



Palm OS[®] Programmer's Companion

**Document Number 3004-003
Print Date 6/00**

CONTRIBUTORS

Written by Christopher Bey, Elly Freeman, and Jean Ostrem

Production by <dot>PS document production services

Engineering contributions by David Fedor, Roger Flores, Steve Lemke, Bob Ebert, Ken Krugler, Bruce Thompson, Jesse Donaldson, Tim Wiegman, Gavin Peacock, Ryan Robertson, and Waddah Kudaimi

Copyright © 1996 - 2000, Palm, Inc. 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 Palm, Inc.

Palm, Inc. reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of Palm, Inc. to provide notification of such revision or changes. PALM, INC. MAKES 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. PALM, INC. MAKES 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, PALM, 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 PALM, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Palm Computing, Palm OS, Graffiti, HotSync, and Palm Modem are registered trademarks, and Palm III, Palm IIIe, Palm IIIx, Palm V, Palm Vx, Palm VII, Palm, More connected., Simply Palm, the Palm Computing platform logo, Palm III logo, Palm IIIx logo, Palm V logo, and HotSync logo are trademarks of Palm, Inc. or its subsidiaries. All other product and brand names may be trademarks or registered trademarks of their respective owners.

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

Palm OS Programmer's Companion
Document Number 3004-003
June 23, 2000

Palm, Inc.
5400 Bayfront Plaza
Santa Clara, CA 95052
USA

www.palm.com/devzone

Table of Contents

About This Document	11
Palm OS SDK Documentation	11
What This Volume Contains	11
Conventions Used in This Guide	13
 1 Programming Palm OS in a Nutshell	 15
Why Programming for Palm OS Is Different	15
Screen Size	15
Quick Turnaround Expected	16
PC Connectivity	16
Input Methods	17
Power	17
Memory	17
File System	18
Backward Compatibility	18
Palm OS Programming Concepts	18
Programming Tools	20
Where to Go From Here	20
 2 Good Design Practices	 23
Designing Your Application	24
Integrating Programs With the Palm OS Environment	24
Naming Conventions	27
Achieving Optimum Performance	28
Assigning a Creator ID	29
Working With Databases	30
Writing Robust Code	30
Avoiding Potential Pitfalls	33
User Interface Guidelines	33
Understanding the Palm OS UI Design Philosophy	34
Creating a Palm OS User Interface	36
Palm OS Resource Selection: List or Table?.	45
Localization Guidelines	45
Making Your Application Run on Different Devices	46

Running New Applications on an Older Device	47
Compiling Older Applications With The Latest SDK	48
3 Application Startup and Stop	49
Launch Codes and Launching an Application	49
Responding to Launch Codes	50
Responding to Normal Launch.	53
Responding to Other Launch Codes	56
Launching Applications Programmatically.	58
Creating Your Own Launch Codes	59
Stopping an Application.	59
Launch Code Summary	61
4 Event Loop	65
The Application Event Loop	67
Low-Level Event Management	71
Event Translation: Pen Strokes to Key Events.	71
Pen Queue Management	72
Key Queue Management	73
Auto-Off Control.	74
System Event Manager Summary	75
5 User Interface	77
Palm OS Resource Summary	78
Drawing on the Palm OS Device	79
The Draw State	80
Drawing Functions	81
Forms, Windows, and Dialogs	82
Alert Dialogs	83
Progress Dialogs	84
Controls.	86
Buttons	86
Popup Trigger	87
Selector Trigger	88
Repeating Button.	89
Push Buttons	90

Check Boxes	91
Sliders and Feedback Sliders	92
Fields	97
Menus	99
Dynamic Menus	101
Menu Shortcuts	102
Tables	104
Table Event	105
Lists	105
Categories	107
Initializing Categories in a Database	108
Initializing the Category Popup Trigger	111
Managing a Category Popup List.	112
The Default Application Category	115
Bitmaps	116
Versions of Bitmap Support	117
Drawing a Bitmap	118
Color Tables and Bitmaps	119
Labels	120
Scroll Bars	120
Custom UI Objects	122
Dynamic UI	125
Dynamic User Interface Functions	126
Color and Grayscale Support.	128
Color Table	128
UI Color List.	130
Insertion Point	132
Text.	132
Working With Text As Strings	133
Fonts in Palm OS 3.0 and Later.	134
Receiving User Input	135
The Graffiti Manager	135
The Key Manager	137
The Pen Manager.	138
Application Launcher	139

Summary of User Interface API.	140
6 Memory	155
Introduction to Palm OS Memory Use	155
Hardware Architecture	155
PC Connectivity	156
Memory Architecture	157
Heap Overview	161
The Memory Manager.	164
Memory Manager Structures.	164
Using the Memory Manager	167
Optimizing Memory Manager Performance	170
Summary of Memory Management	171
7 Files and Databases	173
The Data Manager	173
Records and Databases	174
Structure of a Database Header	175
Using the Data Manager	177
The Resource Manager	180
Structure of a Resource Database Header	180
Using the Resource Manager.	181
File Streaming Application Program Interface	183
Using the File Streaming API	183
Summary of Files and Databases	185
8 Palm System Features	189
Alarms	189
Setting an Alarm	190
Alarm Scenario	192
Setting a Procedure Alarm.	193
Features	195
The System Version Feature	196
Application-Defined Features	197
Using the Feature Manager	197
Feature Memory	198

Notifications	200
Registering for a Notification	201
Writing a Notification Handler	204
Sleep and Wake Notifications	206
Sound	207
Synchronous and Asynchronous Sound	209
Using the Sound Manager	209
Sound Preferences Compatibility Information	214
System Boot and Reset	218
Soft Reset	218
Soft Reset + Up Arrow	219
Hard Reset	219
System Reset Calls	220
Hardware Interaction	220
Palm OS Power Modes	220
Guidelines for Application Developers	222
Power Management Calls	222
The Microkernel	223
Retrieving the ROM Serial Number	224
Time	226
Using Real-Time Clock Functions.	227
Using System Ticks Functions	227
Floating-Point	228
Using Floating Point Arithmetic	228
Using 1.0 Floating-Point Functionality	229
Summary of System Features.	229

9 Serial Communication 233

Serial Hardware	233
Byte Ordering	234
Serial Communications Architecture Hierarchy	235
The Serial Manager	236
Using the Serial Manager	237
The New Serial Manager	240
Checking for the New Serial Manager.	241

What's New About the New Serial Manager	241
About the New Serial Manager.	242
Using the New Serial Manager.	243
New Serial Manager Example	248
Writing a Serial or Virtual Device Driver.	251
The Connection Manager	254
The Serial Link Protocol	255
SLP Packet Structures.	255
Transmitting an SLP Packet	258
Receiving an SLP Packet	258
The Serial Link Manager.	258
Using the Serial Link Manager	259
Summary of Serial Communications	263

10 Beaming (Infrared Communication) 265

Exchange Manager	265
Overview	266
Exchange Manager and Launch Codes	267
IR Library	269
IrDA Stack	269
Accessing the IR Library	271
Summary of Beaming	271

11 Network Communication 273

Net Library	273
About the Net Library	274
Net Library Usage Steps.	277
Obtaining the Net Library's Reference Number	278
Setting Up Berkeley Socket API	279
Setup and Configuration Calls	279
Opening the Net Library	284
Closing the Net Library	286
Version Checking.	287
Network I/O and Utility Calls	287
Berkeley Sockets API Functions	288
Extending the Network Login Script Support	295

Internet Library	300
System Requirements	301
Initialization and Setup	302
Accessing Web Pages	302
Asynchronous Operation	303
Using the Low Level Calls	305
Cache Overview	305
Internet Library Network Configurations	306
Summary of Network Communication	308
12 Internet and Messaging Applications	311
Overview of the Palm.Net System	312
Palm Query Applications	313
Palm.Net System Overview	314
System Version Checking	317
Using Clipper to Display Information	318
Launching Other Applications from Clipper	319
Sending Messages	320
New keyDownEvent Key Codes	321
Over the Air Characters	322
13 Localized Applications	325
Localization Guidelines	326
Using Overlays to Localize Resources	326
Text Manager and International Manager	329
Characters	331
Declaring Character Variables	331
Using Character Constants	332
Missing and Invalid Characters	332
Retrieving a Character's Attributes	333
Virtual Characters	334
Retrieving the Character Encoding	335
Strings	336
Manipulating Strings	337
Performing String Pointer Manipulation.	338
Truncating Displayed Text.	339

Comparing Strings	339
Global Find	340
Dynamically Determining a String's Contents	342
Dates	344
Numbers	345
Compatibility Information	346
Notes on the Japanese Implementation	347
Japanese Character Encoding	347
Japanese Character Input	348
Displaying Japanese Strings on UI Objects	348
Displaying Error Messages	349
Summary of Localization	349
14 Debugging Strategies	351
Displaying Development Errors	351
Using the Error Manager Macros	352
Understanding the Try-and-Catch Mechanism	353
Using the Try and Catch Mechanism	354
Summary of Debugging API.	355
15 Standard IO Applications	357
Creating a Standard IO Application	358
Creating a Standard IO Provider Application.	359
Summary of Standard IO	362
Index	363



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
Palm OS SDK Reference	An API reference document that contains descriptions of all Palm OS function calls and important data structures.
Palm OS Programmer's Companion	A guide to application programming for the Palm OS. This volume contains conceptual and "how-to" information that complements the Reference.
CodeWarrior Constructor for the Palm OS Platform	A guide to using CodeWarrior Constructor to create Palm OS resource files.
Palm OS Programming Development Tools Guide	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 four chapters

About This Document

What This Volume Contains

help you learn necessary tasks and possible features for your application.

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:

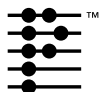
- [Chapter 1, “Programming Palm OS in a Nutshell.”](#) Provides new Palm OS programmers with a summary of what tasks and tools are involved in writing a Palm application and provides pointers to where to look for more information.
- [Chapter 2, “Good Design Practices.”](#) Provides new Palm OS programmers with guidelines for creating a well-designed Palm application with a well-designed user interface.
- [Chapter 3, “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 4, “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 5, “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 6, “Memory.”](#) Describes the memory architecture, memory use on the Palm devices, and the memory manager.
- [Chapter 7, “Files and Databases.”](#) Describes the data storage system, the data manager, resource manager, and the file streaming API.
- [Chapter 8, “Palm System Features.”](#) Describes features unique to the Palm hardware and OS such as alarms, the feature manager, preferences, the sound manager, system boot and reset, the microkernel, time, and floating point arithmetic.
- [Chapter 9, “Serial Communication.”](#) Describes the serial port hardware, the serial communications architecture, the serial link protocol, and the various serial communication managers.

- [Chapter 10, “Beaming \(Infrared Communication\).”](#) Describes the two facilities for beaming, or IR communication: the exchange manager and the IR library.
- [Chapter 11, “Network Communication.”](#) Describes the net library and Internet library and how to perform communications with networking protocols such as TCP/IP and UDP. The net library API maps very closely to the Berkeley UNIX sockets API.
- [Chapter 12, “Internet and Messaging Applications.”](#) Describes the Palm.Net system and how to use the Clipper and iMessenger applications to access and send information using the wireless capabilities of the Palm VII™ device.
- [Chapter 13, “Localized Applications.”](#) Discusses how to make your application localizable. Includes information on the text and international managers, as well as dealing with alternative character encodings, strings, numbers, and dates.
- [Chapter 14, “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 15, “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.

Conventions Used in This Guide

This guide uses the following typographical conventions:

This style...	Is used for...
<code>fixed width font</code>	Code elements such as function, structure, field, bitfield.
<u><code>fixed width underline</code></u>	Emphasis (for code elements).
bold	Emphasis (for other elements).
blue and underlined	Hot links.



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 OS® platform devices and tells you where to go for more in-depth information. It covers:

- [Why Programming for Palm OS Is Different](#)
- [Palm OS Programming Concepts](#)
- [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 OS platform devices, is a bit different from writing desktop applications because the Palm OS platform device is designed differently than a desktop computer. Also, users simply interact with the device differently than they do desktop computers.

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

Screen Size

The Palm OS device's screen is 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 section “[User Interface Guidelines](#)” in the chapter “[Good Design Practices](#)” for more detailed guidelines on designing the user interface.

Note that screen sizes of future Palm OS devices may vary.

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 Palm user uses a Palm 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 hand-held organizers 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 the Palm OS does not provide a wait cursor.)

To maximize performance, the user interface should minimize navigation between windows, opening of dialog boxes, 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 device 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 device (MemoPad, Address Book, etc.). These guidelines are summarized in the chapter “[Good Design Practices](#)” in this book.

PC Connectivity

PC connectivity is an integral component of the Palm OS platform device. The device 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 device with the user’s PC.

Many Palm OS applications have a corresponding application on the desktop. To share data between the device'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 hand-held device. To write a conduit, you use the Conduit SDK, which provides its own documentation.

Input Methods

Handheld users don't have a keyboard or mouse. Users enter data into the device using a pen. They can either write Graffiti® strokes or use the keyboard dialog provided on the device.

While Graffiti 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 device itself.

Power

The Palm OS platform device runs on batteries and thus does not have the same processing power as a desktop PC. It 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 device application.

Memory

The Palm OS device has limited heap space and storage space. Different versions of the device have between 512K and 8MB total of dynamic memory and storage available. The device 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 the Palm OS platform device are available, and each runs a different version of the Palm OS. Users are not expected to upgrade their versions of the 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](#)” in the chapter “[Good Design Practices](#)” 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 3](#), “[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 4](#), “[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 5](#), “[User Interface](#).”)
- All applications should use the memory and data management facilities provided by the system. (See [Chapter 6](#), “[Memory](#),” and [Chapter 7](#), “[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 this guide.

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.

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) from 3Com[®] Corporation. Documentation for the CodeWarrior IDE is provided with CodeWarrior. (See <http://www.palm.com> for information about other development tools.)

As with most applications, the user interface is generally stored in one or more resource files. You use the Palm OS Constructor 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 OS device, or you can use it in conjunction with one of the other debugging tools below.
- The Palm OS Emulator (POSE) tests your application on the desktop computer before downloading it onto the device.
- 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 device.

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 three chapters to learn the principles of Palm OS application and UI 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 OS device is included as examples on your SDK CD. 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 SDK Reference*

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

- Conduit Development Kits and documentation

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



Good Design Practices

This chapter helps you design an application that's fast, robust, and consistent with other applications on the device. The previous chapter described at a very high level the sorts of issues involved with writing a Palm OS® application. This chapter goes into much more detail about what is appropriate application design and user interface design. Its focus is how to:

- Avoid potential problems
- Make your application integrate well with others
- Achieve the best performance possible
- Localize with the minimum amount of work
- Maintain backward compatibility

The information was collected from engineers, testers, and other experts who designed, developed, and tested the four applications shipped with the first Palm OS device.

Paying attention to user interface guidelines and, if applicable, to localization guidelines early in your development cycle will save you time and trouble later. However, there's a lot to digest here. You may want to revisit this chapter from time to time to make sure you haven't forgotten anything.

This chapter discusses these topics:

- [Designing Your Application](#)
- [User Interface Guidelines](#)
- [Localization Guidelines](#)
- [Making Your Application Run on Different Devices](#)

NOTE: Be sure to read the [“Avoiding Potential Pitfalls”](#) and [“Writing Robust Code”](#) sections for information on the problems developers encounter most frequently.

Designing Your Application

This section provides Palm OS application design guidelines. It discusses these topics:

- [Integrating Programs With the Palm OS Environment](#)
- [Naming Conventions](#)
- [Achieving Optimum Performance](#)
- [Assigning a Creator ID](#)
- [Working With Databases](#)
- [Writing Robust Code](#)
- [Avoiding Potential Pitfalls](#)

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[®] and the on-screen 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 3, “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.

- Allow the system to post these messages:
 - alarms
 - low-battery warnings
 - system messages during synchronization
- Be sure your application does not obscure or change the Graffiti area, silk-screened buttons, and power button.
- Don't obscure Graffiti shift indicators.

In addition, follow these rules:

- Store state information in the application preferences database, not in the application record database. Call [PrefGetAppPreferences](#) and [PrefSetAppPreferences](#) to save and restore preferences. This is important if your application returns to the last displayed view by default.
- If your application uses the serial port, be sure to free 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.
- 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, 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

Good Design Practices

Designing Your Application

(available in OS versions 2.0 and later). You assign the name using the Constructor Project Settings panel.

Using the icon name is useful if you plan to localize your application.

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.

Your applications needs to have two icons:

- A large icon of type `tAIB`, with an ID of 1000. For compatibility with 2.0 devices, this icon should be 22 x 32 pixels; if your application only runs on older devices, 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 versions 3.5 and later allows you to create an Application Icon Family. You should not select the App Icon or Multi-bit Icon categories in this Constructor.

- Follow the guidelines listed in [User Interface Guidelines](#) and pay special attention to these points:
 - Ensure that the different user input modes (e.g., Graffiti and keyboard) are available for each field.
 - Ensure that menu items work with shortcuts as advertised.
 - Put limits on the length of fields and test them.
 - Ensure that any growable control, such as the launcher window or the menus, scrolls correctly.
- 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 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.

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;
```

Achieving Optimum Performance

Because the Palm OS device has limited heap space and storage, optimization is critical. The Palm OS device currently has no wait cursor, so users will expect rapid response. Test for performance. Launching, switching, and finding should be fast.

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.
- 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.
- 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 dialog boxes.
- 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 only has from 2K or 4K of stack space depending on the system software version.

Assigning a Creator ID

Each Palm OS application has a distinct creator ID. A creator ID is a 4-byte value used to tie together all the databases related to the application.

Creator IDs are unique to the application, not the creator of the application. Each database on the Palm device has an application value and a type. The type value should be set to `sysFileTApplication` for the executable's database and can be set to any value for other databases associated with an application.

Creator IDs need to be either all caps or mixed case. The Palm OS creator IDs differ from the creator ID and type that appear in the CodeWarrior Project Settings dialog boxes.

The creator ID for a Palm OS application is assigned in the PalmRez Project Settings panel.

- The Type should be set to `APPL`. Type is a 4-byte value.
- For information about creator IDs, and to register a creator ID, see this web page:

<http://www.palm.com/devzone/crid/cridsub.html>

The system uses the creator ID in various ways:

- Creator ID and type is used by the system launcher window to determine which databases are applications that should be displayed for selection.
- The memory application uses a creator ID and type to determine names of applications for display and to calculate total memory used by an application.

Working With Databases

Working properly with databases makes your application run faster and synchronize without problems. Follow these suggestions:

- 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.
- 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.

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.

- **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 devices—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.

Good Design Practices

Designing Your Application

- Check result codes when allocating memory

Because future devices 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 future revisions of the OS may 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 devices 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 database 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.

Avoiding Potential Pitfalls

Certain problems are encountered by application developers again and again. To avoid them, ask yourself these questions:

- Do you have a Creator ID for your application?

Each application (not just each company) has to have a Creator ID. Note that the Creator ID is only needed for the application (database of type APPL) not for all other databases.

- Did you use C library calls in your application? If you did, change them to corresponding Palm OS calls.

User Interface Guidelines

The Palm OS device is designed for rapid entry and quick retrieval of information. To maximize performance, the UI should minimize navigation between windows, opening of dialog boxes, 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 UI of your application is consistent with other applications on the device so users work with familiar patterns.

This section helps you design a user interface that's intuitive, easy to use, and consistent with other applications on the device. You learn about these issues:

- [Understanding the Palm OS UI Design Philosophy](#)
- [Creating a Palm OS User Interface](#)
- [Palm OS Resource Selection: List or Table?](#)

NOTE: Guidelines for implementing specific user-interface objects, such as information on the size of buttons or the font for labels, is provided in "[Palm OS Resources](#)" in the *Palm OS SDK Reference*. Also see the chapter "[User Interface](#)" in this book.

Understanding the Palm OS UI Design Philosophy

This section considers some issues that underlie the design of a user interface for the Palm OS device. It discusses these topics:

- [Creating Fast Applications](#)
- [Matching Use Frequency and Accessibility](#)
- [Creating Easy-to-Use Applications](#)

Creating Fast Applications

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

The Palm OS paradigm, in contrast, resembles that of a watch: People want instant access to information. Speed is therefore a critical design objective for hand-held organizers 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.

The user should be able to keep up with someone on the telephone when setting up appointments, looking up phone numbers, and so on. Priorities include the ability to:

- Execute key commands quickly
- Navigate to key screens quickly
- Find key data quickly (for example, phone numbers)

Matching Use Frequency and Accessibility

PC user interfaces are typically designed to display commands as if they were used equally. In reality, some commands are used very frequently while most are used only rarely. Similarly, some settings are more likely to be used than others. For example, a 3 p.m.- 4 p.m. meeting occurs much more frequently than a 3:25 to 4:15 meeting.

More frequently used commands and settings should be easier to find and faster to execute.

- Frequently executed software commands should be accessible by one tap.

- Infrequently used commands may require more user action.

Frequency	Example	Accessibility
Several times per hour.	Checking today's schedule or to-do items.	One tap.
Several times per day.	One hour meeting starting at the top of the hour.	One tap, write in place.
Several times per week.	Setting a weekly meeting (repeating event).	Several taps, second dialog box.

To make your application easily accessible, follow these guidelines:

- Minimize the number of taps to execute a function or change a setting.
- Provide command buttons for commonly executed multistep operations. Command buttons streamline execution.
- Minimize the need to change screens.
- Minimize the number of dialogs users have to open and close.
- Avoid dialogs within dialogs unless it's an infrequently used feature.

Choose the appropriate UI object when making a speed versus screen layout decision:

- Buttons on the screen provide instant access but take up valuable screen space.
- Push buttons are faster than popup lists and should be used if they fit on the screen reasonably.
- Popup lists are faster than manual input or increment/decrement buttons
- Popup lists can be cumbersome if there are too many items on the list or if the list needs to scroll.

Creating Easy-to-Use Applications

Users must be able to pick up a Palm device and, with no training or instruction, navigate between applications (without getting stuck) and execute basic commands within five minutes. Advanced commands should be easily accessible but should not be in the way.

The design must therefore fit the following criteria:

- Indicate clearly where in an application the user is. The PIM applications and modal dialog boxes have black title bars that usually indicate the application name and view.
- Make it obvious to the user how to get to different views. The command buttons provide the best example of achieving this.
- Use buttons for important commands.
- Accomplishing common tasks should be fast and easy. Minimizing steps helps not only speed but ease of use.

Ease of use amounts to a series of trade-offs. Striking the best balance for the most people is the biggest challenge of UI design. For example:

- Consistency reduces the time needed to learn an application by limiting the number of things that people need to keep in their heads at once. The user should not have to memorize an entire set of rules to use the device easily, for example, the up arrow key should not do different things on different screens.
- Choose the number of buttons on the screen diligently:
 - The fewer buttons on the screen, the less time it takes to learn how to use the product.
 - However, keeping a few frequently used buttons on screen helps reduce the time spent learning basic functionality.
- Advanced features should not be in the way for beginners, but should not require multiple-step searching.
- If possible, make your application consistent with the Palm OS device's native applications; users are used to interacting with them and will easily get used to your application if you follow these rules.

Creating a Palm OS User Interface

The small screen and pen-based user interaction require a different UI paradigm than a desktop computer. Here are some guidelines for making your application's interface consistent with other applications, including the PIM applications.

- Provide an application icon for the Launcher. To launch an application, users navigate to the launcher screen and tap on an icon. Choose a short icon name and an easy to recognize icon.

Specify the Application Icon Name and Application Icon using the Project Settings panel in Constructor.

- Provide a base screen that offers an overview of all available information. This screen is typically a list view. Not all applications need a base screen.
- Allow users to view most record information by pressing the navigation keys. Each event, to-do item, address, memo page, and so on is called a record.
- Organize records into user-defined categories if that makes sense. Categories usually result in more efficient screen use. Users can switch between categories using a popup menu or can display all records at once.
- Detailed information and advanced navigation require the use of a stylus. See [Data Entry Guidelines](#) for different data entry modes.
- Don't require double taps.
- Don't gray out menu commands or other UI elements; instead, remove an element when it's not available.
- If you can, allow finger navigation. For finger navigation, buttons need to be big enough for the system to recognize which button has been pushed. This is done by the Palm OS system software.
- Consider overloading the buttons. If you do overload, release the buttons at every possible opportunity. This is useful only for certain applications, such as games.

This section provides information on a variety of UI design issues:

- [Navigation Guidelines](#)
- [Preferences Guidelines](#)
- [Data Entry Guidelines](#)
- [Command Execution Guidelines](#)
- [Guidelines for Screen Layout](#)
- [Guidelines for Dialog Box Layout](#)

Navigation Guidelines

Users can move through applications by the following methods:

- **Switching applications.** Users press the physical buttons representing the PIM applications or access a launcher to switch applications.

On Palm OS 2.0 or later devices, users can assign each button to the application of their choice using a Preferences panel.

When switching to an application, the user is either presented with a standard first screen or returned to the last place in that application.

- **Switching views.** Each PIM application has two or more views (or modes) typically
 - a list view (or view mode)
 - an edit view (or edit mode)

The user taps on records or uses command buttons to toggle between these views.

Edit mode gives users access to the Details button for settings that affect the entire record. They can also access specific menu commands for records. In many applications, tapping on a record switches the application to edit mode and displays an input cursor.

- **Switching categories of records.** A popup menu in the top right corner lets users switch between categories. The popup menu is found in the list view of applications that support categories.
- **Switching records in applications.** Depending on the application, the user can scroll through lists of records, then tap on a record or a Details button for further information.
- **Graffiti navigation.** Support Graffiti navigation:
 - Left-right-forward-backward movement as part of a field's behavior.
 - Getting to next and previous screen using the down/up and up/down keystrokes.
- **Cycling through categories.** Holding the button on the hard case cycles through all categories.

- **Scrolling.** Records too long to display in one screen are scrollable. On-screen scroll buttons allow users to move up or down one line at a time. The physical arrow buttons allow users to move up and down one page at a time.

Scrollbars were introduced in OS 2.0. Scrollbars are optional. Developers have to consider the trade-off between taking up 7 pixels of horizontal space (the width of the scroll bar) vs. providing convenient scrolling for long lists of records.

Preferences Guidelines

Palm OS 2.0 and later has improved preferences facilities. They are available through launch codes, discussed in the chapter “[Application Launch Codes](#)” in the *Palm OS SDK Reference*.

The system now offers application-specific panels, sticky panels, and quick switch, as follows:

- **Application-specific panels.** Applications can add application-specific preferences panels to follow the system panels when the user cycles through the preferences. To do so, use the common code provided in the `Formats` example application to make the pull-down menu available. If the application uses the common code, a Done button inserts itself if the panel was called from the application, not sequentially following another panel.
- **Sticky panels.** When users bring up a preference panel from the launcher, exit the panel, then bring it up again, the system returns to the last panel used.
- **Quick switch.** Applications can now use the launch codes `sysAppLaunchCmdPanelCalledFromApp` and `sysAppLaunchCmdReturnFromPanel`, which allow an application to let users change preferences without first selecting the launcher, then selecting the application again.

Data Entry Guidelines

Users can enter data by the following methods:

- **Graffiti.** Graffiti characters are written in the text area on the digitizer and appear on the screen at the cursor location. The

user specifies the cursor location by tapping directly on the screen with the stylus.

Some controls accept input from Graffiti: For example, in the time selector dialog, you can write the time into the Graffiti area and it appears as start time or end time. The “next field” stroke switches between start and end time. The “Return” stroke dismisses the dialog.

For 2.0 and later applications, users expect that your application includes the Graffiti Reference option. You can include this option by calling [`SysGraffitiReferenceDialog`](#).

- **On-screen keyboard.** In place of using Graffiti, the user can tap an on-screen keyboard with the stylus. Any text is entered into a temporary window. When the user dismisses the keyboard, the system inserts that text at the cursor location.
- **Controls.** Buttons, check boxes, and popup lists provide a quick way to enter settings and select options.
- **HotSync.** The user can type data on the PC and download it to the Palm OS device.
- **Auto-creation.** Many applications, such as the DateBook or the Memo Pad provide an auto-create feature. If the user starts to write in a list view with no record selected, a new record is created with no additional interaction.

To provide a consistent interface, follow these guidelines when designing the data entry interface for your application:

- Let users perform basic data entry in place.
- Have the cursor ready and visible if there’s only one field for text entry (saves one tap).
- Provide a Details dialog for more elaborate data entry.
- Use the following format in the Details dialog:

Item (right-justified): Value(left-justified)
for example:

Set Date:4-1-96

Auto-off after:2 minutes

- Don't nest dialog boxes too deeply.
- Provide only one interface per function, that is, allow users to interact with an application through either a button, menu, or popup list. Don't provide both a button and a menu for the same actions.

NOTE: All developers are urged to include the rules listed below in their test plan. Applications that don't follow these rules may cause problems for other applications on the device.

- Whenever a field for user input is available, make sure that:
 - System keyboard is available via shortcut
 - System keyboard is available via menu
 - Graffiti input is possible (regular strokes and shortcuts)
 - Cut, copy, paste, and undo are possible
- Be sure to handle the clipboard correctly. If you use it, allow users to copy and paste between applications; if you don't, make sure it's intact when you exit.

Command Execution Guidelines

Users can execute commands by the following methods:

- **Command buttons.** Users execute common commands by tapping on command buttons at the bottom of the screen.
- **Menus.** Commands not represented by command buttons can be accessed via a simple menu system. The user taps on a menu hard icon in the digitizer area to invoke a menu bar. Beginning in Palm OS 3.5, the user may also invoke a menu bar by tapping the form's title. Provide menu shortcuts if possible.

NOTE: If you provide shortcuts, make sure that each shortcut is unique among all commands available at that time.

- **Graffiti menu command shortcuts.** Users can write a special Graffiti stroke and a command keystroke to execute a menu command. This is analogous to keyboard shortcuts on a personal computer. For example, writing the command

Good Design Practices

User Interface Guidelines

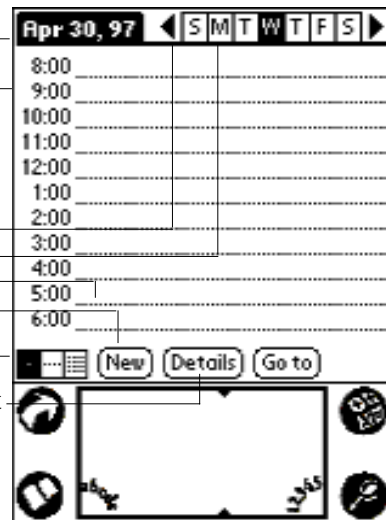
stroke symbol (a bottom-left to top-right line) and “C” allows the user to copy the selected text.

- **Buttons on command toolbar.** Beginning in Palm OS 3.5, entering the Graffiti command stroke symbol displays a command toolbar containing buttons for the commands that are possible in the current user context. For example, if text is selected in a field, the toolbar may display buttons for cut, copy, paste, and undo. Users may either complete the menu command shortcut as described above, or they may tap one of the buttons on the command toolbar.

Guidelines for Screen Layout

The illustration below provides some interface guidelines. Each guideline is numbered and explained in more detail below.

1. Provide a title bar. _____
2. Go to the edge of the screen. _____
3. Use resources provided with environment.
This example uses
 - repeating buttons _____
 - push buttons _____
 - fields _____
 - buttons _____
4. Align buttons at the bottom of the screen. _____
5. Leave one pixel above and below font height _____



1. In the title bar for each screen, provide both the application name and the name of the screen, if possible. Otherwise, provide the most relevant information.
2. Always go to the edge of the screen; that is, don't use borders. This practice maximizes screen real estate available to the application. The non-active area of the LCD and the case provide a natural margin.

3. Use the resources provided with the development environment and use the recommended values for width, height, and so on, provided in “[Palm OS Resources](#)” in the *Palm OS SDK Reference*.
4. Align buttons with the bottom edge of the screen.
5. For text surrounded by borders, leave one pixel above and below the font height.
6. For controls that can be displayed in groups, have at least two pixels to the left and right of the text label. The exception is command buttons, which require wider margins to accommodate the rounded border.
7. Don’t change or obscure the Graffiti status indicator area.
8. Don’t change or obscure the silk-screened icons.

Guidelines for Dialog Box Layout

The illustration below provides some guidelines for dialog box interfaces. Each guideline is numbered and explained in more detail below under the same number.

Good Design Practices

User Interface Guidelines

1. Provide online help for dialogs.
2. Use bold for labels.
Use non-bold for editable items.
3. Use right align:Left align in Details dialog.
4. Leave 3 pixels between edge of dialog and buttons.
5. Align dialog with bottom of screen.



1. Provide online help for dialogs. If you associate a Help ID with a form in Constructor, the system will add the “i” icon and handle presentation of the dialog.
2. Use bold face for labels, nonbold for editable items.
3. In the details dialog, right-align the label and left align the editable field.
4. When using buttons in dialogs, leave a space of 3 pixels between the edge of the dialog and the buttons.
5. Align dialogs with the bottom of the screen. Leave the screen title bar visible if possible.

Palm OS Resource Selection: List or Table?

Many developers find it difficult to decide whether to choose a list or a table for certain components of their application.

Use tables when you need quality text handling (including editing in place). Be careful if you work with non-text items in some of the columns, the selection region may be smaller than you need.

Use lists when users select from a predefined list (e.g. categories) or if the application determines the information to be displayed on the fly (based on previous user selections). Remember that you are responsible for scroll button handling and that editing can be non-trivial.

Localization Guidelines

If you're planning to localize the Palm OS software that you're developing, start by looking at the localized versions of the four PIM applications on the device. Then plan your application's interface, keeping in mind localization issues listed below. Also see the chapter "[Localized Applications](#)", which describes guidelines for writing code in a localized application.

- 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
- Abbreviations may be the best way to accommodate the particularly scarce screen real estate on the Palm OS device.
- 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.
- 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.
- Consider using string templates. For example, the MemoPad application uses the template: Memo # of %. The application

can replace # and % to change the text. Use as many parameters as possible to give localizers greater flexibility. Avoid building sentences by concatenating substrings together, as this often causes translation problems.

- Remember that user interface elements such as lists, fields, and tips scroll if you need more space.

Making Your Application Run on Different Devices

There are many different devices that run Palm OS, and each may have a different version of the OS installed on it (see [Table 2.1](#)). 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 applications.

Table 2.1 Some Palm OS platform devices

Name	Palm OS Version
Pilot 1000 ^a	1.0
Pilot 5000 ^a	1.0
PalmPilot ^a	2.0
PalmPilot Professional ^a	2.0
Palm III TM	3.0
IBM Workpad	2.0 or 3.0
Symbol SPT 1500	3.0
Qualcomm pdQ	3.0
Palm IIIe TM	3.1
Palm IIIx TM	3.1
Palm V TM	3.1
Palm VII TM	3.2
Palm IIIc TM	3.5

a. No longer available.

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

- [Running New Applications on an Older Device](#)
- [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 devices. If you use 2.0 features, your application won't run on the earliest Palm OS platform devices, 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 SDK Reference* has a “[Compatibility Guide](#)” appendix. This guide lists the feature 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 version number to guarantee that a feature exists. For example, Palm OS version 3.2 introduces wireless support, but not all Palm OS devices 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 SDK Reference* to learn how to check for the existence of each specific feature.

Compiling Older Applications With The Latest SDK

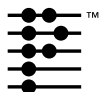
As a rule, all Palm OS applications developed with an earlier version of the Palm OS platform SDK should run error-free on the latest release.

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 devices.
- 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 devices that use prior releases of the OS.

NOTE: If you want to compile an existing application with the Palm OS 3.5 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 SDK Reference* for further details.



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 covers:

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

This chapter does not cover the main application event loop. The event loop is covered in [Chapter 4](#), “[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

Application Startup and Stop

Responding to Launch Codes

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 SDK 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 SDK 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 3.1](#) 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 3.1 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 ();
        StopApplication ();
    }
}
```

Application Startup and Stop

Responding to Launch Codes

```
// 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 was 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 4](#), “[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 3.1](#), the Datebook application example responds to `sysAppLaunchCmdNormalLaunch` by calling a function named `StartApplication`. [Listing 3.2](#) shows the `StartApplication` function.

Listing 3.2 StartApplication from Datebook.c

```
static UInt16 StartApplication (void)  
{  
    UInt16 error = 0;
```

Application Startup and Stop

Responding to Launch Codes

```
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)
    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 3.2](#)) or `sysAppLaunchCmdGoto`; so if the application hasn't received either of these launch codes, its global variables are usually not 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.

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 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` or `SysAppLaunch`, 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 3.1](#) 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 Startup and Stop

Stopping an Application

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 3.3](#) is an example of a `StopApplication` routine from `Datebook.c`.

Listing 3.3 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, 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);
}
```

Launch Code Summary

[Table 3.1](#) 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 3.1 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.
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.
sysAppLaunchCmdExgReceiveData	Notify application that it should receive incoming data via the exchange manager.

Table 3.1 Palm OS Launch Codes (*continued*)

Code	Request
<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 Clipper application and open a URL.
<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.
<u>sysAppLaunchCmdSaveData</u>	Save data. Often sent before find operations.
<u>sysAppLaunchCmdSyncNotify</u>	Notify applications that a HotSync [®] has been completed.
<u>sysAppLaunchCmdSystemLock</u>	Sent to the Security application to request that the system be locked down.

Table 3.1 Palm OS Launch Codes (*continued*)

Code	Request
<u>sysAppLaunchCmdSystemReset</u>	Respond to system reset. No UI is allowed during this launch code.
<u>sysAppLaunchCmdTimeChange</u>	Respond to system time change.
<u>sysAppLaunchCmdURLParams</u>	Launch an application with parameters from Clipper.



Event Loop

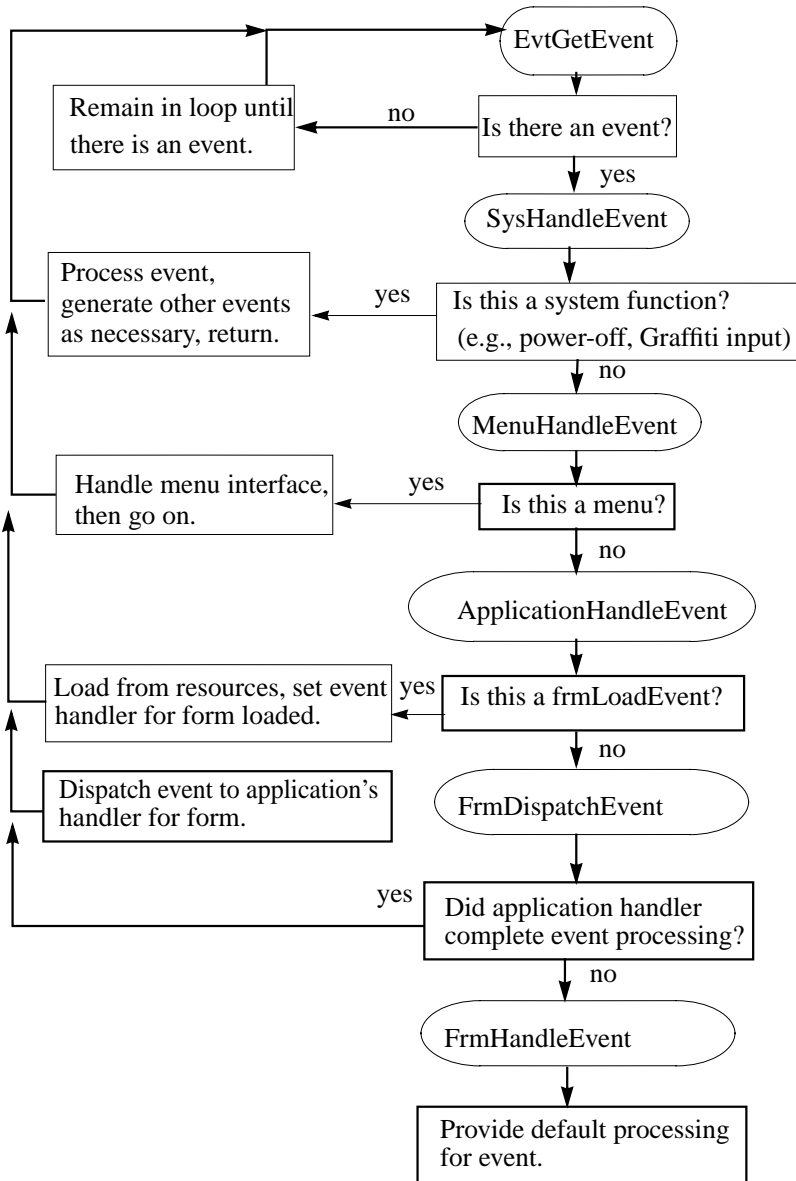
This chapter discusses the event manager, the main interface between the Palm OS® system software and the application. It discusses in some detail what an application does in response to user input, providing code fragments as examples where needed. The topics covered are:

- [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 SDK Reference*. Details for each event are given in [Chapter 3](#), "[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 4.1](#) illustrates control flow in a typical application.

Figure 4.1 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 on the event queue. If there are events on 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 4.1 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))
```

Event Loop

The Application Event Loop

```
        FrmDispatchEvent (&event);

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

In the event loop, the application iterates through these steps (see [Figure 4.1](#) and [Listing 4.1](#))

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[®] input, tapping silk-screened 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 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.
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

Event Loop

The Application Event Loop

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.
- 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 box 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

You can perform low-level event management using System Event Manager functions. The system event manager:

- manages the low-level pen and key event queues.
- translates taps on silk-screened icons into key events.
- sends pen strokes in the Graffiti area to the Graffiti 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.

This section provides information about the system event manager by discussing these topics:

- [Event Translation: Pen Strokes to Key Events](#)
- [Pen Queue Management](#)
- [Auto-Off Control](#)
- [System Event Manager Summary](#)

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 Graffiti area of the digitizer automatically to the Graffiti recognizer for conversion to a key event. Taps on silk-screened icons, such as

Event Loop

Low-Level Event Management

the application launcher, 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 Graffiti area of the digitizer are converted to key events
- All taps on the silk-screened icons are convert to key events
- All other strokes are passed on to the application for processing

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.

[Table 4.1](#) summarizes pen management.

Table 4.1 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.

Table 4.1 Pen Queue Management (*continued*)

The user...	The system...
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 stroke on the digitizer, which is converted in software to a key event

[Table 4.2](#) summarizes key management.

Event Loop

Low-Level Event Management

Table 4.2 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 Graffiti area of digitizer.	System manager calls the Graffiti 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 device. 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](#).

System Event Manager Summary

System Event Manager Functions

Main Event Queue Management

<u>EvtGetEvent</u>	<u>EvtEventAvail</u>
<u>EvtSysEventAvail</u>	<u>EvtAddEventToQueue</u>
<u>EvtAddUniqueEventToQueue</u>	<u>EvtCopyEvent</u>

Pen Queue Management

<u>EvtPenQueueSize</u>	<u>EvtDequeuePenPoint</u>
<u>EvtDequeuePenStrokeInfo</u>	<u>EvtFlushNextPenStroke</u>
<u>EvtFlushPenQueue</u>	<u>EvtGetPen</u>
<u>EvtGetPenBtnList</u>	

Key Queue Management

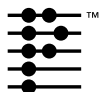
<u>EvtKeyQueueSize</u>	<u>EvtEnqueueKey</u>
<u>EvtFlushKeyQueue</u>	<u>EvtKeyQueueEmpty</u>

Handling pen strokes and key strokes

<u>EvtEnableGraffiti</u>	<u>EvtProcessSoftKeyStroke</u>
--	--

Handling power on and off events

<u>EvtResetAutoOffTimer</u>	<u>EvtWakeup</u>
---	----------------------------------



User Interface

This chapter describes the user interface 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's 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 a C structure is more compact than other objects could be.

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

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

- [Text](#)
- [Receiving User Input](#)
- [Application Launcher](#)

For guidelines on creating a user interface, see the chapter “[Good Design Practices](#)” earlier in this book.

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 5.1](#) maps user interface elements to resources. The ResEdit name is included for developers using that tool. It’s not relevant for Palm OS Constructor users.

All resources are discussed in detail in the chapter “[Palm OS Resources](#)” on page 77 of the *Palm OS SDK Reference*. Specific design recommendations for some of the elements are provided in the chapter “[Good Design Practices](#)” in “[User Interface Guidelines](#).”

Table 5.1 UI resource summary



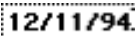


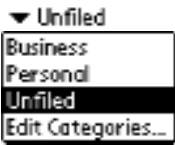

UI Element and Functionality	Example	Resource(s)
Command button— Execute command.		Button (tBTN)
Push button (also called radio button)— Select a value		Push button (tPBN)
Hot text entry— Invoke dialog that changes text of the button.		Selector trigger (tSLT)
Increment arrow— Increment/decrement values, or scroll.		Button (tBTN) or repeating button (tREP)
Check box— Toggle on or off.		Checkbox (tCBX)

Table 5.1 UI resource summary (continued)

UI Element and Functionality	Example	Resource(s)
Popup list— Choose a setting from a list.		Popup trigger (tPUT) Popup list (tPUL) List (tLST)
Menu— Execute commands not found on screen as buttons and so on.		Menu Bar (MBAR) Menu (MENU)
Text field— Display text (single or multiple lines).	Staff meeting	Field (tFLD)
Scroll bar— Use together with fields or tables.		Scroll bar (tSCL)

Drawing on the Palm OS Device

The first Palm OS® platform devices have an LCD screen of 160x160 pixels. The built-in LCD controller maps a portion of system memory to the LCD. This controller can support 2 bits/pixel grayscale; however, the Palm OS software only supported 1 bit/pixel monochrome graphics until version 3.0. Two bits/pixel support was added in Palm OS 3.0 and 4 bits/pixel in Palm OS 3.3. Palm OS 3.5 introduces support for both grayscale and color displays, with system palettes of either 1, 2, 4, or 8 bits/pixel. (See “[Color and Grayscale Support](#)” for more information.) Hardware may still limit the actual displayable depths.

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

Palm OS 3.5 introduces 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:

Table 5.2 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.

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

[Figure 5.1](#) 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 5.1 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 two sections describe special types of forms:

- [Alert Dialogs](#)
- [Progress Dialogs](#)

Alert Dialogs

If you want to display an alert dialog (see [Figure 5.2](#)) or prompt the user for a response to a question, use the alert manager. The alert manager defines the following functions:

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

Figure 5.2 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 (see the chapter "[Palm OS Resources](#)" in the *Palm OS SDK Reference*) 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 SDK 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.

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 higher 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 5.3](#)) display a text 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 rounded 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 5.3](#) shows the system events generated when the user interacts with the button and `CtlHandleEvent`'s response to the events.

Figure 5.3 Buttons

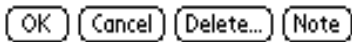


Table 5.3 Event flow for buttons

User Action	System Response	CtlHandleEvent Response
Pen goes down on a button.	penDownEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>ctlEnterEvent</code> 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 <code>EventType</code> .	Adds the ctlSelectEvent to the event queue.
Pen is lifted outside button.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the ctlExitEvent to the event queue.

Popup Trigger

A popup trigger (see [Figure 5.4](#)) displays a text label and a graphic element (always on the left) that signifies the control initiates a popup 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 5.4](#) shows the system events generated when the user interacts with the popup trigger and `CtlHandleEvent`'s response to the events. Because popup triggers are used to display list objects, also see the section "[Lists](#)" in this chapter.

Figure 5.4 Popup trigger



Table 5.4 Event flow for popup triggers

User Action	System Response	CtlHandleEvent Response
Pen goes down on the popup trigger.	penDownEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>ctlEnterEvent</code> to the event queue.
	ctlEnterEvent with popup trigger's ID number.	Inverts the trigger's display.
Pen is lifted from button.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the <code>ctlSelectEvent</code> to the event queue.
	ctlSelectEvent with popup 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. Control then passes to LstHandleEvent .
Pen is lifted outside button.	penUpEvent with the x and y coordinates stored in <code>EventType</code> .	Adds the ctlExitEvent to the event queue.

Selector Trigger

A selector trigger (see [Figure 5.5](#)) 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 5.5](#) shows the system events generated when the user interacts with the selector trigger and `CtlHandleEvent`'s response to the events.

Figure 5.5 Selector trigger

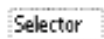


Table 5.5 Event flow for selector triggers

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

Table 5.6 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 <code>EventType</code> .	Adds the <code>ctlEnterEvent</code> to the event queue.
	ctlEnterEvent with button's ID number.	Adds the <code>ctlRepeatEvent</code> to the event queue.

Table 5.6 Event flow for repeating buttons (*continued*)

User Action	System Response	CtlHandleEvent Response
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 5.6](#)) 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 5.7](#) shows the system events generated when the user interacts with the push button and CtlHandleEvent's response to the events.

Figure 5.6 Push buttons

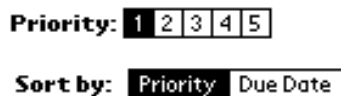


Table 5.7 Event flow for push buttons

User Action	System Response	CtlHandleEvent Response
Pen goes down on a push button.	penDownEvent with the x and y coordinates stored in EventType. ctlEnterEvent with push button's ID number.	Adds the ctlEnterEvent to the event queue. 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 5.7](#)) 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.

Push buttons and check boxes can be arranged into exclusive groups; one and only one control in a group can be on at a time.

[Table 5.8](#) shows the system events generated when the user interacts with the check box and CtlHandleEvent's response to the events.

Figure 5.7 Check boxes

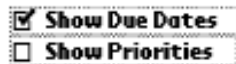


Table 5.8 Event flow for check boxes

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 5.8](#)) 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 5.8 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 5.9](#) shows the system events generated when the user interfaces with a slider and how `CtlHandleEvent` responds to the events.

Table 5.9 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.

Table 5.9 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 5.10](#) shows the system events generated when the user interacts with a feedback slider and CtlHandleEvent's response to the events.

Table 5.10 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 5.10 Event flow for feedback sliders (*continued*)

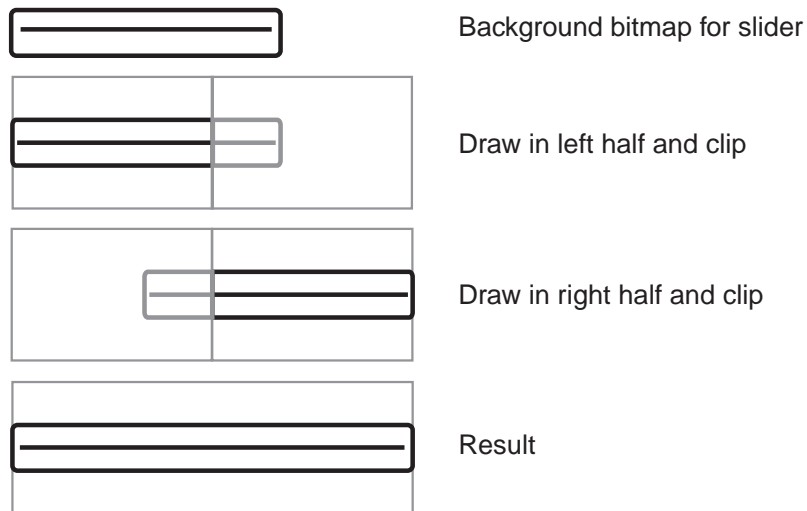
User Action	System Response	CtlHandleEvent Response
	ctlRepeatEvent	Adds or subtracts the slider's page jump value from its current value repeatedly until the thumb reaches the pen position or the slider's minimum or maximum. Then sends a ctlSelectEvent with slider's ID number and new value.
Pen goes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

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

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

The background bitmap you provide can be smaller than the slider's bounding rectangle. This allows you to provide one bitmap for sliders of several different sizes. If the background bitmap isn't as tall as the slider's bounding rectangle, it's vertically centered in the rectangle. If the bitmap isn't as wide as the slider's bounding rectangle, the bitmap is drawn twice. First, it's drawn left-justified in the left half of the bounding rectangle and clipped to exactly half of the rectangle's width. Then, it's drawn right-justified in the right half of the bounding rectangle and clipped to exactly half of the rectangle's width. (See [Figure 5.9](#).) Note that this means that the bitmap you provide must be at least half the width of the bounding rectangle.


Figure 5.9 Drawing a slider background



Fields

A field object displays one or more lines of text. [Figure 5.10](#) is an underlined, left-justified field containing data.

Figure 5.10 Field



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® 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.

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 5.11](#) provides an overview of how `FldHandleEvent` deals with the different events

Table 5.11 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 release 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.

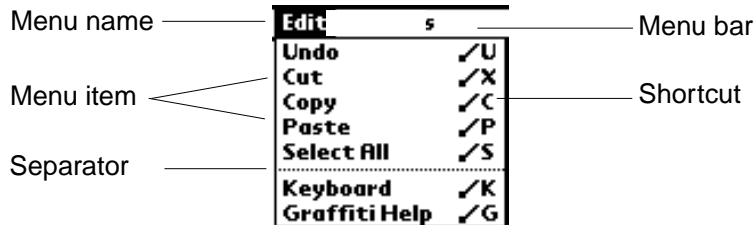
Table 5.11 Event flow for fields (*continued*)

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 5.11](#)).

Figure 5.11 Menu



User actions have the following effect on a menu:

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

A menu has the following features:

- Item separators, which are lines to group menu items.
- Keyboard 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 5.12](#) describes how user actions get translated into events and what [MenuHandleEvent](#) does in response.

Table 5.12 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.

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.

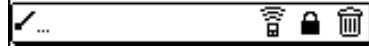
Menu Shortcuts

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 has been extended in Palm OS 3.5.

On all versions of Palm OS, the user can enter a Graffiti command keystroke followed by another Graffiti 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.

Starting in Palm OS 3.5, entering the Graffiti command character displays the command toolbar (see [Figure 5.12](#)). 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 entering the command character, the user has the choice of entering a Graffiti 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 5.12 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 5.1](#) 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 5.1 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 5.13](#) 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 5.13 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 5.1](#).

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 5.13 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 popup 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.

Formatting can be an issue for lists: While it’s possible to imitate a multi-column display, lists really consist of rows of text.

The [LstHandleEvent](#) function handles list events. [Table 5.14](#) provides an overview of how `LstHandleEvent` deals with the different events.

Table 5.14 Event flow for lists

User Action	System Response	LstHandleEvent Response
Pen goes down on popup trigger button.	winEnterEvent identifying list's window. lstEnterEvent with list's ID number and selected item.	Adds the <code>lstEnterEvent</code> to the event queue. 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> . lstSelectEvent with list's ID number and number of selected item.	Adds the <code>lstSelectEvent</code> to the event queue. Stores the new selection. If the list is associated with a popup trigger, adds a popSelectEvent to the event queue. with the popup trigger ID, the popup 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.

Categories

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

You create a category popup list the same way you create any other popup list: create the list resource, create the popup 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 popup list using the category API described in the chapter “[Categories](#)” on page 139 of the *Palm OS SDK Reference*.

For the most part, you can handle a category popup 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 popup trigger’s label when the form is opened (as described in “[Initializing the Category Popup Trigger](#)”).
- Call [CategorySelect](#) when the user selects the category popup trigger (as described in “[Managing a Category Popup 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 7](#), “[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 contains predefined categories that new users see when they start the application for the first time. Follow these guidelines when creating the resource:

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

[Listing 5.2](#) 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 5.2 Creating a database with an app info block

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

User Interface

Categories

```
} 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 Popup Trigger

When a form is opened, you need to set the text that the category popup trigger should display. To do this, use [CategoryGetName](#) to look up the name in the `AppInfoType` structure and then use [CategorySetTriggerLabel](#) to set the popup 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 popup 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 5.3](#) shows how to set the trigger label to match the category for a particular database record.

Listing 5.3 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 Popup List

When the user taps the category popup 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 popup list, manages the user selection, displays the Edit Categories modal dialog as necessary, and sets the popup trigger label to the item the user selected.

Calling CategorySelect

The following is a typical call to `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, popup 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 popup 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 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 SDK 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:

```
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`.

The Default Application Category

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 5.15](#)).

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 5.15 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.

Table 5.15 Launcher predefined categories (*continued*)

Default Launcher Category	Description
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 OS device, 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.

Bitmaps

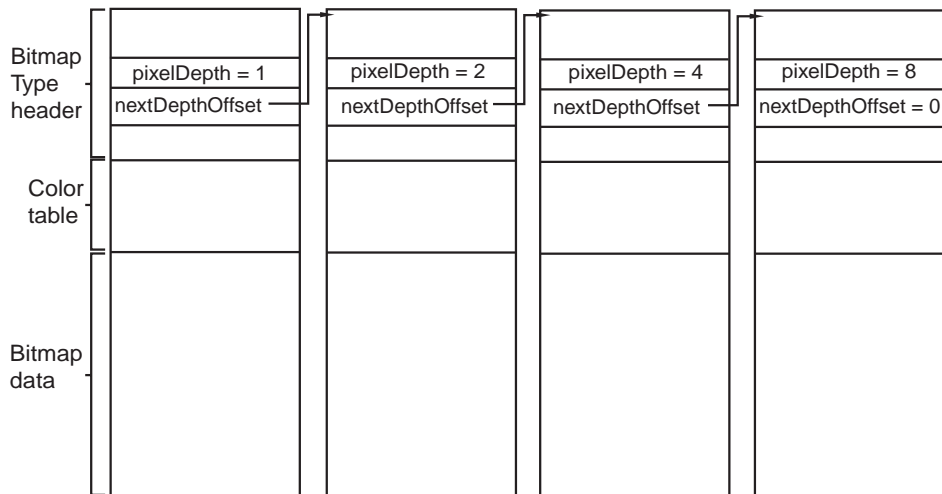
A bitmap is a graphic displayed by Palm OS. There are several ways to create a bitmap resource in Constructor:

- 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 'tbfmF' 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.)

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 5.14](#)). 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 5.14 Bitmap family



Versions of Bitmap Support

There are three different bitmap encodings:

- Version 0 encoding is supported by all Palm OS releases.

- Version 1 encoding is supported on Palm OS 3.0 and higher. 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 higher. 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.

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 [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.

```
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:

```
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 */  
    }  
}
```

`BmpCreate` always creates a version 2 bitmap, even if you don't specify a transparency or compression.

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.

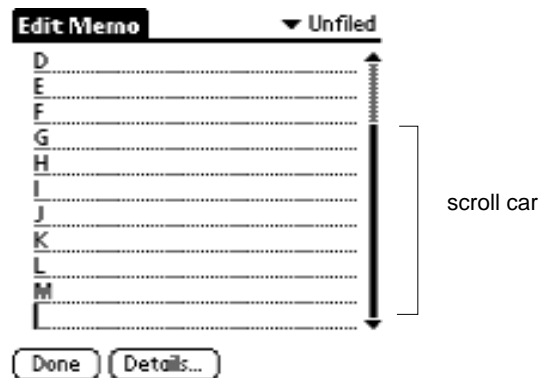
The label resource displays noneditable text or labels on a form (dialog box or full-screen). It's used, for example, to have text appear to the left of a checkbox instead of the right.

You don't interact with a label as a programmatic entity; however, you can use Form and Control 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 5.15](#)).

Figure 5.15 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 SDK 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 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

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 ([FormGadgetHandler](#)) 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 5.4](#) 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 5.5](#) shows how a gadget handler function might be written for Rock Music.

Listing 5.4 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 5.5 Palm OS 3.5 gadget example

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

    switch (cmd) {
        case frmGadgetDrawCmd:
            //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 == fldGadgetEnterEvent) {
                // 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 later provides 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.

Dynamic user interface objects are subject to the following limitations:

- You cannot create tables or Graffiti Shift indicators.

- You cannot create buttons (or repeating buttons) having frames or non-bold frames.

You can use the [FrmNewForm](#) function to create new forms dynamically. Palm's UI guidelines encourage you to keep popup 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. (On debug ROMs they are filled with random colors to help developers notice if they use an invalid value.) 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.

Color Table

The system color table is stored in a 'tblt' 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.

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 SDK 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 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 5.16](#) for a list of colors used.

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

To obtain the color list, the system first tries to load it from the synchronized preferences database using the value `sysResIDPrefUIColorTableBase` plus the current screen depth. The use of a preference allows for the possibility that individual users could customize the look using a third party “personality” or “themes” editor. If the preference is not defined, it loads the default color table from the system color table resource using `systemDefaultUIColorsBase` 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 device or to adjust it within an application. Defining these as color classes ensures that the user interface elements are consistent with each other.

Table 5.16 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 popup triggers	UIObjectFrame, UIObjectFill, UIObjectForeground, UIObjectSelectedFill, UIObjectSelectedForeground

Table 5.16 UI objects and colors (*continued*)

UI Object	Symbolic Colors Used
Labels	Labels on a control and noneditable fields use <code>UIObjectForeground</code> , and text written to a form using <code>WinDrawChars</code> or <code>WinPaintChars</code> use the current text setting in the draw state.
Scroll bars	<code>UIObjectFill</code> , <code>UIObjectForeground</code> , <code>UIObjectSelectedFill</code> , <code>UIObjectSelectedForeground</code>
Insertion point	<code>UIFieldCaret</code>
Front-end processor (currently only used on Japanese systems)	<code>UIFieldFepRawText</code> , <code>UIFieldFepRawBackground</code> , <code>UIFieldFepConvertedText</code> , <code>UIFieldFepConvertedBackground</code> , <code>UIFieldFepUnderline</code>

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.

Insertion Point

The insertion point is a blinking indicator that shows where text is inserted when users write Graffiti characters 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.

Text

This section 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.

NOTE: If your application is going to be localized, you must take special care when working with text. See the chapter “[Localized Applications](#)” for more information.

Working With Text As Strings

The string manager provides a set of string manipulation functions. The string manager API is closely modeled after the standard C string-manipulation functions like `strcpy`, `strcat`, etc.

Applications should use the functions built into the string manager instead of the standard C functions because doing so makes the application smaller:

- When your application uses the string manager functions, the actual code that implements the function is not linked into your application but is already part of the operating system.
- When you use the standard C functions, the code for each function you use is linked into your application and results in a bigger executable.

In addition, many standard C functions don’t work on the Palm OS device at all because the OS doesn’t provide all basic system functions (such as `malloc`) and doesn’t support the subroutine calls used by most standard C functions.

NOTE: If your application is going to be localized, be careful when using string functions. Where possible, use the functions described in the chapter “[Localized Applications](#)” instead.

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 in Palm OS 3.0 and Later

Palm OS 3.0 and later provides a new font (`largeBoldFont`), two new font manipulation routines ([FontSelect](#) and [FntDefineFont](#)), and support for the use of custom fonts.

To use the large, bold font, pass the `largeBoldFont` selector to the [FntSetFont](#) function. Under Palm OS 3.0 and later, if you try to draw with a font that isn't installed, the system uses the standard font by default. Previous versions of Palm OS can crash if told to use a nonexistent font.

The [FontSelect](#) function displays a dialog box in which the user can specify the use of one of the three primary fonts `stdFont`, `boldFont`, or `largeBoldFont`. For more information, see the description of `FontSelect` in the *Palm OS SDK Reference*.

The [FntDefineFont](#) function makes a custom font available to your application. For more information, see the description of `FntDefineFont` in the *Palm OS SDK Reference*.

Currently, Palm has not made available any tools or specifications to convert desktop fonts for use on Palm OS 3.0 or later. If you have an urgent need for such support, send email to devsupp@palm.com for updated information.

Receiving User Input

The three main ways that a user interacts with an application are:

- by entering Graffiti
- by pressing a hardware button on the device
- by tapping the pen on a control in a form or dialog

The Palm OS provides three managers that control these three types of input: [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. This section describes the three input 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 this chapter that covers that object.)

The Graffiti Manager

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

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 Graffiti 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 Graffiti ShortCuts 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 Graffiti 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 Graffiti shift state (caps lock on/off, temporary shift state, etc.)
- Notifying Graffiti when the user selects a different field. Graffiti 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.

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.

Accessing Graffiti ShortCuts

Other routines provide access to the Graffiti ShortCuts database. This is a separate database owned and maintained by the Graffiti manager that contains all of the shortcuts. This database is opened by the Graffiti manager when it initializes and stays open even after applications quit.

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 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 OS device. It converts hardware 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 OS platform device 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 device 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.

Application Launcher

The Application Launcher (accessed via the silkscreen “Applications” button) presents a window or menu from which the user can open other applications present on the Palm device. Applications installed on the Palm device (resource databases of type APPL) appear in the Application Launcher automatically.

NOTE: Versions of Palm OS prior to 3.0 implemented the Launcher as a popup. The [SysAppLauncherDialog](#) function, which provides the API to the old popup launcher, is still present in Palm OS 3.0 for compatibility purposes, but it has not been updated and, in most cases, should not be used.

The Launcher application can beam applications to other Palm devices. Only the application itself is beamed; associated storage databases and preferences are not transmitted. To suppress the beaming of your application by the Launcher, you can set the `dmHdrAttrCopyPrevention` bit in your database header. (For a runtime code example, see the “Dr McCoy” sample application. Note that you can also use compile-time code to suppress beaming.)

Normally, the Launcher represents installed applications graphically as icons that appear in the Launcher window. The Launcher application also provides a list mode that allows the user to see more applications at once than are normally visible in its default viewing mode. You can use the Constructor tool to provide a small icon for the list mode—you'll need to create a `tAIB` resource having 1001 as the value of its ID.

The Launcher displays a version string from each application's `tver` resource, ID 1000. This short string (usually 3 to 6 characters) is displayed in the "Info" dialog.

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 5.6](#).

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

Listing 5.6 Opening the Launcher

```
EventType newEvent;  
newEvent.eType = keyDownEvent;  
newEvent.data.keyDown.chr = launchChr;  
newEvent.data.keyDown.modifiers = commandKeyMask;  
EvtAddEventToQueue (&newEvent);
```

For information on launching other applications programmatically, see "[Launching Applications Programmatically](#)" in the chapter "[Application Startup and Stop](#)."

Summary of User Interface API

Progress Manager Functions

[PrgHandleEvent](#)

[PrgStopDialog](#)

[PrgUserCancel](#)

[PrgStartDialog](#)

[PrgUpdateDialog](#)

Form Functions

Initialization

[FrmInitForm](#)

Event Handling

[FrmSetEventHandler](#)

[FrmHandleEvent](#)

[FrmDispatchEvent](#)

Displaying a Form

[FrmGotoForm](#)

[FrmDrawForm](#)

[FrmSetActiveForm](#)

[FrmPopupForm](#)

[FrmNewForm](#)

Displaying a Modal Dialog

[FrmCustomAlert](#)

[FrmCustomResponseAlert](#)

[FrmAlert](#)

[FrmDoDialog](#)

[FrmHelp](#)

[FrmSaveActiveState](#)

[FrmRestoreActiveState](#)

[FrmNewGsi](#)

Updating the Display

[FrmUpdateForm](#)

[FrmShowObject](#)

[FrmRemoveObject](#)

[FrmReturnToForm](#)

[FrmHideObject](#)

[FrmUpdateScrollers](#)

Form Attributes

[FrmVisible](#)

[FrmSaveAllForms](#)

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](#)

User Interface

Summary of User Interface API

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](#)

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](#)
[WinSetWindowBounds](#)
[WinGetFramesRectangle](#)

[WinWindowToDisplayPt](#)
[WinGetWindowExtent](#)
[WinGetWindowBounds](#)
[WinGetWindowFrameRect](#)

User Interface

Summary of User Interface API

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](#)

Control Functions

Displaying a Control

[CtlShowControl](#)
[CtlSetUsable](#)
[CtlNewGraphicControl](#)

[CtlDrawControl](#)
[CtlNewControl](#)
[CtlNewSliderControl](#)

Control's Value

[CtlGetValue](#)
[CtlGetSliderValues](#)

[CtlSetValue](#)

Label

[CtlSetLabel](#)

[CtlGetLabel](#)

Enabling/Disabling

[CtlSetEnabled](#)
[CtlHideControl](#)

[CtlEnabled](#)
[CtlEraseControl](#)

Event Handling

[CtlHandleEvent](#)

Control Functions

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](#)

Editing Text

[FldSetText](#)
[FldInsert](#)
[FldEraseField](#)

[FldSetTextHandle](#)
[FldDelete](#)

Cut/Copy/Paste

[FldCopy](#)
[FldPaste](#)

[FldCut](#)
[FldUndo](#)

Field Functions

Scrolling

<u>FldScrollField</u>	<u>FldScrollable</u>
<u>FldSetScrollPosition</u>	<u>FldGetScrollPosition</u>
<u>FldGetVisibleLines</u>	<u>FldGetScrollValues</u>
<u>FldGetNumberOfBlankLines</u>	

Field Attributes

<u>FldGetAttributes</u>	<u>FldSetAttributes</u>
<u>FldGetFont</u>	<u>FldSetFont</u>
<u>FldGetMaxChars</u>	<u>FldSetMaxChars</u>
	<u>FldGetBounds</u>

Text Attributes

<u>FldCalcFieldHeight</u>	<u>FldGetTextHeight</u>
<u>FldGetTextAllocatedSize</u>	<u>FldGetTextLength</u>
<u>FldSetTextAllocatedSize</u>	<u>FldWordWrap</u>

Working With the Insertion Point

<u>FldGetInsPtPosition</u>	<u>FldSetInsPtPosition</u>
<u>FldSetInsertionPoint</u>	

Releasing Memory

<u>FldCompactText</u>	<u>FldFreeMemory</u>
---------------------------------------	--------------------------------------

Event Handling

<u>FldHandleEvent</u>	<u>FldSendChangeNotification</u>
<u>FldSendHeightChangeNotification</u>	

Dynamic UI

<u>FldNewField</u>

Menu Functions

MenuDispose	MenuDrawMenu
MenuEraseStatus	MenuInit
MenuHandleEvent	MenuGetActiveMenu
MenuSetActiveMenu	MenuSetActiveMenuRscID
MenuAddItem	MenuCmdBarAddButton
MenuCmdBarDisplay	MenuCmdBarGetButtonData
MenuHideItem	MenuShowItem

Table Functions

Drawing Tables

TblDrawTable	TblSetCustomDrawProcedure
TblSetLoadDataProcedure	

Updating the Display

TblRedrawTable	TblGrabFocus
TblReleaseFocus	TblUnhighlightSelection
TblRemoveRow	TblMarkRowInvalid
TblMarkTableInvalid	TblSelectItem
TblUnhighlightSelection	

Retrieving Data

TblGetItemPtr	TblGetRowData
TblFindRowData	TblGetItemInt
TblGetSelection	TblGetCurrentField
TblSetSaveDataProcedure	

Displaying Data

TblSetItemInt	TblSetItemStyle
TblSetItemPtr	TblSetRowID
TblSetRowData	

Retrieving a Row

TblFindRowID	TblGetRowID
------------------------------	-----------------------------

User Interface

Summary of User Interface API

Table Functions

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](#)

Private Record Functions

[SecSelectViewStatus](#)

[SecVerifyPW](#)

List Functions

Displaying a List

[LstDrawList](#)

[LstPopupList](#)

[LstSetDrawFunction](#)

[LstNewList](#)

Updating the Display

[LstMakeItemVisible](#)

[LstSetListChoices](#)

[LstSetSelection](#)

[LstScrollList](#)

[LstSetHeight](#)

[LstSetTopItem](#)

[LstSetPosition](#)

List Data and Attributes

[LstGetNumberOfItems](#)

[LstGetSelection](#)

[LstGetVisibleItems](#)

[LstGetSelectionText](#)

Removing a List From the Display

[LstEraseList](#)

Event Handling

[LstHandleEvent](#)

Category Functions

[CategoryCreateList](#)

[CategoryEdit](#)

[CategoryFind](#)

[CategoryFreeList](#)

[CategoryGetName](#)

[CategoryGetNext](#)

[CategoryInitialize](#)

[CategorySelect](#)

[CategorySetName](#)

[CategorySetTriggerLabel](#)

[CategorySelect](#)

[CategoryTruncateName](#)

User Interface

Summary of User Interface API

Bitmap Functions

BmpBitsSize	BmpGetBits
BmpColortableSize	BmpGetColortable
BmpCompress	BmpSize
BmpCreate	ColorTableEntries
BmpDelete	

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

String Manager Functions

Length of a String

[StrLen](#)

String Manager Functions

Comparing Strings

[StrCompare](#)
[StrCaselessCompare](#)

[StrNCompare](#)
[StrNCaselessCompare](#)

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](#)

[FntSetFont](#)

Accessing the Font Programmatically

[FntGetFont](#)

[FntGetFontPtr](#)

Wrapping Text

[FntWordWrap](#)

[FntWordWrapReverseNLines](#)

String Width

[FntCharsInWidth](#)
[FntLineWidth](#)

[FntCharsWidth](#)
[FntWidthToOffset](#)

User Interface

Summary of User Interface API

Font Functions

Character Width

[FntAverageCharWidth](#)

[FntCharWidth](#)

Height

[FntCharHeight](#)

[FntLineHeight](#)

[FntBaseLine](#)

[FntDescenderHeight](#)

Scrolling

[FntGetScrollValues](#)

Creating a Font

[FntDefineFont](#)

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

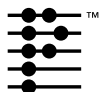
[KeyCurrentState](#)
[KeySetMask](#)

[KeyRates](#)

Pen Manager Functions

[PenCalibrate](#)

[PenResetCalibration](#)



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, handheld devices. Given these constraints, Palm OS is efficient in its use of both memory and processing resources. This section presents two aspects of Palm OS devices that contribute to this efficiency: [Hardware Architecture](#) and [PC Connectivity](#).

Hardware Architecture

The first implementation of Palm OS provides nearly instantaneous response to user input while running on a 16 MHz Motorola® 68000 type processor with a minimum of 128K of nonvolatile storage memory and 512 KB of ROM. Subsequent Palm OS devices provide additional RAM and ROM in varying amounts.

The ROM and RAM for each Palm OS device resides on a memory module known as a **card**. Each memory card can contain ROM,

Memory

Introduction to Palm OS Memory Use

RAM, or both. There is no RAM or ROM storage on the motherboard of the device.

Though all previous and current Palm OS devices hold one card in a user-accessible hardware slot, it is unwise to assume that any Palm OS device has a memory module that can be removed physically. A “card” is simply a logical construct used by the operating system—Palm OS devices can have one card, multiple cards, or no cards. For example, the Simulator provided by the Palm OS SDK on Macintosh can simulate a device that has two cards.

The ROM and RAM on each card is divided into one or more heaps of 64K (in the current implementation) or less. All the RAM-based heaps on a memory card are treated as the RAM store, and all the ROM-based heaps are treated as the ROM store. The heaps for a store do not have to be adjacent to each other in address space—they can be scattered throughout the memory space on the card—but they must all reside on the same card.

The main suite of applications provided with each Palm OS device 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 the Palm OS device. The device 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 device 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 OS device 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 device with a new ROM.

Memory Architecture

IMPORTANT: This section describes the current (3.X) 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 OS device has less than 1 MB of memory, or .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 OS device 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 device is turned "off" (i.e., is in low-power sleep mode.) See "[Palm OS Power Modes](#)" in the

chapter “[Palm System Features](#)” in this book. All of storage memory is preserved even when the device 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 device 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 6.1](#) provides more information about the dynamic heap space that currently available combinations of OS and hardware provide.

Table 6.1 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 6.1 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

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 database manager or resource manager, according to whether it needs to manipulate user data or resources.

NOTE: Starting with Palm OS 3.5, the dynamic heap is sized based on the amount of memory available to the system.

The size and number of storage heaps available on a particular device 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 List, 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.X configures 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.

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 (3.X) 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 OS device 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.

When a Palm OS device 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 OS device 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 OS devices do not provide hardware support for multiple cards, the use of local IDs provides support in software for future devices 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.

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 OS device, 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 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, 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 OS devices, be sure to read "[Writing Robust Code](#)" in the chapter "[Good Design Practices](#)" 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.X removes the 64 KB maximum size restriction on individual heaps and creates just two heaps: one 96K dynamic heap and one storage heap that is the size of all remaining RAM on the card.

NOTE: Starting with PalmOS 3.5, the dynamic heap is sized based on the amount of memory available to the system. The size which will be used is as follows:

Device RAM size	Heap size
< 2 mb of ram	64 k
>= 2 mb	128 k
>= 4 mb	256 k

Optimizing Memory Manager Performance

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 is 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.

Summary of Memory Management

Memory Manager Functions

Allocating and Freeing Memory

<u>MemHandleNew</u>	<u>MemPtrNew</u>
<u>MemHandleLock</u>	<u>MemHandleUnlock</u>
<u>MemLocalIDToLockedPtr</u>	<u>MemPtrUnlock</u>
<u>MemHandleFree</u>	<u>MemPtrFree</u>

Resizing Chunks

<u>MemHandleResize</u>	<u>MemHandleSize</u>
<u>MemPtrResize</u>	<u>MemPtrSize</u>
<u>MemHeapFreeBytes</u>	<u>MemHeapSize</u>

Working With Memory

<u>MemMove</u>	<u>MemSet</u>
<u>MemCmp</u>	<u>MemHeapCompact</u>

Converting Pointers

<u>MemPtrRecoverHandle</u>	<u>MemHandleToLocalID</u>
<u>MemLocalIDKind</u>	<u>MemLocalIDToGlobal</u>
<u>MemPtrToLocalID</u>	<u>MemLocalIDToPtr</u>

Chunk Information

<u>MemHandleCardNo</u>	<u>MemHandleDataStorage</u>
<u>MemHandleHeapID</u>	<u>MemHandleSetOwner</u>
<u>MemPtrCardNo</u>	<u>MemPtrDataStorage</u>
<u>MemPtrSetOwner</u>	

Heap Information

<u>MemPtrHeapID</u>	<u>MemHeapID</u>
<u>MemHeapDynamic</u>	<u>MemHeapCheck</u>
<u>MemHeapFlags</u>	

Memory

Summary of Memory Management

Memory Manager Functions

Card Information

[MemCardInfo](#)

[MemNumHeaps](#)

[MemStoreInfo](#)

[MemNumCards](#)

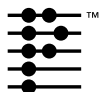
[MemNumRAMHeaps](#)

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.

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 OS devices 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 OS device 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 database 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 OS device. 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

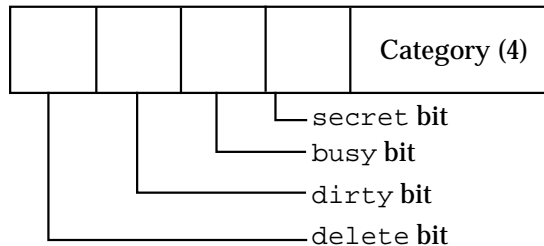
The database header has the following fields:

- 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 7.1](#), 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 7.1 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 OS device 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 device.

When a user “deletes” a record on the Palm OS device, 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 OS device). 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 OS device. For deleted records, the PC just has to delete the same record from the PC before permanently removing the record entry from the Palm OS device.

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 that is stored in the lower four bits of the record's attribute field.

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.

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

Files and Databases

The Resource Manager

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 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 `nil` 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 Hot Sync 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

Files and Databases

File Streaming Application Program Interface

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.

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 Database 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](#)

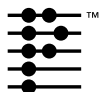
[FileFlush](#)

Error Handling

[FileError](#)

[FileClearerr](#)

[FileGetLastError](#)



Palm System Features

In this chapter, you learn how to work with the features that the Palm OS[®] system provides, such as sound, alarms, 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:

- [Alarms](#)
- [Features](#)
- [Notifications](#)
- [Sound](#)
- [System Boot and Reset](#)
- [Hardware Interaction](#)
- [The Microkernel](#)
- [Retrieving the ROM Serial Number](#)
- [Time](#)
- [Floating-Point](#)
- [Summary of System Features](#)

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.
- Handles alarms by application in a two cycle operation
 - First, it notifies each application that the alarm has occurred.
 - Second, it allows each application to display some UI.
- Allows only one alarm to be set per application.

However, the alarm manager:

- Doesn't provide reminder dialog boxes.
- Doesn't play the alarm sound.

This section looks in some detail at how the alarm manager and applications interact when processing an alarm. It covers:

- [Setting an Alarm](#)
- [Alarm Scenario](#)
- [Setting a Procedure Alarm](#)

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 8.1](#) shows how the Datebook application sets an alarm.

Listing 8.1 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);
}
```

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.

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 8.1](#).) 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 section "[Features](#)" in this 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 will delay other applications from receiving alarms that are set to trigger at the same time.

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.

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`](#).
8. The application can now:
 - Display a dialog box.
 - Display some other type of reminder.
9. The alarm manager processes the alarm queue for the next application that set an alarm for the alarm being triggered and step 6 and 7 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.

Setting a Procedure Alarm

Beginning with Palm OS version 3.2, the system supports setting procedure alarms in addition to the application-based alarms

Palm System Features

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 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.

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.)

`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 will crash 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.

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 Palm, 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 device 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
0x03103000	Palm OS 3.1
0x03103000	Palm OS 3.1
0x03203000	Palm OS 3.2
0x03503000	Palm OS 3.5

Any application can find out the system version by looking for this feature. For example:

```
// See if we're on ROM version 2.0 or later.  
FtrGet(sysFtrCreator, sysFtrNumROMVersion,
```

```
    &romVersion);  
    if (romVersion >= 0x02000000) {  
        ....  
    }
```

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 SDK 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 device 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 64k. 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 device is reset, you only need to open the database once. [Listing 8.2](#) illustrates this example.

Listing 8.2 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.

Notifications

On systems where the [Notification Feature Set](#) is present, your application can receive **notifications** 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 chapter “[Notification Manager](#)” on page 665 in the *Palm OS SDK Reference* for a complete list. (The

notification manager broadcasts the notifications and maintains a list of clients for each notification).

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.

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 devices. 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 8.3](#).

Listing 8.3 Registering an application for a notification

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

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 8.4](#).

Listing 8.4 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 8.3](#) and [Listing 8.4](#) 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 routine. Only use a callback routine 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 routine.

- `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), use a value between -15 and +15 for the priority. Using a value in this range ensures that your code won't collide with the system's handling of notifications.

- `myDataP` is a pointer to any data you need to access in your notification handler routine. 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 8.3](#) and [Listing 8.4](#) 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 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 8.5](#) 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 8.5 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 two functions that broadcast notifications: [SysNotifyBroadcast](#), which immediately broadcasts the notification, and [SysNotifyBroadcastDeferred](#), which waits until the next time [EvtGetEvent](#) is called. 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`
- `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:

```
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.

Sound

The Palm OS platform device has primitive sound generation. A square wave is generated directly from the 68328's PWM circuitry. There is frequency, duration, and volume control. Additionally, Palm OS 3.0 and higher support creating and playing standard MIDI sounds.

The Palm OS sound manager provides an extendable API for playing custom sounds and system sounds, and for controlling default sound settings. Although the sound API accommodates multichannel design, the system provides only a single sound channel at present.

The sound hardware can play only one simple tone at a time through an onboard piezoelectric speaker. Note that for a particular

amplitude level, the Palm III™ device is slightly louder than its predecessors.

Single tones can be played by the [SndDoCmd](#) function and system sounds are played by the [SndPlaySystemSound](#) function. The end-user can control the amplitude of alarm sounds, game sounds, and system sounds by means of the Preferences application. System-supplied sounds include the Information, Warning, Error, Startup, Alarm, Confirmation, and Click sounds.

Palm OS 3.0 introduces support for Standard MIDI Files (SMFs), format 0. An SMF is a note-by-note description of a tune—Palm OS doesn't support sampled sound, multiple voices, or complex "instruments." You can download the SMF format specification from the <http://www.midi.org> Web site.

The alarm sounds used in the built-in Date Book application are SMFs stored in the System MIDI Sounds database and can be played by the [SndPlaySmf](#) function.

All SMF records in the System MIDI Sounds database are available to the user. Developers can add their own alarm SMFs to this database as a way to add variety and personalization to their devices. You can use the `sysFileTMidi` file type and `sysFileCSystem` creator to open this database.

Each record in the database is a single SMF, with a header structure containing the user-visible name. The record includes a song header, then a track header, followed by any number of events. The system only recognizes the `keyDown`, `keyUp` and `tempo` events in a single track; other commands which might be in the SMF are ignored. For more information, see the following:

- [Adding a Standard MIDI File to a Database](#) in this chapter.
- [SndCallbackInfoType](#) in the *Palm OS SDK Reference*.
- [SndMidiRecHdrType](#) in the *Palm OS SDK Reference*.

You can use standard MIDI tools to create SMF blocks on desktop computers, or you can write code to create them on the Palm OS device. The sample code project "RockMusic," particularly the routines in the `MakeSMF.c` file, can be helpful to see how to create an SMF programmatically.

Previous versions of Palm OS don't support SMFs or asynchronous notes; don't use the new routines or commands when the `FtrGet` function returns a system version of less than `0x03000000`. Doing so will crash your application. See the section [“The System Version Feature”](#) for more information.

Synchronous and Asynchronous Sound

The [SndDoCmd](#) function executes synchronously or asynchronously according to the operation it is to perform. The `sndCmdNoteOn` and `sndCmdFrqOn` operations execute asynchronously; that is, they are non-blocking and can be interrupted by another sound command. In contrast, the `sndCmdFreqDurationAmp` operation is synchronous and blocking (it cannot be interrupted).

The [SndPlaySmf](#) function is also synchronous and blocking; however, the Sound Manager polls the key queue periodically during playback and halts playback in progress if it finds events generated by user interaction with the screen, digitizer, or hardware-based buttons. Optionally, the caller can override this default behavior to specify that the [SndPlaySmf](#) function play the SMF to completion without being interrupted by user events.

Using the Sound Manager

Before playing custom sounds that require a volume (amplitude) setting, your code needs to discover the user's current volume settings. To do so in Palm OS 3.X, pass one of the `prefSysSoundVolume`, `prefGameSoundVolume`, or `prefAlarmSoundVolume` selectors to the [PrefGetPreference](#) function.

NOTE: See [“Sound Preferences Compatibility Information”](#) for important information regarding the correct use of sound preferences in various versions of Palm OS.

You can pass the returned amplitude information to the [SndPlaySmf](#) function as one element of a [SndSmfOptionsType](#) parameter block. Alternatively, you can pass amplitude information

to the [SndDoCmd](#) function as an element of a [SndCommandType](#) parameter block.

To execute a sound manager command, pass to the [SndDoCmd](#) function a sound channel pointer (presently, only `NULL` is supported and maps to the shared channel), a pointer to a structure of `SndCommandType`, and a flag indicating whether the command should be performed asynchronously.

To play SMFs, call the `SndPlaySMF` function. This function, which is new in Palm OS 3.0, is used by the built in Date Book application to play alarm sounds.

To play single notes, you can use either of the `SndPlaySMF` or `SndDoCmd` functions. Of course, you can use the `SndPlaySMF` function to play a single MIDI note from an SMF. You can also use the `SndDoCmd` function to play a single MIDI note by passing the `sndCmdNoteOn` command selector to this function. To specify by frequency the note to be played, pass the `sndCmdFrqOn` command selector to the `SndDoCmd` function. You can pass the `sndCmdQuiet` selector to this function to stop playback of the current note.

The system provides no specialized API for playing game sounds or alarm sounds. When an alarm triggers, the application that set the alarm must use the standard Sound Manager API to play the sound associated with that alarm. Similarly, game sounds are implemented by the game developer using any appropriate element of the Sound Manager API. Games should observe the `prefGameSoundVolume` setting, as described in the section “[Sound Preferences Compatibility Information](#).”

To play a default system sound, such as a click or an error beep, pass the appropriate system sound ID to the [SndPlaySystemSound](#) function, which will play that sound at the volume level specified by the user’s system sound preference. For the complete list of system sound IDs, see the `SoundMgr.h` file provided by the Palm OS SDK.

Adding a Standard MIDI File to a Database

To add a format 0 standard MIDI file to the system MIDI database, you can use code similar to the `AddSmfToDatabase` example function shown in the following code listing. This function returns 0 if successful, and returns a non-zero value otherwise. To use a

different database, pass different creator and type values to the [DmOpenDatabaseByTypeCreator](#) function.

Listing 8.6 AddSmfToDatabase

```
// Useful structure field offset macro
#define prvFieldOffset(type, field)
((UInt32)(amp((type*)0)->field))

// returns 0 for success, nonzero for error
Int16 AddSmfToDatabase(MemHandle smfH, Char*
trackName)
{
    Err      err = 0;
    DmOpenRef dbP;
    UInt16*   recIndex;
    MemHandle recH;
    UInt8*    recP;
    UInt8*    smfP;
    UInt32     bMidiOffset;
    UInt32     dwSmfSize;
    SndMidiRecHdrType recHdr;

    bMidiOffset = sizeof(SndMidiRecHdrType) +
                  StrLen(trackName) + 1;
    dwSmfSize = MemHandleSize(smfH);

    recHdr.signature = sndMidiRecSignature;
    recHdr.reserved = 0;
    recHdr.bDataOffset = bMidiOffset;

    dbP = DmOpenDatabaseByTypeCreator(sysFileTMidi,
sysFileCSystem,
                                     dmModeReadWrite |
dmModeExclusive);
    if (!dbP)
        return 1;

    // Allocate a new record for the midi resource
    recIndex = dmMaxRecordIndex;
```

Palm System Features

Sound

```
    recH = DmNewRecord(dbP, &recIndex, dwSmfSize +
bMidiOffset);
    if ( !recH )
        return 2;

    // Lock down the source SMF and target record
and copy the data
    smfP = MemHandleLock(smfH);
    recP = MemHandleLock(recH);

    err = DmWrite(recP, 0, &recHdr, sizeof(recHdr));
    if (!err) err = DmStrCopy(recP,
prvFieldOffset(SndMidiRecType,
    name), trackName);
    if (!err) err = DmWrite(recP, bMidiOffset, smfP,
dwSmfSize);

    // Unlock the pointers
    MemHandleUnlock(smfH);
    MemHandleUnlock(recH);

    //Because DmNewRecord marks the new record as
busy,
    // we must call DmReleaseRecord before closing
the database
    DmReleaseRecord(dbP, recIndex, 1);

    DmCloseDatabase(dbP);

    return err;
}
```

Saving References to Standard MIDI Files

To save a reference to a SMF stored in a particular database, save its record ID and the name of the database in which it is stored. Do not store the database ID between invocations of your application, because various events, such as a HotSync, can invalidate database IDs. Using an invalid database ID can crash your application.

Retrieving a Standard MIDI File From a Database

Standard MIDI Files (SMFs) are stored as individual records in a MIDI record database—one SMF per record. Palm OS defines the database type `sysFileTMidi` for MIDI record databases. The system MIDI database, with type `sysFileTMidi` and creator `sysFileCSystem`, holds multiple system alarm sounds. In addition, your applications can create their own private MIDI databases of type `sysFileTMidi` and your own creator.

To obtain a particular SMF, you need to identify the database in which it resides and the specific database record which holds the SMF data. The database record itself is always identified by record ID. The MIDI database in which it resides may be identified by name or by database ID. If you know the creator of the SMF, you can use the [SndCreateMidiList](#) utility function to retrieve this information. Alternatively, you can use the Data Manager record API functions to iterate through MIDI database records manually in search of this information.

The `SndCreateMidiList` utility function retrieves information about Standard Midi Files from one or more MIDI databases. This information is returned as a table of entries. Each entry contains the name of an SMF; its unique record ID; and the database ID and card number of the record database in which it resides.

Once you have the appropriate identifiers for the record and the database in which it resides, you need to open the MIDI database. If you have identified the database by type and creator, pass the `sysFileTMidi` type and an appropriate creator value to the [DmOpenDatabaseByTypeCreator](#) function. For example, to retrieve a SMF from the system MIDI database, pass type `sysFileTMidi` and creator `sysFileCSystem`. The `DmOpenDatabaseByTypeCreator` function returns a reference to the open database.

If you have identified the database by name, rather than by creator, you'll need to discover its database ID in order to open it. The [DmFindDatabase](#) function returns the database ID for a database specified by name and card number. You can pass the returned ID to the [DmOpenDatabase](#) function to open the database and obtain a reference to it.

Once you have opened the MIDI database, call [DmFindRecordByID](#) to get the index of the SMF record. To retrieve the record itself, pass this index value to either of the functions [DmQueryRecord](#) or [DmGetRecord](#). When you intend to modify the record, use the `DmGetRecord` function—it marks the record as busy. When you intend to use the record in read-only fashion, use the `DmQueryRecord` function—it does not mark the record as busy. You must lock the handle returned by either of these functions before making further use of it.

To lock the database record's handle, pass it to the [MemHandleLock](#) function, which returns a pointer to the locked record holding the SMF data. You can pass this pointer to the [SndPlaySmf](#) function in the `smfP` parameter to play the MIDI file.

When you've finished using the record, unlock the pointer to it by calling the [MemPtrUnlock](#) function. If you've used `DmGetRecord` to open the record for editing, you must call [DmReleaseRecord](#) to make the record available once again to other callers. If you used `DmQueryRecord` to open the record for read-only use, you need not call `DmReleaseRecord`.

Finally, close the database by calling the [DmCloseDatabase](#) function.

Sound Preferences Compatibility Information

The sound preferences implementation and API varies slightly among versions 1.0, 2.0, and 3.X of Palm OS. This section describes how to use sound preferences correctly for various versions of Palm OS.

Because versions 2.0 and 3.X of Palm OS provide backward compatibility with previous sound preference mechanisms, applications written for an earlier version of the sound preferences API will get correct sound preference information from newer versions of Palm OS. However, it is strongly recommended that new applications use the latest API.

Using Sound Preferences on All Palm OS Devices

Because the user chooses sound preference settings, your application should respect them and adhere to their values. Further, you should always treat sound preferences as read-only values.

At reset time, the sound manager reads stored preference values and caches them for use at run time. The user interface controls update both the stored preference values and the sound manager's cached values.

The [PrefSetPreference](#) function writes to stored preference values without affecting cached values. New values are read at the next system reset. The system-use-only `SndSetDefaultVolume` function updates cached values but not stored preferences. Applications should avoid modifying stored preferences or cached values in favor of respecting the user's choices for preferences.

Using Palm OS 1.0 Sound Preferences

To read sound preference values in version 1.0 of Palm OS, call the [PrefGetPreferences](#) function to obtain the data structure shown in [Listing 8.7](#). This `SystemPreferencesTypeV10` structure holds the current values of all system-wide preferences. You must extract from this structure the values of the `sysSoundLevel` and `alarmSoundLevel` fields. These values are the only sound preference information that Palm OS version 1.0 provides.

Each of these fields holds a value of either `s1On` (on) or `s1Off` (off). Your code must interpret the values read from these fields as an indication of whether those volumes should be on or off, then map them to appropriate amplitude values to pass to Sound Manager functions: map the `s1On` selector to the `sndMaxAmp` constant (defined in `SoundMgr.h`) and map the `s1Off` selector to the value 0 (zero).

Listing 8.7 `SystemPreferencesTypeV10` data structure

```
typedef struct {
    UInt16 version; // Version of preference info

    // International preferences
    CountryType country; // Country the device is in
```

Palm System Features

Sound

```
DateFormatType dateFormat;// Format to display
date in
DateFormatType longDateFormat;// Format to
display date in
UInt8 weekStartDay;// Sunday or Monday
TimeFormatType timeFormat;// Format to display
time in
NumberFormatType numberFormat;// Format to
display numbers in

// system preferences
UInt8 autoOffDuration;// Time period before
shutting off
SoundLevelTypeV20 sysSoundLevel;//error beeps
SoundLevelTypeV20 alarmSoundLevel;//alarm only
Boolean hideSecretRecords;// True to not display
records with
                                // their secret bit
attribute set
Boolean deviceLocked; // Device locked until the
system
                                // password is entered
UInt16sysPrefFlags;// Miscellaneous system pref
flags copied into
                                // the global GSysPrefFlags at
boot time.
SysBatteryKind sysBatteryKind;
                                // The type of
batteries installed.
                                // This is copied
into the globals
                                // GSysbatteryKind
at boot time.

} SystemPreferencesTypeV10;
```

Using Palm OS 2.0 Sound Preferences

Version 2.0 of Palm OS introduces a new API for retrieving individual preference values from the system. You can pass any of

the selectors `prefSysSoundLevelV20`, `prefGameSoundLevelV20`, or `prefAlarmSoundLevelV20` to the [PrefGetPreference](#) function to retrieve individual amplitude preference values for alarm sounds, game sounds, or for overall (system) sound amplitude. As in Palm OS 1.0, each of these settings holds values of either `s1On` (on) or `s1Off` (off), as defined in the `Preferences.h` file. Your code must interpret the values read from these fields as an indication of whether those volumes should be on or off, then map them to appropriate amplitude values to pass to Sound Manager functions: map the `s1On` selector to the `sndMaxAmp` constant (defined in `SoundMgr.h` file) and map the `s1Off` selector to the value 0 (zero).

For a complete listing of selectors you can pass to the `PrefGetPreference` function, see the `Preferences.h` file.

Using Palm OS 3.X Sound Preferences

Palm OS version 3.X enhances the resolution of sound preference settings by providing discrete amplitude levels for games, alarms, and the system overall. As usual, do not set preferences yourself, but treat them as read-only values indicating the proper volume level for your application to use.

Palm OS 3.X defines the new sound amplitude selectors `prefSysSoundVolume`, `prefGameSoundVolume`, and `prefAlarmSoundVolume` for use with the [PrefGetPreference](#) function. The values this function returns for these selectors are actual amplitude settings that may be passed directly to Sound Manager functions.

NOTE: The amplitude selectors used in previous versions of Palm OS (all ending with the `Level` suffix, such as `prefGameSoundLevel`) are obsoleted in version 3.0 of Palm OS and replaced by new selectors. The old selectors remain available in Palm OS 3.X to ensure backward compatibility and are suffixed `V20` (for example, `prefGameSoundLevelV20`).

Ensuring Sound Preferences Compatibility

For greatest compatibility with multiple versions of the sound preferences mechanism, your application should condition its sound preference code according to the version of Palm OS on which it is running. See [“The System Version Feature”](#) for more information.

When your application is launched, it should retrieve the system version number and save the results in its global variables (or equivalent structure) for use elsewhere. If the major version number is 3 (three) or greater, then use the 3.0 mechanism for obtaining sound amplitude preferences, since this reflects the user’s selection most accurately. If the major version number is 2 (two), then use the 2.0 mechanism described in [“Using Palm OS 2.0 Sound Preferences.”](#) If it is 1 (one), then use the 1.0 mechanism described in [“Using Palm OS 1.0 Sound Preferences.”](#)

Avoid calling new APIs (including new selectors) when running on older versions of Palm OS that do not implement them. In particular, note that violating any of the following conditions will cause your application to crash:

- Do not call either of the [SndPlaySmf](#) or [SndCreateMidiList](#) functions on versions of Palm OS prior to 3.0.
- Do not pass any selector other than `sndCmdFreqDurationAmp` to the [SndDoCmd](#) function on versions of Palm OS prior to 3.0.

System Boot and Reset

Any reset is normally performed by sticking a bent-open paper clip or a large embroidery needle into the small hole in the back of the device. This hole, known as the “reset switch” is above and to the right of the serial number sticker (on Palm III devices). 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 device 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 device 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 booting the Palm OS device. It calls [SysReset](#) to reset the device. 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 OS device, the Sync application automatically resets the device after the sync is completed to allow the extension to install itself.

The `SysColdBoot` call is similar, but even more dangerous. It performs a hard reset that clears all user storage RAM on the device, destroying all user data.

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 device 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 device 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 device 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 device enters this mode when there is no user activity for a number of minutes or when the user presses the off button. The device 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 device 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 device 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 device 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 device 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 OS platform device is kept out of running mode as much as possible. Any interrupt generated on the device 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.

Palm System Features

Retrieving the ROM Serial Number

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 devices, beginning with the Palm III product, hold a 12-digit serial number that identifies the device uniquely. (Earlier devices 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 8.8](#) retrieves the ROM serial number, calculates the checksum, and draws both on the screen at a specified location. If the device 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 8.8 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;
```

Palm System Features

Time

```
// 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 OS platform device 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 device 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 OS device.

- 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 OS device 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 OS device maintains a tick count that starts at 0 when the device is reset. This tick increments

- 100 times per second when running on the Palm OS device
- 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

Palm OS 1.0 provided 16-bit floating point arithmetic. Instead of using standard mathematical symbols, you called functions like `Fp1Add`, `Fp1Sub`, and so on.

Palm OS 2.0 and later implements floating point arithmetic differently than Palm OS 1.0 did. The floating-point library in OS versions 2.0 and later provides 32-bit and 64-bit floating point arithmetic.

Using Floating Point Arithmetic

To take advantage of the floating-point library, applications can now use the mathematical symbols `+` `-` `*` `/` instead of using functions like `Fp1Add`, `Fp1Sub`, etc.

When compiling the application, you have to link in the floating point library under certain circumstances. Choose from one of these options:

- **Simulator application or application for 1.0 device** — link in the floating point library explicitly.

This library adds approximately 8KB to the size of your `prc` file. The library provides 32-bit and 64-bit floating-point arithmetic. The original Palm OS `Fp1` functions provided only 16-bit floating-point arithmetic. Linking in the library explicitly won't cause problems when you compile for a 2.0 or later device.

- **2.0 or later Palm OS device**—It's not necessary to link in the library.

The compiler generates trap calls to equivalent floating-point functionality in the system ROM.

There are control panel settings in the IDE which let you select the appropriate floating-point model.

Floating-point functionality is identical in either method.

Using 1.0 Floating-Point Functionality

The original `Fp1` calls (documented in the chapter “[Float Manager](#)” in the *Palm OS SDK Reference*) are still available. They may be useful for applications that don’t need high precision, don’t want to incur the size penalty of the float library, and want to run on 1.0 devices only. To get 1.0 behavior, use the 1.0 calls (`Fp1Add`, etc.) and don’t link in the library.

Summary of System Features

Alarm Manager Functions

AlmSetAlarm	AlmGetAlarm
AlmSetProcAlarm	AlmGetProcAlarm

Feature Manager Functions

FtrGet	FtrGetByIndex
FtrSet	FtrUnregister
FtrPtrNew	FtrPtrFree
FtrPtrResize	

Notification Manager Functions

SysNotifyRegister	SysNotifyUnregister
SysNotifyBroadcast	SysNotifyBroadcastDeferred

Sound Manager Functions

SndCreateMidiList	SndDoCmd
SndGetDefaultVolume	SndPlaySmf
SndPlaySystemSound	SndPlaySmfResource

Palm System Features

Summary of System Features

System Manager Functions

Launching Applications

[SysAppLaunch](#)

[SysUIAppSwitch](#)

[SysBroadcastActionCode](#)

System Dialogs

[SysGraffitiReferenceDialog](#)

[SysKeyboardDialog](#)

[SysKeyboardDialogV10](#)

Power Management

[SysBatteryInfo](#)

[SysBatteryInfoV20](#)

[SysSetAutoOffTime](#)

[SysTaskDelay](#)

System Management

[SysLibFind](#)

[SysLibLoad](#)

[SysRandom](#)

[SysReset](#)

[SysGremlins](#)

Working With Strings and Resources

[SysBinarySearch](#)

[SysInsertionSort](#)

[SysQSort](#)

[SysCopyStringResource](#)

[SysCreatePanelList](#)

[SysStringByIndex](#)

[SysFormPointerArrayToStrings](#)

Database Support

[SysCreateDataBaseList](#)

[SysCurAppDatabase](#)

Error Handling

[SysErrString](#)

Event Handling

[SysHandleEvent](#)

System Information

[SysGetOSVersionString](#)

[SysGetStackInfo](#)

[SysGetROMToken](#)

[SysTicksPerSecond](#)

Time Manager Functions

Allowing User to Change Date and Time

<u>DayHandleEvent</u>	<u>SelectTimeV33</u>
<u>SelectDay</u>	<u>SelectDayV10</u>

Changing the Date

<u>DateAdjust</u>	<u>TimAdjust</u>
<u>TimSetSeconds</u>	

Converting to Date Format

<u>DateDaysToDate</u>	<u>DateSecondsToDate</u>
<u>TimSecondsToDateTime</u>	

Converting Dates to Other Formats

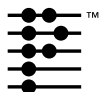
<u>DateToAscii</u>	<u>TimeToAscii</u>
<u>DateToDays</u>	<u>DateToDOWDMFormat</u>
<u>TimGetSeconds</u>	<u>TimDateTimeToSeconds</u>
<u>TimGetTicks</u>	

Date Information

<u>DayOfMonth</u>	<u>DayOfWeek</u>
<u>DaysInMonth</u>	

Float Manager Functions

<u>FplAdd</u>	<u>FplAToF</u>
<u>FplBase10Info</u>	<u>FplDiv</u>
<u>FplFloatToLong</u>	<u>FplFloatToULong</u>
<u>FplFree</u>	<u>FplFToA</u>
<u>FplInit</u>	<u>FplLongToFloat</u>
<u>FplMul</u>	<u>FplSub</u>



Serial Communication

The Palm OS[®] serial communications software provides high-performance serial communications capabilities, including byte-level serial I/O, best-effort packet-based I/O with CRC-16, reliable data transport with retries and acknowledgments, connection management, and modem dialing capabilities.

This chapter helps you understand the different parts of the serial communications system and explains how to use them, discussing these topics:

- [Serial Hardware](#) describes the serial port hardware.
- [Byte Ordering](#) briefly explains the byte order used for all data.
- [Serial Communications Architecture Hierarchy](#) provides an overview of the hierarchy, including an illustration.
- [The Serial Manager](#) and the [The New Serial Manager](#) are responsible for byte-level serial I/O and control of the RS232 signals.
- [The Connection Manager](#) allows other applications to access, add, and delete connection profiles contained in the Connection preferences panel.
- [The Serial Link Protocol](#) provides an efficient mechanism for sending and receiving packets.
- [The Serial Link Manager](#) is the Palm OS implementation of the serial link protocol.

Serial Hardware

The Palm OS[®] platform device serial port is used for implementing desktop PC connectivity or other external communication. The serial communication is fully interrupt-driven for receiving data.

Serial Communication

Byte Ordering

Currently, interrupt-driven transmission of data is not implemented in software, but the hardware does support it. Five external signals are used for this communication:

- SG (signal ground)
- TxD (transmit data)
- RxD (receive data)
- CTS (clear to send)
- RTS (request to send)

The Palm OS platform device has an external connector that provides:

- Five serial communication signals
- General-purpose output
- General-purpose input
- Cradle button input

Palm, Inc. publishes information designed to assist hardware developers in creating devices to interface with the serial communications port on Palm OS platform products. You can obtain this information by joining the Solution Provider Program and enrolling in the Serial Port & Modem Casing Program. For more information about this program and the serial port hardware, see the Palm developer web page at:

<http://www.palm.com/devzone/hw.html>.

Byte Ordering

By convention, all data coming from and going to the Palm OS device use Motorola byte ordering. That is, data of compound types such as UInt16 (2 bytes) and UInt32 (4 bytes), as well as their integral counterparts, are packaged with the most-significant byte at the lowest address. This contrasts with Intel byte ordering.

Serial Communications Architecture Hierarchy

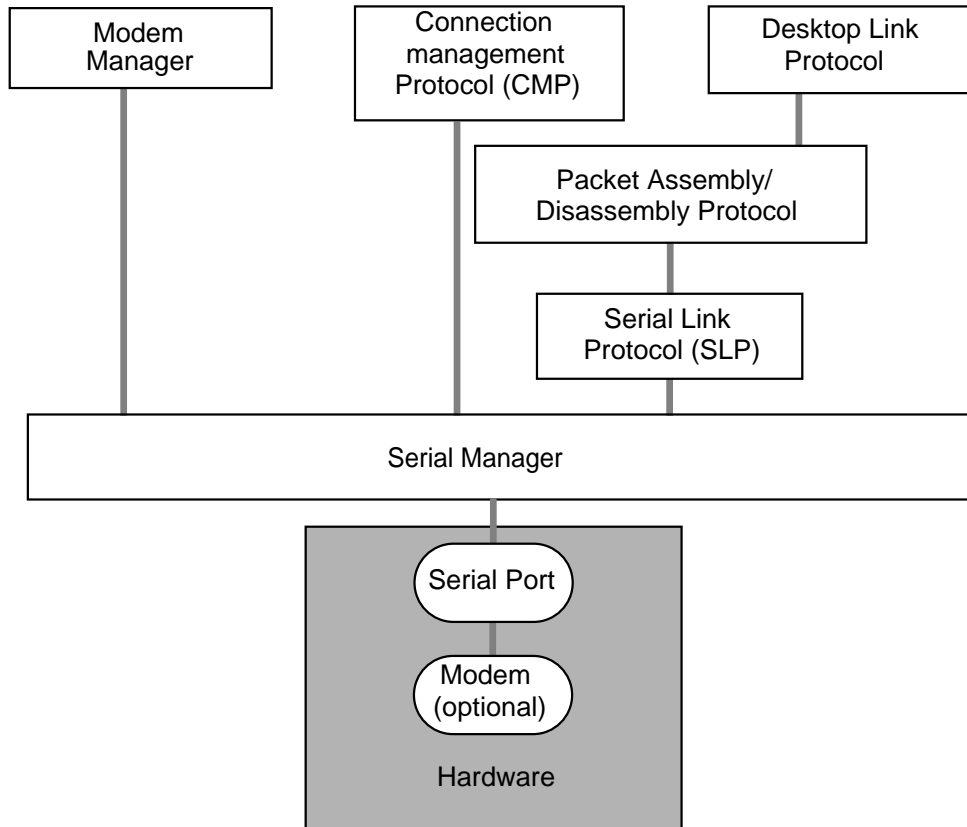
The serial communications software has multiple layers. Higher layers depend on more primitive functionality provided by lower layers. Applications can use functionality of all layers. The software consists of the following layers, described in more detail below:

- The serial manager, at the lowest layer, deals with the Palm device serial port and control of the RS232 signals, providing byte-level serial I/O. See [The Serial Manager](#).
- The modem manager provides modem dialing capabilities.
- The Serial Link Protocol (SLP) provides best-effort packet send and receive capabilities with CRC-16. Packet delivery is left to the higher-level protocols; SLP does not guarantee it. See [The Serial Link Protocol](#).
- The Packet Assembly/Disassembly Protocol (PADP) sends and receives buffered data. PADP is an efficient protocol featuring variable-size block transfers with robust error checking and automatic retries. Applications don't need access to this part of the system.
- The Connection Management Protocol (CMP) provides connection-establishment capabilities featuring baud rate arbitration and exchange of communications software version numbers.
- The Desktop Link Protocol (DLP) provides remote access to Palm OS data storage and other subsystems.

DLP facilitates efficient data synchronization between desktop (PC, Macintosh, etc.) and Palm OS applications, database backup, installation of code patches, extensions, applications, and other databases, as well as Remote Interapplication Communication (RIAC) and Remote Procedure Calls (RPC).

Figure 9.1 illustrates the communications layers.

Figure 9.1 Palm OS Serial Communications Architecture



The Serial Manager

The Palm OS serial manager is responsible for byte-level serial I/O and control of the RS232 signals.

In order to prolong battery life, the serial manager must be very efficient in its use of processing power. To reach this goal, the serial manager receiver is interrupt-driven. In the present implementation, the serial manager uses the polling mode to send data.

Using the Serial Manager

Before using the serial manager, call [SysLibFind](#), passing "Serial Library" for the library name to get the serial library reference number. This reference number is used with all subsequent serial manager calls. The system software automatically installs the serial library during system initialization.

To open the serial port, call [SerOpen](#), passing the serial library reference number (returned by `SysLibFind`), 0 (zero) for the port number, and the desired baud rate. An error code of 0 (zero) or `serErrAlreadyOpen` indicates that the port was successfully opened.

If the serial port is already open when `SerOpen` is called, the port's open count is incremented and an error code of `serErrAlreadyOpen` is returned. This ability to open the serial port multiple times allows cooperating tasks to share the serial port.

All other applications must refrain from sharing the serial port and close it by calling [SerClose](#) when `serErrAlreadyOpen` is returned. Error codes other than 0 (zero) or `serErrAlreadyOpen` indicate failure. The application must open the serial port before making other serial manager calls.

To close the serial port, call `SerClose`. Every successful call to `SerOpen` must eventually be paired with a call to `SerClose`. Because an open serial port consumes more energy from the device's batteries, it is essential not to keep the port open any longer than necessary.

To change serial port settings, such as the baud rate, CTS timeout, number of data and stop bits, parity options, and handshaking options, call [SerSetSettings](#). For baud rates above 19200, use of hardware handshaking is advised.

To retrieve the current serial port settings, call [SerGetStatus](#).

To retrieve the current line error status, call [SerGetStatus](#), which returns the cumulative status of all line errors being monitored. This includes parity, hardware and software overrun, framing, break detection, and handshake errors.

To reset the serial port error status, call [SerClearErr](#), which resets the serial port's line error status. Other serial manager functions,

Serial Communication

The Serial Manager

such as [SerReceive](#), immediately return with the error code `serErrLineErr` if any line errors are pending. Applications should therefore check the result of serial manager function calls and call [SerClearErr](#) if line error(s) occurred.

To send a stream of bytes, call [SerSend](#). In the present implementation, `SerSend` blocks until all data are transferred to the UART or a timeout error (if CTS handshaking is enabled) occurs. If your software needs to detect when all data has been transmitted, consider calling [SerSendWait](#).

NOTE: Both `SerSend` and `SerReceive` were enhanced in version 2.0 of the system. See the function descriptions for more information. The older versions are still available as [SerSend10](#) and [SerReceive10](#).

To wait until all data queued up for transmission has been transmitted, call `SerSendWait`. `SerSendWait` blocks until all pending data is transmitted or a CTS timeout error occurs (if CTS handshaking is enabled).

To flush all bytes from the transmission queue, call [SerSendFlush](#). This routine discards any data not yet transferred to the UART for transmission.

To receive a stream of bytes from the serial port, call `SerReceive`, specifying a buffer, the number of bytes desired, and the interbyte time out. This call blocks until all the requested data have been received or an error occurs.

To read bytes already in the receive queue, call [SerReceiveCheck](#) (see below) to get the number of bytes presently in the receive queue and then call `SerReceive`, specifying the number of bytes desired. Because `SerReceive` returns immediately without any data if line errors are pending, it is important to acknowledge the detection of line errors by calling `SerClearErr`.

To wait for a specific number of bytes to be queued up in the receive queue, call [SerReceiveWait](#), passing the desired number of bytes and an interbyte timeout. This call blocks until the desired number of bytes have accumulated in the receive queue or an error occurs. The desired number of bytes must be less than the current receive

queue size. The default queue size is 512 bytes. Because this call returns immediately if line errors are pending, applications have to call `SerClearErr` to detect any line errors. See also [SerReceiveCheck](#) and [SerSetReceiveBuffer](#).

To check how many bytes are presently in the receive queue, call `SerReceiveCheck`.

To discard all data presently in the receive queue and to flush bytes coming into the serial port, call [SerReceiveFlush](#), specifying the interbyte timeout. This call blocks until a time out occurs waiting for the next byte to arrive.

To replace the default receive queue, call [SerSetReceiveBuffer](#), specifying the pointer to the buffer to be used for the receive queue and its size. The default receive queue must be restored before the serial port is closed. To restore the default receive queue, call `SerSetReceiveBuffer`, passing 0 (zero) for the buffer size. The serial manager does not free the custom receive queue.

To avoid having the system go to sleep while it's waiting to receive data, an application should call [EvtResetAutoOffTimer](#) periodically. For example, the serial link manager automatically calls `EvtResetAutoOffTimer` each time a new packet is received. Note that this facility is not part of the serial manager but part of the event manager. For more information, see "[Auto-Off Control](#)" on page 74.

To perform a control function, applications can call [SerControl](#). This function performs one of the control operations specified by `SerCtlEnum`, whose elements are described in [Table 9.1](#).

Table 9.1 SerCtlEnum Elements

Element	Description
<code>serCtlFirstReserved = 0</code>	Reserve 0
<code>serCtlStartBreak</code>	Turn RS232 break signal on. Applications have to make sure that the break is set long enough to generate a value BREAK! valueP = 0; valueLenP = 0
<code>serCtlStopBreak</code>	Turn RS232 break signal off: valueP = 0; valueLenP = 0

Table 9.1 SerCtlEnum Elements (*continued*)

Element	Description
serCtlBreakStatus	Get RS232 break signal status (on or off): valueP = pointer to UInt16 for returning status (0 = off, != 0 = on) *valueLenP = sizeof(UInt16)
serCtlStartLocalLoopback	Start local loopback test; valueP = 0, valueLenP = 0
serCtlStopLocalLoopback	Stop local loopback test valueP = 0, valueLenP = 0
serCtlMaxBaud	valueP = pointer to UInt32 for returned baud *valueLenP = sizeof(UInt32)
serCtlHandshakeThreshold	Retrieve HW handshake threshold; this is the maximum baud rate that does not require hardware handshaking valueP = pointer to UInt32 for returned baud *valueLenP = sizeof(UInt32)
serCtlEmuSetBlockingHook	Set a blocking hook routine. WARNING! For use with the Simulator on Mac OS only. NOT SUPPORTED ON THE PALM DEVICE. valueP = pointer to SerCallbackEntryType *valueLenP=sizeof(SerCallbackEntryType) Returns the old settings in the first argument.

The New Serial Manager

The new serial manager is capable of managing multiple serial connections within a Palm device.

This section describes the new serial manager and the new capability to write serial drivers that it can use.

The new serial manager is the preferred serial manager API and the Palm OS will eventually phase out support for the original serial manager API.

NOTE: The new serial manager is not available on all Palm devices. It is available by flash ROM update on Palm III™ and upgraded PalmPilot™ devices and some later devices. Before making any new serial manager calls, you must ensure that it is present.

Checking for the New Serial Manager

Because not all Palm devices will (or even can) have the new serial manager installed, it's important that you check for its existence before making any new serial manager calls. You can check by calling [FtrGet](#) as follows:

```
err = FtrGet(sysFileCSerialMgr,  
            sysFtrNewSerialPresent, &value);
```

If the new serial manager is installed, the value parameter will be non-zero and the returned error should also be zero (for no error).

If the new serial manager is installed, it replaces the original serial manager. However, it includes a compatibility layer so that applications that use the original serial manager functions will continue to operate as expected. The compatibility layer simply translates the original serial manager calls into equivalent new serial manager functions.

If you are writing new application code, best performance is achieved by using the new serial library functions directly, assuming the new serial manager is installed on the unit on which your code is executing.

What's New About the New Serial Manager

The main difference between the new serial manager and previous versions is that the new serial manager supports multiple physical serial hardware devices and virtual serial devices, the detailed operation of which is abstracted from the main serial management

Serial Communication

The New Serial Manager

code. Physical serial drivers manage communication with the hardware as needed, and virtual drivers manage blocks of data to be sent to some sort of block-based serial code.

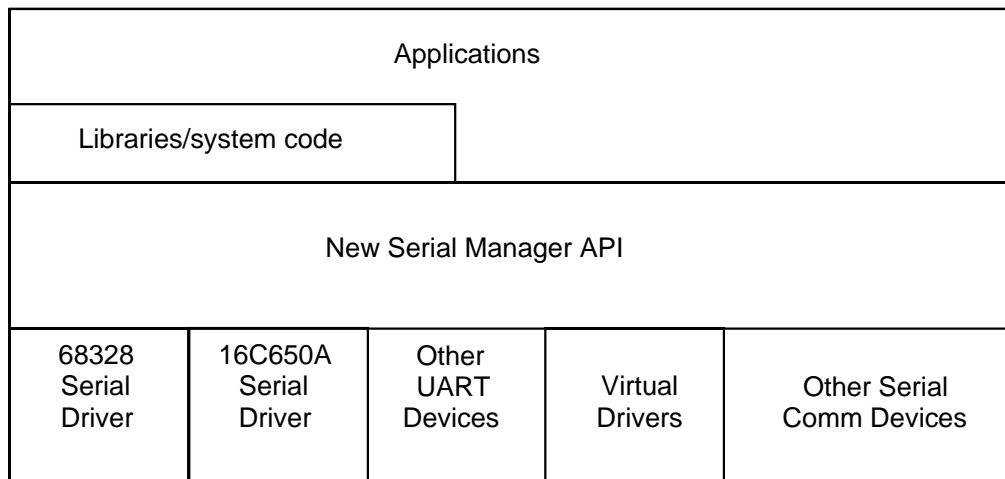
In addition to this big change, a few new functions have been added and there are widespread, minor changes to data structures and API details.

About the New Serial Manager

The new serial manager manages multiple serial devices with minimal duplication of hardware drivers and data structures. In older Palm systems, the serial library managed any and all connections to the serial hardware in the 68328 (Dragonball) processor, which was the only serial device in the system. Newer systems contain additional serial devices, such as an IR port.

The figure below shows the layering of communication software with the new serial manager and hardware drivers.

Figure 9.2 Serial Communications Architecture with New Serial Manager



The new serial manager maintains a database of installed hardware and currently open connections. Applications, libraries, or other serial communication tasks open different pieces of serial hardware by specifying a logical port number or a four-character code identifying the exact piece of serial hardware that a task wishes to open a connection with. The new serial manager then performs the proper actions on the hardware via small hardware drivers that are opened dynamically when the port is needed. One hardware driver is needed for each serial communication hardware device available to the Palm unit.

At system restart, the new serial manager searches for all serial drivers on the Palm device. Serial drivers are independent .prc files with a code resource and a version resource and are of type 'sdrv' or 'vdrv'. Once a driver is found, it is asked to locate its associated hardware and provide information on the capabilities of that hardware. This is done for each driver found and the new serial manager always maintains a list of hardware currently on the device.

Once a port is opened, the new serial manager allocates a structure for maintaining the current information and settings of the particular port. The task or application that opens the port is returned a port ID and must supply the port ID to refer to this port when other new serial manager functions are called.

Upon closing the port, the new serial manager deallocates the open port structure and unlocks the driver code resource to prevent memory fragmentation.

Note that applications can use the connection manager to obtain the proper port ID and other serial port parameters that the user has stored in connection profiles for different connection types. For more information, see the section "[The Connection Manager](#)" on page 254.

Using the New Serial Manager

The new serial manager is installed when the device is booted. Upon opening a new serial manager connection, the calling application receives a unique ID that must be used to refer to this specific connection for all subsequent calls to the new serial manager.

Opening a Connection

Opening a serial connection requires that the application enable the serial hardware by calling the [SrmOpen](#) function and specifying the port ID (logical number or port name) and the initial baud rate of the UART.

The `SrmOpen` call returns a unique port ID for the open port. This port ID is required to perform any other new serial manager functions. If the returned port ID is `NULL` or an error is returned by the `SrmOpen` function, the returned port ID should be considered invalid. Once the `SrmOpen` call is made successfully, it indicates that the new serial manager has successfully allocated internal structures to maintain the port and has successfully loaded the serial driver for this port.

A port may be opened with either a foreground connection (`SrmOpen`) or background connection ([SrmOpenBackground](#)). A foreground connection makes an active connection to the port and controls usage of the port until the connection is closed. A background connection opens the port but relinquishes control to any other task requesting a foreground connection. Background connections are provided to support tasks (such as a keyboard driver) that want to use a serial device to receive data only when no other task is using the port.

Note that background ports have limited functionality: they can only receive data and notify owning clients of what data has been received.

Specifying the portID Parameter

With the new serial manager, ports must be specified using one of the following methods:

- Logical ports

These ports are hardware independent. The OS will map them to the correct physical port. It is better to use logical

ports instead of the physical ports described below. The logical ports are:

\$8000 – Cradle Port, RS-232 serial

\$8001 – IR Port (This is a raw IrDA port with no protocol support)

\$800*n* – reserved for future types of ports

- **Physical ports**

These are 4-character constants ('*uxxx*') that reference the physical hardware of the device. It is usually not a good idea to use these ports because the hardware they reference may not exist on a particular device.

'*u328*' specifies the cradle port using the 68328 UART. This port can be switched between RS232 and raw IrDA mode using the `SerControl` call.

'*u650*' specifies the IR port on the upgrade card for Palm Personal or Palm Professional devices. This will give you a raw IR port like port \$8001, but it only exists on devices that have the upgrade card.

- **Virtual ports**

These ports are not tied to specific hardware; they simulate a hardware interface.

'*ircm*' specifies the IRComm virtual port. This gives you a virtual serial cable over an IrDA link using the IrComm Protocol. It can only be used to talk to another IrComm device.

Note that other 4-character codes for the physical and virtual ports will be added in the future. Also note that the port IDs are 4-character constants, not strings. Therefore, they are enclosed in single quotes (' '), not double quotes (" ").

Closing a Connection

Once an application is finished with the serial port, it must close it using the [SrmClose](#) function. If `SrmClose` returns no error, it indicates that the new serial manager has successfully closed the

driver and deallocated the data structures used for maintaining the port.

Sending and Receiving Data

Sending data is performed synchronously (for example, the process of writing bytes to the serial hardware's transmit FIFO). To send data, the application only needs to have an open connection with a port that has been configured properly and then specify a buffer to send. The larger the buffer to send, the longer the send function operates before returning to the calling application. The [SrmSend](#) function returns the actual number of bytes that were sent.

The [SrmSendCheck](#) function can be used to check and determine if the FIFO is empty. The [SrmSendWait](#) function can be used to wait for the UART to send the contents of its FIFO. The [SrmSendFlush](#) function can be used to flush remaining bytes in the FIFO that have not been sent.

Receiving data is a more involved process because it depends on the receiving application actually listening for data from the port. The [SrmReceiveWait](#) function allows the application to periodically check the serial port to see if data has been received. In this function, you specify a number of bytes to wait for and a timeout value (in ticks). When [SrmReceiveWait](#) returns, you can call [SrmReceive](#) to receive the data.

Applications should not loop indefinitely on the [SrmReceiveCheck](#) and [SrmReceiveWait](#) functions, waiting for serial data to arrive on the port, without allowing the Palm OS to obtain time to execute other tasks running in the same thread (by calling [EvtGetEvent](#) and [SysHandleEvent](#)). Virtual devices often run in the same thread as applications and this can prevent virtual devices and other serial related code from properly handling received data.

Receive Buffer Handling

Functions are provided to support directly changing or accessing the new serial manager's receive queue. This allows substitution of a larger receive buffer to replace the 512-byte default buffer and allows fast access to this buffer to reduce buffer copying. These

functions include [SrmSetReceiveBuffer](#), [SrmReceiveWindowOpen](#), and [SrmReceiveWindowClose](#).

Receive Data Notification

The [SrmSetWakeupHandler](#) and [SrmPrimeWakeupHandler](#) functions are used to install a notification function ([WakeupHandlerProc](#)) that gets called after some number of bytes are received by the new serial manager's interrupt function.

Because wakeup handlers are called during interrupt time, they cannot call any Palm OS system functions that may block the system in any way. Wakeup handlers should also be very short so as to reduce interrupt latency.

Obtaining Information about Serial Hardware

The [SrmGetDeviceCount](#) and [SrmGetDeviceInfo](#) functions can be used by applications to obtain information about all serial devices currently available to the OS. Applications can obtain the number of available serial hardware devices and then get information for those devices by iterating through the list using the [SrmGetDeviceInfo](#) call, until an error is returned.

The [SrmGetStatus](#) function can be used to get status information about the current hardware and return line errors. Typically, [SrmGetStatus](#) is called to retrieve the line errors for the port if some of the send and receive functions return a `serErrLineErr` error code. [SrmClearErr](#) clears line errors.

Handling Custom Operations

The new serial manager handles custom operations via the [SrmControl](#) function. To extend this functionality to the serial drivers, an additional set of control functions has been added (see the [SdrvControl](#) and [VdrvControl](#) functions). These are unique to the serial driver and should be called only by the new serial manager itself. This allows functions that access the hardware directly to go through the same switching mechanism in the driver for both public and private control function operation codes.

New Serial Manager Example

The example code in this section shows how to receive ([Listing 9.1](#)) large blocks of data using the new serial manager.

Listing 9.1 Receiving Data Using the New Serial Manager

```
#include <Pilot.h> // all the system toolbox
headers
#include <SerialMgr.h>
#define k2KBytes 2048
/
*****
*****
*
* FUNCTION: RcvSerialData
*
* DESCRIPTION: An example of how to receive a
large chunk of data
* from the Serial Manager. This function is useful
if the app
* knows it must receive all this data before
moving on. The
* YourDrainEventQueue() function is a chance for
the application
* to call EvtGetEvent and handle other application
events.
* Receiving data whenever it's available during
idle events
* might be done differently than this sample.
*
* PARAMETERS:
* thePort -> valid portID for an open serial port.
* rcvDataP -> pointer to a buffer to put the
received data.
* bufSize <-> pointer to the size of rcvBuffer and
returns
* the number of bytes read.
*
```



```
*****
*****/
Err RcvSerialData(UInt16 thePort, UInt8 *rcvDataP,
UInt32 *bufSizeP)
{
    UInt32 bytesLeft, maxRcvBlkSize, bytesRcvd,
    waitTime, totalRcvBytes = 0;
    UInt8 *newRcvBuffer;
    UInt16 dataLen = sizeof(UInt32);
    Err* error;

    // The default receive buffer is only 512 bytes;
    increase it if
    // necessary. The following lines are just an
    example of how to
    // do it, but its necessity depends on the
    ability of the code
    // to retrieve data in a timely manner.
    newRcvBuffer = MemPtrNew(k2KBytes); // Allocate
    new rcv buffer.
    if (newRcvBuffer)
        // Set new rcv buffer.
        error = SrmSetReceiveBuffer(thePort,
        newRcvBuffer, k2KBytes);
        if (error)
            goto Exit;
        else
            return memErrNotEnoughSpace;

    // Initialize the maximum bytes to receive at
    one time.
    maxRcvBlkSize = k2KBytes;
    // Remember how many bytes are left to receive.
    bytesLeft = *bufSizeP;
    // Only wait 1/5 of a second for bytes to
    arrive.
    waitTime = SysTicksPerSecond() / 5;
```

Serial Communication

The New Serial Manager

```
// Now loop while getting blocks of data and
filling the buffer.
do {
    // Is the max size larger then the number of
bytes left?
    if (bytesLeft < maxRcvBlkSize)
        // Yes, so change the rcv block amount.
        maxRcvBlkSize = bytesLeft;
    // Try to receive as much data as possible,
    // but wait only one second for it.
    bytesRcvd = SrmReceive(thePort, rcvDataP,
maxRcvBlkSize, waitTime, &error);
    // Remember the total number of bytes
received.
    totalRcvBytes += bytesRcvd;
    // Figure how many bytes are left to receive.
    bytesLeft -= bytesRcvd;
    rcvDataP += bytesRcvd; // Advance the
rcvDataP.
    // If there was a timeout and no data came
through...
    if ((error == serErrTimeout) && (bytesRcvd ==
0))
        goto Exit; // ...bail out and report the
error.
    // If there's some other error, bail out.
    if ((error) && (error != serErrTimeout))
        goto Exit;

    // Call a function to handle any pending
events because
    // someone might press the cancel button.
    // YourDrainEventQueue();
    // Continue receiving data until all data has
been received.
} while (bytesLeft);

// Clearing the receive buffer can also be done
right before
// the port is to be closed.
```

```
// Set back the default buffer when we're done.
SrmSetReceiveBuffer(thePort, 0L, 0);
MemPtrFree(newRcvBuffer); // Free the space.

Exit:
    *bufSizeP = totalRcvBytes;
    return error;
}
```

Writing a Serial or Virtual Device Driver

The new serial manager supports the ability to add other serial hardware device drivers to the system. It also supports adding virtual device drivers, which transmit and receive data in blocks, instead of a byte at a time. The following sections discuss writing serial and virtual device drivers, which are installed as code resources on the Palm device.

Serial Driver (sdrv) Code Resources

A serial driver (sdrv) is a code resource (ID = 0) that is independently compiled and installed on a Palm device. It provides a hardware abstraction layer (HAL) for the serial hardware (the UART). Serial driver .prc files are of file type 'sdrv' and their creator type is chosen by the developer (and must be registered with Palm, Inc.) to denote the type of hardware (for example, the 68328 UART driver has creator 'u328'). When the new serial manager is installed, it searches the database manager for code resources of the 'sdrv' file type and then calls the driver's entry point function to determine if the hardware that the driver supports is present and, if so, to get information about the features and capabilities of the hardware.

NOTE: Creator types with all lowercase letters are reserved by Palm, Inc. For more information about assigning and registering creator types, see "[Assigning a Creator ID](#)" on page 29.

Serial drivers are responsible for installing and removing their interrupt handlers. In addition, they must be aware of other hardware that may share the IRQ line and be sure to pass along the

interrupt to other installed handlers, if required. See the [SdrvOpen](#) function for details.

Serial Driver Functions

There are eight functions that each serial driver must minimally support in order to work with the new serial manager. These functions are briefly described in this section. For details on the exact operations each function must perform, see the function descriptions in the *Palm OS SDK Reference*.

The functions a serial driver must implement include:

- [DrvEntryPoint](#) must be the first function defined in a serial driver code resource and must be marked as the `__Startup__` function of the code resource. When the code resource is loaded, the new serial manager jumps to the beginning of the code resource and begins execution at `DrvEntryPoint`. This function is called at system restart, when the new serial manager is building a database of installed drivers and their capabilities, and when a serial port is opened.
- The [SdrvOpen](#) function is responsible for initializing the serial hardware to send and receive data, and installing an interrupt handler.
- The [SdrvClose](#) function must handle all activities needed to power-down the UART and remove the interrupt handler.
- [SdrvControl](#) extends the `SrmControl` function to the level of the hardware.
- [SdrvStatus](#) returns a bitfield that describes the current state of the UART.
- [SdrvWriteChar](#) writes a byte to the appropriate UART register for transmission.
- [SdrvReadChar](#) reads a byte (if available) from the receive FIFO of the UART. It's best to implement the `SdrvReadChar` function in assembly language.
- The [SdrvISP](#) function is called when a hardware interrupt is generated on the IRQ line associated with the serial hardware. It determines if the interrupt is for this particular serial hardware. If so, it calls the `saveDataProc` function (passed to `SdrvOpen`), which handles reading the data from

the UART by calling the `SdrvReadChar` function. It's best to implement the `SdrvISP` function in assembly language.

Virtual Driver (vdrv) Code Resources

A Virtual Driver is a code resource (ID=0) that is independently compiled and installed on a Palm device. Virtual driver .prc files are of file type 'vdrv' and their creator type is chosen by the developer (and must be registered with Palm, Inc.). When the new serial manager is installed, it searches the database manager for code resources of the 'vdrv' type and then calls the driver's entry point function to get information about the features and capabilities of this virtual device. Unlike serial device drivers, virtual device drivers send and receive data in blocks instead of transferring one byte at a time. Their purpose is to abstract a level of communication protocol away from serial devices without forcing applications to work through a different API than the serial manager that may already be used for normal RS-232 serial communication.

Virtual Driver Functions

There are six functions that each virtual driver must minimally support in order to work with the new serial manager. These functions are briefly described in this section. For details on the exact operations each function must perform, see the function descriptions in the *Palm OS SDK Reference*.

The functions a virtual driver must implement include:

- [DrvEntryPoint](#) must be the first function defined in a virtual driver code resource and must be marked as the `__Startup__` function of the code resource. When the code resource is loaded, the new serial manager jumps to the beginning of the code resource and begins execution at `DrvEntryPoint`. This function is called at system restart, when the new serial manager is building a database of installed drivers and their capabilities, and when a virtual port is opened.
- The [VdrvOpen](#) function is responsible for initializing the virtual device to begin communication.
- The [VdrvClose](#) function must handle all activities needed to close the virtual device.

Serial Communication

The Connection Manager

- [VdrvControl](#) extends the `SrmControl` function to the level of the virtual device.
- [VdrvStatus](#) returns a bitfield that describes the current state of the virtual device.
- [VdrvWrite](#) writes a block of bytes to the virtual device.

Note that there is no virtual read function in the current implementation. Virtual devices must save received data by using the functions provided in the [DrvRcvQType](#) when they are notified that data is available via some callback mechanism.

The Connection Manager

The connection manager allows other applications to access, add, and delete connection profiles contained in the Connection preferences panel. The Connection panel replaces the original Modem panel on the Palm device. A connection profile includes information on the hardware port to be used for a particular connection and the port details (speed, flow control, modem initialization string, etc.).

Because there are many more connection choices available to users (serial cable, IR, modem, network, etc.), the connection manager was developed to manage connection profiles that save preferences for various connection types.

The connection manager provides functions that list the saved connection profiles ([CncGetProfileList](#)), return details for a specific profile ([CncGetProfileInfo](#)), add a profile ([CncAddProfile](#)), and delete a profile ([CncDeleteProfile](#)).

NOTE: The connection manager is not available on all Palm devices. It is available by flash ROM update on Palm III and upgraded PalmPilot devices and some later devices. Before making any connection manager calls, you must ensure that it is present.

Because not all Palm devices will (or even can) have the connection manager installed, it's important that you check for its existence before making any connection manager calls. You can check by checking for the existence of the new serial manager, as described in the section "[Checking for the New Serial Manager](#)" on page 241. These managers work together and so are always installed together.

The Serial Link Protocol

The Serial Link Protocol (SLP) provides an efficient packet send and receive mechanism that is used by the Palm desktop software and debugger. SLP provides robust error detection with CRC-16. SLP is a best-effort protocol; it does not guarantee packet delivery (packet delivery is left to the higher-level protocols). For enhanced error detection and implementation convenience of higher-level protocols, SLP specifies packet type, source, destination, and transaction ID information as an integral part of its data packet structure.

SLP Packet Structures

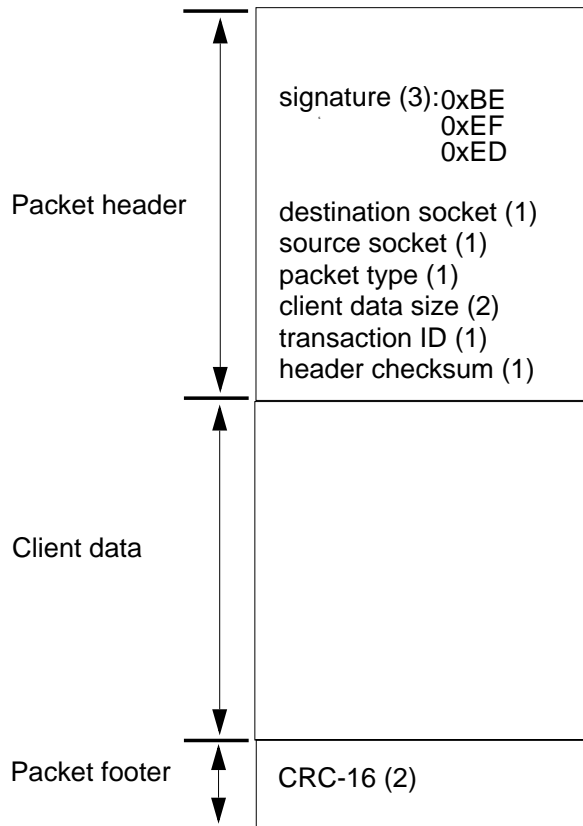
The following sections describe:

- [SLP Packet Format](#)
- [Packet Type Assignment](#)
- [Socket ID Assignment](#)
- [Transaction ID Assignment](#)

SLP Packet Format

Each SLP packet consists of a packet header, client data of variable size, and a packet footer, as shown in [Figure 9.3](#).

Figure 9.3 **Structure of a Serial Link Packet**



- The **packet header** contains the packet signature, the destination socket ID, the source socket ID, packet type, client data size, transaction ID, and header checksum. The packet signature is composed of the three bytes 0xBE, 0xEF, 0xED, in that order. The header checksum is an 8-bit arithmetic checksum of the entire packet header, not including the checksum field itself.
- The **client data** is a variable-size block of binary data specified by the user and is not interpreted by the Serial Link Protocol.
- The **packet footer** consists of the CRC-16 value computed over the packet header and client data.

Packet Type Assignment

Packet type values in the range of 0x00 through 0x7F are reserved for use by the system software. The following packet type assignments are currently implemented:

0x00	Remote Debugger, Remote Console, and System Remote Procedure Call packets.
0x02	PADP packets.
0x03	Loop-back test packets.

Socket ID Assignment

Socket IDs are divided into two categories: static and dynamic. The static socket IDs are “well-known” socket ID values that are reserved by the components of the system software. The dynamic socket IDs are assigned at runtime when requested by clients of SLP. Static socket ID values in the ranges 0x00 through 0x03 and 0xE0 through 0xFF are reserved for use by the system software. The following static socket IDs are currently implemented or reserved:

0x00	Remote Debugger socket.
0x01	Remote Console socket.
0x02	Remote UI socket.
0x03	Desktop Link Server socket.
0x04 - 0xCF	Reserved for dynamic assignment.
0xD0 - 0xDF	Reserved for testing.

Transaction ID Assignment

Transaction ID values are not interpreted by the Serial Link Protocol and are for the sole benefit of the higher-level protocols. The following transaction ID values are currently reserved:

0x00 and 0xFF	Reserved for use by the system software.
---------------	--

0x00	Reserved by the Palm OS implementation of SLP to request automatic transaction ID generation.
0xFF	Reserved for the connection manager's WakeUp packets.

Transmitting an SLP Packet

This section provides an overview of the steps involved in transmitting an SLP packet. The next section describes the implementation.

Transmission of an SLP packet consists of these steps:

1. Fill in the packet header and compute its checksum.
2. Compute the CRC-16 of the packet header and client data.
3. Transmit the packet header, client data, and packet footer.
4. Return an error code to the client.

Receiving an SLP Packet

Receiving an SLP packet consists of these steps:

1. Scan the serial input until the packet header signature is matched.
2. Read in the rest of the packet header and validate its checksum.
3. Read in the client data.
4. Read in the packet footer and validate the packet CRC.
5. Dispatch/return an error code and the packet (if successful) to the client.

The Serial Link Manager

The serial link manager is the Palm OS implementation of the Serial Link Protocol.

Serial link manager provides the mechanisms for managing multiple client sockets, sending packets, and receiving packets both

synchronously and asynchronously. It also provides support for the Remote Debugger and Remote Procedure Calls (RPC).

Using the Serial Link Manager

Before an application can use the services of the serial link manager, the application must open the manager by calling [SlkOpen](#). Success is indicated by error codes of 0 (zero) or `slkErrAlreadyOpen`. The return value `slkErrAlreadyOpen` indicates that the serial link manager has already been opened (most likely by another task). Other error codes indicate failure.

When you finish using the serial link manager, call [SlkClose](#). `SlkClose` may be called only if [SlkOpen](#) returned 0 (zero) or `slkErrAlreadyOpen`. When the open count reaches zero, `SlkClose` frees resources allocated by `SlkOpen`.

To use the serial link manager socket services, open a Serial Link socket by calling [SlkOpenSocket](#). Pass a reference number or port ID (for the new serial manager) of an opened and initialized communications library (see `SlkClose`), a pointer to a memory location for returning the socket ID, and a Boolean indicating whether the socket is static or dynamic. If a static socket is being opened, the memory location for the socket ID must contain the desired socket number. If opening a dynamic socket, the new socket ID is returned in the passed memory location. Sharing of sockets is not supported. Success is indicated by an error code of 0 (zero). For information about static and dynamic socket IDs, see [“Socket ID Assignment” on page 257](#).

When you have finished using a Serial Link socket, close it by calling [SlkCloseSocket](#). This releases system resources allocated for this socket by the serial link manager.

To obtain the communications library reference number for a particular socket, call `SlkSocketRefNum`. The socket must already be open. To obtain the port ID for a socket, if you are using the new serial manager, call [SlkSocketPortID](#).

To set the interbyte packet receive timeout for a particular socket, call [SlkSocketSetTimeout](#).

Serial Communication

The Serial Link Manager

To flush the receive stream for a particular socket, call [SlkFlushSocket](#), passing the socket number and the interbyte timeout.

To register a socket listener for a particular socket, call [SlkSetSocketListener](#), passing the socket number of an open socket and a pointer to the `SlkSocketListenType` structure. Because the serial link manager does not make a copy of the `SlkSocketListenType` structure but instead saves the pointer passed to it, the structure may not be an automatic variable (that is, allocated on the stack). The `SlkSocketListenType` structure may be a global variable in an application or a locked chunk allocated from the dynamic heap. The `SlkSocketListenType` structure specifies pointers to the socket listener procedure and the data buffers for dispatching packets destined for this socket. Pointers to two buffers must be specified:

- Packet header buffer (size of `SlkPktHeaderType`).
- Packet body buffer, which must be large enough for the largest expected client data size.

Both buffers can be application global variables or locked chunks allocated from the dynamic heap.

The socket listener procedure is called when a valid packet is received for the socket. Pointers to the packet header buffer and the packet body buffer are passed as parameters to the socket listener procedure. The serial link manager does not free the `SlkSocketListenType` structure or the buffers when the socket is closed; freeing them is the responsibility of the application. For this mechanism to function, some task needs to assume the responsibility to “drive” the serial link manager receiver by periodically calling [SlkReceivePacket](#).

To send a packet, call [SlkSendPacket](#), passing a pointer to the packet header (`SlkPktHeaderType`) and a pointer to an array of `SlkWriteDataType` structures. [SlkSendPacket](#) stuffs the signature, client data size, and the checksum fields of the packet header. The caller must fill in all other packet header fields. If the transaction ID field is set to 0 (zero), the serial link manager automatically generates and stuffs a new non-zero transaction ID. The array of `SlkWriteDataType` structures enables the caller to specify the client data part of the packet as a list of noncontiguous

blocks. The end of list is indicated by an array element with the size field set to 0 (zero). Listing 3.1 incorporates the processes described in this section.

Listing 9.2 Sending a Serial Link Packet

```
Err          err;
SlkPktHeaderType  sendHdr;
                //serial link packet header
SlkWriteDataType  writeList[2];
                //serial link write data segments
UInt8          body[20];
                //packet body(example packet body)

                // Initialize packet body
                ...

// Compose the packet header
sendHdr.dest = slkSocketDLP;
sendHdr.src = slkSocketDLP;
sendHdr.type = slkPktTypeSystem;
sendHdr.transId = 0;
                // let Serial Link Manager set the transId
// Specify packet body
writeList[0].size = sizeof(body);
                // first data block size
writeList[0].dataP = body;
                // first data block pointer
writeList[1].size = 0;
                // no more data blocks

// Send the packet
err = SlkSendPacket( &sendHdr, writeList );
    ...
}
```

Listing 9.3 Generating a New Transaction ID

```
//  
// Example: Generating a new transaction ID given  
// the previous  
// transaction ID. Can start with any seed value.  
//  
  
UInt8 NextTransactionID (UInt8  
previousTransactionID)  
{  
    UInt8 nextTransactionID;  
  
    // Generate a new transaction id, avoid the  
    // reserved values (0x00 and 0xFF)  
    if ( previousTransactionID >= (UInt8)0xFE )  
        nextTransactionID = 1;           // wrap around  
    else  
        nextTransactionID = previousTransactionID + 1;  
                                           // increment  
  
    return nextTransactionID;  
}
```

To receive a packet, call [SlkReceivePacket](#). You may request a packet for the passed socket ID only, or for any open socket that does not have a socket listener. The parameters also specify buffers for the packet header and client data, and a timeout. The timeout indicates how long the receiver should wait for a packet to begin arriving before timing out. A timeout value of (-1) means “wait forever.” If a packet is received for a socket with a registered socket listener, the packet is dispatched via its socket listener procedure.

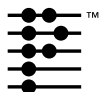
Summary of Serial Communications

Serial Manager Functions	New Serial Manager Functions
<u>SerClearErr</u>	<u>SrmClearErr</u>
<u>SerClose</u>	<u>SrmClose</u>
<u>SerControl</u>	<u>SrmControl</u>
<u>SerGetSettings</u>	<u>SrmGetDeviceCount</u>
<u>SerGetStatus</u>	<u>SrmGetDeviceInfo</u>
<u>SerOpen</u>	<u>SrmGetStatus</u>
<u>SerReceive</u>	<u>SrmOpen</u>
<u>SerReceiveCheck</u>	<u>SrmOpenBackground</u>
<u>SerReceiveFlush</u>	<u>SrmPrimeWakeupHandler</u>
<u>SerReceiveWait</u>	<u>SrmReceive</u>
<u>SerSend</u>	<u>SrmReceiveCheck</u>
<u>SerSendFlush</u>	<u>SrmReceiveFlush</u>
<u>SerSendWait</u>	<u>SrmReceiveWait</u>
<u>SerSetReceiveBuffer</u>	<u>SrmReceiveWindowClose</u>
<u>SerSetSettings</u>	<u>SrmReceiveWindowOpen</u>
	<u>SrmSend</u>
	<u>SrmSendCheck</u>
	<u>SrmSendFlush</u>
	<u>SrmSendWait</u>
	<u>SrmSetReceiveBuffer</u>
	<u>SrmSetWakeupHandler</u>
	<u>WakeupHandlerProc</u>
Serial Driver Functions	Virtual Driver Functions
<u>DrvEntryPoint</u>	<u>DrvEntryPoint</u>
<u>SdrvClose</u>	<u>GetSize</u>
<u>SdrvControl</u>	<u>GetSpace</u>
<u>SdrvISP</u>	<u>VdrvControl</u>
<u>SdrvOpen</u>	<u>VdrvOpen</u>
<u>SdrvReadChar</u>	<u>VdrvStatus</u>
<u>SdrvStatus</u>	<u>VdrvWrite</u>
<u>SdrvWriteChar</u>	<u>WriteBlock</u>
	<u>WriteByte</u>

Serial Communication

Summary of Serial Communications

Connection Manager Functions	Serial Link Manager Functions
<u>CncAddProfile</u>	<u>SlkClose</u>
<u>CncDeleteProfile</u>	<u>SlkCloseSocket</u>
<u>CncGetProfileInfo</u>	<u>SlkFlushSocket</u>
<u>CncGetProfileList</u>	<u>SlkOpen</u>
	<u>SlkOpenSocket</u>
	<u>SlkReceivePacket</u>
	<u>SlkSendPacket</u>
	<u>SlkSetSocketListener</u>
	<u>SlkSocketPortID</u>
	<u>SlkSocketSetTimeout</u>



Beaming (Infrared Communication)

The Palm OS[®] provides two levels of support for beaming, or infrared communication (IR):

- The [Exchange Manager](#) provides a high-level interface that handles all of the communication details transparently.
- The [IR Library](#) provides a low-level, direct interface to the IR communications capabilities of the Palm OS. It is designed for applications that want more direct access to the IR capabilities than the exchange manager provides.

This chapter discusses these two facilities for IR communication.

Exchange Manager

The Palm OS exchange manager provides a simple interface for Palm OS applications to send and receive typed data from any number of remote devices and protocols. The device at the remote end of a connection does not need to know it is talking to a Palm OS device. The exchange manager can be used with industry standard protocols and data formats. The burden of understanding the protocols and data formats is on the Palm OS application using the exchange manager.

The exchange manager was developed to provide a facility by which Palm OS applications could communicate directly with external devices and foreign data formats, without having to be tied to the HotSync[®] mechanism and conduits. In the increasingly complex world of the Internet, wireless communications, and infrared communications, it cannot be expected that all these modes of communication must support HotSync and provide the appropriate conduits on the other end. The Palm OS device must be able to deal directly with foreign data formats since there will not be conduits on the remote end to prepare the data. The data may also

Beaming (Infrared Communication)

Exchange Manager

be sent without regard to the version or even the existence of particular software on the device.

Overview

The exchange manager is designed as a generic communications facility by which typed data objects can be sent and received. It is designed to support a variety of underlying transport mechanisms. Currently, the exchange manager supports only the IR (beaming) capability of the Palm III™ and later devices (and upgraded PalmPilot™ devices).

NOTE: When used for IR communication, the exchange manager uses the OBEX IrDA protocol. The only level of OBEX supported currently is for the Put operation. The Palm III can act as both a client and a server.

The exchange manager API provides a mechanism for exchanging typed data objects between applications. An object is a stream of bytes with some information about its contents attached. The content information includes a creator ID, a MIME data type, and a filename. An application that wants to send data using the exchange manager must provide at least one of these pieces of information. An application that is able to receive an object registers itself with the exchange manager ([ExgRegisterData](#)) and specifies what data types and file extensions it can accept.

A key data structure used by the exchange manager is the [ExgSocketType](#) data type. This exchange socket structure defines information about the connection and the type of data to be exchanged. When you are sending data, you must supply this structure with the appropriate information filled in. When you are receiving, this structure gives you information about the connection and the incoming data. (Note that the use of the term “socket” in the exchange manager API is not related to the term “socket” as used in sockets communication programming.)

NOTE: The current implementation of the IR library does not send data type information, but it may do so in the future. It is recommended that you write information for the data type field of the socket, but do not expect to receive type information. Instead, use a filename including the extension to identify content. When registering, register for a file extension.

Exchange Manager and Launch Codes

When receiving incoming data, the exchange manager communicates with applications via launch codes. The exchange manager sends an application a series of three launch codes when it receives data for it. These are:

- [`sysAppLaunchCmdExgAskUser`](#)
- [`sysAppLaunchCmdExgReceiveData`](#)
- [`sysAppLaunchCmdGoto`](#)

The exchange manager sends the first launch code, `sysAppLaunchCmdExgAskUser`, when it has determined that incoming data is destined for a particular application (based on which application has registered to receive data of that type). This launch code lets the application tell the exchange manager whether or not to display a dialog asking the user if they want to accept the data. If the application chooses not to handle this launch command, the default course of action is that the exchange manager displays a dialog asking the user if they want to accept the incoming data. In most cases, applications won't need to handle this launch code, since the default action is the preferred alternative.

Palm OS 3.5 and higher provide an alternative version of the dialog that displays a category pop-up list so that users can file the incoming data at the same time it is received. The pop-up list is only displayed if you handle the launch code and call [`ExgDoDialog`](#) directly. See that function's description in the *Palm OS SDK Reference* for more information.

The application can respond to this launch code by setting the `result` field in the parameter block to the appropriate value. If it wants to allow the exchange manager to display a dialog, it should

Beaming (Infrared Communication)

Exchange Manager

leave the `result` field set to `exgAskDialog` (the default value). To disable display of the dialog and to automatically accept the incoming data (as if the user had pressed OK in the dialog), set the `result` field to `exgAskOk`. To disable display of the dialog and to automatically reject the incoming data (as if the user had pressed Cancel in the dialog), set the `result` field to `exgAskCancel`. In the later case, the data is discarded and no further action is taken by the exchange manager.

If the application sets the `result` field to `exgAskOk`, or the dialog is displayed and the user presses the OK button, then the exchange manager sends the application the next launch code, `sysAppLaunchCmdExgReceiveData`, so that it can actually receive the data. This launch code notifies the application that it should receive the data.

The application should use the exchange manager functions [ExgAccept](#), [ExgReceive](#), and [ExgDisconnect](#) to receive the data and store it or do whatever it needs to with the data.

The parameter block sent with this launch code is of the `ExgSocketPtr` data type. It is a pointer to the `ExgSocketType` structure corresponding to the exchange manager connection via which the data is arriving. You will need to pass this pointer to the `ExgAccept` function to begin receiving the data. Note that in the socket structure, the `length` field may not be accurate, so in your receive loop you should be flexible in handling more or less data than `length` specifies.

After you have finished receiving the data and before you return from the `PilotMain` routine, you must set up the `goToCreator` and `goToParams` fields in the socket structure. Set in the `goToCreator` field the creator ID of the application that should be launched to view the received data (normally the same application that received the data). If no application should be launched, then set this to `NULL`. Set in the `goToParams` structure information that identifies the record to go to when the application is launched. It is recommended that you use a unique ID to identify the record, rather than the record index, since indexes might change. You can put unique ID information into the `goToParams.matchCustom` field.

Note that the application may not be the active application, and thus may not have globals available when it is launched with this launch code. Be sure to check if you have globals available and don't try to access them if they are not available. In addition, if the application has multiple code segments, you cannot access code outside of segment 0 (the first segment) if the application is launched with this launch code.

Assuming that everything has proceeded normally, the exchange manager again launches the application identified in the `goToCreator` field of the socket structure with the `sysAppLaunchCmdGoto` launch code. This allows the user to view the received item.

IR Library

The IR (InfraRed) library is a shared library that provides a direct interface to the IR communications capabilities of the Palm OS. It is designed for applications that want more direct access to the IR capabilities than the exchange manager provides.

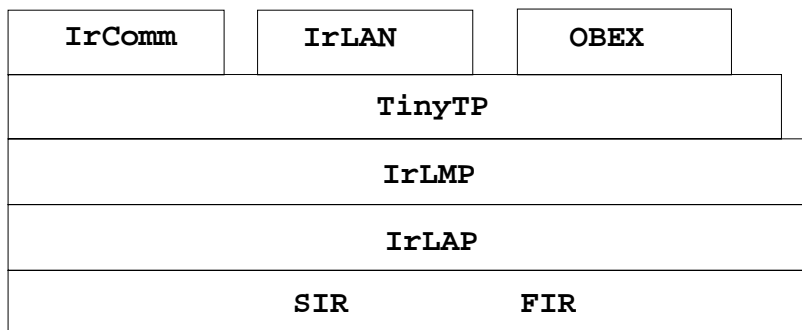
The IR support provided by the Palm OS is compliant with the IrDA specifications. IrDA (Infrared Data Association), is an industry body consisting of representatives from a number of companies involved in IR development. For a good introduction to the IrDA standards, see the IrDA web site at:

<http://www.IrDA.org>.

IrDA Stack

The IrDA stack comprises a number of protocol layers, of which some are required and some are optional. The complete stack looks something like [Figure 10.1](#).

Figure 10.1 IrDA Protocol Stack



The SIR/FIR layer is purely hardware. The SIR (Serial IR) layer supports speeds up to 115k bps while the FIR (Fast IR) layer supports speeds up to 4M bps. IrLAP is the IR Link Access Protocol that provides a data pipe between IrDA devices. IrLMP, the IR Link Management Protocol, manages multiple sessions using the IrLAP. Tiny TP is a lightweight transfer protocol on which some higher-level IrDA layers are built.

One or more of SIR/FIR must be implemented, and Tiny TP, IrLMP and IrLAP must also be implemented. IrComm provides serial and parallel port emulation over an IR link and is optional (it is not currently supported in the Palm OS). IrLAN provides an access point to Local Area Network protocol adapters. It too is optional (and is not supported in the Palm OS).

OBEX is an object exchange protocol that can be used (for instance) to transfer business cards, calendar entries or other objects between devices. It too is optional and is supported in the Palm OS. The capabilities of OBEX are made available through the exchange manager; there is no direct API for it.

The Palm OS implements all the required protocol layers (SIR, IrLAP, IrLMP, and Tiny TP), as well as the OBEX layer, to support the Exchange Manager. Palm III devices provide SIR (Serial IR) hardware supporting the following speeds: 2400, 9600, 19200, 38400, 57600, and 115200 bps. The software (IrOpen) currently limits bandwidth to 57600 bps by default, but you can specify a connection speed of up to 115200 bps if desired.

The stack is capable of connection-based or connectionless sessions.

IrLMP Information Access Service (IAS) is a component of the IrLMP protocol that you will see mentioned in the interface. IAS provides a database service through which devices can register information about themselves and retrieve information about other devices and the services they offer.

Accessing the IR Library

Before you can use the IR library, you must obtain a reference number for it by calling the function [SysLibFind](#), as in this example:

```
err = SysLibFind(irLibName, &refNum);
```

This function returns the library reference number in the `refNum` parameter. This parameter is passed to most of the other functions in the IR library.

Summary of Beaming

Exchange Manager Functions

ExgAccept	ExgPut
ExgDBRead	ExgReceive
ExgDBWrite	ExgRegisterData
ExgDisconnect	ExgSend
ExgDoDialog	

IR Library Functions

IrAdvanceCredit	IrIsNoProgress
IrBind	IrIsRemoteBusy
IrClose	IrLocalBusy
IrConnectIrLap	IrMaxRxSize

Beaming (Infrared Communication)

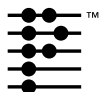
Summary of Beaming

IR Library Functions

<u>IrConnectReq</u>	<u>IrMaxTxSize</u>
<u>IrConnectRsp</u>	<u>IrOpen</u>
<u>IrDataReq</u>	<u>IrSetConTypeLMP</u>
<u>IrDisconnectIrLap</u>	<u>IrSetConTypeTTP</u>
<u>IrDiscoverReq</u>	<u>IrSetDeviceInfo</u>
<u>IrIsIrLapConnected</u>	<u>IrTestReq</u>
<u>IrIsMediaBusy</u>	<u>IrUnbind</u>

IR Library IAS Database Functions

<u>IrIAS_Add</u>	<u>IrIAS_GetUserString</u>
<u>IrIAS_GetInteger</u>	<u>IrIAS_GetUserStringCharSet</u>
<u>IrIAS_GetIntLsap</u>	<u>IrIAS_GetUserStringLen</u>
<u>IrIAS_GetObjectID</u>	<u>IrIAS_Next</u>
<u>IrIAS_GetOctetString</u>	<u>IrIAS_Query</u>
<u>IrIAS_GetOctetStringLength</u>	<u>IrIAS_SetDeviceName</u>
<u>IrIAS_GetType</u>	<u>IrIAS_StartResult</u>



Network Communication

Two different Palm OS® libraries provide network services to applications:

- The net library provides basic network services using TCP and UDP via a socket API. This library is discussed in the section [Net Library](#).
- The Internet library builds on the net library to provide a socket-like API to high-level Internet protocols such as HTTP. This library is discussed in the section [Internet Library](#).

Net Library

The net library allows Palm OS applications to easily establish a connection with any other machine on the Internet and transfer data to and from that machine using the standard TCP/IP protocols.

The basic network services provided by the net library include:

- Stream-based, guaranteed delivery of data using TCP (Transmission Control Protocol).
- Datagram-based, best-effort delivery of data using UDP (User Datagram Protocol).

You can implement higher-level Internet-based services (file transfer, e-mail, web browsing, etc.) on top of these basic delivery services.

IMPORTANT: Applications cannot directly use the net library to make wireless connections. Use the Internet library for wireless connections.

This section describes how to use the net library in your application. It covers:

- [About the Net Library](#)
- [Net Library Usage Steps](#)
- [Obtaining the Net Library's Reference Number](#)
- [Setting Up Berkeley Socket API](#)
- [Setup and Configuration Calls](#)
- [Opening the Net Library](#)
- [Closing the Net Library](#)
- [Version Checking](#)
- [Network I/O and Utility Calls](#)
- [Berkeley Sockets API Functions](#)
- [Extending the Network Login Script Support](#)

About the Net Library

The net library consists of two parts: a netlib interface and a net protocol stack.

The **netlib interface** is the set of routines that an application calls directly when it makes a net library call. These routines execute in the caller's task like subroutines of the application. They are not linked in with the application, however, but are called through the library dispatch mechanism.

With the exception of functions that open, close, and set up the net library, the net library's API maps almost directly to the Berkeley UNIX sockets API, the de facto standard API for Internet applications. You can compile an application written to use the Berkeley sockets API for the Palm OS with only slight changes to the source code.

The **net protocol stack** runs as a separate task in the operating system. Inside this task, the TCP/IP protocol stack runs, and received packets are processed from the network device drivers. The netlib interface communicates with the net protocol stack through an operating system mailbox queue. It posts requests from

applications into the queue and blocks until the net protocol stack processes the requests.

Having the net protocol stack run as a separate task has two big advantages:

- The operating system can switch in the net protocol stack to process incoming packets from the network even if the application is currently busy.
- Even if an application is blocked waiting for some data to arrive off the network, the net protocol stack can continue to process requests for other applications.

One or more network interfaces run inside the net protocol stack task. A **network interface** is a separately linked database containing code necessary to abstract link-level protocols. For example, there are separate network interface databases for PPP and SLIP. A network interface is generally specified by the user in the Network preference panel. In rare circumstances, interfaces can also be attached and detached from the net library at runtime as described in the section “[Settings for Interface Selection](#)” later in this chapter.

Constraints

Because it’s unclear whether all future platforms will need or want network support (especially devices with very limited amounts of memory), network support is an optional part of the operating system. For this reason, the net library is implemented as a system library that is installed at runtime and doesn’t have to be present for the system to work properly.

When the net library is present and running, it requires an estimated additional 32 KB of RAM. This in effect doubles the overall system RAM requirements, currently 32 KB without the net library. It’s therefore not practical to run the net library on any platform that has 128 KB or less of total RAM available since the system itself will consume 64 KB of RAM (leaving only 64 KB for user storage in a 128 KB system).

Because of the RAM requirements, the net library is supported only on PalmPilot Professional and newer devices running Palm OS 2.0 and later.

All applications written for Palm OS must pay special attention to memory and CPU usage because Palm OS runs on small devices with limited amounts of memory and other hardware resources. Applications that use the net library, therefore, must pay even more attention to memory usage. After opening the net library, the total remaining amount of RAM available to an application is approximately 12 KB on a PalmPilot Professional and 36KB on a Palm III™.

The Programmer's Interface

There are essentially two sets of API into the net library: the net library's native API, and the Berkeley sockets API. The two APIs map almost directly to each other. You can use the Berkeley sockets API with no performance penalty and little or no modifications to any existing code that you have.

The header file `<unix/sys_socket.h>` contains a set of macros that map Berkeley sockets calls directly to net library calls. The main difference between the net library API and the Berkeley sockets API is that most net library API calls accept additional parameters for:

- **A reference number.** All library calls in the Palm OS must have the library reference number as the first parameter.
- **A timeout.** In consumer systems such as the Palm OS device, infinite timeouts don't work well because the end user can't "kill" a process that's stuck. The timeout allows the application to gracefully recover from hung connections. The default timeout is 2 seconds.
- **An error code.** The sockets API by convention returns error codes in the application's global variable `errno`. The net library API doesn't rely on any application global variables. This allows system code (which cannot have global variables) to use the net library API.

The macros in `sys_socket.h` do the following:

For...	The macros pass...
reference number	<code>AppNetRefnum</code> (application global variable).

For...	The macros pass...
timeout	AppNetTimeout (application global variable).
error code	Address of the application global errno.

For example, consider the Berkeley sockets call `socket`, which is declared as:

```
Int16 socket(Int16 domain, Int16 type,
             Int16 protocol);
```

The equivalent net library call is `NetLibSocketOpen`, which is declared as:

```
NetSocketRef NetLibSocketOpen(UInt16 libRefnum,
                               NetSocketAddrEnum domain,
                               NetSocketTypeEnum type, Int16 protocol,
                               Int32 timeout, Err* errP)
```

The macro for `socket` is:

```
#define socket(domain,type,protocol) \
    NetLibSocketOpen(AppNetRefnum, domain, type,
                    protocol, AppNetTimeout, &errno)
```

Net Library Usage Steps

In general, using the net library involves the steps listed below. The next several sections describe some of the steps in more detail.

For an example of using the net library, see the example application `NetSample` in the `Palm OS Examples` directory. It exercises many of the net library calls.

- 1. Obtain the net library's reference number.**

Because the net library is a system library, all net library calls take the library's reference number as the first parameter. For this reason, your first step is to obtain the reference number and save it. See "[Obtaining the Net Library's Reference Number](#)."

- 2. Set up for using Berkeley sockets API.**

You can either use the net library's native API or the Berkeley sockets API for the majority of what you do with the net library. If

you're already familiar with Berkeley sockets API, you'll probably want to use it instead of the native API. If so, follow the steps in "[Setting Up Berkeley Socket API](#)."

3. If necessary, configure the net library the way you want it.

Typically, users set up their networking services by using the Network preferences panel. Most applications don't set up the networking services themselves; they simply access them through the net library preferences database. In rare instances, your application might need to perform some network configuration, and it usually should do so before the net library is open. See "[Setup and Configuration Calls](#)."

4. Open the net library right before the first network access.

Because of the limited resources in the Palm OS environment, the net library was designed so that it only takes up extra memory from the system when an application actually needs to use its services. An Internet application must therefore inform the system when it needs to use the net library by opening the net library when it starts up and by closing it when it exits. See "[Opening the Net Library](#)."

5. Make calls to access the network.

Once the net library has been opened, sockets can be opened and data sent to and received from remote hosts using either the Berkeley sockets API or the native net library API. See "[Network I/O and Utility Calls](#)."

6. Close the net library when you're finished with it.

Closing the net library frees up the resources. See "[Closing the Net Library](#)."

Obtaining the Net Library's Reference Number

To determine the reference number, call `SysLibFind`, passing the name of the net library, "Net.lib". In addition, if you intend to use Berkeley sockets API, save the reference number in the application global variable `AppNetRefnum`.

```
err = SysLibFind("Net.lib", &AppNetRefnum);
if (err) { /* error handling here */ }
```

Remember that the net library requires Palm OS version 2.0 or later. If the `SysLibFind` call can't find the net library, it returns an error code.

Setting Up Berkeley Socket API

To set up the use of Berkeley sockets API, do the following:

- Include the header file `<unix/sys_socket.h>`, provided with the Palm OS SDK.
- Link your project with the module `NetSocket.c`, which declares and initializes three required global variables: `AppNetTimeout`, `AppNetRefnum`, and `errno`. `NetLibSocket.c` also contains the glue code necessary for a few of the Berkeley sockets functions.
- As described in the previous section, assign the net library's reference number to the variable `AppNetRefnum`.
- Adjust `AppNetTimeout`'s value if necessary.

This value represents the maximum number of system ticks to wait before a net library call expires. Most applications should adjust this timeout value and possibly adjust it for different sections of code. The following example sets the timeout value to 10 seconds.

```
AppNetTimeout = SysTicksPerSecond() * 10;
```

Setup and Configuration Calls

The setup and configuration API calls of the net library are normally only used by the Network preferences panel. This includes calls to set IP addresses, host name, domain name, login script, interface settings, and so on. Each setup and configuration call saves its settings in the net library preferences database in nonvolatile storage for later retrieval by the runtime calls.

In rare instances, an application might need to perform setup and configuration itself. For example, some applications might allow users to select a particular “service” before trying to establish a connection. Such applications present a pick list of service names and allow the user to select a service name. This functionality is provided via the Network preferences panel. The panel provides

launch codes (defined in `SystemMgr.h`) that allow an application to present a list of possible service names to let the end user pick one. The preferences panel then makes the necessary net library setup and configuration calls to set up for that particular service.

Usually, the setup and configuration calls are made while the library is closed. A subset of the calls can also be issued while the library is open and will have real-time effects on the behavior of the library. [Chapter 54, “Net Library”](#) in *Palm OS SDK Reference*, describes the behavior of each call in more detail.

Settings for Interface Selection

As you learned in the section “[About the Net Library](#),” the net library uses one or more network interfaces to abstract low-level networking protocols. The user specifies which network interface to use in the Network preference panel.

You can also use net library calls to specify which interface(s) should be used:

- [NetLibIFAttach](#) attaches an interface to the library so that it will be used when and if the library is open.
- [NetLibIFDetach](#) detaches an interface from the library.
- [NetLibIFGet](#) returns an interface’s creator and instance number.

Unlike most net library functions, these functions can be called while the library is open or closed. If the library is open, the specific interface is attached or detached in real time. If the library is closed, the information is saved in preferences and used the next time the library is opened.

Each interface is identified by a creator and an instance number. You need these values if you want to attach or detach an interface or to query or set interface settings. You use `NetLibIFGet` to obtain this information. `NetLibIFGet` takes four parameters: the net library’s reference number, an index into the library’s interface list, and addresses of two variables where the creator and instance number are returned.

The creator is one of the following values:

- `netIFCreatorLoop` (Loopback network)

- `netIFCreatorSLIP` (SLIP network)
- `netIFCreatorPPP` (PPP network)

If you know which interface you want to obtain information about, you can iterate through the network interface list, calling `NetLibIFGet` with successive index values until the interface with the creator value you need is returned.

Interface Specific Settings

The net library configuration is structured so that network interface-specific settings can be specified for each network interface independently. These interface specific settings are called IF settings and are set and retrieved through the [NetLibIFSettingGet](#) and [NetLibIFSettingSet](#) calls.

- The [NetLibIFSettingGet](#) call takes a setting ID as a parameter along with a buffer pointer and buffer size for the return value of the setting. Some settings, like login script, are of variable size so the caller must be prepared to allocate a buffer large enough to retrieve the entire setting. (`NetLibIFSettingGet` returns the required size if you pass `NULL` for the buffer. See the `NetLibIFSettingGet` description in the reference documentation for more information.)
- The [NetLibIFSettingSet](#) call also takes a setting ID as a parameter along with a pointer to the new setting value and the size of the new setting.

If you're using `NetLibIFSettingSet` to set the login script, see the next section.

For an example of using these functions, see the `NetSample` example application in the `Palm OS Examples` directory. The function `CmdSettings` in the file `CmdInfo.c`, for example, shows how to loop through and obtain information about all of the network interfaces.

Setting an Interface's Login Script

The `netIFSettingLoginScript` setting is used to store the login script for an interface. The login script is generated either from the script that the user enters in the Network preferences panel or from a script file that is downloaded onto the device during a HotSync®

operation. The format of the script is rigid; if a syntactically incorrect login script is presented to the net library, the results are unpredictable. The basic format is a series of null-terminated command lines followed by a null byte at the end of the script. Each command line has the format:

```
<command-byte> [<parameter>]
```

where the command byte is the first character in the line and there is 1 and only 1 space between the command byte and the parameter string. [Table 11.1](#) lists the possible commands.

Table 11.1 Login Script Commands

Function	Command	Parameter	Example
Send	s	string	s go PPP
Wait for	w	string	w password:
Delay	d	seconds	d 1
Get IP	g		g
Prompt	a	string	a Enter Name:
Wait for prompt	f	string	f ID:
Send CR	s	string	s ^N
Send UserID	s	string	s jdoe
Send Password	s	string	s mypassword
Plugin command ^a	sp	string	sp plugin:cmd:arg

a. See “[Extending the Network Login Script Support](#).”

The parameter string to the send (s) command can contain the escape sequences shown in [Table 11.2](#).

Table 11.2 Send Command Escape Sequences

\$USERID	substitutes user name
\$PASSWORD	substitutes password
\$DBUSERID	substitutes dialback user name
\$DBPASSWORD	substitutes dialback password
^c	if c is '@' -> '_', then byte value 0 -> 31 else if c is 'a' -> 'z', then byte value 1 -> 26 else c
<cr>	carriage return (0x0D)
<lf>	line feed (0x0A)
\"	"
\^	^
\<	<
\\	\

Note also that login scripts can be created on a desktop computer and then installed onto the device during synchronization. The script commands are inspired by the Windows dial-up scripting command language for dial-up networking. For documentation from Microsoft, search for the file `Script.doc` in the Windows folder. The Network preferences panel on Palm OS supports the following subset of commands:

```
set serviceName
set userName
set password
set phoneNumber
set primaryDNS
set secondaryDNS
set ipAddr
set closewait
set inactivityTimeout
set establishmentTimeout
```

```
set protocol
set dynamicIP
waitfor
transmit
getip
delay
prompt
waitforprompt
plugin "pluginname: cmd[ : arg] "
```

The `plugin` command is a Palm OS-specific extension used to perform a command defined in a plugin. See [“Extending the Network Login Script Support”](#) for more information on plugins.

Create a script file with the extension `.pnc` or `.scp` and place it in the user’s install directory. The network conduit will download it to the device during the next HotSync operation. Each script file should contain only one service definition.

General Settings

In addition to the interface-specific settings, there’s a class of settings that don’t apply to any one particular interface. These general settings are set and retrieved through the [NetLibSettingGet](#) and [NetLibSettingSet](#) calls. These calls take setting ID, buffer pointer, and buffer size parameters.

Opening the Net Library

Call [NetLibOpen](#) to open the net library, passing the reference number you retrieved through [SysLibFind](#). Before the net library is opened, most calls issued to it fail with a `netErrNotOpen` error code.

```
err = NetLibOpen(AppNetRefnum, &ifErrs);
if (err || ifErrs) { /* error handling here */ }
```

Multiple applications can have the library open at a time, so the net library may already be open when `NetLibOpen` is called. If so, the function increments the library’s **open count**, which keeps track of how many applications are accessing it, and returns immediately. (You can retrieve the open count with the function [NetLibOpenCount](#).)

If the net library is not already open, `NetLibOpen` starts up the net protocol stack task, allocates memory for internal use by the net library, and brings up the network connection. Most likely, the user has configured the Palm OS device to establish a SLIP or PPP connection through a modem and in this type of setup, `NetLibOpen` dials up the modem and establishes the connection before returning.

If any of the attached network interfaces (such as SLIP or PPP) fail to come up, the final parameter (`ifErrs` in the example above) contains the error number of the first interface that encountered a problem.

It's possible, and quite likely, that the net library will be able to open even though one or more interfaces failed to come up (due to bad modem settings, service down, etc.). Some applications may therefore wish to close the net library using [NetLibClose](#) if the interface error parameter is non-zero and display an appropriate message for the user. If an application needs more detailed information, e.g. which interface(s) in particular failed to come up, it can loop through each of the attached interfaces and ask each one if it is up or not. For example:

```
UInt16 index, ifInstance;
UInt32 ifCreator;
Err err;
UInt8 up;
Char ifName[32];
...
for (index = 0; 1; index++) {
    err = NetLibIFGet(AppNetRefnum, index,
        &ifCreator, &ifInstance);
    if (err) break;

    settingSize = sizeof(up);
    err = NetLibIFSettingGet(AppNetRefnum,
        ifCreator, ifInstance, netIFSettingUp, &up,
        &settingSize);
    if (err || up) continue;
    settingSize = 32;
```

```
err = NetLibIFSettingGet(AppNetRefnum,
    ifCreator, ifInstance, netIFSettingName,
    ifName, &settingSize);
if (err) continue;

//display interface didn't come up message
}
NetLibClose(AppNetRefnum, true);
```

Closing the Net Library

Before an application quits, or if it no longer needs to do network I/O, it should call [NetLibClose](#).

```
err = NetLibClose(AppNetRefnum, false);
```

`NetLibClose` simply decrements the open count. The `false` parameter specifies that if the open count has reached 0, the net library should not immediately close. Instead, `NetLibClose` schedules a timer to shut down the net library unless another [NetLibOpen](#) is issued before the timer expires. When the net library's open count is 0 but its timer hasn't yet expired, it's referred to as being in the **close-wait state**.

Just how long the net library waits before closing is set by the user in the Network preferences panel. This timeout value allows users to quit from one network application and launch another application within a certain time period without having to wait for another network connection establishment.

If `NetLibOpen` is called before the close timer expires, it simply cancels the timer and marks the library as fully open with an open count of 1 before returning. If the timer expires before another `NetLibOpen` is issued, all existing network connections are brought down, the net protocol stack task is terminated, and all memory allocated for internal use by the net library is freed.

It's recommended that you allow the net library to enter the close-wait state. However, if you do need the net library to close immediately, you can do one of two things:

- Set `NetLibClose`'s second parameter to `true`. This parameter specifies whether the library should close immediately or not.

- Call [NetLibFinishCloseWait](#). This function checks the net library to see if it's in the close-wait state and if so, performs an immediate close.

Version Checking

Besides using [SysLibFind](#) to determine if the net library is installed, an application can also look for the net library version feature. This feature is only present if the net library is installed. This feature can be used to get the version number of the net library as follows:

```
UInt32* version;  
err = FtrGet(netFtrCreator, netFtrNumVersion,  
            &version);
```

If the net library is not installed, `FtrGet` returns a non-zero result code.

The version number is encoded in the format `0xMMmf sbbb`, where:

MM	major version
m	minor version
f	bug fix level
s	stage: 3-release, 2-beta, 1-alpha, 0-development
bbb	build number for non-releases

For example:

V1.1.2b3 would be encoded as 0x01122003

V2.0a2 would be encoded as 0x02001002

V1.0.1 would be encoded as 0x01013000

This document describes version 2.01 of the net library (0x02013000).

Network I/O and Utility Calls

For the network I/O and utility calls, you can either make calls using Berkeley sockets API or using the net library's native API.

Several books have been published that describe how to use Berkeley sockets API to perform network communication. Net library API closely mirrors Berkeley sockets API in this regard. However, you should keep in mind these important differences between using networking I/O on a typical computer and using net library on a Palm OS device:

- You can open a maximum of four sockets at once in the net library. This is to keep net library's memory requirements to a minimum.
- When you try to send a large block of data, the net library automatically buffers only a portion of that block because of the limited available dynamic memory. The function call returns the number of bytes of data that it actually transmitted. You must check the return value and if there's more data to send, call the function again until the transmission is finished.
- If you expect to also receive data during a large transmission, you should send a smaller block, then read back whatever is available to read before sending the next block. In this way, the amount of memory in the dynamic heap that must be used to buffer data waiting to send out and data waiting to be read back in by the application is kept to a minimum.

For more information, see the following:

- The next section, "[Berkeley Sockets API Functions](#)," provides tables that list the supported Berkeley sockets calls, the corresponding native net library call, and gives a brief description of what each call does.
- [Chapter 54, "Net Library"](#) of the *Palm OS SDK Reference* provides detailed descriptions of each net library call. Where applicable, it gives the equivalent sockets API call for each net library native call.
- The `NetSample` example application in the `Palm OS Examples` directory shows how to use the Berkeley sockets API in Palm OS applications.

Berkeley Sockets API Functions

This section provides tables that list the functions in the Berkeley sockets API that are supported by the net library. In some cases, the

calls have limited functionality from what's found in a full implementation of the sockets API and these limitations are described here.

Socket Functions

Berkeley Sockets Function	Net Library Function	Description
accept	NetLibSocketAccept	Accepts a connection from a stream-based socket.
bind	NetLibSocketBind	Binds a socket to a local address.
close	NetLibSocketClose	Closes a socket.
connect	NetLibSocketConnect	Connects a socket to a remote endpoint to establish a connection.
fcntl	NetLibSocketOptionSet NetLibSocketOptionGet (...,netSocketOptSockNonBlocking,...)	Supported only for socket <code>refnums</code> and the only commands it supports are <code>F_SETFL</code> and <code>F_GETFL</code> . The commands can be used to put a socket into non-blocking mode by setting the <code>FNDELAY</code> flag in the argument parameter appropriately — all other flags are ignored. The <code>F_SETFL</code> , <code>F_GETFL</code> , and <code>FNDELAY</code> constants are defined in <code><unix/unix_fcntl.h></code> .
getpeername	NetLibSocketAddr	Gets the remote socket address for a connection.
getsockname	NetLibSocketAddr	Gets the local socket address of a connection.

Network Communication

Net Library

Berkeley Sockets Function	Net Library Function	Description
getsockopt	NetLibSocketOptionGet	<p>Gets a socket's control options. Only the following options are implemented:</p> <ul style="list-style-type: none">• TCP_NODELAY Allows the application to disable the TCP output buffering algorithm so that TCP sends small packets as soon as possible. This constant is defined in <code><unix/netinet_tcp.h></code>.• TCP_MAXSEG Get the TCP maximum segment size. This constant is defined in <code><unix/netinet_tcp.h></code>.• SO_KEEPALIVE Enables periodic transmission of probe segments when there is no data exchanged on a connection. If the remote endpoint doesn't respond, the connection is considered broken, and <code>so_error</code> is set to <code>ETIMEOUT</code>.• SO_LINGER Specifies what to do with the unsent data when a socket is closed. It uses the <code>linger</code> structure defined in <code><unix/sys_socket.h></code>.

Berkeley Sockets Function	Net Library Function	Description
		<ul style="list-style-type: none"> • <code>SO_ERROR</code> Returns the current value of the variable <code>so_error</code>, defined in <code><unix/sys_socketvar.h></code> • <code>SO_TYPE</code> Returns the socket type to the caller.
<code>listen</code>	<u>NetLibSocketListen</u>	Sets up the socket to listen for incoming connection requests. The queue size is quietly limited to 1. (Higher values are ignored.)
<code>read, recv, recvmsg, recvfrom</code>	<u>NetLibReceive</u> <u>NetLibReceivePB</u>	Read data from a socket. The <code>recv</code> , <code>recvmsg</code> , and <code>recvfrom</code> calls support the <code>MSG_PEEK</code> flag but not the <code>MSG_OOB</code> or <code>MSG_DONTROUTE</code> flags.
<code>select</code>	<u>NetLibSelect</u>	<p>Allows the application to block on multiple I/O events. The system will wake up the application process when any of the multiple I/O events occurs.</p> <p>This function uses the <code>timeval</code> structure defined in <code><unix/sys_time.h></code> and the <code>fd_set</code> structure defined in <code>sys/types.h</code>.</p>

Network Communication

Net Library

Berkeley Sockets Function	Net Library Function	Description
		<p>Also associated with this function are the following four macros defined in <code><unix/sys_types.h></code>:</p> <ul style="list-style-type: none">• <code>FD_ZERO</code>• <code>FD_SET</code>• <code>FD_CLR</code>• <code>FD_ISSET</code> <p>Besides socket descriptors, this function also works with the “stdin” descriptor, <code>sysFileDescStdIn</code>. This descriptor is marked as ready for input whenever a user or system event is available in the event queue. This includes any event that would be returned by <code>EvtGetEvent</code>. No other descriptors besides <code>sysFileDescStdIn</code> and socket refnums are allowed.</p>
<code>send</code> , <code>sendmsg</code> , <code>sendto</code>	NetLibSend NetLibSendPB	<p>These functions write data to a socket. These calls, unlike the <code>recv</code> calls, do support the <code>MSG_OOB</code> flag. The <code>MSG_PEEK</code> flag is not applicable and the <code>MSG_DONTROUTE</code> flag is not supported.</p>
<code>setsockopt</code>	NetLibSocketOptionSet	<p>This function sets control options of a socket. Only the following options are allowed:</p> <ul style="list-style-type: none">• <code>TCP_NODELAY</code>• <code>SO_KEEPALIVE</code>• <code>SO_LINGER</code>

Berkeley Sockets Function	Net Library Function	Description
shutdown	NetLibSocketShutdown	Similar to <code>close()</code> ; however, it gives the caller more control over a full-duplex connection.
socket	NetLibSocketOpen	Creates a socket for communication. The only valid address family is <code>AF_INET</code> . The only valid socket types are <code>SOCK_STREAM</code> , <code>SOCK_DGRAM</code> , and in Palm OS version 3.0 and higher, <code>SOCK_RAW</code> . The protocol parameter should be set to 0.
write	NetLibSend	Writes data to a socket.

Supported Network Utility Functions

Berkeley Sockets Function	Net Library Function	Description
getdomainname	NetLibSocketOptionGet (..., netSettingDomainName, ...)	Returns the domain name of the local host.
gethostbyaddr	NetLibGetHostByAddr	Looks up host information given the host's IP address. It returns a <code>hostent</code> structure, as defined in <code><netdb.h></code> .
gethostbyname	NetLibGetHostByName	Looks up host information given the host's name. It returns a <code>hostent</code> structure which is defined in <code><netdb.h></code> .

Network Communication

Net Library

Berkeley Sockets Function	Net Library Function	Description
gethostname	NetLibSettingGet (..., netSettingHostName, ...)	Returns the name of the local host.
getservbyname	NetLibGetServByName	Returns a servent structure, defined in <netdb.h> given a service name.
gettimeofday	glue code using TimGetSeconds	Returns the current date and time.
setdomainname	NetLibSettingSet (..., netSettingDomainName, ...)	Sets the domain name of the local host.
sethostname	NetLibSettingSet (..., netSettingHostName, ...)	Sets the name of the local host.
settimeofday	glue code using TimSetSeconds	Sets the current date and time.

Supported Byte Ordering Macros

The byte ordering macros are defined in <unix/netinet_in.h>. They convert an integer between network byte order and the host byte order.

Berkeley Sockets Macro	Description
htonl	Converts a 32-bit integer from host byte order to network byte order.
htons	Converts a 16-bit integer from host byte order to network byte order.
ntohl	Converts a 32-bit integer from network byte order to host byte order.
ntohs	Converts a 16-bit integer from network byte order to host byte order.

Supported Network Address Conversion Functions

The network address conversion functions are declared in the `<unix/arpa_inet.h>` header file. They convert a network address from one format to another, or manipulate parts of a network address.

Berkeley Sockets Function	Net Library Function	Description
<code>inet_addr</code>	NetLibAddrAToIN	Converts an IP address from dotted decimal format to 32-bit binary format.
<code>inet_network</code>	glue code	Converts an IP network number from a dotted decimal format to a 32-bit binary format.
<code>inet_makeaddr</code>	glue code	Returns an IP address in an <code>in_addr</code> structure given an IP network number and an IP host number in 32-bit binary format.
<code>inet_lnaof</code>	glue code	Returns the host number part of an IP address.
<code>inet_netof</code>	glue code	Returns the network number part of an IP address.
<code>inet_ntoa</code>	NetLibAddrINTToA	Converts an IP address from 32-bit format to dotted decimal format.

Extending the Network Login Script Support

Beginning in Palm OS 3.3, you can write a plugin that extends the list of available script commands in the Network preferences panel. You might do so, for example, if:

- You are a corporate IT shop, system integrator, or a token card vendor and want the login script to properly respond to a range of different connection scenarios defined by the authentication server.
- You are a token card vendor and you want to create the Palm OS version of your password generator.

- You want to perform conditional tests and branching during the execution of the script.

The login script enhancement can also be installed on any device that already has network library support (that is, PalmPilot™ Professional and newer devices running Palm OS 2.0 or higher). To do so, you install a file named `Network.prc` along with a PRC file for the network interface you use (i.e., PPP or SLIP). These files provide the new Network preferences panel, which contains support for some new commands and support for the ability to write script plugins.

The sections below describe the basics of how to write a login script plugin. For more detailed information on the API you use to write a plugin, see the chapter “[Script Plugin](#)” on page 1067 in the *Palm OS SDK Reference*.

Writing the Login Script Plugin

To write a login script plugin, you create a project like you normally would; however, specify 'scpt' as the database type instead of 'appl'. (If you're using Metrowerks CodeWarrior, you specify the database type in the PalmRez post linker panel.)

In the `PilotMain` function, the plugin should respond to two launch codes:

- [scptLaunchCmdListCmds](#) to inform the Network preferences panel of the commands your plugin implements.
- [scptLaunchCmdExecuteCmd](#) to execute one of your commands.

Responding to `scptLaunchCmdListCmds`

The Network preferences panel sends the `scptLaunchCmdListCmds` launch code when it is constructing the pull-down list of available commands that it displays in its script view. The panel sends this launch code to all PRCs of type 'scpt'. It passes an empty structure of type [PluginInfoType](#) as its parameter block. Your plugin should respond by filling in the structure with the following information:

- The name of your plugin (the name of the PRC file)

- The number of commands your plugin implements. No more than `pluginMaxNumOfCmds` is allowed.
- An array containing the name of each command your plugin implements and a Boolean value that indicates whether your plugin takes an argument.

A given device might have multiple plugins installed. If so, the resulting pull-down list contains the union of all commands supported by all of the plugins installed on the device. For this reason, you should make sure the command names you supply are unique. You also should make sure the names are as brief as possible, as only 15 characters are allowed for the name.

Responding to `scptLaunchCmdExecuteCmd`

The `scptLaunchCmdExecuteCmd` launch code is sent when the login script is being executed. That is, the user has attempted to connect to the network service specified in the Network preferences panel, and the panel is executing the script to perform authentication.

The `scptLaunchCmdExecuteCmd` parameter block is a structure of type [`PluginExecCmdType`](#). It contains:

- The name of the command to be executed
- The command argument, if it takes one
- A pointer to a network interface function
- A handle to information specific to the current connection

Your plugin should execute the specified command. When a plugin is launched with this code, it is launched as a subroutine and as such does not have access to global variables. Also keep in mind that the network library and a connection application (such as the HotSync application) are already running when the plugin is launched. Thus, available memory and stack space are extremely limited.

To perform most of its work, the plugin command probably needs access to the network interface (such as SLIP or PPP) specified for the selected network service. For this reason, the plugin is passed a pointer to a callback function defined by the network interface. The plugin should call this function when it needs to perform the following tasks:

- Read a number of bytes from the network
- Write a number of bytes to the network
- Get the user's name and password information
- Write a string to the connection log
- Prompt the user for information
- Check to see if the user pressed the Cancel button
- Display a form
- Obtain access to the serial library

The callback's prototype is defined by [ScriptPluginSelectorProc](#). It takes as arguments the handle to the connection-specific data passed in with the launch code, the task that the network interface should perform (specified as a `pluginNetLib...` constant), followed by a series of parameters whose interpretations depend on which task is to be performed.

For example, the following code implements the command "Send Uname", which sends the user's name to the host computer.

Listing 11.1 Simple Script Plugin Command

```
#define pluginSecondCmd "Send Uname"

UInt32 PilotMain(UInt16 cmd, void *cmdPBP,
UInt16 launchFlags) {
    PluginExecCmdPtr execPtr;
    UInt32 error = success;
    Int16 dataSize = 0;
    Char* dataBuffer = NULL;
    ScriptPluginSelectorProcPtr selectorTypeP;

    if (cmd == scptLaunchCmdExecuteCmd) {
        execPtr = (PluginExecCmdPtr)cmdPBP;
        selectorTypeP = execPtr->procP->selectorProcP;

        dataBuffer =
        MemPtrNew(pluginMaxLenTxtStringArg+1);
        if (!dataBuffer) {
            return failure;
        }
    }
}
```

```
    }
    MemSet(dataBuffer,pluginMaxLenTxtStringArg+1,0);

    if (!StrCompare(execPtr->commandName,
pluginSecondCmd)) {

        /* get the user name from the network
interface */
        error = (selectorTypeP)(execPtr->handle,
            pluginNetLibGetUserName, (void*)dataBufferP,
&dataSize, 0,
            NULL);
        if (error) goto Exit;

        dataSize = StrLen((Char*)dataBufferP);

/* have the network interface send the user name
to the host */
        error = (selectorTypeP)(execPtr->handle,
            pluginNetLibWriteBytes, (void*)dataBufferP,
&dataSize, 0,
            NULL);

        return error;
    }
}
```

If your command needs to interact with the user, it must do so through the network interface. When the connection attempt is taking place, the user sees either the Network preferences panel or the HotSync application. Your plugin does not have control of the screen, so you cannot simply display a form. You have two options:

- The network interface can display a prompt for you and return the value that the user enters in response. It can also query the Network preferences panel to see if the user cancelled the connection attempt.
- If you want to do more than simply display a prompt or check the cancel status, you can use the command

`pluginNetLibCallUIProc` to display a form and call your own user interface routine.

To use `pluginNetLibCallUIProc`, you must do the following:

1. Initialize the form using a form resource that you've created.
2. Create a struct that contains your form's handle and any other values that you are going to need in your user interface routine.
3. Call the network interface's callback function with the `pluginNetLibCallUIProc` command, the structure with the form's handle and other pertinent information, and the address of a function in your plugin that will perform the user interface routine. This function should take one argument—the struct you've passed to the network interface—and return `void`.
4. When the call to the network interface returns, close the form.

For an example of using `pluginNetLibCallUIProc`, see the functions `WaitForData` and `promptUser` in the example code `ScriptPlugin.c`.

Internet Library

The Internet library provides Palm applications easy access to World Wide Web documents. The Internet library uses the net library for basic network access and builds on top of the net library's socket concept to provide a socket-like API to higher level internet protocols like HTTP and HTTPS.

Using the Internet library, an application can access a web page with as little as three calls ([INetLibURLOpen](#), [INetLibSockRead](#), and [INetLibSockClose](#)). The Internet library also provides a more advanced API for those applications that need finer control.

NOTE: The information in this section applies only to version 3.2 or later of the Palm OS on Palm VII devices. These features are implemented only if the [Wireless Internet Feature Set](#) is present.

WARNING! In future OS versions, Palm, Inc. does not intend to support or provide backward compatibility for the Internet library API.

The Internet library is implemented as a system library that is installed at runtime and doesn't have to be present for the system to work properly.

This section describes how to use the Internet library in your application. It covers:

- [System Requirements](#)
- [Initialization and Setup](#)
- [Accessing Web Pages](#)
- [Asynchronous Operation](#)
- [Using the Low Level Calls](#)
- [Cache Overview](#)
- [Internet Library Network Configurations](#)

System Requirements

The Internet library is available only on version 3.2 or later of the Palm OS on Palm VII devices. Before making any Internet library calls, ensure that the Internet library is available. You can be sure it is available by using the following [FtrGet](#) call:

```
err = FtrGet(inetLibFtrCreator,  
            inetFtrNumVersion, &value);
```

If the Internet library is installed, the `value` parameter will be non-zero and the returned error will be zero (for no error).

When the Internet library is present and running, it requires an estimated additional 1 KB of RAM, beyond the net library. More additional memory is used for the security library, if that is used (when accessing secure sites), and for opening a cache database, if that is used.

Initialization and Setup

Before using the Internet library, an application must call [SysLibFind](#) to obtain a library reference number, as follows:

```
err = SysLibFind("INet.lib", &libRefNum)
```

Next, it must call [INetLibOpen](#) to allocate an `inetH` handle. The `inetH` handle holds all application specific environment settings and each application that uses the Internet library gets its own private `inetH` handle. Any calls that change the default behavior of the Internet library affect environment settings stored in the application's own `inetH` structure, so these changes will not affect other applications that might be using the Internet library at the same time.

[INetLibOpen](#) also opens the net library for the application. In addition, the application can tell [INetLibOpen](#) the type of network service it prefers: wireline or wireless. [INetLibOpen](#) queries the available network interfaces and attaches the appropriate one(s) for the desired type of service. When the application calls [INetLibClose](#), the previous interface configuration is restored. For more information on configurations, see the section “[Internet Library Network Configurations](#)” on page 306.

The Internet library gets some of its default behavior from the system preferences database, and some of these preference settings are made by the user via the Wireless preferences panel. The preferences set by this panel include the proxy server to use and a setting that determines whether or not the user is warned when the device ID is sent. Other settings stored in the preferences database come from Internet library network configurations (see “[Internet Library Network Configurations](#)” on page 306). All these settings can be queried and/or overridden by each application through the [INetLibSettingGet](#) and [INetLibSettingSet](#) calls. However, any changes made by an application are not stored into the system preferences, but only take effect while that `inetH` handle is open.

Accessing Web Pages

In the Palm.Net environment, all HTML documents are dynamically compressed by the Palm Web Clipping Proxy server before being transmitted to the Palm device.

The procedure for reading a page from the network operates as follows. First, the application passes the desired URL to the [INetLibURLOpen](#) routine, which creates a socket handle to access that web page. This routine returns immediately before performing any required network I/O. Then the application calls [INetLibSockRead](#) to read the data, followed by [INetLibSockClose](#) to close down the socket.

Note that if no data is available to read immediately, [INetLibSockRead](#) blocks until at least one byte of data is available to be read. To implement asynchronous operation using events, see the next section, [Asynchronous Operation](#).

If an application requires finer control over the operation, it can replace the call to [INetLibURLOpen](#) with other lower-level Internet library calls ([INetLibSockOpen](#), [INetLibSockSettingSet](#), etc.) that are described in the section “[Using the Low Level Calls](#)” on page 305.

Asynchronous Operation

A major challenge in writing an Internet application is handling the task of accessing content over a slow network while still providing good user-interface response. For example, a user should be able to scroll, select menus, or tap the Cancel button in the middle of a download of a web page.

To easily enable this type of functionality, the Internet library provides the [INetLibGetEvent](#) call. This call is designed to replace the `EvtGetEvent` call that all traditional, non-network Palm applications use. The [INetLibGetEvent](#) call fetches the next event that needs to be processed, whether that event is a user-interface event like a tap on the screen, or a network event like some data arriving from the remote host that needs to be read. If no events are ready, [INetLibGetEvent](#) automatically puts the Palm device into low-power mode and blocks until the next event occurs.

Using [INetLibGetEvent](#) is the preferred way of performing network I/O since it maximizes battery life and user-interface responsiveness.

With [INetLibGetEvent](#), the process of accessing a web page becomes only slightly more complicated. Instead of calling

`INetLibSockRead` immediately after `INetLibURLOpen`, the application should instead return to its event loop and wait for the next event. When it gets a network event that says data is ready at the socket, then it should call `INetLibSockRead`.

There are two types of network events that `INetLibGetEvent` can return in addition to the standard user-interface events. The first event is a status change event ([inetSockStatusChangeEvent](#)). This event indicates that the status of a socket has changed and the application may want to update its user interface. For example, when calling `INetLibURLOpen` to access an HTTP server, the status on the socket goes from “finding host,” to “connecting with host,” to “waiting for data,” to “reading data,” etc. The event structure associated with an event of this type contains both the socket handle and the new status so that the application can update the user interface accordingly.

The second type of event that `INetLibGetEvent` can return is a data-ready event ([inetSockReadyEvent](#)). This event is returned when data is ready at the socket for reading. This event tells the application that it can call `INetLibSockRead` and be assured that it will not block while waiting for data to arrive.

The general flow of an application that uses the Internet library is to open a URL using `INetLibURLOpen`, in response to a user command. Then it repeatedly calls `INetLibGetEvent` to process events from both the user interface and the newly created socket returned by `INetLibURLOpen`. In response to `inetSockStatusChangeEvent` events, the application should update the user interface to show the user the current status, such as finding host, connecting to host, reading data, etc. In response to `inetSockReadyEvent` events, the application should read data from the socket using `INetLibSockRead`. Finally, when all available data has been read (`INetLibSockRead` returns 0 bytes read), the application should close the socket using `INetLibSockClose`.

Finally, the convenience call [INetLibSockStatus](#) is provided so that an application can query the status of a socket handle. This call never blocks on network I/O so it is safe to call at any time. It not only returns the current status of the socket but also whether or not it is ready for reading and/or writing. It essentially returns the same

information as conveyed via the events `inetSockReadyEvent` and `inetSockStatusChangeEvent`. Applications that don't use `INetLibGetEvent` could repeatedly poll `INetLibSockStatus` to check for status changes and readiness for I/O, though polling is not recommended.

Using the Low Level Calls

Applications that need finer control than `INetLibURLOpen` provides can use the lower level calls of the Internet library. These include [`INetLibSockOpen`](#), [`INetLibSockConnect`](#), [`INetLibSockSettingSet`](#), [`INetLibSockHTTPReqCreate`](#), [`INetLibSockHTTPAttrGet`](#), [`INetLibSockHTTPAttrSet`](#), and [`INetLibSockHTTPReqSend`](#).

A single call to `INetLibURLOpen` for an HTTP resource is essentially equivalent to this sequence: `INetLibSockOpen`, `INetLibSockConnect`, `INetLibSockHTTPReqCreate`, and `INetLibSockHTTPReqSend`. These four calls provide the capability for the application to access non-standard ports on the server (if allowed), to modify the default HTTP request headers, and to perform HTTP PUT and POST operations. The only calls here that actually perform network I/O are `INetLibSockConnect`, which establishes a TCP connection with the remote host, and `INetLibSockHTTPReqSend`, which sends the HTTP request to the server.

`INetLibSockHTTPAttrSet` is provided so that the application can add or modify the default HTTP request headers that `INetLibSockHTTPReqCreate` creates.

`INetLibSockSettingSet` allows an application finer control over the socket settings.

Finally, the routine [`INetLibURLCrack`](#) is provided as a convenient utility for breaking a URL into its component parts.

Cache Overview

The Internet library maintains a cache database of documents that have been downloaded. This is an LRU (Least Recently Used) cache; that is, the least recently used items are flushed when the cache fills. Whether or not a retrieved page is cached is determined by a flag

([inetOpenURLFlagKeepInCache](#)) set in the socket or by [INetLibURLOpen](#). Another flag ([inetOpenURLFlagLookInCache](#)) determines if the Internet library should check the cache first when retrieving a URL.

The same cache database can be used by any application using the Internet library, so that every application can share the same pool of prefetched documents. Alternately, an application can use a different cache database. The cache database to use is specified in the [INetLibOpen](#) call.

Generally, a cached item is stored in one or more database records in the same format as it arrives from the server.

In the cache used by the Clipper application, each record includes a field that contains the “master” URL of the item. This field is set to the URL of the active PQA, so all pages linked from one PQA have the same master URL. This facilitates finding all pages in a hierarchy to build a history list.

The Internet library maintains a list of items in the cache. You can retrieve items in this list, or iterate over the whole list, by calling [INetLibCacheList](#). You can retrieve a cached document directly by using [INetLibCacheGetObject](#).

You can check if a URL is cached by calling [INetLibURLGetInfo](#).

Internet Library Network Configurations

The Internet library supports network configurations. A **configuration** is a specific set of values for several of the Internet library settings (from the [INetSettingEnum](#) type).

The Internet library keeps a list of available configurations and aliases to them. There are three built-in configurations:

- A wireless configuration that uses the Palm.Net wireless system and the Palm Web Clipping Proxy server.
- A wireline configuration that uses the wireline network configuration specified in the Network preferences panel and the Palm Web Clipping Proxy server.
- A generic configuration that uses the wireline network configuration specified in the Network preferences panel and no proxy server.

You can also define your own configuration by modifying an existing one and saving it under a different name.

The Internet library also defines several **configuration aliases** (see “[Configuration Aliases](#)” on page 1144 in the *Palm OS SDK Reference*). An alias is a configuration name that simply points to another configuration. You can specify an alias anywhere in the API you would specify a configuration. This facilitates easy re-assignment of the built-in configurations and eliminates having duplicate settings. You assign an alias by using [INetLibConfigAliasSet](#) and can retrieve an alias by using [INetLibConfigAliasGet](#).

For example, to change the default configuration used by the Internet library for a particular kind of connection, you can set up the appropriate values for a connection, save the configuration, and then set the Internet library’s default alias configuration to point to your custom configuration. When an application specifies which configuration it wants to use, if it specifies the alias, it will use the custom settings.

If you use configurations at all, it will probably be to specify a specific configuration when opening the Internet library via [INetLibOpen](#). The Internet library also contains an API to allow you to manipulate configurations in your application, but doing so is rare. You can list the available configurations ([INetLibConfigList](#)), get a configuration index ([INetLibConfigIndexFromName](#)), select ([INetLibConfigMakeActive](#)) the Internet library network configuration you would prefer to use (wireless, wireline, etc.), rename existing configurations ([INetLibConfigRename](#)), and delete configurations ([INetLibConfigDelete](#)).

The configuration functions are provided primarily for use by Preferences panels while editing and saving configurations. The general procedure is to make the configuration active that you want to edit, set the settings appropriately, then save the configuration using [INetLibConfigSaveAs](#). Note that configuration changes are not saved after the Internet library is closed, unless you call [INetLibConfigSaveAs](#).

Summary of Network Communication

Net Library Functions

Library Open and Close

[NetLibClose](#)

[NetLibConnectionRefresh](#)

[NetLibFinishCloseWait](#)

[NetLibOpen](#)

[NetLibOpenCount](#)

Socket Creation and Deletion

[NetLibSocketClose](#)

[NetLibSocketOpen](#)

Socket Options

[NetLibSocketOptionGet](#)

[NetLibSocketOptionSet](#)

Socket Connections

[NetLibSocketAccept](#)

[NetLibSocketAddr](#)

[NetLibSocketBind](#)

[NetLibSocketConnect](#)

[NetLibSocketListen](#)

[NetLibSocketShutdown](#)

Send and Receive Routines

[NetLibDmReceive](#)

[NetLibReceive](#)

[NetLibReceivePB](#)

[NetLibSend](#)

[NetLibSendPB](#)

Utilities

[NetHToNL](#)

[NetHToNS](#)

[NetLibAddrAToIN](#)

[NetLibAddrINToA](#)

[NetLibGetHostByAddr](#)

[NetLibGetHostByName](#)

[NetLibGetMailExchangeByName](#)

[NetLibGetServByName](#)

[NetLibMaster](#)

[NetLibSelect](#)

[NetLibTracePrintf](#)

[NetLibTracePutS](#)

[NetNTToHL](#)

[NetNTToHS](#)

Net Library Functions

Setup

[NetLibIFAttach](#)
[NetLibIFDetach](#)
[NetLibIFDown](#)
[NetLibIFGet](#)
[NetLibIFSettingGet](#)

[NetLibIFSettingSet](#)
[NetLibIFUp](#)
[NetLibSettingGet](#)
[NetLibSettingSet](#)

Network Utilities

[NetUReadN](#)

[NetUTCPOpen](#)
[NetUWriteN](#)

Internet Library Functions

Library Open and Close

[INetLibClose](#)

[INetLibOpen](#)

Settings

[INetLibSettingGet](#)

[INetLibSettingSet](#)

Event Management

[INetLibGetEvent](#)

High-Level Socket Calls

[INetLibSockClose](#)
[INetLibSockRead](#)

[INetLibURLOpen](#)

Low-Level Socket Calls

[INetLibSockConnect](#)
[INetLibSockOpen](#)
[INetLibSockSettingGet](#)

[INetLibSockSettingSet](#)
[INetLibSockStatus](#)

Internet Library Functions

HTTP Interface

[INetLibSockHTTPAttrGet](#)
[INetLibSockHTTPAttrSet](#)

[INetLibSockHTTPReqCreate](#)
[INetLibSockHTTPReqSend](#)

Utilities

[INetLibCheckAntennaState](#)
[INetLibURLCrack](#)
[INetLibURLGetInfo](#)

[INetLibURLsAdd](#)
[INetLibWiCmd](#)

Cache Interface

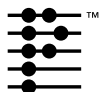
[INetLibCacheGetObject](#)

[INetLibCacheList](#)

Configuration

[INetLibConfigAliasGet](#)
[INetLibConfigAliasSet](#)
[INetLibConfigDelete](#)
[INetLibConfigIndexFromName](#)

[INetLibConfigList](#)
[INetLibConfigMakeActive](#)
[INetLibConfigRename](#)
[INetLibConfigSaveAs](#)



Internet and Messaging Applications

NOTE: The information in this chapter currently applies only to the system software installed on the Palm VII™ device.

The Palm OS® version 3.2 provides support for wireless Internet access and messaging via the Palm.Net wireless network. This chapter discusses the following topics:

- [Overview of the Palm.Net System](#)
- [System Version Checking](#)
- [Using Clipper to Display Information](#)
- [Launching Other Applications from Clipper](#)
- [Sending Messages](#)
- [New keyDownEvent Key Codes](#)
- [Over the Air Characters](#)

Most of the information in this chapter applies to wireline connects as well as wireless connections. It is possible for developers to connect to the Palm.Net network via a wired modem through an Internet Service Provider for testing, though normal users will access Palm.Net via the built-in wireless modem.

For more information about Palm query applications and content style guidelines for the Palm VII device, refer to the *Web Clipping Developer's Guide*.

Overview of the Palm.Net System

Before developing content and applications for the Palm VII device, it's useful to understand the whole Palm.Net system. The Palm VII device is just one part of a system that delivers data wirelessly from the Internet to the Palm device.

The system is designed to work differently from a web browser application running on a desktop computer. The Palm.Net system is designed to best support access to real-time data, not casual browsing. Browsing is possible, but the increased cost and volume of data involved with visiting most standard web sites makes it impractical over a wireless network.

Typical scenarios involve users accessing the following kinds of information on the Internet: news, sports scores, weather, traffic reports, driving directions, airline schedules and flight information, stock quotes, hotel and restaurant information, email, etc.

Constraints on Palm wireless applications include the high cost to users of radio usage, low bandwidth, and increased battery consumption when the radio is on. Palm designed the system to make the best use of resources given these constraints. You must also keep these constraints in mind when designing applications that use the wireless capabilities of the unit.

In particular, note the pricing model for the wireless service. Users are charged a flat monthly fee for a modest number of bytes transmitted and received. Once the limit is exceeded, users are charged for each additional byte sent or received by their Palm device. It's imperative that applications using the wireless services minimize the number of bytes sent and received, to avoid contributing to large airtime charges for users.

Content developers wishing to customize web pages for optimal display on Palm VII devices should follow the design guidelines described in the *Web Clipping Developer's Guide*. A web site that conforms to these style guidelines and contains the

```
<META NAME="PalmComputingPlatform" CONTENT="True">
```

HTML tag is considered Palm friendly.

NOTE: The Internet applications described in this chapter rely on the Internet library (INetLib) for wireless connectivity functions, and the Internet library uses the net library (NetLib). Applications cannot directly use the net library to make wireless connections.

Palm Query Applications

The primary mechanism that Palm has provided for users to interact with the WWW (World Wide Web) is the Palm query application (PQA). Palm query applications encapsulate locally stored HTML content, possibly including one or more query forms, through which the user can submit requests for information from the WWW. Returned data, called web clippings, are displayed by the web clipping viewer application (called Clipper here) that runs on the Palm device.

Note that Clipper does not appear as a separate application in the Launcher; it is invoked automatically when a query application is launched. End users don't see the term "Clipper" anywhere in the user interface or user documentation, so you should not confuse them by using this term in your application documentation, readme files, or help screens.

Palm query applications are created by the Query Application Builder program that runs on a desktop computer. This program translates one or more pages of HTML content into a single compact database (.pqa file) that the user installs on the Palm device.

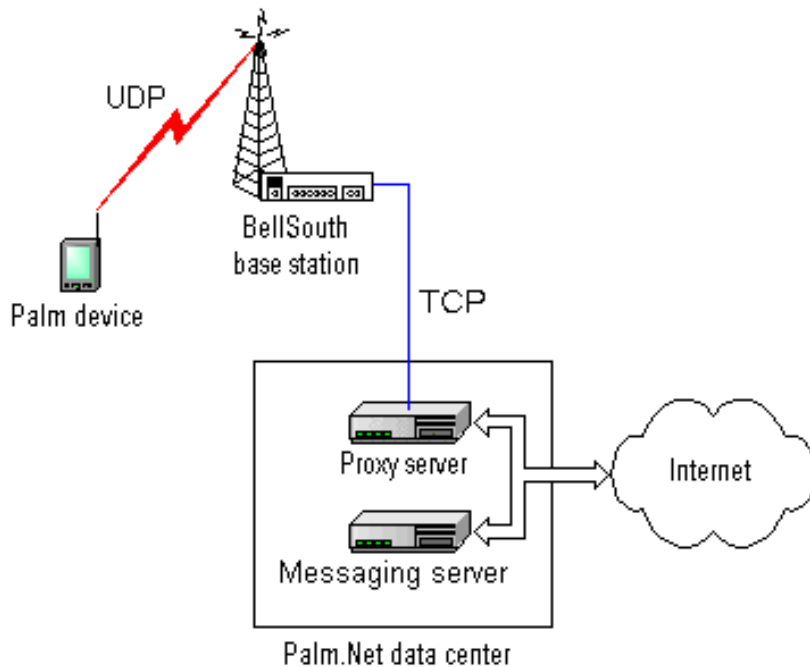
When creating the .pqa file, the Query Application Builder translates HTML into a compressed format. The Clipper application works with this compressed format, rather than HTML directly. The reason for this is that HTML is an inefficient format for the transmission of data over the network and storage of information. Compression minimizes the amount of information sent over the radio and reduces the size of query applications stored on the Palm device.

GIF and JPEG images incorporated into source HTML files are converted to the Palm bitmap format (2-bit graphics) before being stored in the query application file.

Palm.Net System Overview

The physical Palm.Net network is illustrated in [Figure 12.1](#).

Figure 12.1 Palm.Net Network



The Palm VII device communicates via radio modem to a nearby BellSouth Wireless Data network base station. From there, data is sent over a private link to the Palm Web Clipping Proxy server in the Palm data center. The proxy server interprets user requests and passes them to other computers on the Internet, using standard HTTP protocols, to handle as appropriate.

Responses are sent back to the proxy server, which communicates them to the Bell South wireless network and back to the Palm VII device via radio modem.

The wireless radio link operates at approximately 8 kbps, so is best suited for exchanging small amounts of information. After accounting for headers, error correction, and other overhead, the effective data throughput is roughly 2 kbps, so compactness is critical.

Palm Web Clipping Proxy Server

The Palm Web Clipping Proxy server is a key part of the system. This server is responsible for accepting and responding to queries sent by the Palm VII device.

The server supports three high-level protocols: HTTP, HTTPS, and the Palm messaging protocol (used by the iMessenger application). Requests using HTTP and HTTPS are forwarded to the Internet. Requests using the messaging protocol are forwarded to the Palm messaging server, which handles email communication to the Internet.

UDP

One way that Palm optimizes the limited network bandwidth is to use UDP (User Datagram Protocol). All communications between the Palm VII device and the wireless network use UDP. This transmission protocol is extremely efficient and lightweight, resulting in the exchange of the fewest packets possible over the wireless network. Often requests and responses require just a single packet of data each. This is much more efficient than the relatively verbose TCP (Transmission Control Protocol). Using UDP decreases user airtime costs because fewer packets are required for each request and response.

UDP does not normally function as a reliable protocol, however, the wireless connection between the Palm device and the BellSouth Wireless Data network has guaranteed delivery and reliability built into it via other mechanisms, so there is no need for the extra overhead of a full connection-oriented protocol such as TCP.

WWW requests that are passed to the Internet by the proxy server use TCP to guarantee reliability over the Internet.

Note that in a debugging wired connection scenario, TCP is used instead of UDP between the Palm device and the proxy server.

Compressed HTML

Another way that Palm efficiently uses the limited bandwidth of the Palm.Net system is to compress HTML.

Web clippings are rendered on the Palm VII device by the Clipper application. Clipper renders compressed HTML data. Both the query applications and WWW data returned from the Internet are compressed.

- When creating Palm query applications, the Query Application Builder program compresses HTML content and combines multiple HTML pages and images into a single query application.
- All HTML information returned to the Palm device from the Internet is dynamically compressed by the Palm Web Clipping Proxy server before transmission through the wireless network to the Palm device.

It's important to note that the Palm device accesses standard HTML data that resides on standard HTML web servers on the Internet. The compression by the proxy server is transparent to the user and the web server on the Internet.

If a web page that is not Palm-friendly is browsed, the proxy server removes images, scripting code, Java code, frames, and other non-supported elements before sending the content to the Palm device. Additionally, the content is truncated to prevent large amounts of unexpected data from being transmitted. The user can request more data as desired.

Security

All wired parts of the network support security via the SSL (Secure Sockets Layer) protocol widely used by servers and browsers on the Internet. However, SSL is impractical to run over a low bandwidth wireless network because it is quite verbose.

Palm implemented a level of security for the wireless portion of the network that is equivalent to the 128-bit SSL encryption algorithms, but optimized for use on a wireless network. The wireless part of the network is protected by a security system that includes encryption, message integrity checking, and server authentication.

Message encryption is done via an elliptic curve cryptography engine supplied by Certicom Corporation. Message integrity checking protects against transmission errors or message manipulation. Server authentication prevents the wireless session between the Palm device and the proxy server from being hijacked or spoofed.

Note that despite the optimized security scheme, secure transmissions inherently increase the size of the data packet, slowing its transmission over the network relative to unsecure transmissions.

System Version Checking

Before using any special features of the operating system for the Palm VII device, you must check to ensure they are present. You can ensure that you are running on a device that supports the wireless internet access features by checking for the existence of the Clipper and iMessenger applications. Here's an example of how to check for Clipper:

```
DmSearchStateType searchState;
UInt16* cardNo;
LocalID* dbID;
err = DmGetNextDatabaseByTypeCreator(true,
    &searchState, sysFileTApplication,
    sysFileCCLipper, true, &cardNo, &dbID);
```

If Clipper is not present, the

`DmGetNextDatabaseByTypeCreator` routine returns an error.

To check for iMessenger, you can use the creator type `sysFileCMessaging`.

For more information on checking system compatibility, see the appendix “[Compatibility Guide](#)” starting on page 1201.

Using Clipper to Display Information

You can use launch codes to open Clipper and display content.

To launch Clipper and display a PQA, use the launch code [sysAppLaunchCmdOpenDB](#). You pass as parameters the database id and card number of the PQA to display. This is the same mechanism used by the Launcher to “launch” data files.

To launch Clipper and display any URL, use the launch code [sysAppLaunchCmdGoToURL](#). You pass as a parameter a pointer to the URL string. An example of how to use this launch code is shown in [Listing 12.1](#).

IMPORTANT: Keep in mind that browsing web sites that are complex or not Palm-friendly may possibly result in higher latency and airtime charges for the user. If a web page that is not Palm-friendly is browsed, the proxy server removes images, scripting code, Java code, frames, and other non-supported elements before sending the content to the Palm device.

Listing 12.1 Launching Clipper with a URL

```
Err GoToURL(Char* origurl)
{ // parameter is ptr to URL string
  Err err;
  Char* url;
  DmSearchStateType searchState;
  UInt16* cardNo;
  LocalID* dbID;

  // make a copy of the URL, since the OS will
  free
  // the parameter once Clipper quits
  url = MemPtrNew(StrLen(origurl));
  if (!url) return sysErrNoFreeRAM;
  StrCopy(url, origurl);
  MemPtrSetOwner(url, 0);

  // find clipper and launch it
```

```
err = DmGetNextDatabaseByTypeCreator (true,
&searchState, sysFileTApplication,
sysFileCClipper, true, &cardNo, &dbID);
if (err) { // Clipper is not present
    FrmAlert(NoClipperAlert);
    MemPtrFree(url);
}
else
    err =
SysUIAppSwitch(cardNo,dbID,sysAppLaunchCmdGoToURL,
url);

return err; // 0 means no error
}
```

Launching Other Applications from Clipper

Clipper can launch other applications via two special types of URLs: `palm` and `palmcall`. In a query application, you might want to use the `palmcall` URL to hand some data to a different application to process and/or display while Clipper is running. This would be useful for graphing a set of numbers, for example.

Both of these URL types take a URL string in the following form:

`palm:cccc.tttt?params`

or

`palmcall:cccc.tttt?params`

`cccc` is a four character creator name and `tttt` is a four character database type. These parts identify the application to launch. After the question mark (?), the `params` portion of the string can be any text you want. The entire URL string is passed to the application to use in any manner.

Here's an example of an HTML anchor that uses the `palm` URL type to link to the Memo Pad application:

```
<A HREF="palm:memo.appl">Memo Pad</A>
```

Use the `palm` URL to cause Clipper to launch another application with the [SysUIAppSwitch](#) routine. This causes Clipper to quit before the other application is launched.

Use the `palmcall` URL to cause Clipper to sublaunch another application with the [SysAppLaunch](#) routine. Clipper stays in the background and resumes execution when the other application quits. It's important to note that in this situation, the sublaunched application does not have access to its global variables or to code outside segment 0 (in a multi-segment application).

The Clipper application handles these URLs by sending the [sysAppLaunchCmdURLParams](#) launch code to the specified application. The parameter block for this launch code is a pointer to the URL string.

Sending Messages

You can send messages via the built-in iMessenger application in 3 ways:

- Use the standard `mailto` URL in Clipper, passing an email address, for example, "`mailto:info@palm.com`". This launches iMessenger, passing the email address for the "To" field. Optionally, you can include the subject ("`mailto:info@palm.com?subject=foo`") and/or body ("`mailto:info@palm.com?subject=foo&body=bar`") text in the URL. Internally, this launches iMessenger using the next method.
- Use the [sysAppLaunchCmdAddRecord](#) launch code to launch iMessenger with its editor open (optionally filling in some of the fields via the passed parameter block). This allows the user to edit the email. To make iMessenger display the message in its editor, set the `edit` field in the parameter block to `true`.
- Use the [sysAppLaunchCmdAddRecord](#) launch code to silently add an item (the email) to the iMessenger outbox database. You must pass all the needed information in the parameter block. To prevent iMessenger from displaying the

message in its editor, set the `edit` field in the parameter block to `false`.

When launched via the `sysAppLaunchCmdAddRecord` launch code, the iMessenger application returns an error code, or 0 if there was no error.

To send a launch code to the iMessenger application, you will need obtain its database id. You can use

[DmGetNextDatabaseByTypeCreator](#) and pass the constant `sysFileCMessaging` for the `creator` parameter.

Note that adding an item to the iMessenger outbox does not actually send the message over the radio. It simply stores the message in the outbox until the user later opens iMessenger and chooses to send queued messages. This always gives the user control over when the radio is used.

New `keyDownEvent` Key Codes

The OS on the Palm VII device provides special `keyDownEvent` virtual key codes to support the wireless capabilities. These include:

- `vchrHardAntenna`, which signals that the user has raised the antenna, activating the radio
- `vchrRadioCoverageOK`, which signals that the unit is within radio coverage following a coverage check
- `vchrRadioCoverageFail`, which signals that the unit is outside radio coverage following a coverage check, and thus cannot communicate with the Palm.Net system

Virtual key codes are passed in the `chr` field of a `keyDownEvent` data block, with the `commandKeyMask` bit set in the `modifiers` field, as described in the section “[keyDownEvent](#)” on page 125 of the *Palm OS SDK Reference*.

Normally, you ignore these events in your application event handler, and let the system event handler handle them. For example, the `vchrHardAntenna` event causes the system to invoke the Launcher and switch to the Palm.Net category. If you want to do something different in your application, you must trap and handle the event in your application event handler.

Alternatively, if you want your application to have control over the antenna (avoiding having the system switch to the Launcher on a `vchrHardAntenna` event), you can open the Internet library when your application starts, by calling `INetLibOpen`. You need to open the Internet library with the default or wireless configuration. When your application exits, you must close the Internet library by calling `INetLibClose`. For more information about using the Internet library, see [Chapter 11](#), “[Network Communication](#).”

Over the Air Characters

One of the overriding user interface design goals of the Palm VII system is to always give the user control when making a wireless transaction, partly because of the costs associated with doing so. In order that the user can recognize when an action causes a wireless transaction, you must use a special character in user interface buttons that cause wireless transactions. This alerts the user that tapping the button will result in a wireless transaction and its associated cost and latency. The user must never be surprised that a wireless transaction has occurred as a result of an action they initiated.

Applications that cause data to be transmitted from the Palm VII device must use two special characters in their user interface buttons, as shown in [Figure 12.2](#).

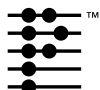
Figure 12.2 Over the Air Characters



If you have a button, that when tapped, causes data to be transmitted, the button text must end with the “Over the air” character (`chrOta`). This alerts the user that tapping the button will cause data transmission and incur possible airtime charges.

If you have a button, that when tapped, causes data to be transmitted securely, the button text must end with the “Over the air secure” character (`chrOtaSecure`). This alerts the user that tapping the button will cause secure data transmission and incur possible airtime charges.

Note that the Clipper application automatically adds these special characters when rendering remote hyperlinks or buttons. You only need to explicitly add these characters if you are building an application that doesn’t use this capability of Clipper.



Localized Applications

When you write an application (or any other type of software) that is going to be localized, 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 for localized applications, focusing on the text manager and international manager, which were introduced in Palm OS® version 3.1, and the overlay manager, which is introduced in Palm OS version 3.5. The chapter covers:

- [Localization Guidelines](#)
- [Using Overlays to Localize Resources](#)
- [Text Manager and International Manager](#)
- [Characters](#)
- [Strings](#)
- [Dates](#)
- [Numbers](#)
- [Compatibility Information](#)
- [Notes on the Japanese Implementation](#)
- [Summary of Localization](#)

This chapter does not cover how to actually perform localization of resources. For more information on this subject, see your tools documentation.

Localization Guidelines

When you start planning for the localized version of your application, follow these guidelines:

- 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](#)" in this chapter 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.
- 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 Determining a String's Contents](#)" in this chapter. 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 OS device.
- Remember that user interface elements such as lists, fields, and tips scroll if you need more space.

The chapter "[Good Design Practices](#)" 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). Note that each Palm OS device supports a single locale.

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 *Using the PRC to Overlay Tool* for more information on creating overlays.

When a PRC file is opened on a system that supports overlays, the overlay manager determines what the locale is for this device 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 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 request a database pointer, you'll receive a pointer to the base database, not the overlay. You can simply make

Localized Applications

Using Overlays to Localize Resources

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 pointer to the most recently opened database, which is typically the overlay version of the database. Passing this pointer 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](#), which returns a structure identifying the locale on this device.

Text Manager and International Manager

The Palm OS provides two managers that help you work with localized 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**.

As you know, 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 OS device supports one language and one character encoding to represent the characters required by that language. Although the Palm OS supports multiple character encodings, a given device uses only one of those encodings. For example, a French device would probably use a character encoding similar to the Microsoft® Windows® code page 1252 character encoding (an extension of ISO Latin 1), while a Japanese device would use a character encoding similar to Microsoft Windows code page 932 (an extension of Shift JIS). Code page 932 is not supported on the French device, and code page 1252 is not supported on the Japanese device even though they both use the same version of Palm OS. No matter what the encoding is on a device, 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 devices and a backslash on all others.

Localized Applications

Text Manager and International Manager

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 device supports (as long as it supports the text manager).

The international manager's job is to detect which character encoding a device 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.

The text manager and international manager are supported starting in Palm OS version 3.1. If your application should work on older systems, you should test for the presence of these managers before using text manager calls. [Listing 13.1](#) shows how.

Listing 13.1 Testing for text and international managers

```
UInt32 intlMgrAttr;
if (FtrGet(sysFtrCreator, sysFtrNumIntlMgr,
    &intlMgrAttr) != 0)
    intlMgrAttr = 0;
if (intlMgrAttr & intlMgrExists) {
    // If international manager exists, so does the
    // text manager.
    // Use text manager calls.
}
```

NOTE: You can still use the text manager and be compatible with earlier releases if you link your application with the PalmOSGlue library. See the section "[Compatibility Information](#)" for more information.

Characters

Depending on the device's supported language, the 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 in a localized application. 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 = 'â'; // WRONG! Don't use.
```

This statement may work on a Latin system, but it would cause problems on an Asian-language system because the `â` 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 a gray square in place of a missing double-byte character and an open rectangle in place of a missing single-byte rectangle (see [Figure 13.1](#)).

Figure 13.1 Missing/invalid characters



Missing single-byte character



Missing or invalid double-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](#) 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 whether the character is alphanumeric,

etc. 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 “[Compatibility Information](#)” for more information on the PalmOSGlue library.)

Therefore, when you check for a virtual character, first check the command bit in the event record. If the command bit is set, then the character is virtual. See [Listing 13.2](#).

Listing 13.2 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 device. You can retrieve the character encoding from the system feature set using the `FtrGet` function as shown in [Listing 13.3](#).

Listing 13.3 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 = TxtEncodingName(encoding);
```

Strings

On systems that support the international manager and 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.

Using characters of different sizes in a string has implications for manipulating strings, searching strings, and implementing the global find facility in your application. This section describes how to perform all of these tasks using text manager functions. It also describes how to create and display dynamically computed strings and how to display error messages.

- [Manipulating Strings](#)
- [Performing String Pointer Manipulation](#)
- [Truncating Displayed Text](#)
- [Comparing Strings](#)
- [Global Find](#)
- [Dynamically Determining a String's Contents](#)

TIP: Many of the existing 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.

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](#) retrieves the next character in a string.
- [TxtGetPreviousChar](#) retrieves the previous character in a string.
- [TxtSetNextChar](#) 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 13.4](#) shows how to iterate through a string character by character until a particular character is found.

Listing 13.4 Iterating through a string or text

```
Char* buffer; // assume this exists
Int16 bufLen = StrLen(buffer);
// Length of the input text.
WChar ch = 0;
UInt16 i = 0;
while ((i < bufLen) && (ch != chrAsterisk))
    i+= TxtGetNextChar(buffer, i, &ch);
```

The text manager also contains functions that let you determine the size of a character without iterating through the string:

- [TxtCharSize](#) returns how much space a given character will take up inside of a string.
- [TxtCharBounds](#) determines the boundaries of a given character within a given string.

Listing 13.5 Working with arbitrary limits

```
UInt32* charStart, charEnd;
Char* fldTextP = FldGetTextPtr(fld);
TxtCharBounds(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.

```
// 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);
TxtSetNextChar(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](#) - 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](#) - 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.

Comparing Strings

Use the text manager functions [TxtCompare](#) and [TxtCaselessCompare](#) 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 SDK Reference* for more information.

Note that `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](#). 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.

[Listing 13.6](#) and [Listing 13.7](#) show how to implement a global find facility on all systems (whether the text manager exists or not), and how to implement a response to [sysAppLaunchCmdGoto](#), which is the system's request that the matching record be displayed. These two listings are only code excerpts. For the complete implementation of these two functions, see the example code in your development environment.

Listing 13.6 Implementing global find

```
static void Search (FindParamsPtr findParams)
{
    UInt16 recordIndex = 0;
    DmOpenRef dbP;
    UInt16 cardNo = 0;
    LocalID dbID;
    MemoDBRecordPtr memoPadRecP;

    // 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 13.7 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 Determining a String's Contents

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](#). `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](#), which allows you to replace up to `^9`; however, `TxtReplaceStr` only allows you to replace one of these sequences at a time.) The `PalmOSGlue` library defines a function

`TxtGlueParamString`, which calls `TxtParamString` if it exists or else implements the Latin equivalent of it.

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 13.8 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);
}
```

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 different for the different languages, though they can be overridden.

To check the system preferences call [PrefGetPreference](#) with one of the values listed in the second column of [Table 13.1](#). The third column lists an enumerated type that helps you interpret the value.

Table 13.1 Date and time preferences

Preference	Name	Returns a value of type
Date formats (i.e., month first or day first)	prefDateFormat	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)

To work with dates in your code, use the Date and Time Manager API. It contains functions such as [DateToAscii](#), [DayOfMonth](#), [DayOfWeek](#), and [DaysInMonth](#), 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 device's country by doing the following (see [Listing 13.9](#)):

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 13.9 Working with numbers

```
// store numbers using US conventions.
Char *jackpot = "20,000,000.00";
Char thou; // thousand separator
Char dp; // decimal separator

// Retrieve current country's preferences.
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);
```

Compatibility Information

If you want to maintain backward compatibility with earlier releases but you still want to use the localization features described in this chapter, you can link your application with the library `PalmOSGlue` (`PalmOSGlue.lib` or `libPalmOSGlue.a`). This library provides these features for versions 2.0 and 3.0.

When you use `PalmOSGlue`, you use the text manager in the same way as described in this chapter, but the names of the functions are different. For example, `TxtFindString` is named `TxtGlueFindString` in the `PalmOSGlue`. (See the chapter [“PalmOSGlue Library”](#) on page 1183 of the *Palm OS SDK Reference* for a complete mapping table.) When you make a call to a glue function (`TxtGlueFunc`, `FntGlueFunc`, or `WinGlueFunc`), the code in `PalmOSGlue` either uses the text manager or international manager on the ROM or, if the managers don't exist, executes a simple Latin equivalent of the function.

`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.

Palm OS version 3.1 contains the following changes from previous releases that affect strings, text, and localization. These changes may affect you if you're updating an application written to run on a prior release or if you want to maintain backward compatibility with prior releases:

- The `KeyDownEvent` structure's `chr` field (which contains the input character) has been changed from a `Word` to a `WChar`. The `chr` field may contain a multi-byte character, so you should never copy the `chr` field into a `Char` variable or pass it to a function using a `Char` parameter. Always use `WChar`.

- Some of the special Palm OS glyphs in the high ASCII range (such as the shortcut stroke and the command stroke) have been moved down into the control code range, and other characters (such as the numeric space and horizontal ellipsis) have been copied into the control range so that they're guaranteed to exist in every encoding. For the numeric space and horizontal ellipsis, you can use the macros [ChrNumericSpace](#) and [ChrHorizEllipsis](#) to return the appropriate character regardless of the character map. In PalmOSGlue, these two macros are named `TxtGlueGetNumericSpaceChar` and `TxtGlueGetHorizEllipsisChar`, respectively.
- The four playing-card characters have been moved from the high ASCII range in the standard four fonts to the 9-point Symbol font.
- Character attribute functions and macros are now obsolete and have been replaced by functions and macros in the text manager.
- The String Manager functions [StrChr](#) and [StrStr](#) now treat buffers as characters, not arbitrary byte arrays. If you previously used these functions to search data buffers, your code may no longer work.

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](#)
- [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 the single-byte Katakana characters and the multi-byte characters are not bolded.

Japanese Character Input

On current Japanese devices, 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 silkscreen buttons added to the Japanese device control the FEP transliteration and conversion process. These four FEP buttons are arranged vertically between the current left-most silkscreen buttons and the Graffiti® 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 space stroke acts as a shortcut for the conversion FEP button and the Graffiti return stroke acts as a shortcut for the confirm FEP button.

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 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

Text Manager

Working With Multi-Byte Characters

[TxtCharBounds](#)

[TxtPreviousCharSize](#)

[TxtByteAttr](#)

[TxtCharSize](#)

[TxtNextCharSize](#)

Changing Text

[TxtReplaceStr](#)

[TxtGetTruncationOffset](#)

[TxtSetNextChar](#)

[TxtTransliterate](#)

Accessing Text

[TxtGetNextChar](#)

[TxtGetChar](#)

[TxtGetPreviousChar](#)

[TxtWordBounds](#)

Searching/Comparing Text

[TxtCaselessCompare](#)

[TxtFindString](#)

[TxtCompare](#)

Localized Applications

Summary of Localization

Text Manager

Obtaining a Character's Attributes

[TxtCharIsAlNum](#)

[TxtCharIsDigit](#)

[TxtCharIsLower](#)

[TxtCharIsSpace](#)

[TxtCharIsValid](#)

[TxtCharIsCntrl](#)

[TxtCharIsPunct](#)

[TxtCharWidth](#)

[TxtCharIsAlpha](#)

[TxtCharIsGraph](#)

[TxtCharIsPrint](#)

[TxtCharIsUpper](#)

[TxtCharXAttr](#)

[TxtCharIsHex](#)

[TxtCharAttr](#)

Obtaining Character Encoding information

[TxtStrEncoding](#)

[TxtMaxEncoding](#)

[TxtEncodingName](#)

[TxtCharEncoding](#)

Localizing Numbers

[StrLocalizeNumber](#)

[LocGetNumberSeparators](#)

[StrDelocalizeNumber](#)

International Manager

[IntlGetRoutineAddress](#)

Overlay Manager

[OmGetCurrentLocale](#)

[OmGetIndexedLocale](#)

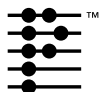
[OmGetRoutineAddress](#)

[OmSetSystemLocale](#)

[OmGetSystemLocale](#)

[OmLocaleToOverlayDBName](#)

[OmOverlayDBNameToLocale](#)



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](#)
- [Understanding the Try-and-Catch Mechanism](#)
- [Using the Error Manager Macros](#)
- [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 device 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 (0)</code>	Doesn't compile in any error calls.
<code>ERROR_CHECK_PARTIAL (1)</code>	Compiles in only <code>ErrDisplay</code> and <code>ErrFatalDisplayIf</code> calls.
<code>ERROR_CHECK_FULL (2)</code>	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.

Understanding the Try-and-Catch Mechanism

The error manager is aware of the machine state of the Palm OS device 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 14.1 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.

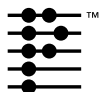
You can nest multiple ErrTry blocks. For example, if you wanted to perform some cleanup at the end of CreateTable in case of error,

- 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

ErrDisplay	ErrDisplayFileLineMsg
ErrFatalDisplayIf	ErrNonFatalDisplayIf
ErrThrow	ErrAlert



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™ device, 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 device, 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 sublaunches the standard IO application when the user types in a command line and Return (using Graffiti®). 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 15.1](#).

Listing 15.1 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 15.2](#).

Listing 15.2 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

<u>fgetc</u>	<u>putchar</u>
<u>fgets</u>	<u>puts</u>
<u>fprintf</u>	<u>SioAddCommand</u>
<u>fputc</u>	<u>SioMain</u>
<u>fputs</u>	<u>sprintf</u>
<u>getchar</u>	<u>system</u>
<u>gets</u>	<u>vfprintf</u>
<u>printf</u>	<u>vsprintf</u>
<u>putc</u>	

Standard IO Provider Functions

<u>SioClearScreen</u>	<u>SioHandleEvent</u>
<u>SioExecCommand</u>	<u>SioInit</u>
<u>SioFree</u>	

Index

Numerics

- 0.01-second timer 227
- 1.0 heaps 169
- 1-second timer 226
- 2.0 heaps 169
- 3.0 heaps 169
- 32K jumps 28
- 68328 processor 157

A

- alarm manager 189–195
 - and alarm sound 190
 - procedure alarms 193
 - reminder dialog boxes 190
- alarm sound 190, 208
- alarms 25
- alert manager 83
- alerts, system-defined 83
- allocating handles 28
- AlmGetAlarm 191
- AlmGetProcAlarm 194
- AlmSetAlarm 190, 192
- AlmSetProcAlarm 194
- ANSI C libraries 19
- AppInfoType 108
- APPL database 29
- application design
 - accessibility 35
 - assigning version number 30
 - buttons 36
 - command buttons 35
 - data entry 39
 - dialogs 35
 - ease of use 35
 - handling system messages 26
 - minimizing taps 35
 - removing deleted records 30
 - switching applications 38
 - using lists 105
- application icon 26, 37
 - name 25
 - size 26
- application launcher 50
- application name 25

- application preferences database 25
- application record database 25
- application startup 49–63
- application-defined features 197
- applications
 - control flow 18
 - event driven 18
- AppNetRefnum 278, 279
- AppNetTimeout 279
- architecture of memory 157
- auto-off 223
 - timer 74
- auto-repeat 74

B

- back-up of data to PC 156
- BarBeamBitmap 104
- BarCopyBitmap 104
- BarCutBitmap 104
- BarDeleteBitmap 104
- BarInfoBitmap 104
- BarPasteBitmap 104
- BarSecureBitmap 104
- BarUndoBitmap 104
- battery 223
 - conservation using modes 222
 - life, maximizing 222
- battery life and serial manager 236
- baud rate, parity options 237
- beaming 265
- Berkeley Sockets API 274
 - mapping example 277
- bind (Berkeley Sockets API) 289
- bitmap family 117
- bitmaps 116
 - bitmap family 117
 - masking 118
 - transparent 118
- BitmapType 117
- bits behind menu bar 101
- BmpCreate 118
- booting 218
- button objects 86
- Button resource 35, 78

Index

- highlighting 86
- buttons
 - assignment by end-user 38
 - choosing number 36
 - in dialog 44
 - position 43
 - traversing categories 38
- byte ordering 234
- C**
- C library
 - and float manager 228
 - and string manager 133
- C library calls 33
- calibrating digitizer 138
- carriage returns 98
- categories 37, 38
 - maximum number 27
 - traversing with button 38
- CategoryGetName 111
- categoryHideEditCategory 113
- CategoryInitialize 109
- CategorySetTriggerLabel 111
- changing serial port settings 237
- Char 331
- Chars.h 332
- check box object 91
- Checkbox 78
- ChrHorizEllipsis 347
- ChrNumericSpace 347
- chunks 165
 - resizing 168
 - size 168
- Click sound 208
- clipboard 41
- clock, real-time 226
- close (Berkeley Sockets API) 289
- close-wait state 286
- closing net library 286
- closing serial link manager 259
- closing serial port 237
- CMP 235
- CodeWarrior IDE 20
- color translation table 129

- colorTableRsc 128
- command buttons 35
- command line applications 357
- command toolbar 102
- conduit 17
- configuration, net library 279
- Confirmation sound 208
- connect (Berkeley Sockets API) 289
- connection management protocol 235
- connection manager 254
- connectivity 233
- connector (external) 234
- conserving battery using modes 222
- Constructor 20
- control flow 18
- control objects 86
- conventions for naming 27
- CoreTraps.h 47
- CRC-16 255
- creating a chunk 168
- creating database 177
- creating resources 183
- creator ID 29
- ctlEnterEvent 87, 88, 89, 91, 92, 93, 95
- ctlExitEvent 90, 95
- CtlHandleEvent 86
- CtlNewControl 126
- ctlRepeatEvent 90, 95
- ctlSelectEvent 88, 89, 91, 92, 95, 112
- CTS timeout 237
- custom UI element 122

D

- data entry, Graffiti 39
- data manager 173
 - using 177
- database headers 175
 - fields 175
- database ID
 - and launch codes 58
- database version number 30
- databases 18, 160, 174
 - getting and setting information 178
 - overlays 326

- date and time manager 226
- DateFormatType 344
- default receive queue, restoring 239
- defaultCategoryRscType 115
- deleted records 26, 30
- deleting database 177
- deleting records 30
- desktop link protocol 235
- Desktop Link Server 257
- Details button 38
- Details dialog format 40
- dialog boxes (reminder) 190
- dialogs 29
 - design 43
 - online help 43
- digitizer 135
 - after reset 219
 - and pen manager 138
 - and pen queue 72
 - calibrating 138
 - dimensions 138
 - pen stroke to key event 72
 - polling 226
 - sampling accuracy 138
- DLP 235
- dmCategoryLength 109
- DmCloseDatabase 214
- DmCreateDatabase 177, 181
- DmDatabaseInfo 30, 178, 181
- DmDatabaseSize 178
- DmDeleteDatabase 177, 181
- DmDeleteRecord 30
- DmFindDatabase 178, 213
- DmFindRecordById 214
- DmFindResource 328
- DmGet1Resource 328
- DmGetDatabase 178
- DmGetRecord 178, 214
- DmGetResource 328
- DmGetResourceIndex 328
- DmNewHandle 109
- DmNewResource 183
- DmNextOpenResDatabase 328
- DmOpenDatabase 213

- DmOpenDatabaseByTypeCreator 211
- DmQueryRecord 178, 214
- dmRecAttrCategoryMask 111
- DmRecordInfo 111
- DmReleaseRecord 178, 214
- DmReleaseResource 182
- DmRemoveRecord 30
- DmResizeRecord 178
- DmSetDatabaseInfo 30, 178
- DmWrite 198
- double taps 37
- down arrow 99
- doze mode 221
- draw state 80
- draw window 82
- drawing state 80
- drivers, restarting 219
- dynamic heap
 - soft reset 218
- dynamic memory 28
- dynamic menus 101
- dynamic RAM 157

E

- editable items
 - labels 44
- edit-in-place 29
- ErrDisplay 351, 353
- ErrFatalDisplayIf 349, 352, 353
- errno 279
- ErrNonFatalDisplayIf 349
- error manager 351–355
 - try-and-catch mechanism 353
- Error sound 208
- ERROR_CHECK_LEVEL 352, 353
- ErrThrow 354
- event loop 67–71
 - example 67
 - example program 31
- event-driven applications 18
- events
 - naming conventions 27
 - overview 65–75
- EvtGetEvent 84, 222

Index

EvtResetAutoOffTimer 74, 239

examples

- event loop 67
- startup routine 53
- stop routine 60

exchange manager 265

- launch codes sent by 267

ExgDoDialog 267

F

fcntl 289

feature manager 195–200

feature memory 198

features

- application-defined 197
- feature memory 198
- system version 196

feedback slider 93

Field 79

field objects 97

- events 98
- line feeds vs. carriage returns 98

file streaming functions 187

finding database 178

FindStrInStr 340

finger navigation 37

FIR 270

flags, launch flags 50

fldEnterEvent 98

fldGadgetEnterEvent 124

FldHandleEvent 98

FldNewField 126

float manager overview 228

flushing serial port 239

FntDefineFont 134, 135

FntSetFont 134

FntWidthToOffset 339

font labels 44

FontSelect 134, 135

Form Bitmap 116

form objects 82

- event flow 83

formGadgetDeleteCmd 125

formGadgetEraseCmd 125

formGadgetHandleEventCmd 124

FormGadgetHandler 122

forms 19

FrmAlert 83

FrmCustomAlert 83

frmGadgetDrawCmd 124

frmGadgetMiscEvent 125

FrmNewBitmap 126

FrmNewForm 126

FrmNewGadget 126

FrmNewLabel 126

frmOpenEvent 83, 89

FrmRemoveObject 126

FrmSetGadgetHandler 122

FrmSetMenu 101

FrmValidatePtr 126

FtrGet 197, 198, 335

FtrPtrNew 198

FtrSet 198

FtrUnregister 198

function naming conventions 27

G

gadget resource 122

getdomainname (Berkeley Sockets API) 293

gethostbyaddr (Berkeley Sockets API) 293

gethostbyname (Berkeley Sockets API) 293

gethostname (Berkeley Sockets API) 294

getpeername (Berkeley Sockets API) 289

getservbyname (Berkeley Sockets API) 294

getsockname (Berkeley Sockets API) 289

getsockopt (Berkeley Sockets API) 290

gettimeofday() (Berkeley Sockets API) 294

global find 25

- and private records 25

global variables 28

- erasing 219

Graffiti 39, 41

- customizing behavior 136

Help 137

Help character 137

Graffiti manager 135

Graffiti navigation 38

Graffiti recognizer 71

Graffiti reference 40
Graffiti Shift
 getting and setting state 136
Graffiti shortcut 102
Graffiti ShortCuts database 137
Graffiti status indicator area
 not obscuring 43
graffitiReferenceChr 137
GrfProcessStroke 136

H

handles, allocation 28
handshaking options 237
hard reset 218, 219
hardware button presses and key manager 137
heap fragmentation 28
heap header 164
heap space 28
heaps
 and soft reset 161
 in Palm OS 1.0 169
 in Palm OS 2.0 169
 in Palm OS 3.0 169
 overview 161
 RAM and ROM based 155
 structure 164
Help ID 44
highlighting button objects 86
HotSync 30
htonl (Berkeley Sockets API) 294
htons (Berkeley Sockets API) 294

I

icons, application 26
ID
 local 163
 See Also creator ID
IDE 20
inet_addr (Berkeley Sockets API) 295
inet_lnaof (Berkeley Sockets API) 295
inet_makeaddr (Berkeley Sockets API) 295
inet_netof (Berkeley Sockets API) 295
inet_network (Berkeley Sockets API) 295
inet_ntoa (Berkeley Sockets API) 295

infrared library 269
initialization
 global variables 53
input devices 17
insertion point object 132
interface(s) used by net library 280
international manager 325
Internet 278
Internet applications 274
Internet library
 RAM requirement 301
interrupting Sync application 224
IR library 269
 accessing 271
IrDA stack 269
IrLAP 270
IrLMP 270

K

kernel 223
key events
 from pen strokes 71
key manager 137
key queue 73
keyboard 40
KeyCurrentState 138
keyDownEvent 98, 99, 138, 140, 207, 331, 346
KeyRates 138

L

label resource 120
labels, font 44
launch codes 18, 49–63
 and returned database ID 58
 code example 51
 creating 59
 handling 24
 launch flags 50
 parameter blocks 50
 predefined 61
 sent by exchange manager 267
 summary 61
 SysBroadcastActionCode 58
 use by application 58

Index

- launch flags 50
- launcher 37
 - application icon name 25
- launching applications 50
- LCD screen 79
- left arrow 99
- libPalmOSGlue.a 346
- line feeds 98
- list objects 105
- List resource 79
- listen (Berkeley Sockets API) 291
- local IDs 163, 174
- localization
 - general guidelines 326
- LocGetNumberSeparators 345
- locking a chunk 168
- Loop-back Test 257
- low-battery warnings 25
- lstEnterEvent 107
- LstHandleEvent 106
- LstNewList 126
- lstSelectEvent 107

M

- mailbox queue 274
- managers
 - naming convention 189
 - overview 19
- masking 118
- master pointer table 165
- maximizing battery life 222
- MemHandleFree 168
- MemHandleLock 168, 214
- MemHandleNew 109, 167
- MemHandleResize 168
- MemHandleSize 168
- MemHandleUnlock 168
- MemMove 169
- memory architecture 157
- memory management
 - architecture 157
 - Introduction 155
- memory manager
 - chunks 160

- memory manager *See Also* data manager
- memory manager *See Also* resource manager
- MemPtrNew 168
- MemPtrRecoverHandle 168
- MemPtrUnlock 214
- MemSet 169
- menu bar objects 99
- Menu Bar resource 79
- menu bars
 - and user actions 100
 - bits behind 101
- Menu Resource 79
- MenuAddItem 101
- MenuCmdBarAddButton 103
- menuCmdBarOpenEvent 103
- menuDownEvent 102
- menuEvent 101, 102
- MenuHandleEvent 101
- MenuHideItem 101
- menuOpenEvent 101
- menus 41
 - dynamic 101
 - shortcut 102
- MenuShowItem 101
- MIME data type 266
- Modem Manager 235
- modes 38, 220
 - efficient use 222
- modifying Graffiti shortcuts 137
- Motorola byte ordering 234
- moving memory 169
- multitasking kernel 223

N

- naming conventions 27
- navigation 38
- net library
 - closing 286
 - open sockets maximum 288
 - opening and closing 284
 - OS requirement 275
 - overview 274–277
 - preferences 279
 - RAM requirement 275

- setup and configuration 279
- version checking 287
- net protocol stack 274
 - as separate task 275
- netIFCreatorLoop 280
- netIFCreatorPPP 281
- netIFCreatorSLIP 281
- netlib interface introduction 274
- NetLibIFAttach 280
- NetLibIFDetach 280
- NetLibIFGet 280
- NetLibIFSettingGet 281
- NetLibIFSettingSet 281
- NetLibSettingGet 284
- NetLibSettingSet 284
- NetSocket.c 279
- network device drivers 274
- network interface 275
- network services 273
- new serial manager 240
- nilEvent 84
- notification client 200
- notification handlers 203, 204
- notification manager 201
- notifications 200
- ntohl (Berkeley Sockets API) 294
- ntohs (Berkeley Sockets API) 294

O

- OBEX 270
- OmGetCurrentLocale 329
- online help 44
- on-screen keyboard 40
- open sockets maximum (net library) 288
- opening net library 284
- opening serial link manager 259
- opening serial port 237
- optimization 28
 - dynamic memory 28
 - sorting 28
- over the air characters 322
- overlays 326
- overloading buttons 37

- overview of net library 274–277
- ovly resource 327

P

- packet assembly/disassembly protocol 235
- packet footer, SLP 256
- packet header, SLP 256
- packet receive timeout 259
- PADP 235, 257
- palettes 128
- PalmOSCompatibility.h 48
- PalmOSGlue.lib 346
- parameter blocks 50
- patches, loading during reset 219
- PC connectivity 16, 156
- pen 80
- pen location polling 138
- pen manager 138
- pen queue 72, 138
- pen strokes and key events 71
- penDownEvent 87, 88, 89, 91, 92, 93, 94, 95, 98, 106, 107
- penUpEvent 87, 88, 89, 90, 91, 92, 94, 95, 98, 101, 107, 136
- performance 28
- physical scrolling 39
- PICT 116
- PilotMain 50
 - code example 51
- PluginInfoType 296
- pluginMaxNumOfCmds 297
- pluginNetLibCallUIProc 300
- popSelectEvent 107
- Popup list 35, 79
- Popup trigger 79
- popup trigger object 87
- port ID for socket 259
- power 17
- power modes 220
- predefined launch codes 61
- prefAlarmSoundLevelV20 217
- prefAlarmSoundVolume 209, 217
- prefDateFormat 344

Index

- preferences 39
 - application-specific 53
 - auto-off 223
 - quick switch 39
 - restoring 25
 - saving 25
 - short cuts 137
 - system 53
- preferences database
 - net library 279
- prefGameSoundLevelV20 217
- prefGameSoundVolume 209, 217
- PrefGetAppPreferences 25
- PrefGetPreference 209, 217, 344, 345
- PrefGetPreferences 215
- PrefSetAppPreferences 25
- PrefSetPreference 215
- prefSysSoundLevelV20 217
- prefSysSoundVolume 209, 217
- prefTimeFormat 344
- prefWeekStartDay 344
- PrgHandleEvent 84
- PrgStartDialog 84, 85
- PrgUpdateDialog 84
- private records 25
- procedure alarms 193
- progress manager 84
- Push button 35, 78
- push button objects 90
 - event flow 91

Q

- quick switch, preferences 39

R

- RAM 17
- RAM store 155
- RAM use 156
- read (Berkeley Sockets API) 291
- real-time clock 226, 227
- receive queue, restoring 239
- receiving SLP packet 258
- records 18, 173
- recv (Berkeley Sockets API) 291

- recvfrom (Berkeley Sockets API) 291
- recvmsg (Berkeley Sockets API) 291
- reference number for socket 259
- registering for a notification 201
- reminder dialog boxes 190
- Remote Console 257
- Remote Console packets 257
- Remote Debugger 257, 259
- remote inter-application communication 235
- Remote Procedure Call packets 257
- remote procedure calls 235, 259
- Remote UI 257
- repeat control objects 89
- Repeating button 78
- ResEdit
 - resource naming conventions 27
- reset 218
 - digitizer screen 219
 - hard reset 219
 - loading patches 219
 - soft reset 219
- resource database header 180
- resource manager 180
 - using 181
- resources
 - gadget 122
 - label 120
 - storing 180
- response time 224
- restoring default receive queue 239
- restoring preferences 25
- resumeSleepChr 207
- RGBColorType 128
- RIAC 235
- right arrow 99
- ROM store 155
- ROM use 156
- ROM, retrieving serial number 224
- RPC 235, 259
- RS232 signals 236
- running mode 221

S

- saving preferences 25
- sclEnterEvent 122

- sclExitEvent 122
- sclRepeatEvent 122
- SclSetScrollBar 121
- scptLaunchCmdListCmds 296
- scptLaunchCmdExecuteCmd 61, 296, 297
- scptLaunchCmdListCmds 61, 296
- screen layout 42
- screen size 15, 79
- scrollbar objects 120
- scrolling 39
- select (Berkeley Sockets API) 291
- Selector trigger 78
- selector trigger object 88
- send (Berkeley Sockets API) 292
- sending stream of bytes 238
- sendmsg (Berkeley Sockets API) 292
- sendto (Berkeley Sockets API) 292
- SerClearErr 237
- serCtlBreakStatus (in SerCtlEnum) 240
- serCtlEmuSetBlockingHook (in SerCtlEnum) 240
- SerCtlEnum 239
- serCtlFirstReserved (in SerCtlEnum) 239
- serCtlHandshakeThreshold (in SerCtlEnum) 240
- serCtlMaxBaud (in SerCtlEnum) 240
- serCtlStartBreak (in SerCtlEnum) 239
- serCtlStartLocalLoopback (in SerCtlEnum) 240
- serCtlStopBreak (in SerCtlEnum) 239
- serCtlStopLocalLoopback (in SerCtlEnum) 240
- serErrAlreadyOpen 237
- serErrLineErr 238
- serial communication 233
- serial link manager 258
 - opening 259
- serial link protocol 235, 255, 256, 258
- serial manager 235, 236, 240
 - prolonging battery life 236
- serial number, retrieving 234
- serial port 25
 - changing settings 237
 - closing 237
 - flushing 239
 - opening 237
- SerOpen 237
- SerReceive 238
- SerReceiveCheck 238
- SerReceiveFlush 239
- SerReceiveWait 238
- SerSend 238
- SerSendWait 238
- SerSetReceiveBuffer 239
- SerSetSettings 237
- setdomainname (Berkeley Sockets API) 294
- sethostname (Berkeley Sockets API) 294
- setsockopt (Berkeley Sockets API) 292
- settimeofday (Berkeley Sockets API) 294
- setup, net library 279
- shortcut for menus 102
- shortcuts 41
- shortcuts, Graffiti 137
- shutdown (Berkeley Sockets API) 293
- silk-screened icons, not obscuring 43
- SIR 270
- sleep mode 220, 221
 - and current time 226
 - and real-time clock 226
- sliders 92
- SlkClose 259
- SlkCloseSocket 259
- slkErrAlreadyOpen 259
- SlkOpen 259
- SlkOpenSocket 259
- SlkPktHeaderType 260
- SlkReceivePacket 260, 262
- SlkSendPacket 260
- SlkSocketListenType 260
- SlkSocketPortID 259
- SlkSocketRefNum 259
- SlkSocketSetTimeout 259
- SlkWriteDataType 260
- SLP 235, 255
- SLP packet 255
 - footer 256
 - header 256
 - receiving 258
 - transmitting 258
- SMF 209
- SMFs in databases 212
- sndCmdFrqOn 209

Index

- SndCommandType 210
- SndCreateMidiList 213, 218
- SndDoCmd 208, 209, 210, 218
- SndPlaySMF 208, 209, 214, 218
- SndPlaySystemSound 208, 210
- SndSetDefaultVolume 215
- SndSmfOptionsType 209
- SO_ERROR (Berkeley Sockets API) 291
- SO_KEEPALIVE (Berkeley Sockets API) 290, 292
- SO_LINGER (Berkeley Sockets API) 290, 292
- SO_TYPE (Berkeley Sockets API) 291
- socket (Berkeley Sockets API) 293
- socket listener 260, 262
- socket listener procedure 260, 262
- sockets, opening serial link socket 259
- soft reset 161, 218, 219
 - dynamic heap 218
- sorting 28
- sound manager 207–218
- stack space 29
- standard IO applications 357
- startup 49–63
- startup routine, example 53
- Startup sound 208
- state information, storing 25
- stop routine example 60
- storage heaps, erasing 219
- storage RAM 157
- StrDelocalizeNumber 345
- string manager 133
- StrLocalizeNumber 345
- strokes
 - capturing 73
- structure elements, naming convention 27
- summary of launch codes 61
- switching applications 38
- switching categories 38
- switching views 38
- Sync application 223
- synchronization messages 25, 26
- sys_socket.h 276, 279
- SysAppLaunch 58, 140
- sysAppLaunchCmdAddRecord 61
- sysAppLaunchCmdAlarmTriggered 61, 191, 192
- sysAppLaunchCmdCountryChange 61
- sysAppLaunchCmdDisplayAlarm 61, 191, 193
- sysAppLaunchCmdExgAskUser 61, 267
- sysAppLaunchCmdExgReceiveData 61, 268
- sysAppLaunchCmdFind 62
- sysAppLaunchCmdGoto 62, 269, 340
- sysAppLaunchCmdGoToURL 62
- sysAppLaunchCmdInitDatabase 62
- sysAppLaunchCmdLookup 62
- sysAppLaunchCmdNormalLaunch 24, 50, 53
- sysAppLaunchCmdNotify 202
- sysAppLaunchCmdOpenDB 62
- sysAppLaunchCmdPanelCalledFromApp 39, 62
- SysAppLaunchCmdReset 219
- sysAppLaunchCmdReturnFromPanel 39, 62
- sysAppLaunchCmdSaveData 62
- sysAppLaunchCmdSyncNotify 62
- sysAppLaunchCmdSystemLock 62
- sysAppLaunchCmdSystemReset 63, 219
- sysAppLaunchCmdTimeChange 63
- sysAppLaunchCmdURLParams 63
- SysAppLauncherDialog 139
- SysBatteryInfo 223
- SysBroadcastActionCode 58
- SysCurAppDatabase 58
- sysFileDescStdIn 292
- sysFtrCreator 196
- sysFtrNumROMVersion 196
- SysGraffitiReferenceDialog 137
- SysLibFind 237, 271
- sysNotifyEarlyWakeupEvent 206
- sysNotifyLateWakeupEvent 206
- sysNotifyNormalPriority 203
- SysNotifyParamType 203
- SysNotifyRegister 201
- sysNotifySleepNotifyEvent 206, 207
- sysNotifySleepRequestEvent 206
- sysNotifySyncFinishEvent 202
- sysNotifySyncStartEvent 202
- SysNotifyUnregister 201
- SysReset 220
- sysResIDPrefUIColorTableBase 130
- sysResTExtPrefs 327

SysSetAutoOffTime 222
SysTaskDelay 222, 227
system event manager 71–75
system keyboard 41
system messages 25, 26
system preferences 24, 53
system tick interrupts 226
system ticks 227
 and Simulator 227
 on Palm OS device 227
system version feature 196
systemDefaultUIColorsBase 130
SystemMgr.h 61, 197, 280
SystemPreferencesTypeV10 215
SysTicksPerSecond 227
SysTraps.h 47
SysUIAppSwitch 58, 140

T

table objects 104
tAIN resource 26
taps
 double taps 37
 minimizing 35
tblSelectEvent 105
Tbmp 116
TCP/IP 273
TCP_MAXSEG (Berkeley Sockets API) 290
TCP_NODELAY (Berkeley Sockets API) 290, 292
text manager 325
tFBM 116
TimDateTimeToSeconds 192, 227
time manager 226
TimeFormatType 344
timeout
 serial link socket 259
timer 226
TimGetSeconds 227
TimGetTicks 227
timing 227
TimSecondsToDateTime 227
TimSetSeconds 227
Tiny TP 270
title bar 42

transmitting SLP packet 258
transparent bitmap 118
try-and-catch mechanism 353
 example 354
TxtCaselessCompare 339
TxtCharBounds 338
TxtCharSize 338
TxtCompare 339
TxtFindString 340
TxtGetNextChar 337
TxtGetPrevChar 337
TxtGlueCharIsVirtual 334
TxtGlueGetHorizEllipsisChar 347
TxtGlueGetNumericSpaceChar 347
TxtGlueParamString 343
TxtIsValidChar 333
TxtParamString 342
TxtReplaceStr 342
TxtSetNextChar 337

U

UDP 273
UI design 16, 33
 avoiding dialog box stacking 29
 button alignment 43
 design elements 78
 design philosophy 16, 33
 dialogs 43
 screen layout 42
 title bar 42
UI design rules
 clipboard 41
 finger navigation 37
 Graffiti navigation 38
 Graffiti status indicator area 43
 overloading buttons 37
 ready cursor 40
 silk-screened icons 43
UI objects 19
 buttons 86
 check box 91
 control objects 86
 field 97
 form 82
 insertion point 132

Index

- list 105
- menu bars 99
- popup trigger 87
- push button 90
- repeat control 89
- scrollbar 120
- selector trigger 88
- table 104
- windows 82
- UI resources
 - custom 122
- UI resources, storing 180
- UIAS 223
- UIColorGetTableEntryIndex 132
- UIColorGetTableEntryRGB 132
- UIColorSetTableEntry 132
- UIResources.r 327
- unlocking a chunk 168
- up arrow 98
- user input 41
 - cut, copy, paste, undo 41
- User Interface Application Shell 223
- user interface elements
 - storing (resource manager) 180

V

vchrHardAntenna 321

vchrRadioCoverageFail 321
vchrRadioCoverageOK 321
version checking, net library 287
version number 30

W

wait cursor 28
Warning sound 208
WChar 331
WinCreateBitmapWindow 118
window objects 82

- off-screen 82

WinDrawBitmap 118
WinDrawTruncChars 339
winEnterEvent 83, 88, 101, 107
winExitEvent 83, 89, 107
WinIndexToRGB 129
WinPaintBitmap 118
WinPalette 119, 129
WinPopDrawState 81
WinPushDrawState 81
WinRGBToIndex 129
WinSetDrawWindow 82
write (Berkeley Sockets API) 293