit](#) during application initialization. `SioInit` saves the object ID of the form that contains the input/output field, the field itself, and the scroll bar.
2. Call [SioHandleEvent](#) from the form's event handler before doing application specific processing of the event. In other words, the form event handler that the application installs with `FrmSetEventHandler` should call `SioHandleEvent` before it does anything else with the event.
3. Call [SioFree](#) during application shutdown.

The application is free to call any of the standard IO macros and functions between the `SioInit` and `SioFree` calls. If the current form is not the standard IO form when these calls are made, they will record changes to the active text and display it the next time the form becomes active.

A typical standard IO provider application will have a routine called `ApplicationHandleEvent`, which gets called from its main event loop after `SysHandleEvent` and `MenuHandleEvent`. An example is shown in [Listing 13.1](#).

Standard IO Applications

Creating a Standard IO Provider Application

Listing 13.1 Standard IO Provider ApplicationHandleEvent Routine

```
static Boolean ApplicationHandleEvent (EventPtr event)
{
    FormType* frm;
    UInt16 formId;

    if (event->eType == frmLoadEvent) {
        formId = event->data.frmLoad.formID;
        frm = FrmInitForm (formId);
        FrmSetActiveForm (frm);

        switch (formId) {
            .....
            case myViewWithStdIO:
                FrmSetEventHandler (frm, MyViewHandleEvent);
                break;
        }
        return (true);
    }

    return (false);
}
```

A typical application form event handler is shown in [Listing 13.2](#).

Listing 13.2 Standard IO Provider Form Event Handler

```
static Boolean MyViewHandleEvent (EventPtr event)
{
    FormType* frm;
    Boolean handled = false;

    // Let StdIO handler do its thing first.
    if (SioHandleEvent(event)) return true;

    // If StdIO did not completely handle the event...
    if (event->eType == ctlSelectEvent) {
        switch (event->data.ctlSelect.controlID) {
            case myViewDoneButtonID:
                FrmGotoForm (networkFormID);
                handled = true;
                break;
        }
    }
}
```

```
    else if (event->eType == menuEvent)
        return MyMenuDoCommand( event->data.menu.itemID );

    else if (event->eType == frmUpdateEvent) {
        MyViewDraw( FrmGetActiveForm() );
        handled = true;
    }

    else if (event->eType == frmOpenEvent) {
        frm = FrmGetActiveForm();
        MyViewInit( frm );
        MyViewDraw( frm );
        handled = true;
    }

    else if (event->eType == frmCloseEvent) {
        frm = FrmGetActiveForm();
        MyViewClose(frm);
    }

    return (handled);
}
```

Summary of Standard IO

Standard IO Macros and Functions

fgetc	Siofgets
fgets	Siofprintf
fprintf	Siofputc
fputc	Siofputs
fputs	Siogets
getchar	Sioprintf
gets	Sioputs
printf	Siosystem
putc	Siovfprintf
putchar	sprintf
puts	system
SioAddCommand	vfprintf
Siofgetc	vsprintf

Application-Defined Functions

[SioMain](#)

Standard IO Provider Functions

SioClearScreen	SioHandleEvent
SioExecCommand	SioInit
SioFree	

Index

Numerics

0.01-second timer 355
1.0 heaps 183
16-bit color 145
1-second timer 355
2.0 heaps 183
3.0 heaps 183
32K jumps 184
68328 processor 171

A

Alarm Manager 306–313
 alarm sound 307
 and Attention Manager 290, 301
 and sleep mode 311
 procedure alarms 302, 311
 reminder dialog boxes 306
alarms 8
 and expansion 217
 delayed in sleep mode 307
 displaying 301
 interaction with menus 313
 multiple 284
 playing a sound 307
 setting 307
alert manager 85
alerts, system-defined 85
allocating handles 184
AlmGetAlarm 308
AlmGetProcAlarm 311
AlmSetAlarm 307, 308
AlmSetProcAlarm 311
ANSI C libraries 6
API naming conventions 6
appInfoStringsRsc 117
AppInfoType 117
appl database 11
application design
 assigning version number 197
 handling system messages 8
 removing deleted records 197
 using lists 112
application icon 154
 name 154

 size 8
application launcher 20
 and expansion 215
application name 154
 on expansion cards 215, 235
application preferences database 8
application record database 8
application startup 19–50
application-defined features 318
applications
 control flow 5
 event driven 5
 running from a card 235
architecture
 expansion 210
architecture of memory 171
ARM-native code
 calling 339
attention indicator 285, 288, 289
 enabling and disabling 305
Attention Manager 283–305
 and Alarm Manager 290, 301
 and existing applications 290, 302
 and hard and soft buttons 288
 and HotSync 300, 303
 and procedure alarms 311
 and SMS application 293
 attention indicator 285, 288, 289, 305
 callback function 291, 293
 deleting items 303
 detail dialog 286
 dismissing items 287
 displaying 289, 306
 displaying alarms 301
 enumerating items 300
 first time flag 295
 getting the user's attention 291
 go to item 288
 insistent items 285, 286, 303
 invalid items 299
 launch codes 293
 list dialog 287
 multiple items 290, 300
 operation 285
 periodically updating 302
 redrawing items 304

- sleep mode operation 305
- snoozing items 287, 301
- sounds 298
- special effects 298
- subtle items 285, 288, 303
- triggering a custom effect 298
- updating items 302, 303, 311
- validating items 300
- Attention Manager commands 294, 298
 - draw detail 295
 - draw list 296
 - go there 299
 - got it 300
 - iterate 300
 - play sound 298
 - snooze 301
 - trigger special effects 301
- Attention Manager dialogs
 - detail dialog 286
 - drawing 294
 - drawing selected item 296
 - list dialog 287
 - text and background colors 297
- AttnDoSpecialEffects 301
- AttnForgetIt 298, 299, 300, 303
- AttnGetAttention 291, 298, 301
- AttnGetCounts 302
- AttnIterate 300
- AttnListOpen 306
- AttnUpdate 303, 311
- auto-off 352
 - timer 65
- auto-repeat 65

B

- back-up of data to PC 170
- BarBeamBitmap 111
- BarCopyBitmap 111
- BarCutBitmap 111
- BarDeleteBitmap 111
- BarInfoBitmap 111
- BarPasteBitmap 111
- BarSecureBitmap 111
- BarUndoBitmap 111
- battery 352

- conservation using modes 351
 - life, maximizing 351
- bitmap family 130
- BitmapFlagsType 150
- bitmapped font 268
- bitmaps 123
 - bitmap family 130
 - masking 124
 - transparent 124
- BitmapType 130, 150
- bits behind menu bar 106
- blueBits 150
- BmpCreate 136
- boldFont 269
- booting 338
- button objects 92
- Button resource 70
 - highlighting 92

C

- Calculator Button 373
- calibrating digitizer 61
- card insertion and removal 217
- CardInfo application 229, 243
- cards
 - expansion 209
- carriage returns 104
- categories
 - maximum number 9
- CategoryGetName 119
- categoryHideEditCategory 122
- CategoryInitialize 117
- CategorySetTriggerLabel 119
- Char 253
- character encoding 252
- character set 252
- CharAttr.h 256
- Chars.h 254
- check box object 97
- Checkbox 71
- checking menu visibility 107
- chunks 179
 - resizing 182
 - size 182

- clock, real-time 355
- CodeWarrior IDE 15
- color translation table 146
- colorTableRsc 145
- command line applications 381
- command toolbar 109
- CompactFlash 208
- conduit 3
- conserving battery using modes 351
- Constructor 16
- control flow 5
- control objects 92
- conventions for API naming 6
- CoreTraps.h 13
- creating a chunk 182
- creating database 194
- creating resources 201
- creator ID 11
- ctlEnterEvent 93, 94, 95, 96, 97, 98, 99, 101
- ctlExitEvent 96, 101
- CtlGlueSetFont 273
- CtlHandleEvent 92
- CtlNewControl 143
- ctlRepeatEvent 96, 101
- ctlSelectEvent 94, 95, 97, 98, 101, 120
- custom UI element 140

D

- Data Manager
 - and the VFS Manager 212
- data manager
 - using 194
- database headers 191
 - fields 191
- database ID
 - and launch codes 28
- database version number 197
- databases 4, 174, 190
 - getting and setting information 194
 - on expansion cards 234
 - overlays 365
- date and time manager 355
- debugging with expansion cards 247
- default directories 215

- determining by file type 242
- for SD slot driver 245
- registered upon initialization 244
- registering new 243
- defaultCategoryRscType 154
- deleted records 8, 197
- deleting database 194
- deleting records 197
- dialog boxes
 - Attention Manager 286, 294
 - reminder 306
- dialogs 185
- digitizer 58
 - after reset 339
 - and pen manager 61
 - and pen queue 63
 - calibrating 61
 - dimensions 61
 - pen stroke to key event 63
 - polling 355
 - sampling accuracy 61
- Direct color 145
- direct color bitmaps 150
- direct color functions 149
- directColor 150
- directories
 - basic operations 240
 - default for file type 242
 - enumerating files within 241
- dmCategoryLength 118
- DmCreateDatabase 194, 199
- DmDatabaseInfo 194, 197, 199
- DmDatabaseSize 194
- DmDeleteDatabase 194, 199
- DmDeleteRecord 197
- DmFindDatabase 194
- DmFindResource 366
- DmGet1Resource 366
- DmGetDatabase 194
- DmGetRecord 194
- DmGetResource 366
- DmGetResourceIndex 366
- DmNewHandle 118
- DmNewResource 201
- DmNextOpenResDatabase 366

- DmQueryRecord 194
- dmRecAttrCategoryMask 120
- DmRecordInfo 120
- DmReleaseRecord 194
- DmReleaseResource 200
- DmRemoveRecord 197
- DmResizeRecord 195
- DmSetDatabaseInfo 194, 197
- DmWrite 319
- down arrow 105
- doze mode 350
- draw state 73
- draw window 75
- drawing state 73
- drivers, restarting 338
- dynamic heap
 - soft reset 338
- dynamic memory 184
- dynamic menus 108
- dynamic RAM 171

E

- edit-in-place 185
- ErrDisplay 376, 377
- ErrFatalDisplayIf 373, 376, 377
- ErrNonFatalDisplayIf 373
- error manager 375–380
 - try-and-catch mechanism 378
- ERROR_CHECK_LEVEL 376, 377
- ErrThrow 378
- event loop 53–56
 - example 53
 - example program 9
- event-driven applications 5
- events
 - naming conventions 6
 - overview 51–67
- EvtGetEvent 86, 351
- EvtResetAutoOffTimer 65
- examples
 - event loop 53
 - startup routine 23
 - stop routine 29
- expansion 207–248
 - and legacy applications 217
 - and Palm databases 234
 - and security 223
 - and the launcher 215
 - applications on cards 216
 - architecture 210
 - auto-start PRC 214, 235
 - card-launched applications 216
 - custom calls 246
 - custom I/O 247
 - debugging 247
 - default directories 215
 - enumerating slots 225
 - file system operations 231
 - file systems 211
 - lifetime of card-launched applications 217
 - mounted volumes 224
 - naming apps on expansion cards 215, 235
 - notifications 217, 218
 - ROM 209
 - slot driver 211
 - slot reference number 211
 - standard directories 214
 - standard directory layout 214
 - start.prc 222
 - volume operations 227
- expansion cards 209
 - capabilities 226
 - checking for 223
 - in slots 226
 - insertion and removal 217
 - reading and writing 236
- Expansion Manager 209, 213
 - checking card capabilities 226
 - custom I/O 247
 - enumerating slots 225
 - functions 214
 - overriding notification handlers 219
 - purpose 213
 - registered notifications 218
 - slot reference number 211, 220
 - verifying presence of 223
- expansion slot 209
- ExpCardInfo 227
- ExpCardPresent 226
- ExpSlotEnumerate 225
- extended font resource 271, 276, 278

F

FAT 212

feature manager 315–320

feature memory 319

features

- application-defined 318

- feature memory 319

- system version 316

feedback slider 99

Field 72

field objects 103

- events 104

- line feeds vs. carriage returns 104

file streaming

- and the VFS Manager 212

file streaming functions 205

file systems 211

- and filenames 233

- and volume names 230

- and volumes 227

- basic operations 231

- custom calls to 246

- FAT 212

- implementation 211

- long filenames 212

- multiple 211

- nonstandard functionality 246

- VFAT 211, 234

filenames

- long 212

files

- enumerating 241

- naming 212, 230, 233

- paths to 233

- reading and writing 236

- referencing 233

finding database 194

FindStrInStr 263

flags, launch flags 20

fldEnterEvent 104

FldHandleEvent 104

FldNewField 143

FldSetFont 273

fntAppFontCustomBase 277

FntAverageCharWidth 275

FntBaseLine 275

FntCharHeight 275

FntDefineFont 277

FntDescenderHeight 275

FntGlueGetDefaultFontID 271

FntLineHeight 275

FntSetFont 273, 275

FntWCharWidth 275

FntWidthToOffset 261, 275

font family resource 276

font ID 276

fontExtRscType 278

fonts 268

- characteristics 274

- custom 275

- extended font resource 271

- high-density displays 271

FontSelect 271

Form Bitmap 123

form objects 83

- event flow 84

formatting volumes 228

formGadgetDeleteCmd 142

formGadgetDrawCmd 142

formGadgetEraseCmd 142

formGadgetHandleEventCmd 142

FormGadgetHandler 140

forms 5

FrmAlert 85

FrmCustomAlert 85

FrmDoDialog 185

frmGadgetEnterEvent 142

frmGadgetMiscEvent 142

FrmGlueSetLabelFont 273

FrmGotoForm 185

FrmNewBitmap 143

FrmNewForm 143

FrmNewGadget 143

FrmNewLabel 143

frmOpenEvent 84, 95

FrmPopupForm 185

FrmRemoveObject 143

FrmSetGadgetHandler 140

FrmSetMenu 108

FrmValidatePtr 143
FtrGet 257, 318, 319
FtrPtrNew 319
FtrSet 318
FtrUnregister 318
function naming conventions 6

G

gadget resource 140
global find 8
 and private records 8
global variables 185
 erasing 338
glyphs 268
Graffiti
 customizing behavior 58
 Help 60
 Help character 60
Graffiti 2
 customizing behavior 58
 Help 60
 Help characters 60
Graffiti 2 recognizer 62
Graffiti manager 58
Graffiti recognizer 62
Graffiti shortcut 109
Graffiti ShortCuts database 59
graffitiReferenceChr 60
greenBits 150
GrfProcessStroke 58, 59

H

handles, allocation 184
hard reset 338, 339
hardware button presses and key manager 60
hasTransparency 151
heap fragmentation 184
heap header 179
heap space 184
heaps
 and soft reset 175
 in Palm OS 1.0 183
 in Palm OS 2.0 183
 in Palm OS 3.0 183

 overview 175
 RAM and ROM based 169
 structure 179
HelperNotifyEventType 38, 41
HelperServiceClass.h 38
highlighting button objects 92
HotSync 197, 209, 300, 303

I

icons, application 154
ID
 local 177
 See also creator ID
IDE 15
initialization
 global variables 23
input devices 3
insertion point object 153
inter-character boundary 259
International Manager 252
international manager 363
interrupting Sync application 353
invalid character 255

K

kAttnCommandCustomEffect 298
kAttnCommandDrawDetail 295
kAttnCommandDrawList 296
kAttnCommandGoThere 299
kAttnCommandGotIt 300
kAttnCommandIterate 300
kAttnCommandPlaySound 298
kAttnCommandSnooze 301
kernel 352
key events
 from pen strokes 62
key manager 60
key queue 64
KeyCurrentState 61
keyDownEvent 36, 61, 104, 105, 156, 254
KeyRates 61
kHelperNotifyActionCodeEnumerate 38
kHelperNotifyActionCodeExecute 38
kHelperNotifyActionCodeValidate 38

L

- label resource 137
- largeBoldFont 269
- largeFont 269
- launch codes 5, 20–48
 - and returned database ID 28
 - code example 21
 - creating 28
 - handling 7
 - launch flags 20
 - parameter blocks 20
 - predefined 46
 - summary 46, 48
 - SysBroadcastActionCode 27
 - use by application 27
- launch flags 20
- launcher
 - and expansion 215
 - application icon name 154
- launching applications 20
- LCD screen 72
- ledFont 270
- left arrow 105
- libPalmOSGlue.a 14
- line feeds 104
- list objects 112
- List resource 71
- LmGetLocaleSetting 370
- LmGlueGetLocaleSetting 370
- local IDs 177, 190
- localization
 - general guidelines 364
- LocGetNumberSeparators 369
- locking a chunk 182
- low-battery warnings 8
- lstEnterEvent 114
- LstGlueSetFont 273
- LstHandleEvent 113
- LstNewList 143
- lstSelectEvent 114

M

- managers
 - naming convention 315

- overview 6
- mapping file types to directories 242
- masking 124
- master pointer table 179
- maximizing battery life 351
- MemHandleFree 182
- MemHandleLock 182
- MemHandleNew 118, 182
- MemHandleResize 182
- MemHandleSize 182
- MemHandleUnlock 182
- MemMove 183
- memory architecture 171
- memory management
 - architecture 171
 - Introduction 169
- memory manager
 - chunks 174
 - See also* data manager
 - See also* resource manager
- Memory Stick 208, 242
- MemPtrNew 183
- MemPtrRecoverHandle 183
- MemSet 183
- menu bar objects 105
- Menu Bar resource 71
- menu bars
 - and user actions 106
 - bits behind 106
- Menu Resource 71
- MenuAddItem 108
- MenuCmdBarAddButton 110
- menuCmdBarOpenEvent 110
- menuDownEvent 109
- menuEvent 107, 109
- MenuHandleEvent 107
- MenuHideItem 108
- menuOpenEvent 108
- menus
 - checking visibility of 107
 - dynamic 108
 - shortcut 109
- MenuShowItem 108
- MIME types 242

missing character 255

modes 350

 efficient use 351

modifying Graffiti shortcuts 60

moving memory 183

MultiMedia (MMC) 208

multitasking kernel 352

N

naming conventions 230, 233

nilEvent 86

notification client 30

notification handlers 33, 34

notifications 30

 expansion 217, 218

 predefined 48

 registering for expansion 219

NotifyMgr.h 48

O

off-screen windows 272

OmGetCurrentLocale 367

optimization 184

 dynamic memory 184

 sorting 184

overlays 365

ovly resource 365

P

palettes 144

PalmOSCompatibility.h 15

PalmOSGlue.lib 14

parameter blocks 20

patches, loading during reset 338

PC connectivity 3, 170

PDB files

 exploring on expansion cards 238

 on expansion cards 234

pen 74

pen location polling 61

pen manager 61

pen queue 61, 63

pen strokes and key events 62

penDownEvent 93, 94, 95, 96, 97, 98, 99, 100, 101, 104, 113, 114

penUpEvent 59, 93, 94, 95, 96, 97, 98, 100, 101, 104, 107, 114

performance 184

PICT 124

PilotMain 21

 code example 21

pixel reading and writing 149

plug and play slot driver 209

popSelectEvent 114

Popup list 71

Popup trigger 71

popup trigger object 93

power 4

power modes 350

PRC files

 exploring on expansion cards 239

 on expansion cards 234

predefined launch codes 46

predefined notifications 48

prefDateFormat 367

preferences

 application-specific 23

 auto-off 352

 restoring 8

 saving 8

 short cuts 60

 system 23

PrefGetPreference 321, 367, 369, 370

prefMinutesWestOfGMT 368

prefTimeFormat 367

prefTimeZone 368

prefWeekStartDay 367

PrgHandleEvent 86

PrgStartDialog 86

PrgUpdateDialog 86

primary storage 208

private records 8

procedure alarms 311

progress manager 86

Push button 71

push button objects 96

 event flow 97

R

- RAM 4
 - expansion 208
- RAM store 169
- RAM use 170
- real-time clock 355, 356
- records 4, 190
- redBits 150
- registering for a notification 31
- reminder dialog boxes 306
- repeat control objects 95
- Repeating button 71
- ResEdit
 - resource naming conventions 6
- reset 338
 - digitizer screen 339
 - hard reset 339
 - loading patches 338
 - soft reset 338
- resource database header 198
- resource manager
 - using 199
- resources
 - gadget 140
 - label 137
 - storing 198
- response time 353
- restoring preferences 8
- resumeSleepChr 36
- RGBColorType 145, 150
- right arrow 105
- ROM
 - expansion 209
- ROM store 169
- ROM use 170
- ROM, retrieving serial number 353
- running mode 351

S

- saving preferences 8
- sclEnterEvent 139
- sclExitEvent 139
- sclRepeatEvent 139
- SclSetScrollBar 139

- scptLaunchCmdExecuteCmd 46
- scptLaunchCmdListCmds 46
- screen size 2, 72
- scrollbar objects 137
- SD slot driver 245
- secondary storage 208
- Secure Digital (SD) 208
- security
 - and expansion 223
- Select Font dialog 271
- Selector trigger 72
- selector trigger object 94
- serial number, retrieving 353
- serial port 8
- shift indicator
 - getting and setting state 59
- shortcut for menus 109
- shortcuts, Graffiti 59
- sleep mode 350
 - and current time 355
 - and real-time clock 355
- sliders 98
- slot custom call 247
- slot driver 211, 229, 245
 - accessing directly 247
 - plug and play 209
- slot reference number 220, 226
- slots 209
 - and volumes 228, 229
 - checking for a card 226
 - enumerating 225
 - referring to 211
- SMS application
 - and Attention Manager 293
- snooze timer (Attention Manager) 288
- snoozing items in Attention Manager 301
- soft reset 175, 338
 - dynamic heap 338
- sorting 184
- sound manager 332–333
- special drawing modes 151
- stack space 185
- standard directories on expansion media 214
- standard IO applications 381

start.prc 214, 222, 235
startup 19–50
startup routine, example 23
state information, storing 8
stdFont 269
stop routine example 29
storage
 primary 208
 secondary 208
storage heaps, erasing 339
storage RAM 171
StrCompare 258
StrDelocalizeNumber 369
String Manager 258
StrLocalizeNumber 369
strokes
 capturing 64
structure elements, naming convention 6
StrVPrintf 267
summary of launch codes 46, 48
symbol11Font 270
symbol7Font 270
symbolFont 270
Sync application 353
synchronization messages 8
SysAppLaunch 27, 156, 299
sysAppLaunchCmdAddRecord 46
sysAppLaunchCmdAlarmTriggered 46, 301, 308, 309
sysAppLaunchCmdAttention 46, 293
sysAppLaunchCmdCardLaunch 46, 216, 222
sysAppLaunchCmdCountryChange 46
sysAppLaunchCmdDisplayAlarm 46, 301, 308, 310
sysAppLaunchCmdExgAskUser 46
sysAppLaunchCmdExgGetData 46
sysAppLaunchCmdExgPreview 47
sysAppLaunchCmdExgReceiveData 47
sysAppLaunchCmdFind 47
sysAppLaunchCmdGoto 47, 263, 299
sysAppLaunchCmdGoToURL 47
sysAppLaunchCmdInitDatabase 47
sysAppLaunchCmdLookup 47
sysAppLaunchCmdNormalLaunch 7, 20, 23, 216, 222
sysAppLaunchCmdNotify 32
sysAppLaunchCmdOpenDB 47
sysAppLaunchCmdPanelCalledFromApp 47
SysAppLaunchCmdReset 338
sysAppLaunchCmdReturnFromPanel 47
sysAppLaunchCmdSaveData 48
sysAppLaunchCmdSyncNotify 42, 48
sysAppLaunchCmdSystemLock 48
sysAppLaunchCmdSystemReset 42, 48, 338
sysAppLaunchCmdTimeChange 48
sysAppLaunchCmdURLParams 48
SysAppLauncherDialog 156
sysAppLaunchStartFlagNoUISwitch 216, 222
SysBatteryInfo 352
SysBroadcastActionCode 27
SysCurAppDatabase 28
sysFtrCreator 316
sysFtrNumROMVersion 316
SysGraffitiReferenceDialog 60
sysMakeROMVersion 316
SysNotifyBroadcast 41
sysNotifyCardInsertedEvent 218
sysNotifyCardRemovedEvent 218
sysNotifyDeviceUnlocked 37
sysNotifyEarlyWakeupEvent 36
sysNotifyHelperEvent 38, 42
sysNotifyLateWakeupEvent 36
sysNotifyNormalPriority 33
SysNotifyParamType 33, 40
SysNotifyRegister 31
sysNotifySleepNotifyEvent 36, 37
sysNotifySleepRequestEvent 35
sysNotifySyncFinishEvent 32
sysNotifySyncStartEvent 32
SysNotifyUnregister 31
sysNotifyVolumeMountedEvent 218, 222
sysNotifyVolumeUnmountedEvent 218
SysReset 339
sysResIDPrefUIColorTableBase 147
sysResTExtPrefs 365
SysSetAutoOffTime 352

SysTaskDelay 351, 356
system event manager 57–66
system extensions
 expansion 208
system messages 8
system preferences 7, 23
system tick interrupts 355
system ticks 355
 and Simulator 356
 on Palm OS device 356
system version feature 316
SystemMgr.h 46, 317
SystemPreferencesChoice 321
SysTicksPerSecond 356
SysTraps.h 13
SysUIAppSwitch 28, 156

T

table objects 111
tAIB resource 154
tAIN resource 215
tblSelectEvent 112
TblSetItemFont 273
Tbmp 123, 124
Text Manager 252
text manager 363
tFBM 123
TimDateTimeToSeconds 308, 356
time manager 355
timer 355
TimGetSeconds 356
TimGetTicks 356
timing 356
TimSecondsToDateTime 356
TimSetSeconds 356
transparent bitmap 124
transparentColor 151
transparentIndex 151
try-and-catch mechanism 378
 example 378
TxtCaselessCompare 262
TxtCharBounds 260
TxtCharIsValid 256

TxtCharSize 260
TxtCompare 258, 262
TxtFindString 263
TxtGetNextChar 259
TxtGetPrevChar 259
TxtGlueCharIsValid 256
TxtParamString 266
TxtPrepFindString 263
TxtReplaceStr 266
TxtSetNextChar 259

U

UI design 2
 avoiding dialog box stacking 185
 design elements 70
 design philosophy 2
UI objects 5
 buttons 92
 check box 97
 control objects 92
 field 103
 form 83
 insertion point 153
 list 112
 menu bars 105
 popup trigger 93
 push button 96
 repeat control 95
 scrollbar 137
 selector trigger 94
 table 111
 windows 84
UI resources
 custom 140
UI resources, storing 198
UIAS 352
UIColorGetTableEntryIndex 149
UIColorGetTableEntryRGB 149
UIColorSetTableEntry 149
UIResources.r 365
universal connector 209
unlocking a chunk 182
up arrow 104
User Interface Application Shell 352

user interface elements
storing (resource manager) 198

V

version number 197
VFAT 211
VFS Manager 209, 213
 and file streaming 212
 and the Data Manager 212
 custom calls 246
 custom I/O 247
 debugging applications 247
 directory operations 240
 enumerating files 241
 file paths 233
 file system operations 231
 filenames 233
 functions 212
 overriding notification handlers 220
 registered notifications 218
 starting apps automatically 222
 verifying presence of 223
 volume operations 227
VFSCustomControl 246
VFSDirCreate 240
VFSDirEntryEnumerate 241
VFSExportDatabaseToFile 236
VFSExportDatabaseToFileCustom 236
VFSFileClose 232, 241
VFSFileCreate 232
VFSFileDBGetRecord 239
VFSFileDelete 232, 240
VFSFileEOF 232
VFSFileGetAttributes 233, 241
VFSFileGetDate 233, 241
VFSFileOpen 232, 240
VFSFileRead 232
VFSFileReadData 232
VFSFileRename 232, 240
VFSFileResize 232
VFSFileSeek 232
VFSFileSetAttributes 233, 241
VFSFileSetDate 233, 241
VFSFileSize 232

VFSFileTell 232
VFSFileWrite 232
VFSGetDefaultDirectory 242
VFSImportDatabaseFromFileCustom 237
VFSVolumeEnumerate 225
VFSVolumeFormat 228
VFSVolumeGetLabel 228
VFSVolumeInfo 228, 229
VFSVolumeMount 227
VFSVolumeSetLabel 228
VFSVolumeSize 228
VFSVolumeUnmount 227
Virtual File System. See VFS Manager
volumes
 and file systems 227
 and slots 211, 228
 automatically mounted 227
 basic operations 227
 enumerating 224
 formatting 228
 hidden 228, 229
 labeling 228
 matching to slots 229
 mounted 224
 mounting 227
 naming 230
 read-only 228
 size 228
 space available 228
 unmounting 227

W

wait cursor 184
WChar 253
WinCreateBitmapWindow 136
window objects 84
 off-screen 84
windowobjects
 off-screen 272
WinDrawBitmap 136
WinDrawTruncChars 261
winEnterEvent 84, 94, 107, 114, 313
winErase 151
winExitEvent 84, 95, 107, 114, 313
WinGetPixel 149

WinGetPixelRGB 149
WinIndexToRGB 147
winInvert 151
winMask 151
winOverlay 151
WinPaintBitmap 136
WinPalette 137, 146
WinPopDrawState 74
WinPushDrawState 74

WinRGBToIndex 147
WinSetBackColor 149, 150
WinSetBackColorRGB 149
WinSetDrawWindow 75
WinSetForeColor 149, 150
WinSetForeColorRGB 149
WinSetTextColor 149, 150
WinSetTextColorRGB 149

