palmsource™

Programming Basics

Exploring Palm OS®

Written by Greg Wilson
Edited by Jean Ostrem
Technical assistance from Jesse Donaldson, Dianne Hackborn, Joe Onorato, Jie Su

PalmSource, Inc. reserves the right to revise this technical documentation from time to time, and is not obligated to notify you of any revisions.

THIS TECHNICAL DOCUMENTATION IS PROVIDED ON AN “AS IS” BASIS. NEITHER PALMSOURCE NOR ITS SUPPLIERS MAKES, AND EACH OF THEM EXPRESSLY EXCLUDES AND DISCLAIMS TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, ANY REPRESENTATIONS OR WARRANTIES REGARDING THIS TECHNICAL DOCUMENTATION, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING WITHOUT LIMITATION ANY WARRANTIES IMPLIED BY ANY COURSE OF DEALING OR COURSE OF PERFORMANCE AND ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, ACCURACY, AND SATISFACTORY QUALITY. PALMSOURCE AND ITS SUPPLIERS MAKE NO REPRESENTATIONS OR WARRANTIES THAT THIS TECHNICAL DOCUMENTATION IS FREE OF ERRORS OR IS SUITABLE FOR YOUR USE. TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, PALMSOURCE, INC. ALSO EXCLUDES FOR ITSELF AND ITS SUPPLIERS ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, EXEMPLARY OR PUNITIVE DAMAGES OF ANY KIND ARISING OUT OF OR IN ANY WAY RELATED TO THIS TECHNICAL DOCUMENTATION, INCLUDING WITHOUT LIMITATION DAMAGES FOR LOST REVENUE OR PROFITS, LOST BUSINESS, LOST GOODWILL, LOST INFORMATION OR DATA, BUSINESS INTERRUPTION, SERVICES STOPPAGE, IMPAIRMENT OF OTHER GOODS, COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR OTHER FINANCIAL LOSS, EVEN IF PALMSOURCE, INC. OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR IF SUCH DAMAGES COULD HAVE BEEN REASONABLY FORESEEN.

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

Exploring Palm OS: Programming Basics
Document Number 3107-003
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

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

Table of Contents

Part I: Concepts

3 Events and the Event Loop 43

4 Notifications 59

Part II: Reference

6 Common Launch Codes 103

About This Document

This book documents those aspects of Palm OS programming that are shared by all Palm OS applications: the basic application structure, the means by which Palm OS keeps your application apprised of user actions, and the mechanism by which applications communicate with one another and with the operating system.

The *Exploring Palm OS* Series

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

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

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

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*

- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*
- *Exploring Palm OS: Palm OS File Formats* (coming soon)

Additional Resources

- Documentation

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

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

- Training

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

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

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

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

Changes to This Document

This section describes the changes made in each version of this document.

3107-002

Minor editorial corrections.

3107-001

The first release of this document for Palm OS Cobalt, version 6.0.

Part I
Concepts

The conceptual material in this part is organized into the following chapters:

1

Programming Palm OS in a Nutshell

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

Read this chapter for a high-level introduction to Palm OS 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. Applications written for devices, specifically Palm Powered devices, are a bit different from those written for desktop computers because the Palm Powered device is designed differently than a desktop computer. As well, the way in which users interact with a device differs from the way they interact with a desktop computer.

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

Screen Size

Palm Powered device screens are often only 160x160 or 320x320 pixels, so the amount of information you can display at one time is limited.

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

Note that the screen size is not necessarily fixed: some Palm Powered devices allow the user to rotate the display or to collapse the input area. If the user collapses the input area, there is more space available to the application.

Quick Turnaround Expected

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

By contrast, the average device user uses a device application 15 to 20 times per day for much briefer periods of time, often just a few seconds. Speed is therefore a critical design objective for devices 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.

To maximize performance, the user interface should minimize navigation between windows, opening of dialogs, and so on. The layout of application screens needs to be simple so that the user can pick up the product and use it effectively after a short time. It's especially helpful if the user interface of your application is consistent with other applications on the 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 (Memo Pad, Address Book, and so on). These guidelines are summarized in the book *Exploring Palm OS: User Interface Guidelines*.

PC Connectivity

PC connectivity is an integral component of the Palm Powered device. The device comes with a cradle or cable 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 to 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 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 device. To write a conduit, you use the Conduit Developer's Kit (CDK). See the documentation provided with the CDK for more information on writing conduits.

Input Methods

Most users of Palm Powered devices don't have a full-sized keyboard or mouse. When entering data directly into the device, depending on the design of the device most users either use a pen—by writing characters in the input area or by tapping on an on-screen keyboard—or they type on a miniature "thumb board."

While pen strokes, the keyboard dialog, or a miniature keyboard 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.

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

Power

Palm Powered devices run on batteries and thus do not have the same processing power as a desktop PC. If your application needs to perform a computationally-intensive task, see if there isn't a way to perform that task in the desktop application instead of the device application.

Memory

Compared to a desktop computer, Palm Powered devices have limited heap and storage space. Different devices within the family of Palm Powered devices might have on the order of 16 Mb to 128 Mb of dynamic memory and storage available, in total. The device does not have a disk drive or PCMCIA support.

File System

Except when working with external storage media (SD cards, Memory Sticks, and the like), 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.

Applications written specifically for Palm OS Cobalt can take advantage of the advanced capabilities of the Schema database format. Schema databases have many of the features of relational databases and give applications greater power and flexibility in working with their data.

Backward Compatibility

Not all Palm Powered devices run the same version of Palm OS. Users are not expected to upgrade their versions of Palm OS as they would an operating system on a desktop computer. New versions of the operating system are designed in such a way that applications that run on a previous version will most likely run on the newer version as well. This allows you to write applications that can be deployed on a wider variety of Palm Powered devices, increasing the potential market for your applications. See “Making Your Application Run on Different Devices” on page 13 for details.

Palm OS Programming Concepts

Palm OS applications are event-driven programs that are most often written in C. Although Palm OS Cobalt supports multiple threads and processes, the user primarily interacts with only one program at a time. To successfully build a Palm OS application, you have to understand how the system itself is structured and how to structure your application.

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

  The sole purpose of the `PilotMain()` function is to receive launch codes and respond to them. (See Chapter 2, "Application Start and Stop," on page 21.)
- Palm OS applications are largely event-driven and so contain an **event loop**; however, this event loop is only started in response to the normal launch. Your application may perform work outside the event loop in response to other launch codes. Chapter 3, "Events and the Event Loop," on page 43, 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 *Exploring Palm OS: User Interface*)
- All applications should use the memory and data management facilities provided by the system. (See *Exploring Palm OS: Memory, Databases, and Files*)
- Applications employ operating system services 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 tables of contents of the various books that make up the *Exploring Palm OS* series.

API Naming Conventions

The following conventions are used throughout most of the Palm OS API:

- Functions start with a capital letter.
- All functions belonging to a particular manager start with a short 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.
- Typedefs start with a capital letter and end with “Type” (for example, `DateFormatType`, found in `DateTime.h`).
- Notifications start with a prefix (most often, “sys”) followed by the word “Notify.” For example, `sysNotifyAppLaunchingEvent`.
- Launch codes have a prefix followed by “LaunchCmd,” as in `sysAppLaunchCmdNormalLaunch`.
- Members of an enumerated type start with a lowercase prefix followed by a name starting with a capital letter, as follows:

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

Integrating Programs with the Palm OS Environment

When users work with a Palm OS application, they expect to be able to switch to other applications, have a way to enter data right on the device, 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 Chapter 2, "Application Start and Stop," on page 21.
- Be sure your application uses the system preferences for numeric formats, date, time, and start day of week. See Chapter 3, "Preferences," on page 37 of *Exploring Palm OS: System Management* for instructions on how to do so.
- Don't obscure shift indicators.

In addition, follow these rules:

- Store state information in the application preferences database, not in the application record database. See Chapter 3, "Preferences," on page 37 of *Exploring Palm OS: System Management* for more information on preferences.
- If your application uses the serial port, be sure to close the port when you no longer need it so that the HotSync application can use it.
- Ensure that your application properly handles the global find. Generally, searches and sorts aren't case sensitive.
- If your application supports private records, be sure they are unavailable to the global find when they should be hidden.
- Integrate with the Launcher application by providing an application name, two application icons, and a version string as described in Chapter 11, "Integrating with the Application Launcher," on page 167 of *Exploring Palm OS: User Interface*.
- Follow the guidelines detailed in *Exploring Palm OS: User Interface Guidelines*.

- Ensure that your application properly handles system messages during and after synchronization.
- Ensure that protected records and masked record contents are not displayed if the user has so indicated.
- Ensure that your application uses a consistent default state when the user enters it:
  - Some applications have a fixed default; for example, the Date Book always displays the current day when launched.
  - Other applications return to the place the user exited last. In that case, remember to provide a default if that place is no longer available. Because of HotSync operations and Preferences, don't assume the application data is the same as it was when the user looked at it last.
- If your application uses sounds, be sure it uses the Warning and Confirmation sounds properly.

Writing Robust Code

To make your programs more robust and to increase their compatibility with the widest variety 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 making frequent use of the `DbgOnlyFatalErrorIf()` macro, 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 `ErrFatalErrorIf()` function.

- Avoid continual polling.

  To conserve the battery, avoid continual polling. If appropriate, take advantage of the `keyUpEvent` or the facilities for performing event-based pen tracking to avoid polling altogether.

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

  In Palm OS Cobalt reading and writing to NULL will cause your application to crash. When calling functions that allocate memory, at least 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 the memory management functions with a version that validates the arguments passed to it.

- Check result codes when allocating memory.

  Because various Palm Powered devices have larger or smaller amounts of available memory, it is always a good idea to check result codes carefully when allocating memory.

- Avoid making assumptions about the screen.

  The size and shape of the screen, the screen buffer, and the number of pixels per bit aren't set in stone—they vary from one Palm Powered device to another. Don't hack around the windowing and drawing functions; the functions provided are optimized to make best use of the underlying hardware and to allow multiple applications and system services to share it.

- 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.

Uniquely Identifying Your Palm OS Application

Each Palm OS application—in fact, each Palm OS database—is uniquely identified by a combination of its name and a four-byte creator ID. By assigning the application's creator ID to all of the databases related to an application, you associate those databases with the application. The OS takes advantage of this; for instance, the Launcher's Info panel uses the creator ID to calculate the total memory used by each application.

Each database on the Palm Powered device has a type as well as a creator ID. The database type allows applications and the OS to distinguish among multiple databases with the same creator ID. For applications, set the database type to `sysFileTApplication` (`'appl'`). For each database associated with an application, set the database type to any other value (as long as it isn't composed entirely of lowercase letters, since those are reserved by PalmSource). Certain predefined types—such as `'appl'` (application) or `'libr'` (library)—have special meaning to Palm OS. For instance, the Launcher looks at the database type to determine which databases are applications.

Types and creator IDs are case-sensitive and are composed of four ASCII characters in the range 32-126 (decimal). Types and creator IDs consisting of all lowercase letters are reserved for use by PalmSource, so any type or creator ID that you choose must contain at least one uppercase letter, digit, or symbol[1].

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

http://dev.palmos.com/creatorid/

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

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

[1] Palm has also reserved 'pqa '.

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

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 operating system installed on it. Users are not expected to upgrade the Palm OS as they would an operating system on a desktop computer. This fact makes backward compatibility more crucial for Palm OS applications.

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

- Processor Differences
- Running New Applications on an Older Device
- Compiling Older Applications with the Latest SDK

Processor Differences

The original Palm OS devices—and, as of this writing, the majority of the installed base—employ a Motorola Dragonball™ processor that is part of the 68000 family. Palm Powered devices running Palm OS Garnet and Palm OS Cobalt use an ARM™ processor (available from a variety of manufacturers). To ensure compatibility, Palm OS Garnet introduced the Palm OS Application Compatibility Environment (PACE), which emulates the instruction set from the earlier class of Palm Powered devices and allows most applications written for those devices to continue to run on an ARM-based device. PACE is present in Palm OS Cobalt as well. While Palm OS Garnet only allowed developers to create applications that worked through PACE, however, Palm OS Cobalt gives developers the choice of developing a traditional Palm OS application that runs through PACE, or developing an application that runs "natively"—one that is compiled for the ARM processor and that can directly call the operating system.

How you choose to develop your Palm OS applications depends on a number of factors. Perhaps most important is what devices you are targeting. If you want to target the largest possible set of customers you'll likely want to write your application to be compatible with a number of earlier versions of Palm OS. If you do this, however, be aware that:

- Applications written expressly for Palm OS Cobalt will run faster than those that must work through PACE.
- PACE does not provide full access to many of the new features introduced in Palm OS Cobalt. Schema databases, advanced graphics, and multithreading are just some of the things you give up by not targeting the Palm OS Cobalt native environment.

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

Running New Applications on an Older Device

PalmSource works hard to maintain binary compatibility between versions of Palm OS. Even with the switch from the 68K-based Dragonball processor to an ARM-based one compatibility is largely maintained through the use of the Palm OS Application Compatibility Environment (PACE). Because of this, applications can be written that will run on all versions of the operating system (provided the application doesn't use any features specific to one version of the operating system). In other words, if you wrote your application using only features available in Palm OS 1.0, then your application should run on all devices. If you use 2.0 features, your application won't run on the earliest Palm Powered devices, but it will run on all those running Palm OS 2.0 and later.

NOTE: As explained in the previous section, this discussion is aimed at applications written using the “68K” APIs—those APIs exposed in Palm OS Cobalt via PACE. Applications written to run natively on the ARM processor will not run on any Palm Powered device running a version of Palm OS prior to 6.0.

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

- The *Palm OS Programmer’s API Reference* has a “Compatibility Guide” appendix. This guide lists the features and functions introduced in each operating system version greater than 1.0.
- The header file `CoreTraps.h` (`SysTraps.h` on versions of Palm OS before 3.5) 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 introduced wireless support, but not all Palm Powered devices have that capability. Thus, checking that the system version is 3.2 (or greater) does not guarantee that wireless support exists. Consult the “Compatibility Guide” in the *Palm OS Programmer’s API Reference* to learn how to check for the existence of each specific feature.

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. This rule applies to Palm OS Cobalt as long as your application continues to use the “68K” APIs supported by PACE. Converting an existing application so that it runs natively on Palm OS Cobalt is a somewhat more involved task; see *Exploring Palm OS: Porting Applications to Palm OS Cobalt* for a complete discussion of this process.

If you want to compile your older application under the latest release, you need to watch out for functions with a changed API. For any of these functions, the old function still exists with a suffix noting the last release that fully supports it, such as "V40" for Palm OS 4.0.

When a given function has been so renamed, you have two options:

- Change the function name in your code to keep using the old API. Your application will then run error free on the newer and the older 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.

Programming Tools

Several tools are available that help you build, test, and debug Palm OS applications. The set of tools you can choose from is large and growing: see `http://www.palmos.com/dev/tools/` for information about your development language and tool options.

The book *Palm OS Programming Development Tools Guide* describes the PalmSource-provided debugging tools available on your development platform. The Palm OS Developer Suite has extensive online help. For information on using third-party tools, refer to the documentation (printed or electronic, depending upon the tool) supplied with the tool.

Where to Go from Here

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

- This book

  The rest of this book explores some of the most basic Palm OS programming concepts. Among other things, the next three chapters cover:

  – the way in which applications are started ("launched"),

  - how applications can be instructed to perform a service on behalf of another,
  - the means by which applications are informed of user actions, and the basic mechanism employed by Palm OS applications to process those actions,
  - how applications can register for and receive notification of important operating system events.

- *Exploring Palm OS: Memory, Databases, and Files*

  As the title implies, this book covers all aspects of the Palm OS memory system. This includes the way in which Palm Powered devices dedicate a portion of their memory to serve as what in a desktop computer would be secondary, or disk, storage. On the device, programs and data are stored in **databases**; these databases can either be structured, like a traditional computer database, or more free-form, like a traditional file. Palm OS doesn't provide a traditional file system except when working with "external" storage (which on a typical Palm OS device takes the form of an SD card or Memory Stick); this book shows you how to interact with these storage devices as well.

- *Exploring Palm OS: User Interface*

  Nearly all Palm OS programs have some sort of user interface, and this book goes into great detail about how you create such an interface. This book covers the nuts-and-bolts of constructing and manipulating your user interface: for more general advice on how best to interact with the user, see *Exploring Palm OS: User Interface Guidelines*.

- *Exploring Palm OS: System Management*

  This book discusses all of the "miscellaneous" system functions: threading, dates and times, floating point, alarms, features and preferences, and so on. It also covers hardware interactions: the real-time clock, expansion media, system boot and reset, battery power, and the like.

- Other books in the *Exploring Palm OS* series

  The remaining books in the series provide more specialized information that might not be of interest to all Palm OS application developers. These books include:

  - *Exploring Palm OS: Text and Localization*

- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP*
- *Exploring Palm OS: Porting Applications to Palm OS Cobalt*
- *Exploring Palm OS: Palm OS File Formats*
- *Introduction to Palm OS Tools*
- *Palm OS Compiler Tools Guide*
- *Palm OS Compiler Reference*
- *Palm OS Debugger Guide*
- *Palm OS Resource Tools Guide*
- *Palm OS Simulator Guide*
- *Virtual Phone Guide*

- Example applications

  The Palm OS Cobalt SDK contains a number of sample applications that can be a valuable aid when you develop your own programs. 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.

- Conduit Development Kit and documentation

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

- Training

  PalmSource offers training in Palm OS programming. See the Training portion of the PalmSource website at http://www.palmos.com/dev/training/.

- PalmSource website

  The PalmSource website is an invaluable source of information about PalmSource, the Palm OS, and Palm OS application development. From this website you can download SDKs, tools, and documentation. You can learn about the various device manufacturers and the devices they produce. And you can get help with your programming questions, either by consulting the PalmSource Knowledge Base or by posting a question to one of the many Internet forums dedicated to Palm OS programming. For all of this and more, go to http://www.palmos.com/dev/.

2

Application Start 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 a Palm Powered™ device. 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 by the global find facility. Palm OS applications don't have an Exit command; instead they exit when a user switches to another application.

This chapter describes how an application launches, how an application stops, and the code you must write to perform these tasks properly. This chapter covers:

This chapter does not cover the main application event loop. The event loop is covered in Chapter 3, "Events and the Event Loop."

Launch Codes and Launching an Application

An application launches when its `PilotMain()` function is called with a **launch code**. Launch codes are a means of communication between the Palm OS and the application or between two applications.

For example, an application typically launches when a user presses one of the buttons on the device or selects an application icon from the Application Launcher. 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 launching 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 launch-code-specific 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" on page 105 for a list of standard Palm OS launch flags.

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 fully described.

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, returning errNone. Otherwise, it performs the action immediately and returns.

Listing 2.1 shows selected parts of PilotMain() from the Datebook application as an example.

Listing 2.1 Parts of Datebook's PilotMain() function

```
uint32_t PilotMain (uint16_t cmd, MemPtr cmdPBP, uint16_t launchFlags) {
   ExgPassableSocketType* passableSocketP;
   ExgSocketType* socketP;
   DmOpenRef dbP;
   uint32_t cursorID = dbInvalidCursorID;
   uint32_t value;
   uint32_t defaultForm;
   uint16_t mode;
   Boolean launched;
   status_t error = errNone; // the error returned by PilotMain

   // Get application dbP
   if ((error = SysGetModuleDatabase(SysGetRefNum(), &gApplicationDbID,
      &gApplicationDbP)) < errNone)
      return error;

   // Assign the local device time zone
   gettimezone(gDeviceTimeZone, TZNAME_MAX);

   switch (cmd){
      // Launch code sent by the launcher or the datebook button.
      case sysAppLaunchCmdNormalLaunch:
         error = PrvStartApplication ();
         if (error < errNone)
            return (error);

         // If the user previously left the Datebook while viewing the agenda,
         // return there. Otherwise, go to the day view
         error = FtrGet (sysFileCDatebook, recentFormFeature, &value);
         if (error)
            defaultForm = defaultRecentForm;
         else
            defaultForm = value;

         FrmGotoForm(gApplicationDbP, (uint16_t) defaultForm);

         PrvEventLoop ();
         PrvStopApplication ();
         break;
```

```
case sysAppLaunchCmdExgGetData:
   // Handle a get request
   ...

case appLaunchCmdExgGetFullLaunch:
   ...

case  appLaunchCmdAlarmEventGoto:
   // This action code is a DateBook specific custom launch code.
   // It will always require that the app launches as it is a result
   // of a SysUIAppSwitch call.
   ...

case  sysAppLaunchCmdGoTo:
   // This action code might be sent to the app when it's already running
   // if the use hits the "Go To" button in the Find Results dialog box.
   launched = launchFlags & sysAppLaunchFlagNewGlobals;
   if (launched) {
      // New start
      error = PrvStartApplication ();
      if (error < errNone)
         break;

      PrvGoToItem ((GoToParamsPtr) cmdPBP, launched);

      PrvEventLoop ();
      PrvStopApplication ();
   } else
      // application was already started
      PrvGoToItem ((GoToParamsPtr) cmdPBP, launched);
   break;

case sysAppLaunchCmdFind:
   // Launch code sent when the user is looking for some text.
   ...

case sysAppLaunchCmdSyncNotify:
   // Launch code sent by sync application to notify the datebook
   // application that its database was been synced.
   ...

case sysAppLaunchCmdNotify:
   ...

case  sysAppLaunchCmdSystemReset:
   // This action code is sent after the system is reset.
   ...
```

```
      case sysAppLaunchCmdExgAskUser:
         ...

      case sysAppLaunchCmdExgReceiveData:
         // Receive the record.  The app will parse the data and add it
         // to the database.
         ...

      case sysAppLaunchCmdExgPreview:
         ...

      case sysAppLaunchCmdInitDatabase:
         // This action code is sent by the DesktopLink server when it creates
         // a new database.  We will initialize the new database.
         ...

      case sysAppLaunchCmdAlarmTriggered:
         // Launch code sent by Alarm Manager to notify the datebook
         // application that an alarm has triggered.
         ...

      case sysAppLaunchCmdAttention:
         // Launch Code sent by Attention Manager to let Datebook draw
         // alarmed events.
         ...

      case sysAppLaunchCmdExportRecordGetCount:
         ...

      case sysAppLaunchCmdExportRecord:
         ...

      case sysAppLaunchCmdImportRecord:
         ...

      case sysAppLaunchCmdDeleteRecord:
         ...
      default:
         break;
   }

   return error;
}
```

NOTE: The above code calls `SysGetModuleDatabase()` and `gettimezone()` for every launch code. Programs should only call functions like these for those launch codes in which the values are needed: both of these calls result in an IPC (interprocess communcation), which should be avoided whenever possible.

Responding to Normal Launch

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

During the start 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. If none exists, create it and initialize it.
3. Get application-specific preferences and initialize related global variables.
4. Initialize any other global variables.

As you saw in Listing 2.1, the Datebook application responds to `sysAppLaunchCmdNormalLaunch` by calling a function named `PrvStartApplication()`. Listing 2.2 shows this function.

Listing 2.2 PrvStartApplication() from Datebook

```
static status_t PrvStartApplication(void) {
   status_t err;
   uint16_t mode;
   DateTimeType dateTime;
   DatebookPreferenceType prefs;
   int16_t prefsVersion;
   time_t rangeStartTime;
   time_t rangeEndTime;


   // Load the ToDo application as a shared lib. Doing this, the ToDo
   // globals will be kept all along the Datebook execution
   err = SysLoadModule(sysFileTApplication, sysFileCToDo, 0, 0, &ToDoRefNum);
   ErrNonFatalDisplayIf(err < errNone,
      "Unable to load the ToDo application as a shared library");

   // Determime if secret record should be shown.
   PrivateRecordVisualStatus = CurrentRecordVisualStatus =
      (privateRecordViewEnum)PrefGetPreference(prefShowPrivateRecords);

   mode = (PrivateRecordVisualStatus == hidePrivateRecords) ?
         dmModeReadWrite : (dmModeReadWrite | dmModeShowSecret);

   // Get the time formats from the system preferences.
   TimeFormat = (TimeFormatType)PrefGetPreference(prefTimeFormat);

   // Get the date formats from the system preferences.
   LongDateFormat = (DateFormatType)PrefGetPreference(prefLongDateFormat);
   ShortDateFormat = (DateFormatType)PrefGetPreference(prefDateFormat);

   // Get the starting day of the week from the system preferences.
   StartDayOfWeek = (uint16_t) PrefGetPreference(prefWeekStartDay);

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

   // Find the application's data file.  If it don't exist create it.
   err = DateDBOpenDatabase (&ApptDB, mode);
   if (err < errNone)
      return err;

   // Create initial cursor based on current date and a 1-day range (day /
agenda view)
   CalculateStartEndRangeTimes(&Date, 1, &rangeStartTime, &rangeEndTime, NULL);
   err =  ApptDBOpenOrRequeryWithNewRange(ApptDB, &gApptCursorID,
      rangeStartTime, rangeEndTime, true);
   if (err < errNone)
      return err;

   PIMAppProfilingBegin("PrvStartApplication, TimeZoneToAscii")

   // Get the devivce localized time zone name
   TimeZoneToAscii(gDeviceTimeZone, gLocalizedTimeZomeName);

   PIMAppProfilingEnd();

   // Read the preferences / saved-state information
   prefsVersion = DatebookLoadPrefs (&prefs);
   DayStartHour = prefs.dayStartHour;
   DayEndHour = prefs.dayEndHour;
   AlarmPreset = prefs.alarmPreset;
   SaveBackup = prefs.saveBackup;
   ShowTimeBars = prefs.showTimeBars;
   CompressDayView = prefs.compressDayView;
   ShowTimedAppts = prefs.showTimedAppts;
   ShowUntimedAppts = prefs.showUntimedAppts;
   ShowDailyRepeatingAppts = prefs.showDailyRepeatingAppts;
   AlarmSoundRepeatCount = prefs.alarmSoundRepeatCount;
   AlarmSoundRepeatInterval = prefs.alarmSoundRepeatInterval;
   AlarmSoundUniqueRecID = prefs.alarmSoundUniqueRecID;
   ApptDescFont = prefs.apptDescFont;
   AlarmSnooze = prefs.alarmSnooze;

   // Get the previous current category
   PrvLoadCurrentCategories(&DateBkCurrentCategoriesCount,
      &DateBkCurrentCategoriesP);

   // Reset selection
   TopVisibleAppt = 0;

   // Set initial active tab for the details dialog in day view
   DetailsSetDefaultEventDetailsTab(DetailsBookOptionsTabId);

   return errNone;
}
```

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.

If your application receives a launch code other than `sysAppLaunchCmdNormalLaunch` or `sysAppLaunchCmdGoTo`, you can find out if it is the current application by checking the launch flags that are sent with the launch code. If the application is the currently running application, the `sysAppLaunchFlagSubCall` flag is set. This flag is set by the system and isn't (and shouldn't be) set by the sender of a launch code.

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

Launching Applications Programmatically

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

TIP: There are other ways for applications to communicate. See "When to Use the Helper API" on page 67 to help you decide which method to use.

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

To send a launch code to another application, use one of the `SysAppLaunch...()` functions from the Application Manager. You use these functions when you want to make use of another application's functionality and eventually return control to your application. The process of calling another application as a subroutine is sometimes referred to as a **sublaunch**.

The Application Manager defines the following `SysAppLaunch...()` functions:

`SysAppLaunch()`: Launches an application as a subroutine of the caller in the caller's process. This function can only be called from the main UI thread. Use with care: most applications will want to use one of the other `SysAppLaunch...()` functions instead.

SysAppLaunchLocal(): Launch an application as a subroutine of the caller in the caller's process, unless the application being launched is already running in another process. In this case, the launch code and parameters are sent to the running application. This function can only be called from the main UI thread.

SysAppLaunchRemote(): Launch an application as a subroutine of the caller in a separate, newly-created process, unless the application being launched is already running in another process in which case the launch code and parameters are sent to the running application. Remote launching allows applications to execute untrusted code without compromising their own security. This function can only be called from the main UI thread.

NOTE: The parameter block you pass in to any of the above cannot contain pointers to other data or objects.

For example, you could use SysAppLaunchLocal() 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.

An alternative, simpler method of sending launch codes is the SysBroadcastActionCode() call. This function automatically finds all other user-interface applications and calls the appropriate SysAppLaunch...() function to send the launch code to each of them.

When an application is launched using one of the SysAppLaunch...() functions, 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 SysGetModuleDatabase() to get the database ID of its own database.

If you want to actually close your application and open another application, use SysUIAppSwitch() instead. This function notifies the system which application to launch next and feeds an appStopEvent event into the event queue. If and when the current application responds to the quit event and returns, the system launches the new application.

WARNING! Do not use the `SysUIAppSwitch()` or `SysAppLaunch...()` functions to open the Application Launcher application. If another application has replaced the default launcher with one of its own, this function will open the system-supplied launcher instead of the custom one. To open the correct Launcher reliably, enqueue a `keyDownEvent` that contains a `launchChr`.

When you launch an application using `SysUIAppSwitch()` you have the option to pass a parameter block (using the *`cmdPBP`* parameter) containing application-specific information to the application being launched. To create this parameter block, allocate a block of memory using `MemPtrNew()` and then call `MemPtrSetOwner()` to set the block's owner ID to 0. This assigns ownership of the block to the system; memory blocks owned by the system aren't automatically freed when the calling application exits. Once ownership of the block has been assigned to the system, neither the launching nor the launched application need worry about freeing the block since the operating system will do this itself after the launched application exits.

Note that your parameter block must be self contained. That is, it must not have pointers to anything on the stack or to memory blocks that are owned by an application. If you don't need to pass a parameter block to the application being launched, pass `NULL` for the *`cmdPBP`* parameter.

Sublaunching in Another Process

Each sublaunch takes place in its own transient process, except when the currently running application receives a request to sublaunch itself, in which case the sublaunch takes place in the Application Process. The sublaunched thread, whether in the main Application process or a sublaunched process, effectively becomes the Application process and thread for the duration of the sublaunch. In other words, the thread requesting the sublaunch is effectively suspended while the sublaunched thread executes. Once the sublaunched application exits, the sublaunched process that was created to accommodate the sublaunch is then completely torn down.

See "Processes and Applications" on page 87 of *Exploring Palm OS: System Management* for a diagram showing all of the Palm OS Cobalt processes.

Creating Your Own Launch Codes

Palm OS contains a large number of predefined launch codes, which are listed in "Launch Code Summary" on page 33. 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 an unsigned 16-bit value. All launch codes with values 0–32767 are reserved for use by the system and for future enhancements. Launch codes beginning at `sysAppLaunchCmdCustomBase` (that is, those from 32768 to 65535) are available for private use by applications.

Stopping an Application

An application shuts itself down when it receives the event `appStopEvent`. Note that this is an event, not a launch code. The application must detect this event and terminate. (Events are covered in detail in Chapter 3, "Palm OS Events.") Applications typically call a "StopApplication" function in response to the `appStopEvent`, before returning from `PilotMain()`.

The `appStopEvent` gives the a application an opportunity to perform cleanup activities including closing databases and saving state information. In the stop function, an application should first flush all active records, close the application's database, and save those aspects of the current state needed for the next time the application is started. Listing 2.3 is an example of a stop function—this is from the Datebook application.

Listing 2.3 PrvStopApplication() from Datebook

```
static void PrvStopApplication (void) {
   // Save the preferences
   DatebookSavePrefs();

   // Save current categories
   PrvSaveCurrentCategories(DateBkCurrentCategoriesCount,
      DateBkCurrentCategoriesP);

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

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

   // Close the application's cursor
   ApptCloseCursor(&gApptCursorID);

   // Close the application's data file.
   DbCloseDatabase (ApptDB);
   ApptDB = NULL;

   // Unload the ToDo application loaded as a shared library
   if (ToDoRefNum != kRALInvalidRefNum)
      SysUnloadModule(ToDoRefNum);
}
```

Launch Code Summary

The following tables list all Palm OS standard launch codes. These launch codes are declared in `CmnLaunchCodes.h`, `TelephonyLib.h`, and `Preferences.h`. All the parameters for a launch code are passed in a single parameter block, and the results are returned in the same parameter block.

Table 2.1 Palm OS Launch Codes

Code	Request
`prefAppLaunchCmdSetActivePanel`	
`sysAppLaunchCmdAddRecord`	Add a record to a database.
`sysAppLaunchCmdAlarmTriggered`	Schedule next alarm or perform quick actions such as sounding alarm tones.
`sysAppLaunchCmdAttention`	Perform the action requested by the Attention Manager.

Table 2.1 Palm OS Launch Codes *(continued)*

Code	Request
sysAppLaunchCmdBackground	Sent to the executable module that is launched in a background thread.
sysAppLaunchCmdCardLaunch	Launch the application. This launch code signifies that the application is being launched from an expansion card.
sysAppLaunchCmdCountryChange	Respond to country change.
sysAppLaunchCmdDeleteRecord	Instructs the application to delete a specified database record.
sysAppLaunchCmdDisplayAlarm	Display specified alarm dialog or perform time-consuming alarm-related actions.
sysAppLaunchCmdEventHook	Allow the application to process an event.
sysAppLaunchCmdExportRecord	Instructs the application to export a specified database record.
sysAppLaunchCmdExportRecordGetCount	Instructs the application to return the number of records in the application's database.
sysAppLaunchCmdFailedAppNotify	Indicates a failure in an application that was just switched to.
sysAppLaunchCmdFepPanelAddWord	Add a word to the FEP user dictionary.
sysAppLaunchCmdFinalizeUI	Instructs the application's start-up code to de-initialize the process's UI.
sysAppLaunchCmdFind	Find a text string.
sysAppLaunchCmdGoTo	Go to a particular record, display it, and optionally select the specified text.

Table 2.1 Palm OS Launch Codes *(continued)*

Code	Request
`sysAppLaunchCmdHandleSyncCallApp`	Perform some application-specific operation at the behest of the application's conduit.
`sysAppLaunchCmdImportRecord`	Presents the application with a record to be added to or updated in the application's database.
`sysAppLaunchCmdInitDatabase`	Initialize database.
`sysAppLaunchCmdInitializeUI`	Instructs the application's start-up code to initialize the process's UI.
`sysAppLaunchCmdLookup`	Look up data. In contrast to `sysAppLaunchCmdFind`, a level of indirection is implied. For example, look up a phone number associated with a name.
`sysAppLaunchCmdLookupWord`	Look a word up in the FEP dictionaries.
`sysAppLaunchCmdMultimediaEvent`	
`sysAppLaunchCmdNormalLaunch`	Launch normally.
`sysAppLaunchCmdNotify`	Receive a notification.
`sysAppLaunchCmdOpenDB`	Launch application and open a database.
`sysAppLaunchCmdPanelCalledFromApp`	Tell preferences panel that it was invoked from an application, not the Preferences application.
`sysAppLaunchCmdPinletLaunch`	Sent to an application that is launched as a pinlet instead of `sysAppLaunchCmdNormalLaunch` in order to launch the application.

Table 2.1 Palm OS Launch Codes *(continued)*

Code	Request
`sysAppLaunchCmdReturnFromPanel`	Tell an application that it's restarting after preferences panel had been called.
`sysAppLaunchCmdRun68KApp`	Launch a 68K-based application.
`sysAppLaunchCmdSaveData`	Save data. Often sent before find operations.
`sysAppLaunchCmdSlipLaunch`	System use only.
`sysAppLaunchCmdSyncCallApplicationV10`	Obsolete launch code.
`sysAppLaunchCmdSyncNotify`	Notify applications that a HotSync has been completed.
`sysAppLaunchCmdSyncRequest`	Request a HotSync.
`sysAppLaunchCmdSyncRequestLocal`	Request a "local" HotSync.
`sysAppLaunchCmdSyncRequestRemote`	Request a "remote" HotSync.
`sysAppLaunchCmdSystemLock`	Sent to the Security application to request that the system be locked down.
`sysAppLaunchCmdSystemReset`	Respond to system reset. No UI is allowed during this launch code.
`sysAppLaunchCmdTimeChange`	Respond to system time change.
`sysAppLaunchPnpsPreLaunch`	Pre-launch code for "plug-and-play" devices.
`sysBootAppLaunchCmdNoSublaunch`	Informs the boot application that a no-notify reset has occurred. *This launch code is for system use only.*
`sysLaunchCmdAppExited`	An application has exited from its `PilotMain()` function.

Table 2.1 Palm OS Launch Codes *(continued)*

Code	Request
`sysLaunchCmdBoot`	Informs operating system initialization procedures that the system is booting. *This launch code is for system use only.*
`sysLaunchCmdFinalize`	An executable module is being unloaded; gives the module a last chance to do any needed "de-initialization."
`sysLaunchCmdGetGlobals`	Retrieve a pointer to an executable module's globals structure.
`sysLaunchCmdGetModuleID`	Retrieve an executable module's module ID. *This launch code is for system use only.*
`sysLaunchCmdGraphicsAccelInit`	System use only.
`sysLaunchCmdInitialize`	An executable module has been loaded; gives the executable a chance to do any needed initialization.
`sysLaunchCmdInitRuntime`	A newly-loaded executable module should initialize its module ID and linker stub. *This launch code is for system use only.*
`sysLibLaunchCmdGet68KSupportEntry`	Determine if a shared library can be called from a 68K application.
`sysPackageLaunchAttachImage`	A package has been loaded and should supply an image context used by the package to determine when the package should be unloaded.
`sysPackageLaunchGetInstantiate`	Asks a package for the function used to instantiate the package's components.

Table 2.1 Palm OS Launch Codes *(continued)*

Code	Request
`sysPatchLaunchCmdClearInfo`	Informs a patch that a target shared library has been unloaded.
`sysPatchLaunchCmdSetInfo`	Informs a patch that one of the shared libraries it wants to patch is being loaded.
`sysPinletLaunchCmdLoadProcPtrs`	Requests pointers to the functions used by the Pen Input Manager when interacting with a pinlet.
`sysSvcLaunchCmdGetQuickEditLabel`	Get a “quick edit” label for one of the standard service panels.
`sysSvcLaunchCmdGetServiceID`	Get a standard service panel’s service ID.
`sysSvcLaunchCmdGetServiceInfo`	Obtain the name and service ID for a given system service.
`sysSvcLaunchCmdGetServiceList`	Obtain a list of system services.
`sysSvcLaunchCmdSetServiceID`	Set a standard service panel’s service ID.

Table 2.2 Communications-related Launch Codes

Code	Request
`kTelNwkLaunchCmdNetworkStatusChange`	
`kTelNwkLaunchCmdSignalLevelChange`	
`kTelNwkLaunchCmdUssdAnswer`	
`kTelPowLaunchCmdBatteryChargeLevelChange`	
`kTelPowLaunchCmdBatteryConnectionStatusChange`	

Table 2.2 Communications-related Launch Codes *(continued)*

Code	Request
`kTelPowLaunchCmdConnectionOff`	
`kTelPowLaunchCmdConnectionOn`	
`kTelPowLaunchCmdPhonebookNotReady`	
`kTelPowLaunchCmdPhonebookReady`	
`kTelPowLaunchCmdSmsNotReady`	
`kTelPowLaunchCmdSmsReady`	
`kTelSmsLaunchCmdIncomingMessage`	
`kTelSpcLaunchCmdCallAlerting`	
`kTelSpcLaunchCmdCallConnect`	
`kTelSpcLaunchCmdCallDialing`	
`kTelSpcLaunchCmdCallerIdAvailable`	
`kTelSpcLaunchCmdCallHeld`	
`kTelSpcLaunchCmdCallIncoming`	
`kTelSpcLaunchCmdCallReleased`	
`kTelSpcLaunchCmdCallWaiting`	
`kTelStyLaunchCmdAuthenticated`	
`kTelStyLaunchCmdAuthenticationCanceled`	
`sysAppLaunchCmdAntennaUp`	The antenna has been raised on a device that is appropriately equipped.
`sysAppLaunchCmdExgAskUser`	Let the application override display of the dialog asking user if they want to receive incoming data via the Exchange Manager.

Table 2.2 Communications-related Launch Codes *(continued)*

Code	Request
sysAppLaunchCmdExgGetData	Notify the application that it should send data using the Exchange Manager.
sysAppLaunchCmdExgPreview	Notify the application that it should display a preview using the Exchange Manager.
sysAppLaunchCmdExgReceiveData	Notify the application that it should receive incoming data using the Exchange Manager.
sysAppLaunchCmdGoToURL	Launch an application and open a URL.
sysAppLaunchCmdURLParams	Obsolete launch code.
sysAppLaunchNppiNoUI	Launch a network panel plug-in without UI, and load NetLib.
sysAppLaunchNppiUI	Launch a network panel plug-in with UI.
sysBtLaunchCmdExecuteService	Let Bluetooth service applications know that there is an inbound-connected data socket.
sysBtLaunchCmdPrepareService	Let Bluetooth service applications know that a listener socket has been created and to request an SDP service record.
sysCncPluginLaunchCmdGetPlugins	Request for plug-in descriptions from Connection Manager plug-in modules
sysCncPluginLaunchCmdRegister	Instructs Connection Manager plug-ins to initialize themselves.
sysCncPluginLaunchCmdUnregister	The Connection Manager plug-in is being removed.
sysCncWizardLaunchCmdEdit	Edit a Connection Manager profile.

Table 2.2 Communications-related Launch Codes *(continued)*

Code	Request
`sysDialLaunchCmdDial`	Dial the modem.
`sysDialLaunchCmdHangUp`	Hang the modem up.
`sysIOSDriverInstall`	System use only.
`sysIOSDriverRemove`	System use only.

Application Manager Function Summary

Launching Applications

`SysAppLaunch()`
`SysAppLaunchLocal()`
`SysAppLaunchRemote()`
`SysAppLaunchV40()`
`SysUIAppSwitch()`
`SysUIAppSwitchV40()`

Other Application Manager Functions

`SysBroadcastActionCode()`
`SysCurAppDatabase()`
`SysCurAppDatabaseV40()`
`SysGetStackInfo()`
`SysReset()`

3

Events and the Event Loop

This chapter discusses **events**—the primary mechanism by which the operating system communicates with an application—and the event loop that forms the heart of all Palm OS® applications. The topics covered are:

When working with events, you use APIs declared in `Event.h`. These APIs are documented in Chapter 7, "Event," on page 139; additional event-related APIs are documented in Chapter 13, "System Event Manager," on page 247. The event codes representing the various events are primarily declared in `EventCodes.h`, documented in Chapter 8, "Event Codes," on page 169. Many events are only of interest to developers working with a particular technology; accordingly, those events have been documented in the corresponding *Exploring Palm OS* volumes. See "Palm OS-Generated Events" on page 54 for a complete list of all events organized according to the book, and thus the technology, with which it is most commonly associated.

Palm OS Events

Palm OS applications are event-driven: user actions and some system requests are placed in an **event queue** from which the events can be retrieved and acted upon. Each thread running in either the

Application or Background process can have its own event queue into which events destined for that thread are placed (but note that the event queue is part of the UI context of the thread, and a UI context is relatively heavyweight, so unless you need it you shouldn't create the UI context). The operating system places into each queue only those events that are relevant for that queue.

Certain events—such as pen and key events—are classified as **low-level events**. Applications rarely work with low-level events directly; the operating system translates low-level events into higher-level events that are then posted to the appropriate event queue. Applications that do work with low-level events directly might do so to enqueue key events or to retrieve each of the pen points that comprise a pen stroke.

Because Palm OS translates low-level pen and key events into higher-level events, most Palm OS events can be ignored by the typical application. For instance, if the user taps an on-screen button a low-level `penDownEvent` is generated. This event is passed on to the control object, which then posts a `ctlEnterEvent`. Then as the user moves the pen, a series of `penMoveEvent`s are posted. Finally,when the user lifts the pen a `penUpEvent` is posted. If the pen was lifted within the control, the operating system realizes that the user just tapped an on-screen button and posts a `ctlSelectEvent` to the event queue. Of these events, an application that is only interested in knowing when the user taps an on-screen button need only watch for the `ctlSelectEvent`; it can ignore the other events entirely.

The Structure of an Event

The `EventType` structure describes an event. It consists of three main parts:

- A 32-bit value that identifies the event that has taken place.
- A standard data block that, for many events, contains the state of the pen at the time the event occurred.
- An optional event-specific data block.

The `EventType` structure looks like this (with the event-specific data structures that comprise the `data` union omitted for clarity):

```
typedef struct EventType {
   eventsEnum eType;
   Boolean penDown;
   uint8_t padding_1;
   uint16_t padding_2;
   uint32_t tapCount;
   Coord screenX;
   Coord screenY;
   union {
      ...
   } data;
} EventType;
```

In this structure the `data` union is used only by those events that have additional data associated with them. For instance, a `keyDownEvent`'s `data` field contains the following structure:

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

On the other hand, an `appStopEvent`—an indication to the application that it should stop—needs no additional data.

The Application Event Loop

Upon receiving a `sysAppLaunchCmdNormalLaunch` launch code, a typical Palm OS application does the following:

1. Perform any needed application-specific initialization.
2. Display the application's main form.
3. Enter a loop, retrieving and handling events until an `appStopEvent` is retrieved. This part of the program is known as the **application event loop**.
4. Perform any necessary cleanup, and exit.

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

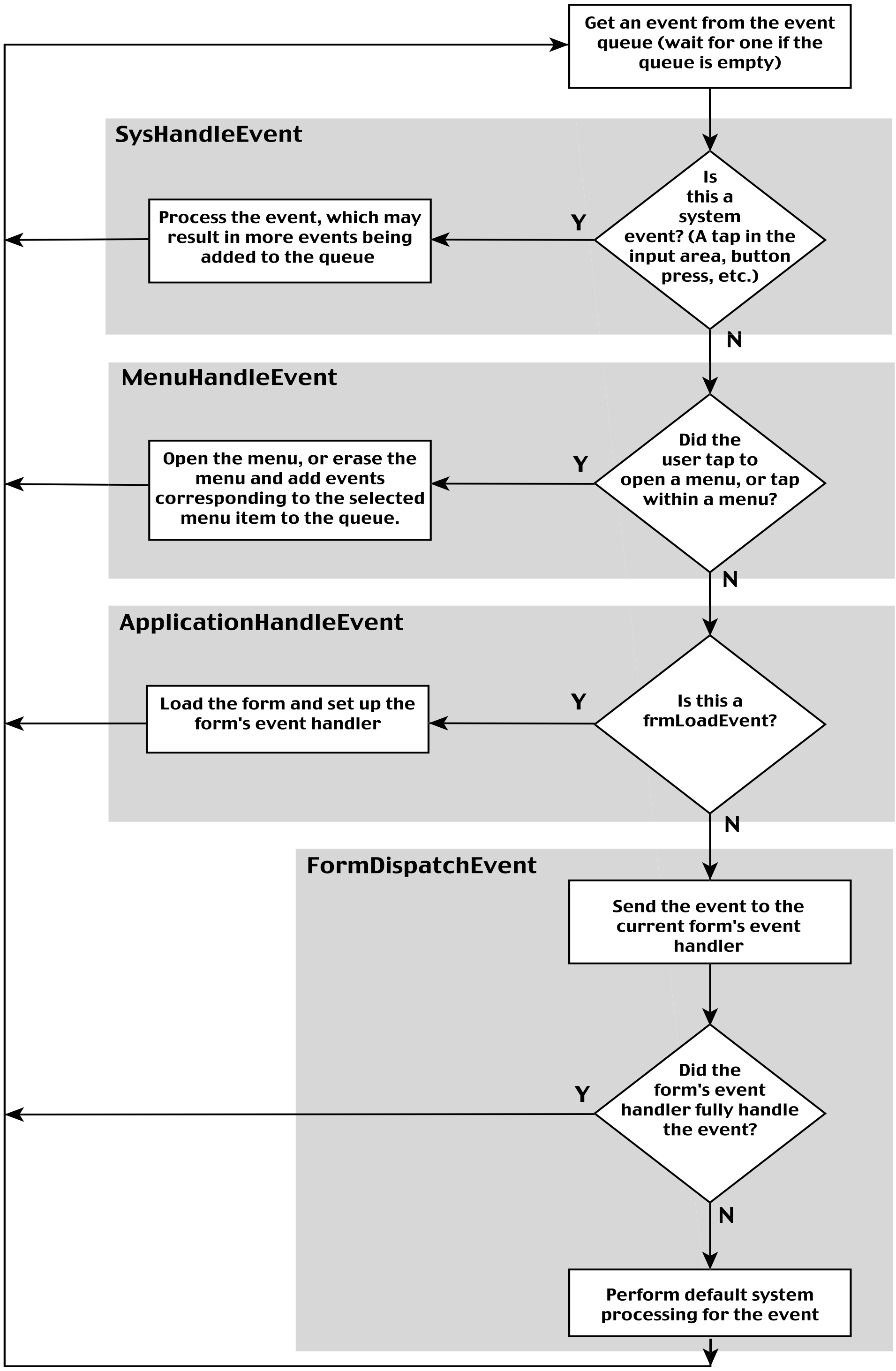
The application typically remains in the event loop until the system tells it to shut itself down by sending an `appStopEvent` through the event queue. The application must detect this event and terminate.

Listing 3.1 shows a typical application event loop. Figure 3.1 graphically illustrates this same event loop, with additional explanation for each of the steps.

Listing 3.1 Sample application event loop

```
static void AppEventLoop(void){
   status_t error;
   EventType event;

   do {
      EvtGetEvent(&event, evtWaitForever);

      if (SysHandleEvent(&event))
         continue;

      if (MenuHandleEvent(0, &event, &error))
         continue;

      if (AppHandleEvent(&event))
         continue;

      FrmDispatchEvent(&event);

   } while (event.eType != appStopEvent);
}
```

Figure 3.1 Control flow in a typical application


Get an event from the event queue (wait for one if the queue is empty)
SysHandleEvent
Is this a system event? (A tap in the input area, button press, etc.)
Y
Process the event, which may result in more events being added to the queue
N
MenuHandleEvent
Did the user tap to open a menu, or tap within a menu?
Y
Open the menu, or erase the menu and add events corresponding to the selected menu item to the queue.
N
ApplicationHandleEvent
Is this a frmLoadEvent?
Y
Load the form and set up the form's event handler
N
FormDispatchEvent
Send the event to the current form's event handler
Did the form's event handler fully handle the event?
Y
N
Perform default system processing for the event

As illustrated both in the code and the flowchart, within the basic application event loop the application performs the following steps (Each of which is discussed in greater detail in the following sections):

1. Fetch an event from the event queue.
2. Call `SysHandleEvent()` to give the system an opportunity to handle the event.
3. If `SysHandleEvent()` did not completely handle the event, the application gives the menu system a chance to handle it by calling `MenuHandleEvent()`.
4. If `MenuHandleEvent()` did not completely handle the event, your application now gets a chance to deal with it. `ApplicationHandleEvent()` is a function your application has to provide. Your `ApplicationHandleEvent()` function typically only handles the `frmLoadEvent`s, since each form typically handles its own events.
5. If `ApplicationHandleEvent` did not completely handle the event, the application calls `FrmDispatchEvent()`. This function sends the event to the active form's event handler and, if the form's event handler didn't deal with it, `FrmDispatchEvent()` passes it off to the operating system to do any default processing of the event.

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 a `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 initialize the elements on the form and then wait for events it can handle to arrive from the queue.

Retrieving Events

Applications call `EvtGetEvent()` to obtain the next available event from the current thread's event queue. Pass a pointer to an `EventType` structure into which the event will be copied, and a timeout value indicating the length of time the function should wait for an event if there are no events currently on the queue. In most instances you simply pass `evtWaitForever`, indicating that `EvtGetEvent()` shouldn't return until there is at least one event in the queue.

`EvtGetEvent()` has no return value: it always returns a valid event. If there are no events in the queue and the specified timeout period elapses, `EvtGetEvent()` generates and returns a `nilEvent`.

If your application needs to perform a lengthy process, such as a data transfer during a communications session, it should periodically call `EvtGetEvent()`. 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. Note that in situations like these you'll probably also want to display a progress dialog. For more information on progress dialogs, see *Exploring Palm OS: User Interface*.

Handling System Events

The system handles events like power on/power off, Graffiti® 2 input, tapping inputarea icons; these events are not posted to your thread's event queue. Other events, like the pressing of the "hard" buttons on the device, are posted to your thread's event queue to give your application the opportunity to handle them. Those events that your application does not handle entirely by itself should be passed on to the operating system; you do this by calling `SysHandleEvent()`.

`SysHandleEvent()` returns `true` if the event was completely handled and no further processing of the event is required. The application is then free to pick up the next event from the queue.

Handling Menu Events

`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.

Handling Form Load Events: the AppHandleEvent() Function

Your `ApplicationHandleEvent()` function typically only handles the `frmLoadEvent`s, since each form typically handles its own events. The `EventType` structure that comprises a `frmLoadEvent` contains both the `formID` of the form to be loaded and a `DmOpenRef` to the open resource database that contains the form. Using this information, your `AppHandleEvent()` function should do the following:

1. Load and initialize the form. This is most commonly done by calling `FrmInitForm()`. At this point the form has not yet been drawn, nor is it active.
2. Make the form the active form. Call `FrmSetActiveForm()` to do this. The active form receives all key and pen input, and all drawing is performed on the active form (until otherwise specified).
3. Set the form's event handler using `FrmSetEventHandler()`. You supply the address of a callback function that will receive all events intended for the form. Generally, you create a separate callback function for each form in your application.
4. Call `FrmInitLayout()` to prepare the form for automatic resizing. This last step is optional, but recommended: you need only call this function if your form should be automatically resized in response to a `winResizedEvent`.

Your `AppHandleEvent()` function should return `true` if it handled the event, or `false` if the event should be passed on to be handled elsewhere.

Listing 3.2 illustrates a typical `AppHandleEvent()` function.

Listing 3.2 A sample AppHandleEvent() function

```
static Boolean AppHandleEvent(EventType* pEvent) {
   uint16_t formId;
   FormType *pForm;

   if (pEvent->eType == frmLoadEvent) {
      // Load the form resource.
      formId = pEvent->data.frmLoad.formID;

      pForm = FrmInitForm(gAppDB, formId);
      FrmSetActiveForm(pForm);

      // Set the event handler for the form.  The handler of
      // the currently active form is called by
      // FrmHandleEvent each time is receives an event.
      switch (formId) {
         case MainForm:
            FrmSetEventHandler(pForm, MainFormHandleEvent);
            FrmInitLayout(pForm, gMainFormLayout);
            break;

         case Form2Form:
            FrmSetEventHandler(pForm, Form2FormHandleEvent);
            FrmInitLayout(pForm, gForm2FormLayout);
            break;

         default:
            ErrFatalDisplay("Invalid Form Load Event");
            break;
      }

      return true;
   }

   return false;
}
```

Handling Form-Specific Events

`FrmDispatchEvent()` begins by sending the event to the application's event handler for the active form. This is the event handler routine that was established in `ApplicationHandleEvent()`. This gives the application's code 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`, in which case `FrmDispatchEvent()` returns to the application's event loop. Otherwise, `FrmDispatchEvent()` calls `FrmHandleEvent()` to provide the system's default processing for the event. In many cases this default handling is sufficient; see the documentation for `FrmHandleEvent()` for an explanation of how that function deals with various events.

In the process of handling an event, an application may have to first close the current form and then open another one. This happens as follows:

1. The application calls `FrmGotoForm()` to bring up another form. `FrmGotoForm()` enqueues a `frmCloseEvent` for the currently active form, and then enqueues a `frmLoadEvent` and a `frmOpenEvent` for the new form.
2. When the application gets the `frmCloseEvent,` it closes and erases the currently active form.
3. 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 pre-loading 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.
4. Upon receipt of the `frmOpenEvent` the application performs any required initialization of the form.
5. Upon receipt of a `frmUpdateEvent` the application draws the form on the display.

After `FrmGotoForm()` has been called, any further events that make their way to the application event loop's `FrmDispatchEvent()` call are dispatched to the event handler for the form that's currently active—the form specified in the `FrmGotoForm()` call.

Using Events to Communicate Between Threads

NOTE: This section describes how the Palm OS event mechanism can be used to facilitate communications between separate threads of execution. For more complete information on writing multi-tasking Palm OS applications, see *Exploring Palm OS: System Management*.

Each thread can have its own event queue. While the Palm OS event mechanism is most commonly used to keep applications apprised of user actions, you can also employ it to communicate with other threads, either in the same or a different process.

Communicating Between Threads in a Single Process

Given a handle to an event queue, you can post an arbitrary event to that queue by calling `EvtAddEventToEventQueue()` or `EvtAddUniqueEventToEventQueue()`. Note that the former is different from `EvtAddEventToQueue()`, which always posts events to the default queue for the current thread.

To obtain a handle to the event queue of another thread in the same process you simply call `EvtGetThreadEventQueue()` from within that thread. When you are done with the queue be sure to call `EvtReleaseEventQueue()`.

Communicating Between Threads in Different Processes

As when posting events to the queue of another thread in the same process, you use `EvtAddEventToEventQueue()` or `EvtAddUniqueEventToEventQueue()` to post events to a queue in a separate process. How you obtain the handle to the other thread's queue, however, differs. The mechanism you use depends on whether you created the other thread yourself or whether you need to attach to an already-running thread in another process:

- If you are creating the thread yourself, you'll likely use `EvtCreateBackgroundThread()`. This function returns a handle to the newly-created background thread's event queue. Note that although threads don't necessarily have an event queue—you need to call WinStartThreadUI() to create a UI context (and thus an event queue) for a thread created with SysThreadCreate()—background threads created with EvtCreateBackgroundThread() do always have an event queue.

- To obtain a handle to the event queue of an already-running thread in another process, that thread must have published its queue by name using EvtPublishEventQueue(). Then, as long as the posting thread knows the name by which the queue was published, it need only call EvtLookupEventQueue() to obtain the queue handle.

Two-way communication is enabled by the *replyQueue* parameter to the EvtAddEventToEventQueue() call: if task A supplies a handle to its own queue when posting an event to task B's event queue, task B can send a reply by posting an event back to task A's queue. In order to obtain the handle to task A's event queue, task B must call EvtGetReplyEventQueue() while processing the original event posted by task A. Note that the reply queue is associated with a single event posting; this allows a background server task to service multiple clients at the same time.

Palm OS-Generated Events

The following is complete list of all events generated by Palm OS that are of interest to developers. Because most events are generated by or handled by specific areas of the system, they are documented in other books in the *Exploring Palm OS* series, as listed in the following tables.

Palm OS-Generated Events	
General Events	
appStopEvent	nilEvent
Events documented in *Exploring Palm OS: Input Services*	
gsiStateChangeEvent	keyDownEvent
keyHoldEvent	keyHoldEvent5

keyUpEvent	keyUpEvent5
penDownEvent	penMoveEvent
penUpEvent	
Events documented in *Exploring Palm OS: User Interface*	
ctlEnterEvent	ctlExitEvent
ctlRepeatEvent	ctlSelectEvent
daySelectEvent	fldChangedEvent
fldEnterEvent	fldHeightChangedEvent
frmCloseEvent	frmGadgetEnterEvent
frmGadgetMiscEvent	frmGotoEvent
frmLoadEvent	frmOpenEvent
frmSaveEvent	frmScrollPrvRefreshEvent
frmStopDialogEvent	frmTitleEnterEvent
frmTitleSelectEvent	frmUpdateEvent
insertionPointOffEvent	insertionPointOnEvent
lstEnterEvent	lstExitEvent
lstSelectEvent	menuCloseEvent
menuCmdBarOpenEvent	menuCmdBarTimeoutEvent
menuEvent	menuOpenEvent
popSelectEvent	prgUpdateEvent
sclEnterEvent	sclExitEvent
sclRepeatEvent	sysClearUIEvent
tblEnterEvent	tblExitEvent
tblSelectEvent	winEnterEvent
winExitEvent	winFocusGainedEvent

winFocusLostEvent

winResizedEvent

winUpdateEvent

winVisibilityChangedEvent

Events to be documented in *Exploring Palm OS: Creating a FEP*

tsmConfirmEvent

tsmFepButtonEvent

tsmFepChangeEvent

tsmFepDisplayOptionsEvent

tsmFepModeEvent

tsmFepSelectOptionEvent

Events documented in *Exploring Palm OS: Telephony and SMS*

telAsyncReplyEvent (kTelTelephonyEvent)

Events used internally by the operating system or reserved for future use

amWorkerDoneEvent

attnIndicatorEnterEvent

attnIndicatorSelectEvent

certMgrWorkerDoneEvent

debugEvent

exgLocalEvtNotify

exgLocalEvtDie

reservedEventCode1

reservedEventCode2

reservedEventCode3

reservedFindEvent

stringInputEvent

tunneledEvent

Summary of Event APIs

Functions Declared in Event.h

Main Event Queue Management

EvtAddEventToQueue

EvtAddEventToQueueAtTime

EvtAddUniqueEventToQueue

EvtAddUniqueEventToQueueAtTime

EvtEventAvail

EvtGetEvent

EvtSetNullEventTick

EvtSysEventAvail

Pen Queue Management

EvtDequeuePenPoint

EvtDequeuePenStrokeInfo

EvtFlushNextPenStroke

EvtFlushPenQueue

EvtGetPen

EvtGetPenNative

Key Queue Management

EvtDequeueKeyEvent

EvtEnqueueKey

EvtFlushKeyQueue

EvtKeydownIsVirtual

EvtKeyQueueEmpty

Handling Power On

EvtWakeup

EvtWakeupWithoutNilEvent

Communicating Between Threads in the Same Process

EvtAddEventToEventQueue

EvtGetThreadEventQueue

EvtReleaseEventQueue

Communicating Between Threads in Different Processes

EvtAcquireEventQueue

EvtCreateBackgroundThread

EvtGetReplyEventQueue

EvtLookupEventQueue

EvtPublishEventQueue

Blocking on the Event Queue’s IOS File Descriptor

EvtFinishLastEvent

EvtGetEventDescriptor

Getting the Current Focus

EvtGetFocusWindow

Debugging

EvtEventToString

4

Notifications

Applications can register for **notifications** that are sent when certain system-level events or application-level events occur. Notifications are similar to application launch codes, with one important difference: notifications are only sent to applications or code resources that have specifically registered to receive them, making them more efficient than launch codes.

This chapter describes the Palm OS notification mechanism. It shows how to register for a notification and how to deal with the notifications that you then receive. It provides some detail on some of the more commonly-used notifications: those that signal when the device is about to go to sleep and those that indicate that it is waking up. This chapter then discusses a special class of notifications—helper notifications—that can be used to publish and request application services. Finally, it concludes with a complete list of all of the notifications defined by Palm OS.

Reference material for many of the APIs discussed in this chapter can be found in Chapter 4, “Notifications,” on page 59.

Notification Overview

The Palm OS system and the built-in applications send notifications when certain events occur. (For a complete list, see “Notification Summary” on page 72.) It’s also possible for your application to create and broadcast its own notifications. However, applications rarely do so. It's more likely that you'll want to register to receive the predefined notifications or that you'll broadcast the predefined `sysNotifyHelperEvent` described under "Helper Notifications" on page 66.

A given notification is sent to each of the **notification clients** that register for it. 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.

- 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, the notification is deleted, 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 4.1.

Listing 4.1 Registering for a notification

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

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

- The 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.
- The notification for which you are registering. In the above examples, `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 means by which the notification should be received. Applications should 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 priority with which the notification should be sent. `sysNotifyNormalPriority` means that you don't want your code to receive any special consideration when receiving the notification. Notifications are broadcast synchronously in priority order. The lower the number you specify here, the earlier you receive the notification in the list.

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

After you've made the calls shown in Listing 4.1 and the system is about to begin a HotSync operation, it broadcasts the sysNotifySyncStartEvent notification to both clients. Along with the notification your code receives 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.)

Writing a Notification Handler

The application's (or a library's) response to sysAppLaunchCmdNotify is called a **notification handler**. A notification handler may perform any processing necessary, including displaying a user interface or broadcasting other notifications.

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

If you need to perform some lengthy process in a notification handler, one way to ensure that you aren't blocking other events is to send yourself a deferred notification. For example, Listing 4.2 shows a notification handler for the sysNotifyTimeChangeEvent notification that performs no work other than setting up a deferred notification (myDeferredNotifyEvent--which is a custom notification) 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 4.2 Deferring notification within a handler

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

      /* initialize myData here */

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

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

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

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

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

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

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

Sleep and Wake Notifications

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

`sysNotifySleepRequestEvent`: Broadcast during `SysHandleEvent()` processing when the system has decided to go to sleep.

sysNotifySleepNotifyEvent: Broadcast during
SysHandleEvent() immediately before the system is put to sleep. After the broadcast is complete, the system is put to sleep.

sysNotifyEarlyWakeupEvent: Broadcast during
SysHandleEvent() immediately after the system has finished sleeping. The screen may still be turned off, and the system may not fully wake up. It may simply handle an alarm or a battery charger event and go back to sleep.

sysNotifyLateWakeupEvent: Broadcast during
SysHandleEvent() immediately after the device has finished waking up.

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

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

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

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

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

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

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

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

During a wake-up event, the other two notifications listed above are broadcast. The `sysNotifyEarlyWakeupEvent` is broadcast very early on in the wake-up 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 wake-up event will probably want to register for `sysNotifyLateWakeupEvent` instead. At this stage, the screen has been turned on and the system is guaranteed to fully wake up.

When the handheld receives the `sysNotifyLateWakeupEvent` notification, it may be locked and waiting for the user to enter the password. If this is the case, you must wait for the user to unlock the handheld before you display a user interface. Therefore, if you intend to display a user interface when the handheld wakes up, you should make sure the handheld is not locked. If the handheld is locked, you should register for `sysNotifyDeviceUnlocked` notification and display your user interface when it is received. See Listing 4.3.

Listing 4.3 Responding to Late Wake-up Notification

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

Helper Notifications

The helper notification, `sysNotifyHelperEvent`, is a way for one application to request a service from another application. Currently, the Dial application is the only application that performs a service through `sysNotifyHelperEvent`. Specifically, the Dial application dials a phone in response to this notification. The Address Book uses the Dial application to dial the phone number that the user has selected. You can use the Dial application in a similar way by broadcasting the `sysNotifyHelperEvent` from your application. You may also choose to write a provider of services.

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

A helper registers for the `sysNotifyHelperEvent` notification. In the notification handler, the helper responds to action requests pertaining to the **service** that it provides.

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

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

The possible services are defined in `HelperServiceClass.h` and described in Chapter 10, "Helper Service Class," on page 181. These services are to dial a number, email a message, send an SMS message, or send a fax. If you want to define your own service, you must register a unique creator ID for that service. Alternatively, you can use the creator ID of your application.

When to Use the Helper API

There are several means by which one application can communicate with another application on the same handheld. Specifically, an application can send a launch code to another application (see "Launching Applications Programmatically" on page 29, can use the Exchange Manager and Local Exchange Library to send data to another application (see Chapter 4, "Object Exchange," in *Exploring Palm OS: High-Level Communications*), or can use the helper API to request that a service be performed.

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

Requesting a Helper Service

Listing 4.4 shows how an application should request the dial service. In general, you should do the following to request a service:

- Broadcast a `sysNotifyHelperEvent` with a `kHelperNotifyActionCodeValidate` action each time you want to verify that the service is available.

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

- Broadcast a `sysNotifyHelperEvent` with a `kHelperNotifyActionCodeExecute` action when you want the service performed. See Listing 4.4.
- If you want to obtain a list of all possible services, broadcast a `sysNotifyHelperEvent` with a `kHelperNotifyActionCodeEnumerate` action. You might do so when your application is launched, upon system reset, or any time the user performs a task where you might want to provide a service.

Listing 4.4 Requesting a helper service

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

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

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

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

   SysNotifyBroadcast(&param);

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

```
    return (execute.err == errNone);
}
```

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

- Always use `SysNotifyBroadcast()`, which broadcasts the notification synchronously.
- The notification’s *`notifyDetailsP`* parameter points to a `HelperNotifyEventType`. This structure allows the broadcaster to communicate with the helper.
- The helper may allocate memory and add it to the `HelperNotifyEventType` structure. In particular, if the action code is `kHelperNotifyActionCodeEnumerate`, the helper allocates at least one structure of type `HelperNotifyEnumerateListType` and adds it to the `data` field in the `HelperNotifyEventType` structure. The broadcaster must free this memory, even though the helper allocated it.
- The broadcaster uses the `helperAppID` field to communicate directly with a particular provider of the requested service. For example, suppose two applications provide a dial service. The broadcaster might discover these two applications through the enumerate action and then allow the user to specify which application should dial the phone number. When broadcasting the enumerate action, no helper ID is specified, so all helpers respond. After the user has set the preferred helper, the broadcaster sets the `helperAppID` field for the validate and execute actions to that helper’s creator ID. A helper must check the `helperAppID` field and only respond to the notification if its creator ID matches the value in that field or if that field is 0.
- The `dataP` field contains the data required to perform the service. For the dial service, `dataP` contains the phone number to dial. If any extra information is required or desired, then it is provided in the `detailsP` field. If you’re requesting the email or SMS service, you use `detailsP` to provide the message to be sent. See Chapter 10, “Helper Service Class,” on page 181 for more information.
- The `handled` field of `SysNotifyParamType` and the `err` field of the `HelperNotifyEventType` structure are used to

return the result. Always set `handled` to `false` and `err` to `errNone` before broadcasting and check their values after the broadcast is complete. The helper uses `handled` to indicate if it attempted to handle the service. If `handled` is `true`, it uses `err` to indicate the success or failure of performing that service.

Implementing a Helper

To implement a helper, do the following:

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

Listing 4.5 and Listing 4.6 show how the Dial application responds to the enumerate and validate actions. Note that the enumerate action requires the helper to allocate memory and add that memory to the `HelperNotifyEventType` structure pointed to by `notifyDetailsP` in the `SysNotifyParamType` parameter block. In this case, the `notifyDetailsP->dataP` field is a linked list of `HelperNotifyEnumerateListType` structures. Each helper must allocate one of these structure per service and add it to the end of the list. The broadcaster is responsible for freeing all of these structures after the notification broadcast is complete.

Listing 4.5 Enumerating services provided

```
Boolean PrvAppEnumerate
(HelperNotifyEventType *helperNotifyEventP)
{
   HelperNotifyEnumerateListType* newNodeP;
   MemHandle handle;
   MemPtr stringP;
   newNodeP = MemPtrNew
      (sizeof(HelperNotifyEnumerateListType));

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

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

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

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

   return true;
}
```

Listing 4.6 show how the Dial application responds to the validate action.

Listing 4.6 Responding to validate action

```
Boolean PrvAppValidate (SysNotifyParamType *sysNotifyParamP)
{
   HelperNotifyEventType* helperNotifyEvent;
   helperNotifyEvent = sysNotifyParamP->notifyDetailsP;
   // Check version
   if (helperNotifyEvent->version < 1)
      return false;

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

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

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

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

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

- Always check the `helperAppID` field and only respond if it is 0 or if it matches your creator ID. For the validate and execute actions, a broadcaster may use `helperAppID` to only communicate with the desired helper.
- If you handle the action, set `handled` to `true`. If the handling of the service was unsuccessful, set the `err` field in `notifyDetailsP`.
- Always check the `handled` field before performing the service. If any helper can perform the service, you must make sure that the service has not already been performed before you perform it. If `handled` is `true`, the service has already been performed.

Notification Summary

Table 4.1 lists the standard notifications that are supported in Palm OS Cobalt. These notifications are declared in the header

NotifyMgr.h. All the parameters for a notification are passed in a SysNotifyParamType structure and the results are returned in that same structure.

Table 4.1 Notification Constants

Constant	Description
cncNotifyConnectionStateEvent	Broadcast by the Connection Manager whenever a persistent profile is either connected or disconnected.
shortCutNotifyAddDbgMacrosEvent	
sysExternalConnectorAttachEvent	A device has been attached to an external connector.
sysExternalConnectorDetachEvent	A device has been detached from an external connector.
sysNotifyAltInputSystemDisabled	An alternative input system (such as an external keyboard) has become disabled.
sysNotifyAltInputSystemEnabled	An alternative input system (such as an external keyboard) has been enabled.
sysNotifyAntennaRaisedEvent	The antenna has been raised on a Palm VII series handheld.
sysNotifyAppServicesEvent	
sysNotifyCardInsertedEvent	An expansion card has been inserted into the expansion slot.
sysNotifyCardRemovedEvent	An expansion card has been removed from the expansion slot.
sysNotifyDBAddedEvent	A new database has been added to the device.
sysNotifyDBChangedEvent	Database info has been set on a database, such as with DmSetDatabaseInfo().
sysNotifyDBCreatedEvent	A database has been created.
sysNotifyDBDeletedEvent	A database has been deleted.

Table 4.1 Notification Constants *(continued)*

Constant	Description
`sysNotifyDBDirtyEvent`	An overlay has been opened, a database has been opened for write, or another event has occurred which has made the database info "dirty."
`sysNotifyDeleteProtectedEvent`	The Launcher has attempted to delete a protected database.
`sysNotifyDeviceUnlocked`	The user has unlocked the handheld.
`sysNotifyDisplayChangeEvent`	The color table or bit depth has changed.
`sysNotifyEarlyWakeupEvent`	The system is starting to wake up.
`sysNotifyEvtGotAttnEvent`	System use only.
`sysNotifyForgotPasswordEvent`	The user has tapped the Lost Password button in the Security application.
`sysNotifyHelperEvent`	An application has requested that a particular service be performed.
`sysNotifyHostFSInitDone`	System use only.
`sysNotifyLateWakeupEvent`	The system has finished waking up.
`sysNotifyLocaleChangedEvent`	The system locale has changed.
`sysNotifyMenuCmdBarOpenEvent`	The system is about to display the menu command toolbar.
`sysNotifyPhoneEvent`	Reserved for future use.
`sysNotifyPOSEMountEvent`	System use only.
`sysNotifyResetFinishedEvent`	The system has finished a reset.
`sysNotifyRetryEnqueueKey`	The Attention Manager has failed to post a virtual character to the key queue.
`sysNotifySecuritySettingEvent`	The device security level has been changed.
`sysNotifySleepNotifyEvent`	The system is about to go to sleep.

Table 4.1 Notification Constants *(continued)*

Constant	Description
`sysNotifySleepRequestEvent`	The system has decided to go to sleep.
`sysNotifySyncFinishEvent`	A HotSync operation has just completed.
`sysNotifySyncStartEvent`	A HotSync operation is about to begin.
`sysNotifyTimeChangeEvent`	The system time has just changed.
`sysNotifyVolumeMountedEvent`	A file system has been mounted.
`sysNotifyVolumeUnmountedEvent`	A file system has been unmounted.
telNotifyEnterCodeEvent	
telNotifyErrorEvent	

Notification Function Summary

Notification Manager Functions

`SysNotifyBroadcast`
`SysNotifyBroadcastDeferred`
`SysNotifyRegister`
`SysNotifyRegisterBackground`
`SysNotifyRegisterV40`
`SysNotifyUnregister`
`SysNotifyUnregisterV40`

Part III
Reference

This part contains reference documentation for the following:

5

Application Manager

This chapter provides reference documentation for the Application Manager, which you use to launch Palm OS applications programmatically. The contents of this chapter are organized as follows:

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

For more information on how Palm OS applications are launched, see "Application Start and Stop" on page 21.

Application Manager Structures and Types

ARMAppLaunchPrefsType Struct

Purpose ARM application's launch preferences.

Declared In `AppMgr.h`

Prototype

```
typedef struct ARMAppLaunchPrefsType {
   uint32_t version;
   uint32_t reserved1;
   uint32_t reserved2;
   uint32_t stackSize;
   uint32_t flags;
} ARMAppLaunchPrefsType
```

Fields `version`

Version of this structure. See "Launch Preferences Structure Versions" on page 90 for the values that this field can assume.

`reserved1`
Reserved for future use.

`reserved2`
Reserved for future use.

`stackSize`
Not used in Palm OS Cobalt. This field should always have a value of 0.

`flags`
Any combination of the launch flags listed under "Launch Preference Flags" on page 89.

Example You can obtain an application's launch preferences using code similar to the following:

```
DmOpenRef openRef;

// Open the database
openRef = DmOpenDatabase(dbID, dmModeReadOnly);
if (openRef) {

   // Look for its launch preferences.
   MemHandle resH = DmGetResource(openRef,
      sysResTAppLaunchPrefsLE32, sysResIDDefault);

   if (resH) {
      ARMAppLaunchPrefsType *launchPrefs =
         (ARMAppLaunchPrefsType *)MemHandleLock(resH);

      // Do something with the launch prefs here

      MemHandleUnlock(resH);
      DmReleaseResource(resH);
   }
}
```

ImportExportRecordParamsType Struct

Purpose Parameter block passed with the `sysAppLaunchCmdImportRecord`, `sysAppLaunchCmdMoveRecord`, sysAppLaunchCmdExportRecord and sysAppLaunchCmdDeleteRecord launch codes.

Declared In AppMgr.h

Prototype

```
typedef struct {
   uint32_t index;
   uint32_t destIndex;
   uint32_t uniqueID;
   MemHandle vObjectH;
} ImportExportRecordParamsType
typedef ImportExportRecordParamsType
*ImportExportRecordParamsPtr;
```

Fields

index

Index of the database record to be exported, moved, or deleted, or dmMaxRecordIndex if the uniqueID field identifies the record. When importing, this value is ignored. This value is updated after the sublaunch.

destIndex

Index of the destination location for the record to be moved when the launch code is sysAppLaunchCmdMoveRecord.

uniqueID

The record's unique ID. This field is ignored unless the index field is set to dmMaxRecordIndex. When importing, if this field is set to a valid record unique ID (other than dmUnusedRecordID) the imported record should replace the one specified.

vObjectH

Memory handle for the location that contains the record being exported or the location where the record being imported is to be stored.

SysAppLaunchCmdCardType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdCardLaunch` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   status_t err;
   uint16_t volRefNum;
   uint16_t _reserved1;
   const char *path;
   uint16_t startFlags;
   uint16_t padding;
} SysAppLaunchCmdCardType
```

Fields

← *`err`*

Initially set to `expErrUnsupportedOperation`, applications that recognize `sysAppLaunchCmdCardLaunch` and that don't want to receive the subsequent `sysAppLaunchCmdNormalLaunch` launch code should set this field to `errNone`.

→ *`volRefNum`*

The reference number of the volume from which the application is being launched.

`_reserved1`

Reserved for future use.

→ *`path`*

The complete path to the application being launched.

↔ *`startFlags`*

A combination of the flags listed under "Expansion Card Launch Flags" on page 89.

`padding`

Padding bytes.

SysAppLaunchCmdFailedAppNotifyType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdFailedAppNotify` launch code. This structure identifies both the failed application and the reason for failure.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   uint32_t creator;
   uint32_t type;
   status_t result;
} SysAppLaunchCmdFailedAppNotifyType
```

Fields

`creator`
The failed application's creator ID.

`type`
The failed application's type.

`result`
The error code returned from the failed application.

SysAppLaunchCmdHandleSyncCallAppType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdHandleSyncCallApp` launch code. This structure contains all of the information passed to `SyncCallRemoteModule()` on the desktop plus the fields needed to pass the result back to the desktop.

Declared In `AppMgr.h`

Prototype

```
typedef struct
SysAppLaunchCmdHandleSyncCallAppType {
   uint16_t pbSize;
   uint16_t action;
   void *paramP;
   uint32_t dwParamSize;
   void *dlRefP;
   Boolean handled;
   uint8_t _reserved1;
   uint16_t _reserved2;
   status_t replyErr;
   uint32_t dwReserved1;
   uint32_t dwReserved2;
} SysAppLaunchCmdHandleSyncCallAppType
```

Fields

`pbSize`

Size, in bytes, of this parameter block. Set to `sizeof(SysAppLaunchCmdHandleSyncCallAppType)`.

`action`

Call action ID (application-specific).

`paramP`

Pointer to parameter block (call action ID specific).

`dwParamSize`

Parameter block size, in bytes.

`dlRefP`

DesktopLink reference pointer. Supply this value in the `DlkCallAppReplyParamType` structure when calling `DlkControl()` with the `dlkCtlSendCallAppReply` control code.

`handled`

Initialized to `false` by DLServer; if handled, your application must set it to `true` (and your handler the handler must call `DlkControl` with the `dlkCtlSendCallAppReply`control code). If your handler is not going to send a reply back to the conduit, leave this field set to `false`, in which case the DesktopLink Server will send the default "unknown request" reply.

`_reserved1`

Reserved. Set to `NULL`.

`_reserved2`
: Reserved. Set to `NULL`.

`replyErr`
: Error code returned from the call to `DlkControl()` with the `dlkCtlSendCallAppReply` control code.

`dwReserved1`
: Reserved. Set to `NULL`.

`dwReserved2`
: Reserved. Set to `NULL`.

SysAppLaunchCmdInitDatabaseType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdInitDatabase` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct SysAppLaunchCmdInitDatabaseType {
   DmOpenRef dbP;
   uint32_t creator;
   uint32_t type;
   uint16_t version;
   uint16_t padding;
} SysAppLaunchCmdInitDatabaseType
```

Fields

`dbP`
: Handle of the newly-created database, already open for read/write access.

`creator`
: Creator ID of the newly-created database.

`type`
: Type of the newly-created database.

`version`
: Version number of the newly-created database.

`padding`
: Padding bytes.

Comments

IMPORTANT: The `sysAppLaunchCmdInitDatabase` launch code handler *must* leave the database handle open on return.

SysAppLaunchCmdOpenDBType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdOpenDB` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   MemHandle dbH;
} SysAppLaunchCmdOpenDBType
```

Fields

`dbH`
: Handle to the database to open.

SysAppLaunchCmdPnpsType Struct

Purpose Parameter block that accompanies a `sysAppLaunchPnpsPreLaunch` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   status_t error;
   uint16_t volRefNum;
   uint16_t slotLibRefNum;
   uint16_t slotRefNum;
   uint16_t _reserved1;
} SysAppLaunchCmdPnpsType
```

Fields

`error`
: Error code returned from the pre-launch application. Set this field `errNone` to prevent the application from receiving a `sysAppLaunchCmdNormalLaunch` launch code.

`volRefNum`
: Volume reference number, or zero if a file system wasn't mounted.

`slotLibRefNum`
: Slot driver library reference number. This field is always valid for a slot driver call.

`slotRefNum`
: Slot reference number. This field is always valid for a slot driver call.

`_reserved1`
: Reserved for future use.

SysAppLaunchCmdSaveDataType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdSaveData` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   Boolean uiComing;
   uint8_t reserved1;
} SysAppLaunchCmdSaveDataType
```

Fields

`uiComing`

`true` if the system dialog is displayed before launch code arrives.

`reserved1`

Reserved for future use.

SysAppLaunchCmdSyncCallApplicationTypeV10 Struct

Purpose

Declared In `AppMgr.h`

Prototype

```
typedef struct
SysAppLaunchCmdSyncCallApplicationTypeV10 {
   uint16_t action;
   uint16_t paramSize;
   void *paramP;
   uint8_t remoteSocket;
   uint8_t tid;
   Boolean handled;
   uint8_t reserved1;
} SysAppLaunchCmdSyncCallApplicationTypeV10
```

Fields

`action`

`paramSize`

`paramP`

`remoteSocket`

`tid`

`handled`

`reserved1`

SysAppLaunchCmdSystemResetType Struct

Purpose Parameter block that accompanies a `sysAppLaunchCmdSystemReset` launch code.

Declared In `AppMgr.h`

Prototype

```
typedef struct {
   Boolean hardReset;
   Boolean createDefaultDB;
} SysAppLaunchCmdSystemResetType
```

Fields

`hardReset`
`true` if system was hard reset. `false` if system was soft reset.

`createDefaultDB`
If `true`, application has to create default database.

PilotMainType Typedef

Purpose Type used to declare pointers to a `PilotMain()` function.

Declared In `AppMgr.h`

Prototype

```
typedef uint32_t (PilotMainType) (uint16_t cmd,
    void *cmdPBP, uint16_t launchFlags)
```

See Also `PilotMain()`

Application Manager Constants

Expansion Card Launch Flags

Purpose Used in combination to specify how sysAppLaunchCmdCardLaunch is to operate. Supply one or more of these flags to the `startFlags` field of the SysAppLaunchCmdCardType structure that accompanies the launch code.

Declared In `AppMgr.h`

Constants `#define sysAppLaunchStartFlagAutoStart 0x0001`
Indicates that the application is being run automatically upon card insertion.

`#define sysAppLaunchStartFlagNoAutoDelete 0x0004`
Prevents the VFS Manager from deleting the copy of the application in main memory when the associated volume is unmounted.

`#define sysAppLaunchStartFlagNoUISwitch 0x0002`
Prevents a UI switch to the auto-launched application.

Launch Preference Flags

Purpose Flags that control how an ARM application is launched. These flags can be used in combination to make up the value of the ARMAppLaunchPrefsType structure's `flags` field.

Declared In `AppMgr.h`

Constants `#define ARMAppLaunchPrefsFindNotification 0x02`
If set, the application is a sent a sysAppLaunchCmdFind launch code upon launch.

`#define ARMAppLaunchPrefsNoOverlay 0x20`

`#define ARMAppLaunchPrefsResetNotification 0x01`
If set, the application is a sent a sysAppLaunchCmdSystemReset launch code upon launch.

`#define ARMAppLaunchPrefsTimeChangeNotification 0x04`

If set, the application is a sent a `sysAppLaunchCmdTimeChange` launch code upon launch.

`#define ARMAppLaunchPrefsReserved 0xffffffd8`

Reserved flag bits. The corresponding bits in the `ARMAppLaunchPrefsType` structure's `flags` field must be set to zero.

Launch Preferences Structure Versions

Purpose Identify the version of the `ARMAppLaunchPrefsType` structure. That structure's `version` field should contain one of these values.

Declared In `AppMgr.h`

Constants `#define ARMAppLaunchPrefsTypeVersion60 1`

The first version of the structure as defined in Palm OS Cobalt, version 6.0.

`#define ARMAppLaunchPrefsTypeVersionCurrent ARMAppLaunchPrefsTypeVersion60`

The current version of the structure.

Miscellaneous Application Manager Constants

Purpose The Application Manager header file also defines these constants.

Declared In `AppMgr.h`

Constants `#define ImpExpInvalidRecIndex 0xFFFFFFFF`

`#define ImpExpInvalidUniqueID dmUnusedRecordID`

Application Manager Functions and Macros

PilotMain Function

Purpose The entry point for all Palm OS applications, this function's sole purpose is to receive and respond to launch codes.

Declared In `AppMgr.h`

Prototype

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

Parameters

→ *`cmd`*

The launch code to which your application is to respond. See Chapter 6, "Common Launch Codes," on page 103 for a list of predefined launch codes. You may create additional launch codes; see "Creating Your Own Launch Codes" on page 32.

→ *`cmdPBP`*

A pointer to a structure containing any launch-command-specific parameters, or `NULL` if the launch code has none. See the description of each launch code for a description of the parameter structure that accompanies it, if any.

→ *`launchFlags`*

Flags that indicate whether your application's global variables are available, whether your application is now the active application, whether it already was the active application, and so on. See "Launch Flags" on page 105 for a list of launch flags.

Returns Return `errNone` if your application processed the launch code successfully, or an appropriate error code if there was a problem. When another application invokes your application using `SysAppLaunch()`, this value is returned to the caller.

Comments See Chapter 2, "Application Start and Stop," on page 21 for a discussion on how applications receive and handle launch codes, with examples.

SysAppLaunch Function

Purpose Launch an application as a subroutine of the caller in the caller's process (irrespective of whether or not the application being launched is already running in another process).

> **NOTE:** Applications should avoid this function; they should use `SysAppLaunchLocal()` or `SysAppLaunchRemote()` instead.

Declared In `AppMgr.h`

Prototype

```
status_t SysAppLaunch (DatabaseID dbID,
   uint16_t cmd, void *cmdPBP, uint32_t *resultP)
```

Parameters

→ *dbID*
The database ID of the resource database containing the application to launch.

→ *cmd*
Launch code passed to the launched application's `PilotMain()` function.

→ *cmdPBP*
Pointer to the launch code parameter block.

← *resultP*
The value returned from the application's `PilotMain()` routine.

Returns Returns `errNone` if the application was launched successfully.

Comments Applications can use `SysAppLaunch()` to send a specific launch code to another application and have control return to the calling application when finished. This function in effect makes the specified application a subroutine of the caller. If you want to actually close your application and call another application, use `SysUIAppSwitch()` instead of this function. `SysUIAppSwitch()` sends the current application an `appStopEvent` and then starts the specified application.

Do not use this function to open the system-supplied Launcher application. If another application has replaced the default launcher with one of its own, this function will open the custom launcher instead of the system-supplied one. To open the Launcher reliably, enqueue a `keyDownEvent` that contains a `launchChr`.

You can call this function only in the context of the main UI application thread. To invoke the `PilotMain()` procedure of any application in the context of another thread, use `SysLoadModule()` and `SysGetEntryAddresses()` instead.

If the target application happens to be the same as the root application of the calling process, the target application's `PilotMain()` is re-entered in the context of the calling thread. In this case the `sysAppLaunchFlagSubCall` launch flag is set.

Before the `PilotMain()` procedure of the target application is entered, the database of the target application is opened and added to the default resource search chain. After the target application's `PilotMain()` exits, that database is closed. If the closing causes the open count of the database to become zero, the database is removed from the default resource search chain.

NOTE: For important information regarding the correct use of this function, see Chapter 2, "Application Start and Stop," on page 21.

See Also `SysBroadcastActionCode()`, `SysUIAppSwitch()`, `SysCurAppDatabase()`

SysAppLaunchLocal Function

Purpose Launch an application as a subroutine of the caller in the caller's process, unless the application is already running in another process. If the application is already running in another process the launch code and parameters are sent to the running application.

Declared In `AppMgr.h`

Prototype

```
status_t SysAppLaunchLocal (DatabaseID dbID,
   uint16_t cmd, void *cmdPBP,
   uint32_t cmdPBSize, uint32_t *resultP)
```

Parameters

→ *dbID*

The database ID of the resource database containing the application to launch.

→ *`cmd`*
Launch code passed to the launched application's `PilotMain()` function.

→ *`cmdPBP`*
Pointer to the launch code parameter block.

→ *`cmdPBSize`*
Size, in bytes, of the launch code parameter block.

← *`resultP`*
The value returned from the application's `PilotMain()` routine.

Returns Returns `errNone` if the application was launched successfully.

Comments A local sublaunch becomes a local subroutine invocation in the same process.

See Also `SysAppLaunch()`, `SysAppLaunchRemote()`

SysAppLaunchRemote Function

Purpose Launch an application as a subroutine of the caller in a separate, newly-created process, unless the application is already running in another process. If the application is already running in another process the launch code and parameters are sent to the running application.. Remote launching allows applications to execute untrusted code without compromising their own security.

Declared In `AppMgr.h`

Prototype

```
status_t SysAppLaunchRemote (DatabaseID dbID,
   uint16_t cmd, void *cmdPBP,
   uint32_t cmdPBSize, uint32_t *resultP)
```

Parameters

→ *`dbID`*
The database ID of the resource database containing the application to launch.

→ *`cmd`*
Launch code passed to the launched application's `PilotMain()` function.

→ *`cmdPBP`*
Pointer to the launch code parameter block. If *`cmdPBSize`* is non-zero, *`cmdPBP`* is interpreted as the address of a block of

memory whose size is *cmdPBSize* bytes. If the target application is started in a separate transient process, the contents of that memory block are copied to the transient process, and the address of that copy is passed to the target application's `PilotMain()` procedure as the *cmdPBP* parameter.

If *cmdPBSize* is zero, the value of *cmdPBP* is passed as-is to the target application's `PilotMain()` procedure as the *cmdPBP* parameter. No memory is copied even if the target application is started in a separate process.

→ *cmdPBSize*

Size, in bytes, of the launch code parameter block, or zero if the *cmdPBP* parameter is to be passed as-is to the launched application's `PilotMain()` function.

← *resultP*

The value returned from the application's `PilotMain()` function.

Returns Returns `errNone` if the application was launched successfully.

Comments This function creates a separate transient process in which to execute the target application, unless the target application happens to be the same as the root application of the calling process—in which case `SysAppLaunchRemote()` simply performs a local sublaunch as `SysAppLaunch()` does.

See Also `SysAppLaunch()`, `SysAppLaunchLocal()`

SysAppLaunchV40 Function

Purpose Launch a specified application as a subroutine of the caller.

Declared In `AppMgr.h`

Prototype

```
status_t SysAppLaunchV40 (uint16_t cardNo,
    LocalID dbID, uint16_t launchFlags,
    uint16_t cmd, MemPtr cmdPBP,
    uint32_t *resultP)
```

Parameters → *cardNo*

The card number of the resource database containing the application to launch.

→ *dbID*
The local ID of the resource database containing the application to launch.

→ *launchFlags*
Set to 0.

→ *cmd*
Launch code.

→ *cmdPBP*
Launch code parameter block.

← *resultP*
The value returned from the application's `PilotMain()` routine.

Returns Returns `errNone` if no error, or one of `sysErrParamErr`, `memErrNotEnoughSpace`, or `sysErrOutOfOwnerIDs`.

Comments Applications can use `SysAppLaunch()` to send a specific launch code to another application and have control return to the calling application when finished. This function in effect makes the specified application a subroutine of the caller. If you want to actually close your application and call another application, use `SysUIAppSwitch()` instead of this function. `SysUIAppSwitch()` sends the current application an `appStopEvent` and then starts the specified application.

Do not use this function to open the system-supplied Application Launcher application. If another application has replaced the default launcher with one of its own, this function will open the custom launcher instead of the system-supplied one. To open the system-supplied launcher reliably, enqueue a `keyDownEvent` that contains a `launchChr`.

NOTE: For important information regarding the correct use of this function, see Chapter 2, "Application Start and Stop," on page 21.

Compatibility This function is provided for compatibility purposes only. Applications should use `SysAppLaunch()` instead.

See Also `SysBroadcastActionCode()`, `SysUIAppSwitch()`, `SysCurAppDatabase()`

SysBroadcastActionCode Function

Purpose Send the specified action code (launch code) and parameter block to the latest version of every UI application.

Declared In `AppMgr.h`

Prototype

```
status_t SysBroadcastActionCode (uint16_t cmd,
    void *cmdPBP)
```

Parameters

→ *`cmd`*
Launch code to send.

→ *`cmdPBP`*
Launch code parameter block to send.

Returns Returns `errNone` if no error, or one of the following errors: `sysErrParamErr`, `memErrNotEnoughSpace`, or `sysErrOutOfOwnerIDs`.

See Also `SysAppLaunch()`, Chapter 2, "Application Start and Stop," on page 21

SysCurAppDatabase Function

Purpose Get the database ID of the current application's resource database.

Declared In `AppMgr.h`

Prototype

```
status_t SysCurAppDatabase (DatabaseID *dbIDP)
```

Parameters

← *`dbIDP`*
Pointer to the location in memory where the database ID is to be written.

Returns Returns `errNone` if no error, or `sysErrParamErr` if an error occurs.

See Also SysAppLaunch(), SysGetModuleDatabase(), SysUIAppSwitch()

SysCurAppDatabaseV40 Function

Purpose Get the card number and database ID of the current application’s resource database.

Declared In `AppMgr.h`

Prototype

```
status_t SysCurAppDatabaseV40 (uint16_t *cardNoP,
   LocalID *dbIDP)
```

Parameters

← *`cardNoP`*

Pointer to the location in memory where the card number is to be written.

← *`dbIDP`*

Pointer to the location in memory where the database ID is to be written.

Returns Returns `errNone` if no error, or `sysErrParamErr` if an error occurs.

Compatibility This function is provided for compatibility purposes only. Applications should use `SysCurAppDatabase()` instead.

See Also SysAppLaunch(), SysUIAppSwitch()

SysGetStackInfo Function

Purpose Locate the start and end of the current thread’s stack.

Declared In `AppMgr.h`

Prototype

```
Boolean SysGetStackInfo (void **startPP,
   void **endPP)
```

Parameters

← *`startPP`*

Upon return, points to the start of the stack.

← *`endPP`*

Upon return, points to the end of the stack.

Returns Returns `true` if the stack has not overflowed, that is, the value of the stack overflow address has not been changed. Returns `false` if the stack overflow value has been overwritten, meaning that a stack overflow has occurred.

SysReset Function

Purpose Perform a soft reset and reinitialize the globals and the dynamic memory heap.

Declared In `AppMgr.h`

Prototype

```
void SysReset (void)
```

Parameters None.

Returns Nothing.

Comments This routine resets the system, reinitializes the globals area and all system managers, and reinitializes the dynamic heap. All database information is preserved. This routine is called when the user presses the reset switch on the device.

SysUIAppSwitch Function

Purpose Try to make the current UI application quit and then launch the UI application specified by database ID.

Declared In `AppMgr.h`

Prototype

```
status_t SysUIAppSwitch (DatabaseID dbID,
   uint16_t cmd, void *cmdPBP,
   uint32_t cmdPBSize)
```

Parameters

→ *`dbID`*
Database ID of the new application's resource database.

→ *`cmd`*
Launch code.

→ *`cmdPBP`*
Pointer to the launch code parameter block, or `NULL` if you don't need to pass a parameter block to the application.

→ *`cmdPBSize`*
Size, in bytes, of the parameter block pointed to by *`cmdPBP`*.

Returns Returns `errNone` if the application switch was performed successfully.

Comments When you launch an application using `SysUIAppSwitch()` you have the option to pass a parameter block (using the *`cmdPBP`* parameter) containing application-specific information to the application being launched. To create this parameter block, allocate it using `MemPtrNew()` and then call `MemPtrSetOwner()` to set the block's owner ID to 0. This assigns ownership of the block to the system; memory blocks owned by the system aren't automatically freed when the calling application exits. Once ownership of the block has been assigned to the system, neither the launching nor the launched application need worry about freeing the block since the operating system will do this itself after the launched application exits.

Note that your parameter block must be self contained. That is, it must not have pointers to anything on the stack or to memory blocks that are owned by an application. If you don't need to pass a parameter block to the application being launched, pass `NULL` for the *`cmdPBP`* parameter.

Do not use `SysUIAppSwitch()` to open the system-supplied Application Launcher application. If a third-party launcher is installed, you'll likely want to launch that one instead. To do this, enqueue a `keyDownEvent` that contains a `launchChr`. This will run whatever is run whenever the user taps the Applications icon.

See Also `SysAppLaunch()`, Chapter 2, "Application Start and Stop," on page 21

SysUIAppSwitchV40 Function

Purpose Try to make the current UI application quit and then launch the UI application specified by card number and database ID.

Declared In `AppMgr.h`

Prototype

```
status_t SysUIAppSwitchV40 (uint16_t cardNo,
   LocalID dbID, uint16_t cmd, MemPtr cmdPBP)
```

Parameters

→ *`cardNo`*
Card number for the new application; currently only card 0 is valid.

→ *`dbID`*
Local ID of the new application's resource database.

→ *`cmd`*
Action code (launch code).

→ *cmdPBP*
Action code (launch code) parameter block.

Returns Returns `errNone` if the application switch was performed successfully.

Comments May display a fatal error message if the *cardNo* parameter is invalid. On debug ROMs, displays a fatal error message if there is no currently running application.

Do not use this function to open the system-supplied Application Launcher application. If a third-party launch is installed, you'll likely want to launch that one instead. To do this, enqueue a `keyDownEvent` that contains a `launchChr`. This will run whatever is run whenever you tap on the Applications icon.

If you are passing a parameter block (the *cmdPBP* parameter), you must set the owner of the parameter block chunk to the operating system. To do this, and for more information, see `MemPtrSetOwner()`. 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 using `MemHandleSetOwner()` or `MemPtrSetOwner`. If you set the owner of this parameter block properly, the system maintains the parameter block and frees it when the second application quits. If you don't set the owner of the parameter block, the system frees the parameter block as soon as the calling application quits, causing unpredictable results.

Compatibility This function is provided for compatibility purposes only. Applications should use `SysUIAppSwitch()` instead.

See Also `SysAppLaunch()`, Chapter 2, "Application Start and Stop," on page 21

6

Common Launch Codes

This chapter provides detailed descriptions of many of the Palm OS application launch codes and flags. Launch codes that are specific to a particular technology are documented with that technology. For instance, launch codes used exclusively with the Alarm Manager are documented in the Alarm Manager reference chapter within *Exploring Palm OS: System Management*.

This chapter is organized into the following sections:

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

Further information on working with launch codes and flags, plus a complete listing of all Palm OS launch codes, can be found in Chapter 2, "Application Start and Stop," on page 21.

Common Launch Codes Structures and Types

GoToParamsType Struct

Purpose Parameter block for the `sysAppLaunchCmdGoTo` launch code. An application receives this launch code if the user selects one of its matching records in the Find Results dialog or to display data that has just been received using the Exchange Manager.

Declared In `Find.h`

Prototype

```
typedef struct {
   DatabaseID dbID;
   uint32_t recordNum;
   uint32_t recordID;
   size_t matchPos;
   size_t matchLen;
   uint32_t matchFieldNum;
   size_t searchStrLen;
   uint32_t matchCustom;
   char string[maxFindStrLen+1];
   uint8_t reserved1;
   uint8_t reserved2;
   uint8_t reserved3;
} GoToParamsType
typedef GoToParamsType *GoToParamsPtr
```

Fields

`dbID`

Database ID of the record database to open.

`recordNum`

Index of the database record to display.

`recordID`

Unique ID of the database record to display.

`matchPos`

Byte offset of the start of the matching text within the record. The Exchange Manager does not use this field.

`matchLen`

The number of bytes of matched text found in the record. The Exchange Manager does not use this field.

`matchFieldNum`

Index of the text field in which the matching text should be displayed.

If your application's database is a schema database, use this field to set the column ID that contains the matching text.

The Exchange Manager does not use this field.

`searchStrLen`

Length of normalized search string. This is *not* the length of the matching string. `matchLen` contains the length of the matching string.

The Exchange Manager does not use this field.

`matchCustom`

Application-specific information.

`string`

The `strAsTyped` field from `FindParamsType`. The Exchange Manager does not use this field.

`reserved1`

Reserved for future use.

`reserved2`

Reserved for future use.

`reserved3`

Reserved for future use.

Comments Some multi-byte character encodings represent certain characters both as a single-byte character and a multi-byte character. When the search is performed, the single-byte character is accurately matched against its multi-byte equivalent. For this reason, the length of the string searched for does not always equal the length of the matching string.

See Also `FindParamsType`, `FindSaveMatch()`

Common Launch Codes Constants

Launch Flags

Purpose Flags that compose the *`launchFlags`* argument of an application's `PilotMain()`.

Declared In `CmnLaunchCodes.h`

Constants `#define sysAppLaunchFlagDataRelocated 0x80`

Indicates that global data (static pointers) have been relocated. *Note: This flag is for internal use by `SysAppLaunch()` only. It should never be set by the caller.*

`#define sysAppLaunchFlagGlobalsAvailable 0x20`

Indicates that the application can access globals. This flag is set whenever `sysAppLaunchFlagNewGlobals` is set, or when the application has an unique runtime ID. *Note: This flag is for internal use by `SysAppLaunch()` only. It should never be set by the caller.*

`#define sysAppLaunchFlagNewGlobals 0x04`

Set this flag to create a new globals environment for the application being launched. Note that a new globals environment implies a new owner ID for memory chunks.

`#define sysAppLaunchFlagNewStack 0x02`

Set this flag to have the launched application use its own, newly-created, stack.

`#define sysAppLaunchFlagNewThread 0x01`

Set this flag to have the application launched in a new thread. Applications launched with this flag set will also get a new stack, irrespective of the `sysAppLaunchFlagNewStack` flag.

```
#define sysAppLaunchFlagPrivateSet
  (sysAppLaunchFlagSubCall |
  sysAppLaunchFlagDataRelocated |
  sysAppLaunchFlagGlobalsAvailable)
```

The set of private, internal flags that should never be set by the caller.

`#define sysAppLaunchFlagSubCall 0x10`

Set this flag to indicate that the application is calling its own entry point as a subroutine call. When this flag is set, the A5 (globals) pointer remains valid through the call. *Note: This flag is for internal use by `SysAppLaunch()` only. It should never be set by the caller.*

`#define sysAppLaunchFlagUIApp 0x08`

Indicates to the application being launched that it is a UI application.

Miscellaneous Common Launch Codes Constants

Purpose The header file `CmnLaunchCodes.h` also declares these constants.

Declared In `CmnLaunchCodes.h`

Constants `#define sysAppLaunchCmdCustomBase 0x8000`
Application-specific launch codes should be defined starting with this value.

`#define sysDialLaunchCmdLast 39`
The last of the standard dialer service launch codes.

`#define sysSvcLaunchCmdLast 49`
The last of the standard service panel launch codes.

Common Launch Codes

sysAppLaunchCmdAddRecord

Purpose Add a record to an application's database.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdAddRecord 19`

Parameters The launch code's parameter block pointer references a `MailAddRecordParamsType` structure. This structure looks like this:

```
typedef struct {
   Boolean secret;
   Boolean signature;
   Boolean confirmRead;
   Boolean confirmDelivery;
   MailMsgPriorityType priority;
   UInt8 padding;
   Char *subject;
   Char *from;
   Char *to;
   Char *cc;
   Char *bcc;
   Char *replyTo;
   Char *body;
} MailAddRecordParamsType;

typedef MailAddRecordParamsType
   *MailAddRecordParamsPtr;
```

where:

- A `true` value for `secret` means that the message should be marked secret.
- A `true` value for `signature` indicates that the signature from the Mail application's preferences should be attached to the message.
- A `true` value for `confirmRead` means that a confirmation should be sent when the message is read.
- A `true` value for `confirmDelivery` means that a confirmation should be sent when the message is delivered.
- `priority` is either `high`, `normal`, or `low`.
- `subject` is a pointer to a null-terminated string containing the message's subject. Set this pointer to `NULL` to omit the subject line.
- `from` is a pointer to a null-terminated string containing the sender's address. This field is not currently used.
- `to` is a pointer to a null-terminated string containing the email addresses to which the message is to be sent.
- `cc` is a pointer to a null-terminated string containing any additional email address to which the message is to be sent. This pointer is required; if the message isn't to be sent to any additional addresses, `cc` should point to a NUL character.
- `bcc` is a pointer to a null-terminated string containing any "blind carbon copy" email address to which the message is to be sent. This pointer is required; if the message isn't to be sent to any bcc addresses, `bcc` should point to a NUL character.
- `replyTo` is a pointer to a null-terminated string containing the email address to which any replies should be sent.
- `body` is a pointer to a null-terminated string containing the text of the email message.

Comments This launch code is used to add a message to the Mail or iMessenger™ (on the Palm VII™) application's outbox. You pass information about the message such as address, body text, etc. in the parameter block. For iMessenger, you can set the `edit` field of the parameter block to control whether or not the iMessenger editor is displayed. Set it to `true` to display the editor or `false` not to display it.

See Also `sysAppLaunchCmdMoveRecord`

sysAppLaunchCmdAntennaUp

Purpose Sent when the antenna is raised on devices that are appropriately equipped.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdAntennaUp 53`

Parameters None.

Comments This launch code is typically used to switch to the Launcher (or whichever application the user has specified for this action).

sysAppLaunchCmdCardLaunch

Purpose Sent to an application that is being run from an expansion card, before the application is copied into the device's main memory.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdCardLaunch 58`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdCardType` structure.

Comments The application is copied into the device's main memory prior to being sent this launch code. If the application doesn't respond to `sysAppLaunchCmdCardLaunch`, it is then sent a `sysAppLaunchCmdNormalLaunch` launch code. Applications that can profit from the knowledge that they are being launched from an expansion card may want to consult the fields in the parameter block that accompanies `sysAppLaunchCmdCardLaunch`.

When the Launcher sends `sysAppLaunchCmdCardLaunch` to an application, it sets the `sysAppLaunchFlagNewStack`, `sysAppLaunchFlagNewGlobals`, and `sysAppLaunchFlagUIApp` flags (see "Launch Flags" on page 105 for documentation on these flags). These flags are not sent to `start.prc`, however. Applications should never interact with the user upon receiving this launch code, and should not depend on globals being available. This launch code is intended to notify the application that it is being launched from a card. Applications typically save some state information upon receiving this launch code and do the bulk of their processing when they receive `sysAppLaunchNormalLaunch`.

sysAppLaunchCmdCountryChange

Purpose Sent when the user has changed their country preference. As a result, various locale-specific formats should change.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdCountryChange 8`

Parameters None.

Comments Applications should change the display of numbers to use the proper number separators. To do this, call `LocGetNumberSeparators`, `StrLocalizeNumber`, and `StrDelocalizeNumber`.

sysAppLaunchCmdDeleteRecord

Purpose Generally sent to the PIM applications, this launch code instructs the application to delete a specified database record.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdDeleteRecord 67`

Parameters The launch code's parameter block pointer references a `ImportExportRecordParamsType` structure.

Comments Note that the record may be identified either by index or unique ID. See the `ImportExportRecordParamsType` structure documentation for details.

See Also `sysAppLaunchCmdExportRecord`, `sysAppLaunchCmdImportRecord`

sysAppLaunchCmdEventHook

Purpose Sent to an application to allow the application to process an event.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdEventHook 25`

Parameters The launch code's parameter block pointer references an `EventType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

sysAppLaunchCmdExportRecord

Purpose Generally sent to the PIM applications, this launch code instructs the application to export a specified database record.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExportRecord 65`

Parameters The launch code's parameter block pointer references a `ImportExportRecordParamsType` structure.

Comments In response to this launch code, the application should place a copy of the specified database record, properly formatted for export, in the location specified by the `vObjectH` field of the `ImportExportRecordParamsType` structure. Note that the record may be identified either by index or unique ID. See the `ImportExportRecordParamsType` structure documentation for details.

See Also `sysAppLaunchCmdDeleteRecord`, `sysAppLaunchCmdExportRecordGetCount`, `sysAppLaunchCmdImportRecord`

sysAppLaunchCmdExportRecordGetCount

Purpose Generally sent to the PIM applications, this launch code instructs the application to return the number of records in the application's database.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdExportRecordGetCount 64`

Parameters The launch code's parameter block pointer references a single 32-bit integer into which the record count is to be written.

sysAppLaunchCmdFailedAppNotify

Purpose Indicates a failure in an application that was just switched to.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdFailedAppNotify 24`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdFailedAppNotifyType` structure. This structure identifies the failed application and contains the error code returned from that application.

sysAppLaunchCmdFepPanelAddWord

Purpose Send this launch code to the FEP panel to add a word to the FEP user dictionary.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdFepPanelAddWord 87`

Parameters The launch code's parameter block pointer references a structure that indicates the word to be added. This structure is simply a pointer to the word to be added, followed by a `uint16_t` containing the length of the word, like this:

```
typedef struct {
   const char *wordP;
   uint16_t wordLen;
} SysAppLaunchCmdFepPanelAddWordType;
```

See Also `sysAppLaunchCmdLookupWord`

sysAppLaunchCmdFinalizeUI

Purpose Sent only to the root application of the Application process, this launch code instructs the application's startup code to de-initialize the process's UI.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdFinalizeUI 0x7ff7`

Parameters None.

See Also `sysAppLaunchCmdInitializeUI`

sysAppLaunchCmdFind

Purpose Used to implement the global find. When the user enters a text string in the Find dialog, the system sends this launch code to each application. The application should search for the string that the user entered and return any records matching the find request.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdFind 1`

Parameters The launch code's parameter block pointer references a `FindParamsType`structure.

Comments The system displays the results of the query in the Find results dialog. The system continues the search with each application until it has a full screen of matching records or until all of the applications on the device have had a chance to respond. If the screen is full, a Find More button appears at the bottom of the dialog. If the user clicks the Find More button, the search resumes. Applications can test whether the current find is a continuation of a previous one by checking the `continuation` field in the parameter block.

Most applications that use text records should support this launch code. When they receive it, they should search all records for matches to the find string and return all matches.

An application can also integrate the find operation in its own user interface and send the launch code to a particular application.

Applications that support this launch code should also support `sysAppLaunchCmdSaveData` and `sysAppLaunchCmdGoTo`.

sysAppLaunchCmdGoTo

Purpose Sent in conjunction with `sysAppLaunchCmdFind` or `sysAppLaunchCmdExgReceiveData` to allow users to actually inspect the record that the global find returned or that was received by the Exchange Manager.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdGoTo 2`

Parameters The launch code's parameter block pointer references a GoToParamsTypestructure.

Comments Applications should do most of the normal launch actions, then display the requested item. The application should continue running unless explicitly closed.

An application launched with this code does have access to global variables, static local variables, and code segments other than segment 0 (in multi-segment applications).

Applications that receive this launch code should test the `sysAppLaunchFlagNewGlobals` launch flag to see if they need to initialize global variables. `sysAppLaunchFlagNewGlobals` indicates that the system has just allocated your global variables.

For example:

```
case sysAppLaunchCmdGoTo:
   if (launchFlags & sysAppLaunchFlagNewGlobals)
      StartApplication();
```

Note that you shouldn't automatically initialize the global variables in response to this launch code. Test the launch flag first. Your application receives this launch code when the user selects a record in the global find results. If your application was the current application before the user selected the Find command, the launch flag is clear to indicate that your globals should not be re-initialized.

sysAppLaunchCmdGoToURL

Purpose Retrieve and display the specified URL.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdGoToURL 54`

Parameters The parameter block for this launch command is simply a pointer to a string containing the URL.

Comments The `ExgRequest()` function launches an application with this launch code if it cannot find an exchange library that is registered for the URL it has received. To receive the launch code, the application must first use `ExgRegisterDatatype()` to register for a URL scheme.

sysAppLaunchCmdHandleSyncCallApp

Purpose Sent by the Desktop Link server when `SyncCallRemoteModule()` is called from a conduit to request that the handheld application do some processing on the conduit's behalf.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdHandleSyncCallApp 18`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdHandleSyncCallAppType` structure.

Comments The `SysAppLaunchCmdHandleSyncCallApp` structure contains all of the information passed to `SyncCallRemoteModule()` on the desktop plus the fields needed to pass the result back to the desktop. Pass the results back to the conduit by calling `DlkControl()`. See that function's documentation for an example of how to handle this launch code.

sysAppLaunchCmdImportRecord

Purpose Generally sent to the PIM applications, this launch code presents the application with a record to be added to or updated in the application's database.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdImportRecord 66`

Parameters The launch code's parameter block pointer references a `ImportExportRecordParamsType` structure.

Comments If the `uniqueID` field of the `ImportExportRecordParamsType` structure contains the ID of an existing record, the imported record should replace the specified existing record. Otherwise, the imported record should be added to the application's database.

See Also `sysAppLaunchCmdDeleteRecord`, `sysAppLaunchCmdExportRecord`

sysAppLaunchCmdInitDatabase

Purpose Sent by the Desktop Link server in response to a request to create a database. It is sent to the application whose creator ID matches that of the requested database.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdInitDatabase 11`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdInitDatabaseType` structure.

Comments The most frequent occurrence of this is when a `'data'` database is being installed or restored from the desktop. In this case, HotSync creates a new database on the device and passes it to the application via a `sysAppLaunchCmdInitDatabase` command, so that the application can perform any required initialization. HotSync will then transfer the records from the desktop database to the device database.

When a Palm OS application crashes while a database is installed using HotSync, the reason may be that the application is not handling the `sysAppLaunchCmdInitDatabase` command properly. Be especially careful not to access global variables.

The system will create a database and pass it to the application for initialization. The application must perform any initialization required, then pass the database back to the system, unclosed.

IMPORTANT: The `sysAppLaunchCmdInitDatabase` launch code handler *must* leave the database handle (the *`dbP`* field in the `SysAppLaunchCmdInitDatabaseType` structure) open on return.

sysAppLaunchCmdInitializeUI

Purpose Sent only to the root application of the Application process, this launch code instructs the application's startup code to initialize the process's UI.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdInitializeUI 0x7ff8`

Parameters None.

See Also `sysAppLaunchCmdFinalizeUI`

sysAppLaunchCmdLookup

Purpose The system or an application sends this launch command to retrieve information from another application. In contrast to Find, there is a level of indirection; for example, this launch code could be used to retrieve the phone number based on input of a name.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdLookup 15`

Parameters The parameter block is defined by the application that supports this launch code. For an example, see the source code for the standard Palm OS Address Book.

Comments This functionality is currently supported by the standard Palm OS Address Book.

Applications that decide to handle this launch code must search their databases for the supplied string and perform the match operation specified in the launch code's parameter block.

If an application wants to allow its users to perform lookup in other applications, it has to send it properly, including all information necessary to perform the match. An example for this is in `Address.c` and `AppLaunchCmd.h`, which are included in your SDK.

sysAppLaunchCmdLookupWord

Purpose Send to the dictionary application to look a word up in the FEP dictionaries.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdLookupWord 88`

Parameters The launch code's parameter block pointer references a structure that indicates the word to be added. This structure is simply a pointer to the word to be added, followed by a `uint16_t` containing the length of the word, like this:

```
typedef struct {
   const char *wordP;
   uint16_t wordLen;
} SysAppLaunchCmdFepPanelAddWordType;
```

Comments The specified word is automatically entered into the word lookup form, and the results are displayed to the user.

See Also `sysAppLaunchCmdFepPanelAddWord`

sysAppLaunchCmdMoveRecord

Purpose Move a record from one position to another in an application's database.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdMoveRecord 68`

Parameters The launch code's parameter block pointer references an `ImportExportRecordParamsType` structure. Within this structure, the `index` field indicates the record to be moved, and the `destIndex` field indicates the new position for the record (both indexes are zero-based). The `uniqueID` field is updated if this launch code succeeds.

Comments If the application doesn't support this launch code, `dmErrInvalidParam` is returned.

See Also `sysAppLaunchCmdAddRecord`

sysAppLaunchCmdMultimediaEvent

Purpose

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdMultimediaEvent 63`

Parameters

sysAppLaunchCmdNormalLaunch

Purpose Launch an application.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdNormalLaunch 0`

Parameters None.

See Also `sysAppLaunchCmdPinletLaunch`, `sysAppLaunchCmdSlipLaunch`

sysAppLaunchCmdNotify

Purpose The system or an application sends this launch code to notify applications that an event has occurred.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdNotify 51`

Parameters The `SysNotifyParamType` structure declared in `NotifyMgr.h` defines the format of this launch code's parameter block. See its description in the "Notifications" chapter.

Comments The parameter block specifies the type of event that occurred, as well as other pertinent information. To learn which notifications are broadcast by the system, see the chapter titled "Notifications" on page 59.

sysAppLaunchCmdOpenDB

Purpose You can send this launch code to the Web Clipping Application Viewer application to launch the application and cause it to open and display a Palm™ query application stored on the device. This is the same mechanism that the Launcher uses to launch query applications.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdOpenDB 52`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdOpenDBType` structure.

sysAppLaunchCmdPanelCalledFromApp

Purpose Lets a preferences panel know whether it was switched to from the Preferences application or whether an application invoked it to make a change. The panel may be a preference panel owned by the application or a system preferences panel.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdPanelCalledFromApp 13`

Parameters None.

Comments

In conjunction with `sysAppLaunchCmdReturnFromPanel`, this launch code allows an application to let users change preferences without switching to the Preferences application. For example, for the calculator, you may launch the Formats preferences panel, set up a number format preference, then directly return to the calculator that then uses the new format.

Examples of these system panels that may handle this launch code are:

- Network panel (called from network applications)
- Modem panel (called if modem selection is necessary)

All preferences panels must handle this launch code. If a panel is launched with this command, it should:

- Display a Done button.
- *Not* display the panel-switching pop-up trigger used for navigation within the preferences application.

sysAppLaunchCmdPinletLaunch

Purpose

Sent to an application that is launched as a pinlet instead of `sysAppLaunchCmdNormalLaunch` in order to launch the application.

Declared In

`CmnLaunchCodes.h`

Prototype

```
#define sysAppLaunchCmdPinletLaunch 83
```

Parameters

None.

See Also

`sysAppLaunchCmdSlipLaunch`

sysAppLaunchCmdReturnFromPanel

Purpose

Informs an application that the user is done with a called preferences panel.

Declared In

`CmnLaunchCodes.h`

Prototype

```
#define sysAppLaunchCmdReturnFromPanel 14
```

Parameters

None.

Comments This launch code is used in conjunction with `sysAppLaunchCmdPanelCalledFromApp`. The system passes this launch code to the application when a previously-called preferences panel exists.

sysAppLaunchCmdRun68KApp

Purpose Sent to PACE in order to launch a 68K-based application.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdRun68KApp 0x7ffc`

Parameters The launch code's parameter block pointer references a structure that specifies the application to be run. This structure looks something like this:

```
typedef struct AppSwitchInfoType {
   DatabaseID dbID;
   DatabaseID agent;
   MemPtr cmdPBP;
   uint32_t cmdPBSize;
   uint16_t cmd;    // if (!dbID), doubles as error code
   uint16_t flags;  // hold launch flags fo the app
   uint16_t rsrcID;
   uint8_t padding[2];
} AppSwitchInfoType;
```

sysAppLaunchCmdSaveData

Purpose Instructs the application to save all current data. For example, before the system performs a global find, an application should save all data.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSaveData 10`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdSaveDataType` structure.

Comments Any application that supports the Find command and that can have buffered data should support this launch code. The system sends this launch code to the currently active application before it begins the search. The application receiving this launch code should respond by saving all buffered data so that the search is able to find matches in the text just entered.

sysAppLaunchCmdSlipLaunch

Purpose Sent to any application that is launched within a Slip at the time of launch.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSlipLaunch 82`

Parameters None.

Comments Applications so launched can only draw during update events.

See Also `sysAppLaunchCmdNormalLaunch`, `sysAppLaunchCmdPinletLaunch`

sysAppLaunchCmdSyncCallApplicationV10

Purpose Used by the Desktop Link Server's "call application" command.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSyncCallApplicationV10 12`

Parameters

sysAppLaunchCmdSyncNotify

Purpose Sent to applications to inform them that a HotSync operation has occurred.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSyncNotify 3`

Parameters None.

Comments This launch code is sent only to applications whose databases were changed during the HotSync operation, including when the application itself has been installed by HotSync. The record database(s) must have the same creator ID as the application in order for the system to know which application to send the launch code to.

This launch code provides a good opportunity to update, initialize, or validate the application's new data, such as resorting records, setting alarms, and so on.

Because applications only receive `sysAppLaunchCmdSyncNotify` when their databases are updated, this launch code is not a good place to perform any operation that must occur after every HotSync operation. Instead, you may register to receive the `sysNotifySyncFinishEvent` notification. This notification is sent at the end of a HotSync operation, and it is sent to all applications registered to receive it, whether the application's data changed or not. Note that there is also a `sysNotifySyncStartEvent` notification.

sysAppLaunchCmdSyncRequest

Purpose Sent to the HotSync application to request a HotSync. This launch code is equivalent to `sysAppLaunchCmdSyncRequestLocal`.

Declared In `CmnLaunchCodes.h`

Prototype

```
#define sysAppLaunchCmdSyncRequest
   sysAppLaunchCmdSyncRequestLocal
```

Parameters None.

sysAppLaunchCmdSyncRequestLocal

Purpose Sent to the HotSync application to request a "local" HotSync. A local HotSync occurs when the HotSync button is pressed.

Declared In `CmnLaunchCodes.h`

Prototype

```
#define sysAppLaunchCmdSyncRequestLocal 9
```

Parameters None.

See Also `sysAppLaunchCmdSyncRequest`, `sysAppLaunchCmdSyncRequestRemote`

sysAppLaunchCmdSyncRequestRemote

Purpose Sent to the HotSync application to request a "remote" HotSync. A remote HotSync occurs when the "Remote HotSync" button (`vchrHardCradle2`) is pressed.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSyncRequestRemote 17`

Parameters None.

See Also `sysAppLaunchCmdSyncRequest`, `sysAppLaunchCmdSyncRequestLocal`

sysAppLaunchCmdSystemLock

Purpose Sent to the system-internal security application to lock the device.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSystemLock 16`

Parameters None.

Comments As a rule, applications don't need to do respond to this launch code. If an application replaces the system-internal security application, it must handle this launch code.

sysAppLaunchCmdSystemReset

Purpose Respond to a soft or hard reset.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdSystemReset 5`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdSystemResetType` structure.

Comments Applications can respond to this launch code by performing initialization, indexing, or other setup that they need to do when the system is reset. For more information about resetting the device, see Chapter 7, "System Reset," in *Exploring Palm OS: System Management*.

NOTE: Your application will not receive this launch code unless the the `ARMAppLaunchPrefsResetNotification flag in` the application's launch preferences resource is set to `TRUE`. See the description of the Application Launch Preferences Resource—in particular, the `ALPF_FLAG_NOTIFY_RESET` flag—in *Palm OS Resource File Formats* for more information on setting this flag.

sysAppLaunchCmdTimeChange

Purpose Respond to a time change initiated by the user.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdTimeChange 4`

Parameters None.

Comments Applications that are dependent on the current time or date need to respond to this launch code. For example, an application that sets alarms may want to cancel an alarm or set a different one if the system time changes.

Applications should register to receive the `sysNotifyTimeChangeEvent` notification instead of responding to this launch code. The `sysAppLaunchCmdTimeChange` launch code is sent to all applications. The `sysNotifyTimeChangeEvent` notification is sent only to applications that have specifically registered to receive it, making it more efficient than `sysAppLaunchCmdTimeChange`.

sysAppLaunchCmdURLParams

Purpose Sent from the Web Clipping Application Viewer application to launch another application.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdURLParams 50`

Parameters The parameter block consists of a pointer to a special URL string, which the application must know how to parse. The string is the

URL used to launch the application and may contain encoded parameters.

Comments An application launched with this code may or may not have access to global variables, static local variables, and code segments other than segment 0 (in multi-segment applications). It depends on the URL that caused the Web Clipping Application Viewer to send this launch code. 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 best way to test if you have global variable access is to test the `sysAppLaunchFlagNewGlobals` launch flag sent with this launch code. If this is flag is set, then you have global variable access.

sysAppLaunchNppiNoUI

Purpose Sent to a network panel plug-in to launch it without UI, and load NetLib.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchNppiNoUI 55`

Parameters The launch code's parameter block pointer references a `uint16_t` that contains the network library reference number.

See Also `sysAppLaunchNppiUI`

sysAppLaunchNppiUI

Purpose Send to a network panel plug-in to launch it with UI.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchNppiUI 56`

Parameters None.

See Also `sysAppLaunchNppiNoUI`

sysAppLaunchPnpsPreLaunch

Purpose Pre-launch code for “plug-and-play” devices.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchPnpsPreLaunch 61`

Parameters The launch code’s parameter block pointer references a `SysAppLaunchCmdPnpsType` structure.

sysAppLaunchPreDelete

Purpose Sent to PalmSource-created applications before they're deleted.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchPreDelete 62`

Parameters None.

sysCncPluginLaunchCmdGetPlugins

Purpose Sent to Connection Manager plug-in modules to request plug-in descriptions.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysCncPluginLaunchCmdGetPlugins 81`

Parameters The launch code’s parameter block pointer references a `CncGetPluginsPBType` structure.

Comments **NOTE:** This launch code is intended for use by Connection Manager plug-ins only. Applications should not send nor respond to this launch code.

Upon receipt of this launch code, each plug-in should set the `CncGetPluginsPBType` structure’s `plugins` field so that it points to one or more contiguous `CncPlgDefinitionType` structures, each containing details about a particular plug-in. Set the `CncGetPluginsPBType` structure’s `n` field to the number of plug-ins structures being returned.

See Also `sysCncPluginLaunchCmdRegister`

sysCncPluginLaunchCmdRegister

Purpose The Connection Manager sends this launch code to each plug-in module after it has been registered with the Connection Manager. This launch code gives the plug-in a chance to initialize itself.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysCncPluginLaunchCmdRegister 79`

Parameters The launch code's parameter block pointer references a `CncRegisterPBType` structure.

Comments

NOTE: This launch code is intended for use by Connection Manager plug-ins only. Applications should not send nor respond to this launch code.

A Connection Manager plug-in must handle this launch code if it wants to define more objects than just plug-ins, such as interfaces, edges or profiles. Plug-ins have already been added to the Connection Manager database before this code is sent.

See Also `sysCncPluginLaunchCmdUnregister`

sysCncPluginLaunchCmdUnregister

Purpose Sent to Connection Manager plug-in modules when removing plug-ins

Declared In `CmnLaunchCodes.h`

Prototype `#define sysCncPluginLaunchCmdUnregister 80`

Parameters None.

Comments

NOTE: This launch code is intended for use by Connection Manager plug-ins only. Applications should not send nor respond to this launch code.

A Connection Manager plug-in must handle this launch code to free any data Connection Manager related. The Connection Manager has the charge to delete any profiles and relations that reference a deleted plug-in.

See Also `sysCncPluginLaunchCmdRegister`

sysCncWizardLaunchCmdEdit

Purpose The `CncProfileEdit()` function sends this launch code to the Connection Manager's configuration application in order to edit a connection profile.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysCncWizardLaunchCmdEdit 84`

Parameters

NOTE: This launch code is intended for use by Connection Manager plug-ins only. Applications should not send nor respond to this launch code.

The launch code's parameter block pointer references a `CncWizardEditPBType` structure.

sysDialLaunchCmdDial

Purpose Dials the modem (optionally displaying the dial progress), given the service ID and serial library reference number.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysDialLaunchCmdDial 30`

Parameters

See Also `sysDialLaunchCmdHangUp`

sysDialLaunchCmdHangUp

Purpose Hangs up the modem (optionally displaying the disconnect progress), given the service ID and serial library reference number.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysDialLaunchCmdHangUp 31`

Parameters

See Also `sysDialLaunchCmdDial`

sysIOSDriverInstall

Purpose Sent to a code module in the I/O process when it is installed.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysIOSDriverInstall 74`

Parameters The launch code's parameter block pointer references an `IOSDriverInstallType` structure.

Comments The code module typically initializes the driver in response to this launch code.

sysIOSDriverRemove

Purpose Sent to a code module in the I/O process when it is removed.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysIOSDriverRemove 75`

Parameters None.

sysLaunchCmdAppExited

Purpose Sent by the Application Manager to all loaded modules after an application exits from its `PilotMain()` function.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdAppExited 0x7ff9`

Parameters None.

sysLaunchCmdBoot

Purpose Sent to operating system initialization procedures at boot time, upon receipt of this launch code those procedures do whatever is necessary to initialize their component.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdBoot 70`

Parameters The launch code's parameter block pointer references a `AppInitProcParamsType` structure. This structure is private.

Comments The procedure receiving this launch code runs in the System process. Drivers typically install themselves at this time (using `IOSInstallDriver()`). Many other initialization procedures take this opportunity to register plug-ins (with `CncRegisterPluginModule()`).

sysLaunchCmdFinalize

Purpose Sent to all kinds of executable modules right before a module gets unloaded, this launch code gives your executable a last chance to release resources or do any other needed "de-initialization."

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdFinalize 0x7fff`

Parameters None.

See Also `sysLaunchCmdAppExited()`, `sysLaunchCmdInitialize()`

sysLaunchCmdGetGlobals

Purpose Sent to an executable module to retrieve a pointer to its global structure.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdGetGlobals 0x7ffa`

Parameters The launch code's parameter block pointer references a location into which the executable module should write either the location of its globals structure, if there are globals to export, or `NULL` if the executable doesn't export any globals.

Comments An executable module that wants to make all or part of its globals accessible by other modules can do this by putting those globals in a single C structure and returning the address of this global structure in the memory location pointed to by the launch code's parameter block pointer.

To prevent globals from being retrieved, simply return `NULL` in response to this launch code.

See Also "Exporting Globals" on page 73 of *Exploring Palm OS: System Management*.

sysLaunchCmdGetModuleID

Purpose Sent to an executable module to retrieve its module ID.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdGetModuleID 0x7ff5`

Parameters None.

sysLaunchCmdGraphicsAccelInit

Purpose Sent by mini-GL to the graphics accelerator, requesting that it initialize itself.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdGraphicsAccelInit 78`

Parameters The launch code's parameter block pointer references a private structure that contains the mini-GL context in which the graphics accelerator is running.

sysLaunchCmdInitialize

Purpose Sent to an executable module right after the module is loaded, this launch code gives your executable a chance to allocate resources or do any other needed initialization.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdInitialize 0x7ffe`

Parameters In some cases the launch code's parameter block pointer references a private structure that contains information about the heap. Often the parameter block pointer is set to `NULL`.

See Also `sysLaunchCmdFinalize()`

sysLaunchCmdInitRuntime

Purpose Sent to an executable module when it is loaded to initialize its module ID and linker stub.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdInitRuntime 0x7ff6`

Parameters The launch code's parameter block pointer references a structure that contains the module ID and a pointer to the module's linker stub. This structure looks like this:

```
struct {
   uint32_t id;
   void *linkerP;
} cmdPB
```

sysLibLaunchCmdGet68KSupportEntry

Purpose Sent to a shared library to determine if it can be called from a 68K application.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLibLaunchCmdGet68KSupportEntry 0x7ffd`

Parameters The launch code's parameter block pointer indicates the location to which the launch code handler should write the address of the shared library's main entry point.

Comments Shared libraries that can be called from 68K applications (via PACE) should respond to this launch code by returning (using the parameter block pointer) the address of the shared library's main entry point.

sysLaunchCmdProcessDestroyed

Purpose

Declared In `CmnLaunchCodes.h`

Prototype `#define sysLaunchCmdProcessDestroyed 0x7ff4`

Parameters

sysPackageLaunchAttachImage

Purpose Sent to a package when it is loaded in order to supply an image context used by the package to determine when the package should be unloaded.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysPackageLaunchAttachImage 71`

Parameters The launch code's parameter block pointer references a private structure that contains information about the image context.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

See Also `sysPackageLaunchGetInstantiate`

sysPackageLaunchGetInstantiate

Purpose Sent to a package when it is loaded in order to locate the function used to instantiate the package's components.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysPackageLaunchGetInstantiate 72`

Parameters The launch code's parameter block pointer indicates a private structure. One of this structure's fields is the location to which the launch code handler should write the component instantiation function's address.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

See Also `sysPackageLaunchAttachImage`

sysPinletLaunchCmdLoadProcPtrs

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

Declared In `CmnLaunchCodes.h`

Prototype `#define sysPinletLaunchCmdLoadProcPtrs 85`

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

sysSvcLaunchCmdGetQuickEditLabel

Purpose Get a "quick edit" label for one of the standard service panels. The standard service panels include the Network panel and the Dialer panel.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysSvcLaunchCmdGetQuickEditLabel 40`

Parameters The launch code's parameter block pointer references a private `SvcQuickEditLabelInfoType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

sysSvcLaunchCmdGetServiceID

Purpose Get a standard service panel's service ID. The standard service panels include the Network panel and the Dialer panel.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysSvcLaunchCmdGetServiceID 21`

Parameters The launch code's parameter block pointer references a `PrvNetSvcServiceIDType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

See Also `sysSvcLaunchCmdGetServiceInfo`, `sysSvcLaunchCmdSetServiceID`

sysSvcLaunchCmdGetServiceInfo

Purpose Obtain the name and service ID for a given system service.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysSvcLaunchCmdGetServiceInfo 23`

Parameters The launch code's parameter block pointer references a `ServiceInfo68KType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

sysSvcLaunchCmdGetServiceList

Purpose Obtain a list of system services.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysSvcLaunchCmdGetServiceList 22`

Parameters The launch code's parameter block pointer references a `ServiceList68KType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

sysSvcLaunchCmdSetServiceID

Purpose Set a standard service panel's service ID. The standard service panels include the Network panel and the Dialer panel.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysSvcLaunchCmdSetServiceID 20`

Parameters The launch code's parameter block pointer references a `PrvNetSvcServiceIDType` structure.

Comments This launch code is for internal use only. Applications should not send or respond to this launch code.

See Also `sysSvcLaunchCmdGetServiceID`

7

Event

This chapter provides reference documentation for the structures and functions that you use to manipulate events and event queues. This includes functions that allow you to create and communicate with threads in the background process.

The contents of this chapter are organized as follows:

The header file `Event.h` declares the API that this chapter describes. For reference documentation on some common system events, see Chapter 8, "Event Codes." For conceptual information on events and event queues, see Chapter 3, "Events and the Event Loop."

Event Structures and Types

EventType Struct

Purpose The `EventType` structure contains all of the data associated with a system event. All event types have some common data. Most events also have data specific to those events. The event-specific data uses a union that is part of the `EventType` data structure.

The common data is documented below the structure. Chapter 8, "Event Codes," documents each event and provides details on the important data associated with each type of event.

Declared In `Event.h`

Prototype

```
typedef struct EventType {
   eventsEnum eType;
   Boolean penDown;
   uint8_t padding_1;
   uint16_t padding_2;
   uint32_t tapCount;
   Coord screenX;
   Coord screenY;
   union {
      ...
   } data;
} EventType;
typedef EventType *EventPtr
```

Fields

`eType`

The type of the event. See "Event Codes Events" on page 171 for a complete list of events.

`penDown`

`true` if the pen was down at the time of the event, otherwise `false`.

`padding_1`

Padding bytes, for alignment purposes.

`padding_2`

Padding bytes, for alignment purposes.

`tapCount`

The number of taps received at this location. This value is used mainly by fields. When the user taps in a text field, two taps selects a word, and three taps selects the entire line.

`screenX`

Window-relative position of the pen in pixels (number of pixels from the left bound of the window).

`screenY`

Window-relative position of the pen in pixels (number of pixels from the top left of the window).

`data`

The specific data for an event, if any. The data is a union, and its exact contents depend on the *`eType`* field. See Chapter 8, "Event Codes," for more information on the event types and the structures that may accompany them.

EvtQueueHandle Typedef

Purpose A handle for a thread's event queue.

Declared In `Event.h`

Prototype

```
typedef void *EvtQueueHandle
```

See Also `EvtAddEventToEventQueue()`, `EvtCreateBackgroundThread()`, `EvtGetReplyEventQueue()`, `EvtGetThreadEventQueue()`

SysAppLaunchCmdBackgroundType Struct

Purpose Structure that accompanies a `sysAppLaunchCmdBackground` launch code and provides data supplied to the `EvtCreateBackgroundThread()` call that created the background thread.

Declared In `Event.h`

Prototype

```
typedef struct SysAppLaunchCmdBackgroundType {
   EvtQueueHandle callerQueue;
   MemPtr data;
   size_t dataSize;
} SysAppLaunchCmdBackgroundType
```

Fields

`callerQueue`

Event queue handle supplied when `EvtCreateBackgroundThread()` was called, or `NULL` if no handle was supplied. This handle is automatically released by the system when the thread terminates. This queue allows the background thread to post events back to the calling application.

`data`

Pointer to a data block supplied when `EvtCreateBackgroundThread()` was called, or `NULL` if no data block pointer was supplied. This pointer is automatically released by the system when the thread terminates.

`dataSize`

Size of the data block pointed to by the `data` field. This is the value supplied to the call to `EvtCreateBackgroundThread()` that created the background thread.

Event Constants

Event Flags Enum

Purpose Flags that accompany certain events.

Declared In `Event.h`

Constants `evtPenPressureFlag = 0x0001`

This flag is set in the `flags` field of the `penDownMove` structure that accompanies a `penDownEvent` or `penMoveEvent` if there is pen pressure information available. The pen pressure value, if available, can be found in the `penDownMove` structure’s `pressure` field.

Event Dispatch Types Enum

Purpose Values returned by your pen event filter function indicating how a given pen event should be handled.

Declared In `Event.h`

Prototype `typedef uint32_t EvtDispatchType;`

Constants `evtDispatchAbsorb = 0`

Deliver the event to the window and consume it.

`evtDispatchFallthrough = 1`

Ignore the pen event, and send it to the next window. Note that once you have allowed an event to fall through, you will not see this or any more events in the current motion delivered to your window.

Event Error Codes

Purpose Error codes returned by the various Event and System Event Manager functions (those defined here and those defined in `SysEvtMgr.h`).

Declared In `Event.h`

Constants `#define evtErrNoQueue (evtErrorClass | 5)`

The specified event queue does not exist.

`#define evtErrParamErr (evtErrorClass | 1)`
One of the specified parameters is invalid.

`#define evtErrQueueBusy (evtErrorClass | 4)`

`#define evtErrQueueEmpty (evtErrorClass | 3)`
There are no events in the specified event queue.

`#define evtErrQueueFull (evtErrorClass | 2)`
The event could not be added to the queue because the event queue is full.

Miscellaneous Event Constants

Purpose The header file `Event.h` defines these constants.

Declared In `Event.h`

Constants `#define evtNoWait 0`
A timeout value you can supply to `EvtGetEvent()` to cause it to return immediately if there are no events waiting in the event queue.

`#define evtWaitForever -1`
A timeout value you can supply to `EvtGetEvent()` to cause the CPU to go into doze mode until the user provides input.

```
#define virtualKeyMask (appEvtHookKeyMask |
  libEvtHookKeyMask | commandKeyMask)
```
Mask value used by the `EvtKeydownIsVirtual()` macro to determine if a given event is a virtual character key down event.

Event Launch Codes

sysAppLaunchCmdBackground

Purpose Sent to the executable module that is launched in a background thread with `EvtCreateBackgroundThread()`.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdBackground 73`

Parameters The launch code's parameter block pointer references a `SysAppLaunchCmdBackgroundType` structure.

Event Functions and Macros

EvtAcquireEventQueue Function

Purpose Acquires a reference on the given queue so that it won't become invalid until another `EvtReleaseEventQueue()` call is made on it.

Declared In `Event.h`

Prototype `void EvtAcquireEventQueue (EvtQueueHandle queue)`

Parameters → *queue*

Handle to the event queue.

Returns Nothing.

Comments This function is useful if, for example, you've gotten an `EvtQueueHandle` from somewhere and now want to give it to another thread. Call `EvtAcquireEventQueue()` and then pass the `EvtQueueHandle` to the other thread. When the other thread is done with the event queue, it should call `EvtReleaseEventQueue()`. Unlike `EvtGetThreadEventQueue()`, `EvtAcquireEventQueue()` allows you to pass a handle to any event queue, not just the one associated with your thread.

EvtAddEventToEventQueue Function

Purpose Send an event to a specific event queue.

Declared In `Event.h`

Prototype

```
status_t EvtAddEventToEventQueue
    (EvtQueueHandle queue, const EventType *event,
    EvtQueueHandle replyQueue)
```

Parameters

→ *`queue`*
Handle of the event queue to which the event is to be sent.

→ *`event`*
Pointer to the event structure representing the event to be sent.

→ *`replyQueue`*
Handle of the event queue to which a reply is to be sent, or `NULL` if the event handler doesn't need to generate an event in reply. The event handler can retrieve this value by calling `EvtGetReplyEventQueue()`.

Returns Returns `errNone` if the event was successfully added to the queue, or one of the following otherwise:

`bndErrorDead`
The process that was hosting the event queue has gone away.

`evtErrQueueFull`
The event queue is full, or the target thread associated with the queue has gone away.

`evtErrNoQueue`
`queue` is `NULL`.

Comments Events can be sent with the restriction that only the top-level contents of the event structure will be copied. The event structure cannot contain pointers to strings or other data or objects.

See Also `EvtCreateBackgroundThread()`, `EvtGetThreadEventQueue()`, `EvtLookupEventQueue()`

EvtAddEventToQueue Function

Purpose Add an event to the event queue.

Declared In `Event.h`

Prototype

```
status_t EvtAddEventToQueue
    (const EventType *event)
```

Parameters → *`event`*

Pointer to the structure that contains the event.

Returns Returns `errNone` if the event was successfully added to the event queue, or `evtErrQueueFull` otherwise.

Comments This function makes a copy of the structure that you pass in and adds it to the event queue.

EvtAddEventToQueueAtTime Function

Purpose Add an event to the event queue at a specified time.

Declared In `Event.h`

Prototype

```
status_t EvtAddEventToQueueAtTime
    (uint64_t absoluteTime,
    const EventType *event)
```

Parameters → *`absoluteTime`*

The time, in milliseconds. This value is the number of milliseconds since the device was last reset; you can get the current time by calling `SysGetRunTime()` and converting the resulting value to milliseconds with the `P_NS2MS()` macro.

→ *`event`*

Pointer to the structure that contains the event.

Returns Returns `errNone` if the event was successfully added to the event queue.

Comments This function makes a copy of the structure that you pass in and adds it to the event queue.

EvtAddUniqueEventToEventQueue Function

Purpose Add an event to a specific event queue, replacing one of the same type if it is found.

Declared In `Event.h`

Prototype

```
status_t EvtAddUniqueEventToEventQueue
    (EvtQueueHandle queue, const EventType *event,
    uint32_t userCookie, Boolean inPlace)
```

Parameters

→ *queue*
Handle of the event queue to which the event is to be sent.

→ *event*
Pointer to the structure that contains the event.

→ *userCookie*
Event identifier. If this value is 0, this function matches on the first existing event that has the given type. Otherwise, it matches only on an event that has both the given type and the identifier supplied in this parameter.

→ *inPlace*
If `true`, any existing event is replaced. If `false`, the existing event is deleted and a new event is added to end of queue.

Returns Returns `errNone` if the event was successfully added to the event queue, or `evtErrQueueFull` otherwise.

Comments This function looks in the specified event queue for an event of the same event type and *userCookie* (if specified). The function replaces it with the new event, if found.

If no existing event is found, the new event is copied to the specified queue.

If an existing event is found, the function proceeds as follows:

- If *inPlace* is `true`, the existing event is replaced with a copy of the new event.
- If *inPlace* is `false`, the existing event is removed and the new event is added to the end of the queue.

EvtAddUniqueEventToQueue Function

Purpose Add an event to the event queue, replacing one of the same type if it is found.

Declared In `Event.h`

Prototype

```
status_t EvtAddUniqueEventToQueue
   (const EventType *eventP, uint32_t userCookie,
   Boolean inPlace)
```

Parameters

→ *`eventP`*
Pointer to the structure that contains the event.

→ *`userCookie`*
Event identifier. If this value is 0, this function matches on the first existing event that has the given type. Otherwise, it matches only on an event that has both the given type and the identifier supplied in this parameter.

→ *`inPlace`*
If `true`, any existing event is replaced. If `false`, the existing event is deleted and a new event is added to end of queue.

Returns Returns `errNone` if the event was successfully added to the event queue, or `evtErrQueueFull` otherwise.

Comments This function looks in the event queue for an event of the same event type and *`userCookie`* (if specified). The routine replaces it with the new event, if found.

If no existing event is found, the new event is copied to the queue.

If an existing event is found, the routine proceeds as follows:

- If *`inPlace`* is `true`, the existing event is replaced with a copy of the new event.
- If *`inPlace`* is `false`, the existing event is removed and the new event is added to the end of the queue.

EvtAddUniqueEventToQueueAtTime Function

Purpose Add an event to the event queue at a specified time, replacing one of the same type if it is found.

Declared In `Event.h`

Prototype

```
status_t EvtAddUniqueEventToQueueAtTime
    (uint64_t absoluteTime,
    const EventType *eventP, uint32_t userCookie,
    Boolean inPlace)
```

Parameters

→ *`absoluteTime`*
The time, in milliseconds. This value is the number of milliseconds since the device was last reset; you can get the current time by calling `SysGetRunTime()` and converting the resulting value to milliseconds with the `P_NS2MS()` macro.

→ *`eventP`*
Pointer to the structure that contains the event.

→ *`userCookie`*
Event identifier. If this value is 0, this function matches on the first existing event that has the given type. Otherwise, it matches only on an event that has both the given type and the identifier supplied in this parameter.

→ *`inPlace`*
If `true`, any existing event is replaced. If `false`, the existing event is deleted and a new event is added to end of queue.

Returns Returns `errNone` if the event was successfully added to the event queue, or `evtErrQueueFull` otherwise.

Comments This function looks for an event in the event queue of the same event type and *`userCookie`* (if specified). The routine replaces it with the new event, if found.

If no existing event is found, the new event is copied to the queue.

If an existing event is found, the routine proceeds as follows:

- If *`inPlace`* is `true`, the existing event is replaced with a copy of the new event.

If *`inPlace`* is `false`, the existing event is removed and the new event is added to the end of the queue.

EvtCreateBackgroundThread Function

Purpose Create a thread in the background process and return a handle to the thread's event queue, through which you can communicate with the background thread.

Declared In `Event.h`

Prototype

```
EvtQueueHandle EvtCreateBackgroundThread
    (DatabaseID db, size_t stackSize,
    uint8_t priority, EvtQueueHandle callerQueue,
    MemPtr data, size_t dataSize)
```

Parameters

→ *`db`*
Unique identifier for the database containing the code to be executed in the background thread.

→ *`stackSize`*
Size, in bytes, to be allocated to the background thread's stack.

→ *`priority`*
The requested thread priority. See "Thread Priorities" on page 454 of *Exploring Palm OS: System Management* for constants that represent commonly-used thread priorities.

→ *`callerQueue`*
Event queue handle for the queue that the background thread is to use to communicate with the calling thread, or `NULL` if the background thread doesn't need to send events back to the calling thread. This is usually the calling application's event queue, which can be obtained by calling `EvtGetThreadEventQueue()`.

→ *`data`*
Pointer to a block of data that will be made accessible to the background thread. This pointer accompanies the `sysAppLaunchCmdBackground` launch code. The data block cannot be more than 3kb in size (approximately).

→ *`dataSize`*
Size, in bytes, of the data block pointed to by *`data`*. This value is passed to the code running in the background thread.

Returns Returns a handle to the background thread's event queue, or `NULL` if the background thread couldn't be started.

Comments Events can be sent to the background thread through the queue as in the local process, with the restriction that only top-level contents of the event structure will be copied. The event structure cannot contain pointers to strings or other data or objects.

The caller queue and data are propagated to the new thread through the launch code as described below. Supplying `NULL` for any of these is valid.

Be sure to call `EvtReleaseEventQueue()` when you are done with this queue (though only doing that will *not* make the thread go away). You'll need a handle to the caller queue; this means that you should *not* do something like this:

```
myHandle = EvtCreateBackgroundThread(...,
EvtGetThreadEventQueue(), ...);
```

The above function enters `PilotMain()` with the launch code `sysAppLaunchCmdBackground` and a `SysAppLaunchCmdBackgroundType` data structure.

NOTE: `EvtCreateBackgroundThread()` does not guarantee that the requested priority will be satisfied. A return value of `errNone` does not guarantee that the thread has been created at requested priority. Depending upon the context in which the function was called, the actual thread priority may be lower than what was requested.

See Also `SysThreadCreate()`

EvtDequeueKeyEvent Function

Purpose Obtain the next key event from the key queue.

Declared In `Event.h`

Prototype
```
status_t EvtDequeueKeyEvent (EventType *event,
   Boolean peek)
```

Parameters ← *`event`*

Pointer to an event structure that is filled in with the details of the next event on the key queue.

→ *peek*
If `false`, the key event is removed from the key queue. If `true`, it is left in the key queue.

Returns Returns `errNone` if the key queue contained at least one key event, or `evtErrQueueEmpty` if there are no key events in the key queue.

EvtDequeuePenPoint Function

Purpose Get the next pen point out of the pen queue. This function is called by recognizers.

Declared In `Event.h`

Prototype

```
status_t EvtDequeuePenPoint (PointType *retP)
```

Parameters

← *retP*
Return point.

Returns Always returns `errNone`.

Comments Called by a recognizer that wishes to extract the points of a stroke. Returns the point (-1, -1) at the end of a stroke.

Before calling this routine, you must call `EvtDequeuePenStrokeInfo()`.

EvtDequeuePenStrokeInfo Function

Purpose Initiate the extraction of a stroke from the pen queue.

Declared In `Event.h`

Prototype

```
status_t EvtDequeuePenStrokeInfo
   (PointType *startPtP, PointType *endPtP)
```

Parameters

← *startPtP*
Start point returned here.

← *endPtP*
End point returned here.

Returns Always returns `errNone`.

Comments This routine must be called before `EvtDequeuePenPoint()` is called.

Subsequent calls to `EvtDequeuePenPoint()` return points at the starting point in the stroke and including the end point. After the end point is returned, the next call to `EvtDequeuePenPoint()` returns the point -1, -1.

EvtEnqueueKey Function

Purpose Place keys into the key queue.

Declared In `Event.h`

Prototype
```
status_t EvtEnqueueKey (wchar32_t ascii,
   uint16_t keycode, uint16_t modifiers)
```

Parameters

→ *`ascii`*
Character code for the key.

→ *`keycode`*
Virtual key code of key. This is the `keyCode` field of the `keyDownEvent` and is currently unused.

→ *`modifiers`*
Modifiers for `keyDownEvent`.

Returns Returns `errNone` if successful, or `evtErrParamErr` if an error occurs.

Comments **IMPORTANT:** Make sure you pass a `wchar32_t` as the `ascii` parameter, not a `char`. If you pass a high-ASCII `char`, the compiler sign-extends it to be a 32-bit value, resulting in the wrong character being added to the key queue.

EvtEventAvail Function

Purpose Determine if an event is available.

Declared In `Event.h`

Prototype
```
Boolean EvtEventAvail (void)
```

Parameters None.

Returns Returns `true` if an event is available, `false` otherwise.

EvtEventToString Function

Purpose Creates a string representation of an event, for debugging purposes.

Declared In `Event.h`

Prototype

```
void EvtEventToString (EventType *event,
   char *str, uint32_t bufsize)
```

Parameters

→ *`event`*
The event for which a string representation is to be created.

← *`str`*
A string buffer into which the string representation is written.

→ *`bufsize`*
The size of the string buffer *`str`*.

Returns Nothing.

Comments The string representation includes the event type, an indication of whether the pen was touching the screen at the time the event was generated, the tap count, and the x and y coordinates identifying the pen location. It also includes event-specific information, if appropriate. You can use the string produced by this function to log events as an aid to debugging.

EvtFinishLastEvent Function

Purpose Indicate that you are done processing an event obtained before blocking on the IOS file descriptor for the event queue.

Declared In `Event.h`

Prototype `void EvtFinishLastEvent (void)`

Parameters None.

Returns Nothing.

Comments Normally you don’t need to call this function. `EvtGetEvent()` and `IOSPoll()` both call it for you.

See Also `EvtGetEventDescriptor()`

EvtFlushKeyQueue Function

Purpose Flush all keys out of the key queue.

Declared In `Event.h`

Prototype `status_t EvtFlushKeyQueue (void)`

Parameters None.

Returns Always returns `errNone`.

EvtFlushNextPenStroke Function

Purpose Flush the next stroke out of the pen queue.

Declared In `Event.h`

Prototype `status_t EvtFlushNextPenStroke (void)`

Parameters None.

Returns Always returns `errNone`.

Comments Called by recognizers that need only the start and end points of a stroke. If a stroke has already been partially dequeued (by `EvtDequeuePenStrokeInfo()`) this routine finishes the stroke dequeueing. Otherwise, this routine flushes the next stroke in the queue.

See Also `EvtDequeuePenPoint()`

EvtFlushPenQueue Function

Purpose Flush all points out of the pen queue.

Declared In `Event.h`

Prototype `status_t EvtFlushPenQueue (void)`

Parameters None.

Returns Always returns `errNone`.

EvtGetEvent Function

Purpose Return the next available event from the current thread's event queue.

Declared In `Event.h`

Prototype

```
void EvtGetEvent (EventType *event,
    int32_t timeout)
```

Parameters ← *`event`*

Pointer to the structure to hold the event returned.

→ *`timeout`*

Maximum number of ticks to wait before an event is returned (`evtWaitForever` means wait indefinitely, `evtNoWait` means don't wait at all).

Returns Nothing.

Comments Pass `evtWaitForever` as the timeout in most instances. When running on the device, this makes the CPU go into doze mode until the user provides input. For applications that do animation, pass a *`timeout`* value greater than or equal to zero (`evtNoWait` has a value of zero).

Note that a timeout value greater than or equal to zero is simply the *maximum* number of ticks which can elapse before `EvtGetEvent()` returns an event. If any other event—including a `nilEvent`—occurs before this time has elapsed, `EvtGetEvent()` returns that event. Otherwise, once the specified time has elapsed `EvtGetEvent()` generates and returns a `nilEvent`. If you supply a value of zero for the timeout parameter, `EvtGetEvent()` returns the event currently in the queue, or, if there aren't any events in the queue, it immediately generates and returns a `nilEvent`.

EvtGetEventDescriptor Function

Purpose Get an IOS file descriptor that you can block on until events arrive in your queue.

Declared In `Event.h`

Prototype `int32_t EvtGetEventDescriptor (void)`

Parameters None.

Returns Returns the IOS file descriptor for the event queue, or a value less than zero if an error occurred while obtaining the file descriptor.

Comments This function only works for the main UI thread. Outside of the main UI thread you should use the multithreading APIs to do I/O instead of multiplexing with an event thread.

Rather than making repeated calls to `EvtGetEvent()`, you can instead obtain an IOS file descriptor using this function and block on that file descriptor. When an event is posted to your event queue, your application will wake up and can then process the event. Note that when using this technique you must let the operating system know when you are done with the event. `IOSPoll()` does this for you, or you can make a call to `EvtFinishLastEvent()`.

On debug ROMs, this function displays a fatal alert if the calling thread is not the main UI thread.

EvtGetFocusWindow Function

Purpose Get a handle to the window that currently has the focus.

Declared In `Event.h`

Prototype `WinHandle EvtGetFocusWindow (void)`

Parameters None.

Returns Returns a handle to the last window that received a `winFocusGainedEvent`, or, if a `winFocusGainedEvent` has not been returned from `EvtGetEvent()` since the last `winFocusLostEvent`, returns `invalidWindowHandle` (this constant is defined in `Window.h`).

EvtGetPen Function

Purpose Return the current status of the pen.

Declared In `Event.h`

Prototype

```
status_t EvtGetPen (Coord *pScreenX,
   Coord *pScreenY, Boolean *pPenDown)
```

Parameters

← *pScreenX*
x location, in standard coordinates, relative to the draw window.

← *pScreenY*
y location, in standard coordinates, relative to the draw window.

← *pPenDown*
`true` or `false`, indicating whether or not the pen is currently touching the screen.

Returns Always returns `errNone`.

See Also `EvtGetPenNative()`

EvtGetPenNative Function

Purpose Get the current status of the pen using a window's active coordinate system.

Declared In `Event.h`

Prototype

```
status_t EvtGetPenNative (WinHandle winH,
   Coord *pScreenX, Coord *pScreenY,
   Boolean *pPenDown)
```

Parameters

→ *winH*
Handle to a valid window.

← *pScreenX*
x location, in active coordinates, relative to the window.

← *pScreenY*
y location, in active coordinates, relative to the window.

← *pPenDown*
`true` if the pen is down, `false` otherwise.

Returns Always returns `errNone`.

Comments This function is a variation on `EvtGetPen()`. `EvtGetPen()` returns a pen sample using the standard coordinate system, relative to the draw window, whereas `EvtGetPenNative()` returns a pen sample using the active coordinate system of *`winH`*, relative to the window origin. If the active coordinate system is high density, the returned pen sample uses high-density coordinates.

On a debug ROM this function displays an error if *`winH`* doesn't reference a valid window object.

EvtGetReplyEventQueue Function

Purpose Obtain the event queue through which you can post a reply to the event being processed.

Declared In `Event.h`

Prototype `EvtQueueHandle EvtGetReplyEventQueue (void)`

Parameters None.

Returns A handle to the reply event queue, or `NULL` if the event handler isn't expected to post a reply.

Comments Used by event handlers to post a response to the thread that sent the event. The reply queue handle is specified when the event is originally sent, as a parameter to `EvtAddEventToEventQueue()`.

You must call `EvtReleaseEventQueue()` when done with the queue returned by this function.

See Also `EvtGetThreadEventQueue()`, `EvtLookupEventQueue()`

EvtGetThreadEventQueue Function

Purpose Obtain a handle to the current thread's event queue.

Declared In `Event.h`

Prototype `EvtQueueHandle EvtGetThreadEventQueue (void)`

Parameters None.

Returns Returns a handle to the event queue.

Comments Given a handle to a thread's event queue, you can use `EvtAddEventToEventQueue()` to add events to the thread's event queue from any other thread in the process. When you are done with the thread's event queue, call `EvtReleaseEventQueue()` to allow the system to reclaim the queue's resources.

See Also `EvtGetReplyEventQueue()`, `EvtLookupEventQueue()`

EvtKeydownIsVirtual Macro

Purpose Determine if a given event is a virtual character key down event.

Declared In `Event.h`

Prototype `#define EvtKeydownIsVirtual (`*`eventP`*`)`

Parameters → *`eventP`*

Pointer to an `EventType` structure.

Returns Evaluates to `true` if the character is a letter in an alphabet or a numeric digit, `false` otherwise.

Comments The macro assumes that the caller has already determined the event is a `keyDownEvent`.

This macro is intended for use by the system.

EvtKeyQueueEmpty Function

Purpose Determine whether the key queue is currently empty.

Declared In `Event.h`

Prototype `Boolean EvtKeyQueueEmpty (void)`

Parameters None.

Returns Returns `true` if the key queue is currently empty, otherwise returns `false`.

EvtLookupEventQueue Function

Purpose Look up an event queue by name.

Declared In `Event.h`

Prototype

```
EvtQueueHandle EvtLookupEventQueue
    (const char *name)
```

Parameters → *name*

The name of the event queue, as published by `EvtPublishEventQueue()`.

Returns Returns a handle to the event queue if the named queue was found, or `NULL` if an event queue with the given name couldn't be located.

Comments You must call `EvtReleaseEventQueue()` when you are done with the queue returned by this function.

Published queues persist across application switches, but note that if the queue refers to a thread in the Application process, after an application switch that queue will be dead and errors will be returned if you attempt to use it.

See Also `EvtGetReplyEventQueue()`, `EvtGetThreadEventQueue()`, `EvtPublishEventQueue()`

EvtPublishEventQueue Function

Purpose Publish (or withdraw from publication) an event queue name, so that code executing in other threads can gain access to the queue simply by knowing the event queue name.

Declared In `Event.h`

Prototype

```
status_t EvtPublishEventQueue (const char *name,
    EvtQueueHandle queue)
```

Parameters → *name*

The name by which the event queue is to be known (or the name of the published event queue that is to be withdrawn from publication).

→ *queue*

The event queue's handle, if publishing, or `NULL` to withdraw an event queue from publication.

Returns Returns `errNone` if the operation was successfully completed, or an error value otherwise.

Comments The functionality provided by this function and by `EvtLookupEventQueue()` allows an application that operates in conjunction with a background thread to attach to its already running background thread whenever the application starts. For instance, a media player that uses a background thread to perform playback or recording operations could use this to reestablish communications with the background thread after the user has switched away from and then back to the media player UI application.

Event queues should use Java-style naming conventions. For example, "com.palmsource.someapp.myqueue".

Published queues persist across application switches, but note that if the queue refers to a thread in the Application process, after an application switch that queue will be dead and errors will be returned if you attempt to use it.

See Also `EvtLookupEventQueue()`

EvtReleaseEventQueue Function

Purpose Release a reference on an event queue.

Declared In `Event.h`

Prototype `void EvtReleaseEventQueue (EvtQueueHandle` *`queue`*`)`

Parameters → *`queue`*
Handle to the event queue to be released.

Returns Nothing.

Comments Call this function to release a reference on the queue. Once all references are gone—including the one implicitly held by the thread running the queue and from publishing the queue—the system will reclaim the queue's resources.

See Also `EvtCreateBackgroundThread()`, `EvtGetReplyEventQueue()`, `EvtGetThreadEventQueue()`, `EvtLookupEventQueue()`

EvtSetNullEventTick Function

Purpose Make sure that a `nilEvent` occurs in at least the specified amount of time.

Declared In `Event.h`

Prototype
```
Boolean EvtSetNullEventTick
    (int64_t milliseconds)
```

Parameters → *milliseconds*
Maximum amount of time, in milliseconds, that should elapse before a `nilEvent` is added to the queue.

Returns Returns `true` if timeout value changed, or `false` if it did not change.

EvtSetPenDispatchFunc Function

Purpose Set the pen event filter function for a given window.

Declared In `Event.h`

Prototype
```
extern status_t EvtSetPenDispatchFunc
    (WinHandle winHandle,
    EvtPenDispatchFunc penDispatch,
    void *userData)
```

Parameters → *winHandle*
Handle to the window for which the pen event filter function is being set.

→ *penDispatch*
Pointer to the filter function, which must have a prototype as defined by `EvtPenDispatchFunc()`. If this parameter is `NULL`, the default filter function (which always returns `evtDispatchAbsorb`) is used.

→ *userData*
Pointer that can be used to pass application-specific data to the pen event filter function. If the filter function requires no such data, pass `NULL` for this parameter.

Returns Always returns `errNone`. On a debug ROM, this function generates a fatal error if the supplied window handle is invalid.

Comments A pen event filter function is a function that you write that allows you to control which pen taps are passed on to your window's event queue, and which are passed on to other windows that may be beneath yours. Such "pen event filters" are used primarily by overlay pinlets, although they can be used by any window; they are not limited to use by pinlets.

EvtSysEventAvail Function

Purpose Return `true` if a low-level system event (such as a pen or key event) is available.

Declared In `Event.h`

Prototype `Boolean EvtSysEventAvail (Boolean` *`ignorePenUps`*`)`

Parameters → *`ignorePenUps`*

If `true`, this function ignores pen-up events when determining if there are any system events available.

Returns Returns `true` if a system event is available.

Comments Call `EvtEventAvail()` to determine whether high-level software events are available.

EvtWakeup Function

Purpose Force the Event Manager to wake up and send a `nilEvent` to the current application.

Declared In `Event.h`

Prototype `status_t EvtWakeup (void)`

Parameters None.

Returns Always returns `errNone`.

Comments Called by interrupt routines, like the Sound Manager and Alarm Manager.

See Also `EvtWakeupWithoutNilEvent()`

EvtWakeupWithoutNilEvent Function

Purpose Force the Event Manager to wake up without sending a `nilEvent` to the current application.

Declared In `Event.h`

Prototype `status_t EvtWakeupWithoutNilEvent (void)`

Parameters None.

Returns Always returns `errNone`.

Comments Called by interrupt routines.

See Also `EvtWakeup()`

Application-Defined Functions

EvtPenDispatchFunc Function

Purpose A callback function that allows you to control which pen taps are passed on to your window's event queue, and which are passed on to other windows that may be beneath yours. Such "pen event filters" are used primarily by overlay ("on screen input") pinlets, although they can be used by any window; they are not limited to use by pinlets. They can be used to implement windows with irregular shapes and more sophisticated effects.

Declared In `Event.h`

Prototype

```
typedef EvtDispatchType (*EvtPenDispatchFunc)
   (const EventType *penEvent,
   const RectangleType *nativeFrame,
   void *userData)
```

Parameters

→ *`penEvent`*
The pen event to be filtered.

→ *`nativeFrame`*
The target window's frame.

↔ *`userData`*
Pointer to an optional application-defined data block specified during the call to `EvtSetPenDispatchFunc()`.

Returns Return one of the values defined by the Event Dispatch Types enum to indicate whether the event should be absorbed or passed on to the next window layer.

Comments This function is called for each pen event delivered to the window, allowing you to decide what to do with the event. Return `evtDispatchAbsorb` for those events that should be placed on your window's event queue, or `evtDispatchFallthrough` for those that should "fall through" to the next window beneath. Note that once you have allowed an event to fall through, *any subsequent events in the current motion will not be delivered to your window.*

NOTE: This function is called from outside of the window's event thread. You cannot access any UI state from it.

A typical `EvtPenDispatchFunc()` implementation will usually do nothing more than check if the pen is in a certain region of the window (and possibly check some internal state of the pinlet) before returning the appropriate value.

It is important to understand that the dispatch function set here is called as part of the system's lower-level event dispatching mechanism, before the event is placed in the target window's event queue. This means:

- The function is called in a system thread, *not* in the window's event thread, and so it cannot access any of that thread's UI state. In particular it can't make any Window Manager or standard Event Manager calls. You can use the multi-threaded Event Manager functions to communicate from this thread to the UI thread, however. You can also count on the thread running in the same process as your window, so you can access common globals and the *`userData`* parameter can contain a pointer to a shared data structure on the local heap.
- When using this function you must have a good understanding of multithreading to correctly synchronize calls to the dispatch function with whatever is going on in the UI thread. Not properly taking care of multi-threaded issues can result in application crashes and other bad behavior.

- This dispatch function is called as part of the low-level system event dispatching, and thus should do as little possible to decide what to do with each event it is given.

This function provides very direct access to the operating system's event processing, and as such developers should be very careful when using it. Take care to call as few operating system functions as possible: avoid the Data Manager, any UI functions besides the multithreaded Event Manager functions, and any other high level functions such as those involving the status bar, the dynamic input area, and the like. While some of these functions may happen to work in current versions of the operating system, future versions of the system may not be able to support them.

8

Event Codes

The file `EventCodes.h` defines the Palm OS-generated events. This chapter documents that header file, and is organized as follows:

For information on the structures that accompany most events, and the functions that can be used to manipulate the event queue, see Chapter 7, "Event," on page 139. For conceptual information on events and the event queue, see Chapter 3, "Events and the Event Loop," on page 43.

Event Codes Structures and Types

eventsEnum Typedef

Purpose Defines a type that can be used to hold an event value.

Declared In `EventCodes.h`

Prototype `typedef uint32_t eventsEnum`

Comments See "Event Codes Events" on page 171 for the set of event values defined by Palm OS.

Event Codes Constants

Miscellaneous Event Codes Constants

Purpose In addition to the enum that defines the events themselves, the `EventCodes.h` header file declares these constants.

Declared In `EventCodes.h`

Constants `invalidEvent = 100`

An invalid event value, used for error checking. This event is not normally posted to the event queue.

`firstINetLibEvent = 0x1000`

Base value for Internet Library events.

`firstWebLibEvent = 0x1100`

Base value for Web Library events.

`firstUserEvent = 0x6000`

Base value for events generated by third-party applications. All events generated by Palm OS have a value less than `firstUserEvent`. Third-party application event values should fall in the range:
`firstUserEvent >=` *`n`* `>= lastUserEvent`

`lastUserEvent = 0x7FFF`

The maximum value that should be used for an event generated by a third-party application. Third-party application event values should fall in the range:
`firstUserEvent >=` *`n`* `>= lastUserEvent`

Event Codes Events

NOTE: The events documented in this section represent general events of interest to most Palm OS programmers. Other events declared in `EventCodes.h` are generated by, or handled by, specific portions of the system and thus are only of interest to developers working with the corresponding operating system features. Those events are documented in other books in the *Exploring Palm OS* series, as listed under "Palm OS-Generated Events" on page 54.

appStopEvent

Purpose Request for the current application to terminate.

Prototype There is no event-specific data associated with this event.

Comments When the system wants to launch a different application than the one currently running, the event manager sends this event to request the current application to terminate. In response, an application has to exit its event loop, close any open files and forms, and exit.

If an application doesn't respond to this event by exiting, the system can't start the other application.

nilEvent

Purpose Event that is sent by the Event Manager when there are no events in the event queue.

Prototype There is no event-specific data associated with this event.

Comments A `nilEvent` is useful for animation, polling, and similar situations.

The Event Manager sends this event when there are no events in the event queue. This can happen if the routine `EvtGetEvent()` is passed a time-out value (a value other than `evtWaitForever`). If `EvtGetEvent()` is unable to return an event in the specified time, it returns a `nilEvent`. Different Palm OS versions and different devices can send `nilEvents` under different circumstances, so you might receive a `nilEvent` even before the timeout has expired.

prgMakeCallback

Purpose This event is for use by the operating system only. Applications should not post or act upon this event

prgUpdateDialog

Purpose This event is for use by the operating system only. Applications should not post or act upon this event

9

Helper

This chapter describes the Helper API declared in the header files `Helper.h` and `HelperServiceClass.h`. The Helper API is used when an application broadcasts a `sysNotifyHelperEvent` to all interested parties. The broadcaster of the notification and the notification clients (called **helpers**) use the Helper APIs to communicate with each other. The chapter discusses the following topics:

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

For more information on using the Helper API, see the section "Helper Notifications" on page 66.

Helper Structures and Types

HelperNotifyActionCodeType Typedef

Purpose Contains an action code that specifies what action the broadcasting application is requesting. See "Action Codes" on page 178 for the set of defined action codes.

Declared In `Helper.h`

Prototype `typedef uint16_t HelperNotifyActionCodeType`

HelperNotifyEnumerateListType Struct

Purpose The `HelperNotifyEnumerateListType` provides the broadcaster of the helper notification with information about the services that the helper can provide. This structure is used as the `data` field of the `HelperNotifyEventType` structure when the action code is `kHelperNotifyActionCodeEnumerate`.

Declared In `Helper.h`

Prototype

```
typedef struct HelperNotifyEnumerateListTypeTag {
   struct HelperNotifyEnumerateListTypeTag *nextP;
   Char helperAppName[kHelperAppMaxNameSize];
   Char actionName[kHelperAppMaxActionNameSize];
   uint32_t helperAppID;
   uint32_t serviceClassID;
} HelperNotifyEnumerateListType
```

Fields

`nextP`
A pointer to the next element in the list or `NULL` to signal the end of the list.

`helperAppName`
A null-terminated string containing the name of the helper application, suitable for display in the user interface. If more than one application can perform the same service, this name can be displayed as one of the choices in a pop-up list. The name should not exceed `kHelperAppMaxNameSize` bytes in length.

`actionName`
A null-terminated string containing the name of the service that can be performed, suitable for display in the user interface. The action name should be short enough to display on a button, and should never exceed `kHelperAppMaxActionNameSize` bytes in length.

`helperAppID`
The helper's creator ID or any other ID that uniquely identifies the helper.

`serviceClassID`
The ID of the service that the helper performs. See "Helper Service Class IDs" on page 182.

Comments The helper allocates this structure and then adds it to the linked list of structures pointed to by `notifyDetailsP->data.enumerateP` in the `SysNotifyParamType` that is sent to the helper. The helper should allocate one structure per supported service.

Even though the helper allocates this structure, the helper is not responsible for freeing the structure. Instead, the application that broadcast the notification must free the structure.

HelperNotifyEventType Struct

Purpose The `HelperNotifyEventType` structure contains all data associated with a helper notification (`sysNotifyHelperEvent`). A pointer to this structure is passed as the `notifyDetailsP` field in the `SysNotifyParamType` for that notification.

Declared In `Helper.h`

Prototype

```
typedef struct HelperNotifyEventTypeTag {
   uint16_t version;
   HelperNotifyActionCodeType actionCode;
   union {
      struct HelperNotifyEnumerateListTypeTag
         *enumerateP;
      struct HelperNotifyValidateTypeTag
         *validateP;
      struct HelperNotifyExecuteTypeTag
         *executeP;
   } data;
} HelperNotifyEventType
```

Fields

`version`
The version number for this structure. The current version is `kHelperNotifyCurrentVersion`.

`actionCode`
The action that the helper application should perform. See "Action Codes" on page 178.

`data`
Data specific to the action code. See "Action Codes" on page 178.

Comments The `HelperNotifyEventType` structure specifies which action is to be performed and contains data necessary for that action. All actions have some common data. Actions also have data specific to that action. The specific data uses a union that is part of the `HelperNotifyEventType` structure.

HelperNotifyExecuteType Struct

Purpose The `HelperNotifyExecuteType` structure identifies the service to perform and contains the data necessary to perform that service. This structure is used as the `data` field of the `HelperNotifyEventType` structure when the action code is `kHelperNotifyActionCodeExecute`.

Declared In `Helper.h`

Prototype

```
typedef struct HelperNotifyExecuteTypeTag {
   uint32_t serviceClassID;
   uint32_t helperAppID;
   Char *dataP;
   Char *displayedName;
   void *detailsP;
   status_t err;
} HelperNotifyExecuteType
```

Fields

`serviceClassID`

The ID of the service to be performed. See "Helper Service Class IDs" on page 182.

`helperAppID`

The unique ID of the helper; a value of 0 indicates that any available helper for the specified service class should perform the service.

`dataP`

A null-terminated string specific to this service, such as a phone number for the dial service or an email address for the email service. See "Helper Service Class IDs" on page 182. Multiple fields must be separated by semicolons (;).

`displayedName`

A null-terminated string containing an optional, human-readable description of the string in `dataP`. For example, if dataP contains a phone number, this field might contain the name of the person at that number.

detailsP

A pointer to a data structure containing extra information that this service requires. See "Helper Service Class IDs" on page 182. If the service does not require extra information, this field is NULL.

err

An error code that indicates whether the service was performed successfully. If the service was successful, this field contains errNone, and the handled field in the notification data structure should be set to true.

HelperNotifyValidateType Struct

Purpose The HelperNotifyValidateType structure identifies a service that should be validated and the helper that should validate it. This structure is used as the data field of the HelperNotifyEventType structure when the action code is kHelperNotifyActionCodeValidate.

Declared In Helper.h

Prototype

```
typedef struct HelperNotifyValidateTypeTag {
   uint32_t serviceClassID;
   uint32_t helperAppID;
} HelperNotifyValidateType
```

Fields

serviceClassID

The ID of the service to be validated. See "Helper Service Class IDs" on page 182.

helperAppID

The creator ID of the helper application. 0 indicates that any available helper for the specified service should respond. If nonzero, only the helper with the matching creator ID should respond.

Comments The helper returns true in the handled field of the SysNotifyParamType structure to indicate that the service can be performed or false to indicate that the service cannot be performed.

Helper Constants

Action Codes

Purpose Codes that specify the action that a helper application is expected to take. The code is passed to each helper application registered to receive a `sysNotifyHelperEvent` notification as part of the `HelperNotifyEventType` structure that accompanies the notification.

Declared In `Helper.h`

Constants

```
#define kHelperNotifyActionCodeEnumerate
   ((HelperNotifyActionCodeType)1)
```

Send a list of available services. The `HelperNotifyEventType` structure's `data` field contains a `HelperNotifyEnumerateListType` structure.

```
#define kHelperNotifyActionCodeExecute
   ((HelperNotifyActionCodeType)3)
```

Perform the specified service. The `HelperNotifyEventType` structure's `data` field contains a `HelperNotifyExecuteType` structure.

```
#define kHelperNotifyActionCodeValidate
   ((HelperNotifyActionCodeType)2)
```

Perform the specified service. The `HelperNotifyEventType` structure's `data` field contains a `HelperNotifyValidateType` structure.

Miscellaneous Helper Constants

Purpose The `Helper.h` file also declares these constants.

Declared In `Helper.h`

Constants

```
#define kHelperAppMaxActionNameSize 48
```

The maximum length, in bytes, of a string containing the name of the service that can be performed, suitable for display in the user interface. This string is passed to broadcasting applications as part of the `HelperNotifyEnumerateListType` structure.

`#define kHelperAppMaxNameSize 72`

The maximum length, in bytes, of a string containing the name of the helper application, suitable for display in the user interface. This string is passed to broadcasting applications as part of the `HelperNotifyEnumerateListType` structure.

`#define kHelperNotifyCurrentVersion 1`

The version of the `HelperNotifyEventType` structure.

Helper Notifications

sysNotifyHelperEvent

Purpose Broadcast by applications to request a service from another application. For example, the Address Book application broadcasts this notification to request that the Dial application dial a phone number.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyHelperEvent 'hlpr'`

Parameters `notifyDetailsP` points to a `HelperNotifyEventType` structure.

Comments For the `sysNotifyHelperEvent`, the notification client (that is, the application or shared library that registers for the notification) is called a **helper**. The application that broadcasts this notification specifies one of the action codes listed under "Action Codes" on page 178. These action codes request all helper applications to enumerate (list the services they perform), validate (ensure that the service will succeed), and execute (perform the action). The helper responds to the notification by returning the required data in the appropriate portion of the `notifyDetailsP` structure and by setting the `handled` field to `true` or `false` to indicate the success or failure of the action.

10

Helper Service Class

This chapter documents those APIs that you employ when using the standard helper services included in Palm OS. The material in this chapter is divided up as follows:

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

For more information on using the Helper API, see the section "Helper Notifications" on page 66 and Chapter 9, "Helper."

Helper Service Class Structures and Types

HelperServiceEMailDetailsType Struct

Purpose Provides additional data for the email service. It is used as the `detailsP` field in the `HelperNotifyExecuteType` when the service class ID is `kHelperServiceClassIDEMail`.

Declared In `HelperServiceClass.h`

Prototype

```
typedef struct _HelperServiceEMailDetailsType {
   uint16_t version;
   Char *cc;
   Char *subject;
   Char *message;
} HelperServiceEMailDetailsType
```

Fields `version`

The version number for this structure. The current version is 1.

`cc`
A null-terminated string containing an email address that should be sent a carbon copy of the message. Multiple addresses are separated by a semi-colon (;). May be `NULL` if there are no email addresses to carbon copy.

`subject`
A null-terminated string containing the subject line. May be `NULL`.

`message`
Initial message body string or `NULL`.

HelperServiceSMSDetailsType Struct

Purpose The `HelperServiceSMSDetailsType` structure provides the SMS message to be sent. It is used as the `detailsP` field in the `HelperNotifyExecuteType` when the service class ID is `kHelperServiceClassIDSMS`.

Declared In `HelperServiceClass.h`

Prototype

```
typedef struct _HelperServiceSMSDetailsType {
   uint16_t version;
   Char *message;
} HelperServiceSMSDetailsType
```

Fields

`version`
The version number for this structure. The current version is 1.

`message`
A null-terminated string containing the body of the message to be sent, or `NULL`.

Helper Service Class Constants

Helper Service Class IDs

Purpose Identify the service that the helper performs. Pass one of these service class IDs within a `HelperNotifyValidateType` structure when validating the existence of a service, or within a

HelperNotifyExecuteType when requesting that the service be performed. When enumerating possible services, the returned HelperNotifyEnumerateListType structures contain service class IDs to identify the services that they perform.

Declared In `HelperServiceClass.h`

Constants

`#define kHelperServiceClassIDEMail 'mail'`
Send an email message. `dataP` points to the email address to which the message is to be sent, while `detailsP` points to a structure of type HelperServiceEMailDetailsType.

`#define kHelperServiceClassIDFax 'fax_'`
Send a fax. `dataP` points to the fax number to which the fax is to be sent, while `detailsP` is `NULL`.

`#define kHelperServiceClassIDSMS 'sms_'`
Send an SMS message. `dataP` points to the SMS mailbox number to which the message is to be sent, while `detailsP` points to a structure of type HelperServiceSMSDetailsType.

`#define kHelperServiceClassIDVoiceDial 'voic'`
Dial a phone number for a voice telephone call. `dataP` points to the telephone number to dial, while `detailsP` is `NULL`.

Comments Third party developers may define their own service classes. To do so, you must register a 32-bit identifier with PalmSource, Inc. on this web site:

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

Alternatively, you can use a creator ID that you already own.

11

Notification Manager

This chapter provides reference documentation for the Notification Manager APIs. These APIs include both the functions that you use to register and unregister to receive a notification and the functions you use to broadcast a notification, plus the definitions of many of the notifications themselves.

The contents of this chapter are organized as follows:

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

See Chapter 4, "Notifications," on page 59 for an introduction to notifications and their use. For a complete list of all notifications broadcast by Palm OS, see "Notification Summary" on page 72.

Notification Manager Structures and Types

SleepEventParamType Struct

Purpose Notification-specific data that accompanies a `sysNotifySleepRequestEvent` notification. This structure indicates why the device is going to sleep and allows a handler to prevent the device from sleeping.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SleepEventParamTag {
   uint16_t reason;
   uint16_t deferSleep;
} SleepEventParamType
```

Fields

`reason`
The reason the system is going to sleep. See "Reasons for Device Sleep" on page 197 for the set of possible values for this field.

`deferSleep`
Initially set to 0. If a notification handler wants to defer sleep, then it should increment this value. When *`deferSleep`* is greater than 0, the system waits before going to sleep.

SysNotifyAppLaunchOrQuitType Struct

Purpose Notification-specific data intended to accompany a `sysNotifyAppLaunchingEvent` notification. Because that notification is not sent in Palm OS Cobalt, however, this structure isn't used.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyAppLaunchOrQuitTag {
   uint32_t version;
   uint32_t dbID;
   uint16_t cardNo;
   uint16_t padding;
} SysNotifyAppLaunchOrQuitType
```

SysNotifyDBAddedType Typedef

Purpose Notification-specific data that accompanies a `sysNotifyDBAddedEvent` notification. This structure identifies the newly-added database.

Declared In `NotifyMgr.h`

Prototype

```
typedef SysNotifyDBCreatedType
SysNotifyDBAddedType;
```

Comments `SysNotifyDBAddedType` is equivalent to `SysNotifyDBCreatedType`. See that structure's documentation for details.

SysNotifyDBChangedType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDBChangedEvent` notification. This structure indicates what about the database has changed.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDBChangedTag {
   DatabaseID dbID;
   char name[dmDBNameLength];
   uint32_t creator;
   uint32_t type;
   uint16_t attributes;
   uint16_t version;
   uint32_t crDate;
   uint32_t modDate;
   uint32_t bckUpDate;
   uint32_t modNum;
   MemHandle appInfoH;
   MemHandle sortInfoH;
   char displayName[dmDBNameLength];
   uint16_t encoding;
   uint16_t fields;
   char oldName[dmDBNameLength];
   uint32_t oldCreator;
   uint32_t oldType;
   uint16_t oldAttributes;
   uint16_t reserved;
} SysNotifyDBChangedType
```

Fields

`dbID`
Database ID.

`name`
New name of database.

`creator`
New database creator ID.

`type`
New database type.

`attributes`
New database attributes.

`version`
: New database version.

`crDate`
: New database creation date.

`modDate`
: New database modification date.

`bckUpDate`
: New database backup date.

`modNum`
: New database modification number.

`appInfoH`
: New database application info block.

`sortInfoH`
: New database sort info block.

`displayName`
: New database display name.

`encoding`
: New database encoding.

`fields`
: Flags that indicate what about the database changed, and thus which of the above fields are set. See "Database Changed Flags" on page 198 for the set of flags that can be combined to make up this value.

`oldName`
: Name of database prior to the call to `DmSetDatabaseInfo`.

`oldCreator`
: Database creator ID prior to the call to `DmSetDatabaseInfo`.

`oldType`
: Database type prior to the call to `DmSetDatabaseInfo`.

`oldAttributes`
: Database attributes prior to the call to `DmSetDatabaseInfo`.

`reserved`
: Reserved for future use.

SysNotifyDBCreatedType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDBCreatedEvent` notification. This structure identifies the newly-created database.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDBCreatedTag {
   DatabaseID newDBID;
   char name[dmDBNameLength];
   uint32_t creator;
   uint32_t type;
   uint16_t attributes;
   uint16_t reserved;
} SysNotifyDBCreatedType
```

Fields

`newDBID`
Database ID of the newly-created database.

`name`
Database name.

`creator`
Database creator ID.

`type`
Database type.

`attributes`
Database attributes.

`reserved`
Reserved for future use.

SysNotifyDBDeletedType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDBDeletedEvent` notification. This structure identifies the newly-deleted database.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDBDeletedTag {
   DatabaseID oldDBID;
   char name[dmDBNameLength];
   uint32_t creator;
   uint32_t type;
   uint16_t attributes;
   uint16_t reserved;
} SysNotifyDBDeletedType
```

Fields

`oldDBID`
The database ID of the deleted database. This ID is no longer valid.

`name`
The name of the deleted database.

`creator`
The creator ID of the deleted database.

`type`
The type of the deleted database.

`attributes`
The deleted database's attributes.

`reserved`
Reserved for future use.

Comments

WARNING! The ID in `oldDBID` is invalid by the time the notification is broadcast. If you try to pass it to a Data Manager function, the system will crash.

SysNotifyDBDirtyType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDBDirtyEvent` notification. This structure identifies the newly-deleted database.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDBDirtyTag {
   DatabaseID dbID;
   char name[dmDBNameLength];
   uint32_t creator;
   uint32_t type;
   uint16_t attributes;
   uint16_t reserved;
} SysNotifyDBDirtyType
```

Fields

`dbID`
Database ID.

`name`
Database name.

`creator`
Database creator ID.

`type`
Database type.

`attributes`
Database attributes.

`reserved`
Reserved for future use.

SysNotifyDBInfoType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDeleteProtectedEvent` notification. This structure identifies the newly-deleted database.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDBInfoTag {
   MemHandle dbID;
   uint16_t cardNo;
   uint16_t attributes;
   char dbName[dmDBNameLength];
   uint32_t creator;
   uint32_t type;
} SysNotifyDBInfoType
```

Fields

`dbID`
Handle to the database to be deleted.

`cardNo`
The number of the card on which the database resides.

`attributes`
The database's attributes.

`dbName`
The name of the database to be deleted.

`creator`
The creator ID of the database to be deleted.

`type`
The type of the database to be deleted.

SysNotifyDisplayChangeDetailsType Struct

Purpose Notification-specific data that accompanies a `sysNotifyDisplayChangeEvent` notification. This structure contains the old and new display depths.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyDisplayChangeDetailsTag {
   uint32_t oldDepth;
   uint32_t newDepth;
} SysNotifyDisplayChangeDetailsType
```

Fields `oldDepth`

The old bit depth.

`newDepth`

The new bit depth.

SysNotifyLocaleChangedType Struct

Purpose Notification-specific data that accompanies a `sysNotifyLocaleChangedEvent` notification. This structure contains the old and new locales.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyLocaleChangedTag {
   LmLocaleType oldLocale;
   LmLocaleType newLocale;
} SysNotifyLocaleChangedType
```

Fields `oldLocale`

The old locale. See `LmLocaleType`.

`newLocale`

The new locale.

SysNotifyParamType Struct

Purpose Contains all of the data associated with a notification. This structure is passed as the parameter block for the `sysAppLaunchCmdNotify` launch code or as a parameter to the notification callback function.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyParamType {
   uint32_t notifyType;
   uint32_t broadcaster;
   void *notifyDetailsP;
   void *userDataP;
   Boolean handled;
   uint8_t reserved2;
   uint16_t padding;
} SysNotifyParamType
```

Fields

`notifyType`
The type of event that occurred. See Chapter 4, "Notifications," on page 59.

`broadcaster`
The creator ID of the application that broadcast the notification, or `sysNotifyBroadcasterCode` if the system broadcast the event.

`notifyDetailsP`
Pointer to data specific to this notification.

`userDataP`
Custom data that your notification handler requires. You create this data and pass it to `SysNotifyRegister()`.

`handled`
Set this field to `true` if the notification has been handled; set to `false` otherwise. In some cases, `handled` is treated as a bit field that notification handlers can use to indicate that certain conditions are true.

`reserved2`
Reserved for future use.

`padding`
Padding bytes.

Comments The `SysNotifyParamType` structure contains all of the data associated with a notification. This structure is passed as the parameter block for the `sysAppLaunchCmdNotify` launch code or as a parameter to the notification callback function. All notifications have some common data. Most notifications also have data specific to that notification. The specific data is pointed to by the `notifyDetailsP` field.

The common data for each notification is documented below the following structure declaration. Chapter 4, "Notifications," on page 59 section gives details on the important data associated with each type of notification.

SysNotifyPenStrokeType Struct

Purpose Notification-specific data that accompanies a `sysNotifyProcessPenStrokeEvent` notification. Because that notification is not sent in Palm OS Cobalt, however, this structure isn't used.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyPenStrokeTag {
   uint32_t version;
   PointType startPt;
   PointType endPt;
} SysNotifyPenStrokeType
```

SysNotifyVirtualCharHandlingType Struct

Purpose Notification-specific data that accompanies a `sysNotifyVirtualCharHandlingEvent` notification. Because that notification is not sent in Palm OS Cobalt, however, this structure isn't used.

Declared In `NotifyMgr.h`

Prototype

```
typedef struct SysNotifyVirtualCharHandlingTag {
   uint32_t version;
   struct _KeyDownEventType keyDown;
} SysNotifyVirtualCharHandlingType
```

Chapter 5, "Low-Level Events Reference,"Notification Manager Constants

Reasons for Device Sleep

Purpose These constants are part of the notification-specific data that accompanies a `sysNotifySleepRequestEvent` notification, and indicate why the device is going to sleep.

Declared In `NotifyMgr.h`

Constants

`#define sysSleepAutoOff 1`
The idle time limit has been reached.

`#define sysSleepPowerButton 0`
The user pressed the power off button.

`#define sysSleepResumed 2`
The sleep event was deferred by one of the notification handlers but has been resumed through the use of the `resumeSleepChr`.

`#define sysSleepUnknown 3`
Unknown reason.

Database Changed Flags

Purpose Flags that accompany a `sysNotifyDBChangedEvent` and indicate what about the database changed. Each flag corresponds to one of the fields in the `SysNotifyDBChangedType` structure.

Declared In `NotifyMgr.h`

Constants

`#define DBChangedFieldSetAppInfo 0x80`
New database application info block. `AppInfoH` contains a handle to the new application info block.

`#define DBChangedFieldSetAttributes 0x200`
New database attributes. `attributes` contains the new database attributes, while `oldAttributes` contains the database attributes as they were before the change.

`#define DBChangedFieldSetBckUpDate 0x20`
New database backup date. `bckUpDate` contains the new backup date.

`#define DBChangedFieldSetCrDate 0x8`
New database creation date. `crDate` contains the new creation date.

`#define DBChangedFieldSetCreator 0x2`
New database creator ID. `creator` contains the new creator ID, while `oldCreator` contains the creator ID as it was before the change.

`#define DBChangedFieldSetDisplayName 0x800`
New database display name. `displayName` contains the new display name.

`#define DBChangedFieldSetEncoding 0x1000`
New database encoding. `encoding` contains the new database encoding.

`#define DBChangedFieldSetModDate 0x10`
New database modification date. `modDate` contains the new modification date.

`#define DBChangedFieldSetModNum 0x40`
New database modification number. `modNum` contains the new modification number.

`#define DBChangedFieldSetName 0x1`
: New name of database. `name` contains the new database name, while `oldName` contains the database name as it was before the change.

`#define DBChangedFieldSetSortInfo 0x100`
: New database sort info block. `sortInfoH` contains a handle to the new sort info block.

`#define DBChangedFieldSetType 0x4`
: New database type. `type` contains the new database type, while `oldType` contains the database type as it was before the change.

`#define DBChangedFieldSetVersion 0x400`
: New database version. `version` contains the new database version.

Miscellaneous Notification Manager Constants

Purpose Miscellaneous constants defined by the Notification Manager.

Declared In `NotifyMgr.h`

Constants `#define sysNotifyBroadcasterCode sysFileCSystem`

`#define sysNotifyDefaultQueueSize 100`

`#define sysNotifyNoDatabaseH ((DatabaseID) 0xFFFFFFFF)`

`#define sysNotifyNormalPriority 0`
: Notification priority value used with `SysNotifyRegister()`. This value indicates “normal” priority.

`#define sysNotifyVersionNum 1`

Notification Manager Notifications

cncNotifyConnectionStateEvent

Purpose Broadcast by the Connection Manager whenever a persistent profile is either connected or disconnected.

Declared In `NotifyMgr.h`

Prototype `#define cncNotifyConnectionStateEvent 'cncc'`

Parameters The `notifyDetailsP` field isn't a pointer but instead is a `uint32_t` that has one of two values: `kCncConnectedState` if a persistent profile has been connected, or `kCncDisconnectedState` if a persistent profile has been disconnected.

See Also Chapter 4, "Notifications"

sysExternalConnectorAttachEvent

Purpose Broadcast when a USB cradle, RS-232 cradle or peripheral, a power cable, or a modem is attached to the universal connector.

Declared In `NotifyMgr.h`

Prototype `#define sysExternalConnectorAttachEvent 'ecna'`

Parameters The `notifyDetailsP` field points to a `uint16_t` that identifies which type of device was attached.

Compatibility This notification is broadcast only on devices that have the universal connector.

See Also `sysExternalConnectorDetachEvent`, Chapter 4, "Notifications"

sysExternalConnectorDetachEvent

Purpose Broadcast when a USB cradle, a RS-232 cradle or peripheral, a power cable, or a modem is detached from the universal connector.

Declared In `NotifyMgr.h`

Prototype `#define sysExternalConnectorDetachEvent 'ecnd'`

Parameters The `notifyDetailsP` field points to a `uint16_t` that identifies which type of device was detached.

Compatibility This notification is broadcast only on devices that have the universal connector.

See Also `sysExternalConnectorAttachEvent`, Chapter 4, "Notifications"

sysNotifyAltInputSystemDisabled

Purpose Broadcast by an alternative input system (such as an external keyboard) driver when it becomes disabled.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAltInputSystemDisabled 'aisd'`

Parameters

See Also `sysNotifyAltInputSystemEnabled`, Chapter 4, "Notifications"

sysNotifyAltInputSystemEnabled

Purpose Broadcast by an alternative input system (such as an external keyboard) driver when it becomes enabled.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAltInputSystemEnabled 'aise'`

Parameters

See Also `sysNotifyAltInputSystemDisabled`, Chapter 4, "Notifications"

sysNotifyAntennaRaisedEvent

Purpose Broadcast by `SysHandleEvent()` when the antenna is raised on a device that is so equipped.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAntennaRaisedEvent 'tena'`

Parameters None.

Comments Register for this notification if you want to handle the antenna key down event. To ensure that no other code handles the antenna key down event after yours, set the `handled` parameter of the `SysNotifyParamType` structure to `true`.

See Also Chapter 4, "Notifications"

sysNotifyAppServicesEvent

Purpose

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyAppServicesEvent 'apsv'`

Parameters None.

Comments

See Also Chapter 4, "Notifications"

sysNotifyCardInsertedEvent

Purpose Broadcast when an Expansion Manager card is inserted into a slot. When a new card is inserted, the Expansion Manager attempts to mount the volume on that card and plays a sound (indicating success or failure) once the attempt is complete.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyCardInsertedEvent 'crdi'`

Parameters `notifyDetailsP` points to a `uint16_t` containing the slot reference number.

Comments Most applications will want to register for sysNotifyVolumeMountedEvent instead of this notification. Register for sysNotifyCardInsertedEvent if you need to know when a card is inserted or if you want to prevent the Expansion Manager from performing its default handling of the notification.

To prevent the Expansion Manager from mounting the volume, set the expHandledVolume bit in the handled field. To prevent the Expansion Manager from playing the sound, set the expHandledSound bit in the handled field. For example:

```
cmdPBP->handled |= expHandledSound;
```

See Also sysNotifyCardRemovedEvent, sysNotifyVolumeMountedEvent, Chapter 4, "Notifications"

sysNotifyCardRemovedEvent

Purpose Broadcast when an Expansion Manager card is removed from a slot. When a card is removed, the Expansion Manager responds to this notification by playing a goodbye sound and then attempting to unmount the volume.

Declared In NotifyMgr.h

Prototype `#define sysNotifyCardRemovedEvent 'crdo'`

Parameters notifyDetailsP points to a uint16_t containing the slot reference number.

Comments Most applications will want to register for sysNotifyVolumeUnmountedEvent instead of this notification. Register for sysNotifyCardRemovedEvent if you need to know when a card is removed or if you want to prevent the Expansion Manager from performing its default handling of the notification.

To prevent the Expansion Manager from unmounting the volume, set the expHandledVolume bit in the handled field. To prevent the Expansion Manager from playing the sound, set the expHandledSound bit in the handled field. For example:

```
cmdPBP->handled |= expHandledSound;
```

See Also sysNotifyCardInsertedEvent, sysNotifyVolumeUnmountedEvent, Chapter 4, "Notifications"

sysNotifyDBAddedEvent

Purpose Broadcast when a new database has been added to the device.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDBAddedEvent 'dbs+'`

Parameters `notifyDetailsP` points to a `SysNotifyDBAddedType` structure.

Compatibility Palm OS Cobalt does not broadcast this notification.

See Also `sysNotifyDBChangedEvent`, `sysNotifyDBCreatedEvent`, `sysNotifyDBDeletedEvent`, `sysNotifySyncFinishEvent`, Chapter 4, “Notifications”

sysNotifyDBChangedEvent

Purpose The `sysNotifyDBChangedEvent` is broadcast sometime *after* database info is set with `DmSetDatabaseInfo()`.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDBChangedEvent 'dbmn'`

Parameters `notifyDetailsP` points to a `SysNotifyDBChangedType` structure. The contents of `fields` in this structure indicates what about the database changed, and thus which of the other structure fields contain valid data.

Comments Register for this notification if you keep an internal list of databases that needs to be updated when database info changes.

IMPORTANT: The `sysNotifyDB`*`xxx`*`Event` notifications are deferred notifications. So, for instance, if your application creates a database, opens it for write, and then renames it, all before `EvtGetEvent()` is called, the three corresponding notifications will all go out together. A `sysNotifyDBDirtyEvent` handler would fail if it tried to open the database, since the database will already have been renamed. You must be aware of the ramifications of a deferred notification when writing your notification handler.

See Also `sysNotifyDBAddedEvent`, `sysNotifyDBCreatedEvent`, `sysNotifyDBDeletedEvent`, `sysNotifyDBDirtyEvent`, `sysNotifySyncFinishEvent`, Chapter 4, "Notifications"

sysNotifyDBCreatedEvent

Purpose Broadcast sometime *after* a database is created with `DmCreateDatabase()`.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDBCreatedEvent 'dbcr'`

Parameters `notifyDetailsP` points to a `SysNotifyDBCreatedType` structure.

Comments Register for this notification if you keep an internal list of databases that needs to be updated when a new database is created.

IMPORTANT: The `sysNotifyDB`*`xxx`*`Event` notifications are deferred notifications. So, for instance, if your application creates a database, opens it for write, and then renames it, all before `EvtGetEvent()` is called, the three corresponding notifications will all go out together. A `sysNotifyDBDirtyEvent` handler would fail if it tried to open the database, since the database will already have been renamed. You must be aware of the ramifications of a deferred notification when writing your notification handler.

See Also `sysNotifyDBAddedEvent`, `sysNotifyDBChangedEvent`, `sysNotifyDBDeletedEvent`, `sysNotifySyncFinishEvent`, Chapter 4, "Notifications"

sysNotifyDBDeletedEvent

Purpose Broadcast sometime *after* a database is removed from the device.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDBDeletedEvent 'dbs-'`

Parameters `notifyDetailsP` points to a `SysNotifyDBDeletedType` structure.

Comments Register for this notification if you keep an internal list of databases that needs to be updated upon removal of a database. For example, the Attention Manager and Connection Manager register for this notification to maintain their internal lists of databases.

IMPORTANT: The `sysNotifyDB`*`xxx`*`Event` notifications are deferred notifications. So, for instance, if your application creates a database, opens it for write, and then renames it, all before `EvtGetEvent()` is called, the three corresponding notifications will all go out together. A `sysNotifyDBDirtyEvent` handler would fail if it tried to open the database, since the database will already have been renamed. You must be aware of the ramifications of a deferred notification when writing your notification handler.

See Also `sysNotifyDBAddedEvent`, `sysNotifyDBChangedEvent`, `sysNotifyDBCreatedEvent`, `sysNotifyDeleteProtectedEvent`, `sysNotifySyncFinishEvent`, Chapter 4, "Notifications"

sysNotifyDBDirtyEvent

Purpose Broadcast sometime *after* a database is opened for write or in some other way has been made modifiable. Note that the database may not have actually been modified yet.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDBDirtyEvent 'dbdr'`

Parameters `notifyDetailsP` points to a `SysNotifyDBDirtyType` structure.

Comments Register for this notification if you keep an internal list of databases that needs to be updated when a database becomes "dirty." For instance, upon reset the Launcher normally checks over such databases and updates its internal list.

IMPORTANT: The `sysNotifyDBxxxEvent` notifications are deferred notifications. So, for instance, if your application creates a database, opens it for write, and then renames it, all before `EvtGetEvent()` is called, the three corresponding notifications will all go out together. A `sysNotifyDBDirtyEvent` handler would fail if it tried to open the database, since the database will already have been renamed. You must be aware of the ramifications of a deferred notification when writing your notification handler.

See Also `sysNotifyDBChangedEvent`, Chapter 4, "Notifications"

sysNotifyDeleteProtectedEvent

Purpose Broadcast when the Launcher attempts to delete a database that has the protected flag set. The Launcher broadcasts the notification and then attempts to delete the database again. Any third party application that deletes databases should broadcast this notification as well.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDeleteProtectedEvent '-pdb'`

Parameters `notifyDetailsP` points to a `SysNotifyDBInfoType` structure.

Comments Register for this notification if you have a protected database but you still want to allow users to delete your application or other code resource if they choose. A notification handler should check the information in the `notifyDetailsP` struct to see if its database is the one being deleted. If so, it should respond to this notification to perform any necessary cleanup and to clear the protected flag. In this way, when the Launcher attempts to delete the database again, it will succeed. Note that if an application has multiple protected databases, this notification may be sent out more than once.

See Also `sysNotifyDBDeletedEvent`, `sysNotifySecuritySettingEvent`, Chapter 4, "Notifications"

sysNotifyDeviceUnlocked

Purpose Broadcast by the Security application when the user unlocks the device. The notification is broadcast immediately after the device has finished unlocking.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDeviceUnlocked 'unlk'`

Parameters None.

Comments If you display UI in response to the `sysNotifyLateWakeupEvent` notification, you should also register to receive the `sysNotifyDeviceUnlocked` notification. When a locked device receives the `sysNotifyLateWakeupEvent`, your UI should not be displayed if the device is waiting for the user to enter the password. The `sysNotifyDeviceUnlocked` notification is broadcast after the password is entered, which indicates that the user interface is ready.

See Also `sysNotifyForgotPasswordEvent`, `sysNotifySecuritySettingEvent`, Chapter 4, “Notifications”

sysNotifyDisplayChangeEvent

Purpose Broadcast whenever the display mode changes. Either the color table has been set to use a specific palette using the `WinPalette()` function or the bit depth has changed using the `WinScreenMode()` function.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyDisplayChangeEvent 'scrd'`

Parameters `notifyDetailsP` points to a `SysNotifyDisplayChangeDetailsType` structure.

Comments The `notifyDetailsP` field indicates how the bit depth changed. If the two values in the struct are equal, it means that the color palette has changed instead of the bit depth.

See Also Chapter 4, “Notifications”

sysNotifyEarlyWakeupEvent

Purpose Broadcast during `SysHandleEvent()` immediately after the system has finished sleeping.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyEarlyWakeupEvent 'worm'`

Parameters None.

Comments The screen may still be turned off, and the system may not 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.

IMPORTANT: This notification is *not* guaranteed to be broadcast. Thus, it is not suitable for applications where external hardware must be turned on when the system is powered on.

See Also `sysNotifyResetFinishedEvent`, `sysNotifySleepNotifyEvent`, `sysNotifySleepRequestEvent`, Chapter 4, "Notifications"

sysNotifyForgotPasswordEvent

Purpose Broadcast after the user taps the Lost Password button in the Security application. The notification is sent after the user has confirmed that all private records should be deleted but before the deletion actually occurs.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyForgotPasswordEvent 'bozo'`

Parameters None.

See Also `sysNotifyDeviceUnlocked`, `sysNotifySecuritySettingEvent`, Chapter 4, "Notifications"

sysNotifyHostFSInitDone

Purpose Broadcast by the Host File System library when the library has been initialized.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyHostFSInitDone 'hfid'`

Parameters None.

Comments This notification allows the AutoMounter to mount volumes on POSE slots.Applications should not register for this notification; it is intended for system use only.

See Also Chapter 4, “Notifications”

sysNotifyLateWakeupEvent

Purpose Broadcast during `SysHandleEvent()` immediately after the device has finished waking up.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyLateWakeupEvent 'lazy'`

Parameters None.

Comments This notification is sent at the late stage of wakeup, after the screen has been turned on. When this notification is broadcast, the system is guaranteed to fully wake up. Register for this notification if you need to perform startup tasks each time the system wakes up.

IMPORTANT: This notification is *not* guaranteed to be broadcast. Thus, it is unsuitable for applications where external hardware must be powered on when the device wakes up.

When the device receives this notification, it may be locked and waiting for the user to enter the password. If this is the case, you must wait for the user to unlock the device before you display a user interface. Therefore, if you intend to display a user interface when the device wakes up, you should make sure the device is not locked. If the device is locked, you should register for sysNotifyDeviceUnlocked notification and display your user interface when it is received.

Example The following code excerpt show how you might respond to this notification:

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

See Also sysNotifyEarlyWakeupEvent, sysNotifyResetFinishedEvent, sysNotifySleepNotifyEvent, sysNotifySleepRequestEvent, Chapter 4, “Notifications”

sysNotifyLocaleChangedEvent

Purpose Broadcast immediately after the system locale has changed. Currently, the user has the opportunity to change the locale only when the device first starts up and after a hard reset.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyLocaleChangedEvent 'locc'`

Parameters `notifyDetailsP` points to a `SysNotifyLocaleChangedType` structure.

Comments RAM-based applications and other code resources should obtain locale information by passing the `prefLocale` constant to `PrefGetPreference()`. They should not register for this notification. This notification is used by the built-in applications, which respond to it by rebuilding their default databases to use the newly selected language and character set.

See Also sysNotifyTimeChangeEvent, Chapter 4, “Notifications”

sysNotifyMenuCmdBarOpenEvent

Purpose Broadcast during `MenuHandleEvent()` when it is about to display the menu shortcut command bar.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyMenuCmdBarOpenEvent 'cbar'`

Parameters None.

Comments Register for this notification if you are writing a system extension (such as a “hack” installed with the HackMaster program) that needs to add a button to the menu command bar or to suppress the menu command bar. To add a button, call `MenuCmdBarAddButton()`. To suppress the command toolbar, set the `handled` field to `true`.

Applications that need to add their own buttons to the menu command bar should do so in response to a `menuCmdBarOpenEvent`. They should *not* register for this notification because an application should only add buttons if it is already the active application. The notification is sent after the event has been received, immediately before the command toolbar is displayed.

See Also Chapter 4, “Notifications”

sysNotifyPhoneEvent

Purpose Reserved for future use.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyPhoneEvent 'fone'`

Parameters

Compatibility This notification is not broadcast in Palm OS Cobalt.

See Also `telNotifyEnterCodeEvent`, `telNotifyErrorEvent`,Chapter 4, “Notifications”

sysNotifyPOSEMountEvent

Purpose Broadcast by the Host File System to communicate with itself.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyPOSEMountEvent 'pose'`

Parameters `notifyDetailsP` points to a `uint32_t` that contains the POSE slot number in its lower 16 bits and the `HostFSCustomControl()` function selector in its upper 16 bits.

Compatibility This notification is not broadcast in Palm OS Cobalt.

See Also Chapter 4, "Notifications"

sysNotifyResetFinishedEvent

Purpose Broadcast immediately after the system has finished a reset.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyResetFinishedEvent 'rstf'`

Parameters None.

Comments Because the notification registry is cleared upon a reset, only internal system components use this notification. Applications that need to be informed of a system reset can respond to the `sysAppLaunchCmdSystemReset` launch code.

See Also `sysNotifyEarlyWakeupEvent`, `sysNotifyLateWakeupEvent`, `sysNotifySyncFinishEvent`, Chapter 4, "Notifications"

sysNotifyRetryEnqueueKey

Purpose Broadcast at the top of the event loop if the Attention Manager has attempted to post a virtual character to the key queue and failed because the queue is full. The notification signals that the Attention

Manager is going to retry enqueuing the virtual character until it is successful.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyRetryEnqueueKey 'retk'`

Parameters `notifyDetailsP` points to a `wchar_t` containing the virtual character to be enqueued.

Comments Most applications do not need to register for this notification. It is used only by the Attention Manager to schedule retries of enqueuing the virtual character. When enqueueing a virtual character fails, the Attention Manager retries at the top of the event loop. It uses this notification to schedule retries so that they occur even if the user switches applications.

See Also Chapter 4, "Notifications"

sysNotifySecuritySettingEvent

Purpose Broadcast after the security level is successfully changed with a call to `SecSvcsSetDeviceSetting()`.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifySecuritySettingEvent 'ssch'`

Parameters None.

Comments This notification is not broadcast if the level isn't changed, either because the user doesn't have permission to change the level or the level value supplied to `SecSvcsSetDeviceSetting()` is invalid.

See Also `sysNotifyDeleteProtectedEvent`, `sysNotifyDeviceUnlocked`, `sysNotifyForgotPasswordEvent`, Chapter 4, "Notifications"

sysNotifySleepNotifyEvent

Purpose Broadcast during `SysHandleEvent()` immediately before the system is put to sleep. After the broadcast is complete, the system is put to sleep.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifySleepNotifyEvent 'slp!'`

Parameters None.

Comments Register for this notification if you have a small amount of cleanup that needs to be performed before the system goes to sleep. 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 this notification.

If your code is in the middle of a lengthy computation and needs to defer sleep, it should register for the `sysNotifySleepRequestEvent` notification instead.

IMPORTANT: This notification is *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.

See Also `sysNotifyEarlyWakeupEvent`, `sysNotifyLateWakeupEvent`, Chapter 4, “Notifications”

sysNotifySleepRequestEvent

Purpose Broadcast during `SysHandleEvent()` processing when the system has decided to go to sleep.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifySleepRequestEvent 'slpq'`

Parameters `notifyDetailsP` points to a `SleepEventParamType` structure.

Comments Register for this notification if you need to delay the system from going to sleep while your code performs a lengthy operation, such as disconnecting from the network. 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.

Note that you may receive this notification several times before the system goes to sleep because notification handlers can delay the system sleep and resume it later.

IMPORTANT: This notification is *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.

See Also `sysNotifyEarlyWakeupEvent`, `sysNotifyLateWakeupEvent`, `sysNotifySleepNotifyEvent`, Chapter 4, “Notifications”

sysNotifySyncFinishEvent

Purpose Broadcast immediately after a HotSync operation has completed. Register for this notification if you need to perform post-processing after HotSync operations.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifySyncFinishEvent 'sync'`

Parameters None.

See Also `sysNotifyDBAddedEvent`, `sysNotifyDBChangedEvent`, `sysNotifyDBDeletedEvent`, `sysNotifyResetFinishedEvent`, `sysNotifySyncStartEvent`, Chapter 4, "Notifications"

sysNotifySyncStartEvent

Purpose Broadcast immediately before a HotSync operation is begun. Register for this notification if you need to perform preprocessing before a HotSync operation.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifySyncStartEvent 'hots'`

Parameters None.

See Also `sysNotifySyncFinishEvent`, Chapter 4, "Notifications"

sysNotifyTimeChangeEvent

Purpose Broadcast just after the system time has been changed using `TimSetSeconds()`. Register for this notification if you need to know when the time has changed.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyTimeChangeEvent 'time'`

Parameters None.

See Also `sysNotifyLocaleChangedEvent`, Chapter 4, "Notifications"

sysNotifyVolumeMountedEvent

Purpose Broadcast when a Virtual File System Manager volume is mounted.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyVolumeMountedEvent 'volm'`

Parameters `notifyDetailsP` points to a `VFSSlotMountParamType` or `VFSPOSEMountParamType` structure.

Comments When a volume is mounted, the VFS Manager activates the `start.prc` application on the newly mounted volume and switches applications to the Launcher or to the `start.prc` application on that volume if it has a user interface.

Register for this notification if you need to know when a volume is mounted or if you want to prevent the default behavior of the VFS Manager.

To prevent the VFS Manager from activating the `start.prc` application, set the `vfsHandledStartPrc` bit in the `handled` field. To prevent the VFS Manager from switching applications, set the `vfsHandledUIAppSwitch` bit.

See Also `sysNotifyCardInsertedEvent`, `sysNotifyVolumeUnmountedEvent`, Chapter 4, “Notifications”

sysNotifyVolumeUnmountedEvent

Purpose Broadcast when a Virtual File System Manager volume is unmounted. Register for this notification if you need to know when a volume is unmounted.

Declared In `NotifyMgr.h`

Prototype `#define sysNotifyVolumeUnmountedEvent 'volu'`

Parameters `notifyDetailsP` contains the volume reference number.

See Also `sysNotifyCardRemovedEvent`, `sysNotifyVolumeMountedEvent`, Chapter 4, “Notifications”

Deprecated Notifications

Purpose These notifications, although declared in `NotifyMgr.h`, are not used in Palm OS Cobalt.

Declared In `NotifyMgr.h`

Constants `sysNotifyGotUsersAttention`

Notification Manager Functions and Macros

SysNotifyBroadcast Function

Purpose Synchronously send a notification to all applications registered for it.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyBroadcast
   (SysNotifyParamType *notify)
```

Parameters ↔ *`notify`*

Identifies the notification to be broadcast. See `SysNotifyParamType`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`sysNotifyErrBroadcastBusy`

The broadcast stack limit has already been reached.

`sysErrParamErr`

The background thread is broadcasting the notification and the *`notify`* parameter is `NULL`.

`sysNotifyErrNoStackSpace`

There is not enough space on the stack for the notification.

Comments When you call this function, the notification you specify is broadcast to all applications, shared libraries, and other code resources that have registered to receive that notification. The broadcast is performed synchronously, meaning that the system broadcasts the notification immediately and waits for each notification client to perform its notification handler and return before the

`SysNotifyBroadcast` call returns. This notification occurs in priority order.

The system allows nested notifications. That is, the recipient of a notification might broadcast a new notification, whose recipient might broadcast another new notification and so on. The constant `sysNotifyDefaultQueueSize` specifies how many levels of nested notification are allowed. If you reach this limit, the error `sysNotifyErrBroadcastBusy` is returned and your notification is not broadcast. To avoid reaching the limit, use `SysNotifyBroadcastDeferred()` instead of `SysNotifyBroadcast()` in your notification handlers. This ensures that the notification will not be broadcast until the top of the event loop.

WARNING! Do not call `SysNotifyBroadcast()` from outside of the main UI thread. Use `SysNotifyBroadcastDeferred()` instead.

WARNING! This function is not secure. it dispatches the notifications all in the local process by loading the PRCs being executed. Secure applications should always use `SysNotifyBroadcastDeferred()`.

See Also `SysNotifyRegister()`

SysNotifyBroadcastDeferred Function

Purpose Enqueue a notification for later broadcast.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyBroadcastDeferred
    (SysNotifyParamType *notify,
    uint32_t paramSize)
```

Parameters

↔ *notify*
The notification to enqueue. See `SysNotifyParamType`.

→ *paramSize*
Size of the data pointed to by the field `notify->notifyDetailsP`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrNotEnoughSpace`
There is not enough memory to allocate a new notification entry in the queue.

`sysErrParamErr`
paramSize is a negative number.

`sysNotifyErrQueueFull`
The queue has reached its maximum number of entries.

Comments This function is similar to `SysNotifyBroadcast()` except that the broadcast does not take place until the top of the event loop (specifically, the next time `EvtGetEvent()` is called). The system copies the *notify* structure to a new memory chunk, which is disposed of upon completion of the broadcast. (The *paramSize* value is used when copying the `notifyDetailsP` portion of the *notify* structure.)

See Also `SysNotifyRegister()`

SysNotifyRegister Function

Purpose Register to receive a notification.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyRegister (DatabaseID database,
   uint32_t notifyType,
   SysNotifyProcPtr callback, int32_t priority,
   void *userData, uint32_t userDataSize)
```

Parameters → *database*
ID of the database containing the application or code resource that is to receive the notification.

→ *notifyType*
The notification that the application wants to receive. See Chapter 4, “Notifications,” on page 59.

→ *`callback`*

Set to `NULL` to receive the notification as an application launch code. Note that you can only request a call with a callback from the main UI thread; use `SysNotifyRegisterBackground()` to register to receive a notification in the background thread.

→ *`priority`*

The priority with which the application should receive the event. Most applications and other code resources should always use `sysNotifyNormalPriority`. In rare circumstances, you may need to ensure that your code is notified toward the beginning or toward the end of the notification sequence. If so, be sure to leave some space so that your code won't collide with the system's handling of notifications or with another application's handling of notifications. In general, PalmSource recommends using a value whose least significant bits are 0 (such as 32, 64, 96, and so on). The smaller the priority, the earlier your code is notified.

→ *`userData`*

Caller-defined data to pass to the notification handler.

→ *`userDataSize`*

Size, in bytes, of the data block pointed to by *`userData`*.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`sysErrParamErr`

The specified database ID is `NULL`.

`sysNotifyErrDuplicateEntry`

This application is already registered to receive this notification.

`sysNotifyErrNoServer`

The notification server could not be contacted.

Comments Call this function when your code should receive a notification that a specific event has occurred or is about to occur. See Chapter 4, "Notifications," on page 59 for a list of the possible notifications. Once you register for a notification, you remain registered to receive it until a system reset occurs or until you explicitly unregister using `SysNotifyUnregister()`.

Pass `NULL` for the *`callbackP`* parameter so that the system will notify your application by sending it the `sysAppLaunchCmdNotify` launch code. This launch code's parameter block points to a `SysNotifyParamType` structure containing details about the notification.

The notification handler may perform any processing necessary. As with most launch codes, it's not possible to access global variables. If the handler needs access to any particular value to respond to the notification, pass a pointer to that value in the *`userDataP`* parameter. The system passes this pointer back to your code in the launch code's parameter block.

The notification handler may unregister for this notification or register for other notifications. It may also broadcast another notifications; however, it's recommended that you use `SysNotifyBroadcastDeferred()` to do this so as not to overflow the broadcast stack.

You may display a user interface in your notification handler; however, you should be careful when you do so. Many of the notifications are broadcast during `SysHandleEvent()`, which means your application event loop might not have progressed to the point where it is possible for you to display a user interface, or you might overflow the stack by displaying a user interface at this stage. See Chapter 4, "Notifications," on page 59 to learn which notifications are broadcast during `SysHandleEvent()`.

See Also `SysNotifyBroadcast()`, `SysNotifyBroadcastDeferred()`, `SysNotifyRegisterBackground()`, `SysNotifyUnregister()`

SysNotifyRegisterBackground Function

Purpose Register to receive a notification in the background thread.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyRegisterBackground
    (DatabaseID database, uint32_t notifyType,
    int32_t priority, void *userData,
    uint32_t userDataSize)
```

Parameters

→ *`database`*
ID of the database containing the application or code resource that is to receive the notification.

→ *`notifyType`*
The notification that the application wants to receive. See Chapter 4, "Notifications," on page 59.

→ *`priority`*
The priority with which the application should receive the event. Most applications and other code resources should always use `sysNotifyNormalPriority`. In rare circumstances, you may need to ensure that your code is notified toward the beginning or toward the end of the notification sequence. If so, be sure to leave some space so that your code won't collide with the system's handling of notifications or with another application's handling of notifications. In general, PalmSource recommends using a value whose least significant bits are 0 (such as 32, 64, 96, and so on). The smaller the priority, the earlier your code is notified.

→ *`userData`*
Caller-defined data to pass to the notification handler.

→ *`userDataSize`*
The size, in bytes, of the data block pointed to by *`userData`*.

Returns

Comments This function allows you to register for a notification to be received in the background thread. When the notification occurs, the Notification Manager loads the PRC indicated by *`database`* into the background process and performs the sublaunch there.

NOTE: Background notifications happen completely independently of the main applications, so these notifications are often received out-of-order with the notifications delivered to the main UI thread.

See Also `SysNotifyBroadcast()`, `SysNotifyBroadcastDeferred()`, `SysNotifyRegister()`, `SysNotifyUnregister()`

SysNotifyRegisterV40 Function

Purpose Register to receive a notification.

NOTE: This function is provided for compatibility purposes only; applications should use `SysNotifyRegister()` instead.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyRegisterV40 (uint16_t cardNo,
   LocalID dbID, uint32_t notifyType,
   SysNotifyProcPtr callback, int8_t priority,
   void *userData)
```

Parameters

→ *`cardNo`*
Number of the storage card on which the application or code resource resides.

→ *`dbID`*
Local ID of the application or code resource.

→ *`notifyType`*
The notification that the application wants to receive. See Chapter 4, "Notifications," on page 59.

→ *`callback`*
Set to `NULL` to receive the notification as an application launch code, or pass a pointer to a function that should be called when the notification is broadcast. See `SysNotifyProcPtr()`.

→ *`priority`*
The priority with which the application should receive the event. Most applications and other code resources should always use `sysNotifyNormalPriority`. In rare

circumstances, you may need to ensure that your code is notified toward the beginning or toward the end of the notification sequence. If so, be sure to leave some space so that your code won't collide with the system's handling of notifications or with another application's handling of notifications. In general, PalmSource recommends using a value whose least significant bits are 0 (such as 32, 64, 96, and so on). The smaller the priority, the earlier your code is notified.

→ *`userData`*
Caller-defined data to pass to the notification handler.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrCardNotPresent`
The *`cardNo`* parameter is non-zero.

`sysErrParamErr`
The specified database ID is `NULL`.

`sysNotifyErrDuplicateEntry`
This application is already registered to receive this notification.

`sysNotifyErrNoServer`
The notification server could not be contacted.

Comments Call this function when your code should receive a notification that a specific event has occurred or is about to occur. See Chapter 4, "Notifications," on page 59 for a list of the possible notifications. Once you register for a notification, you remain registered to receive it until a system reset occurs or until you explicitly unregister using `SysNotifyUnregister()`.

If your code is running in the main UI thread, you can register a function to be called when the notification is broadcast. In your call to `SysNotifyRegisterV40()` pass a pointer to a callback function in *`callbackP`*. This callback should follow the prototype shown in `SysNotifyProcPtr()`. Note that you should always supply a card number and database ID to `SysNotifyRegisterV40()`, even if you specify a callback function.

IMPORTANT: Because the *callbackP* 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 loaded in the applicatoin process 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.

If the code registering for notification isn't in the main UI thread, or if you don't have a function that can be called directly, pass NULL as the callbackP parameter. In this case, the system notifies your application by sending it the sysAppLaunchCmdNotify launch code. This launch code's parameter block points to a SysNotifyParamType structure containing details about the notification.

Whether the notification handler is responding to sysAppLaunchCmdNotify or uses the callback function, the notification handler may perform any processing necessary. As with most launch codes, it's not possible to access global variables. If the handler needs access to any particular value to respond to the notification, pass a pointer to that value in the *userDataP* parameter. The system passes this pointer back to your application or callback function in the launch code's parameter block.

The notification handler may unregister for this notification or register for other notifications. It may also broadcast another notifications; however, it's recommended that you use SysNotifyBroadcastDeferred() to do this so as not to overflow the broadcast stack.

You may display a user interface in your notification handler; however, you should be careful when you do so. Many of the notifications are broadcast during SysHandleEvent(), which means your application event loop might not have progressed to the point where it is possible for you to display a user interface, or you might overflow the stack by displaying a user interface at this stage.

See Chapter 4, "Notifications," on page 59 to learn which notifications are broadcast during `SysHandleEvent()`.

Compatibility This function is provided for compatibility purposes only; applications should use `SysNotifyRegister()`—which omits the obsolete *`cardNo`* parameter, identifies the database containing the application or code resource using a `DatabaseID`, and adds the *`userDataSize`* parameter—instead.

See Also `SysNotifyBroadcast()`, `SysNotifyBroadcastDeferred()`, `SysNotifyRegister()`, `SysNotifyUnregisterV40()`

SysNotifyUnregister Function

Purpose Cancel notification of the given event.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyUnregister
    (DatabaseID database, uint32_t notifyType,
    int32_t priority)
```

Parameters

→ *`database`*
Database ID of the database containing the application or code resource that is receiving the notification.

→ *`notifyType`*
The notification for which to unregister. See Chapter 4, "Notifications," on page 59..

→ *`priority`*
The priority value you passed when calling `SysNotifyRegister()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`sysNotifyErrEntryNotFound`
The application wasn't registered to receive this notification.

`sysNotifyErrNoServer`
The notification server could not be contacted.

Comments Use this function to remove your code from the list of those that receive notifications about a particular event. This function is particularly necessary if you use a notification callback; when the resource containing the callback is unloaded, it must unregister for all of its notifications, or the system will crash when the notification is broadcast.

See Also `SysNotifyRegister()`

SysNotifyUnregisterV40 Function

Purpose Cancel notification of the given event.

> **NOTE:** This function is provided for compatibility purposes only; applications should use `SysNotifyUnregister()` instead.

Declared In `NotifyMgr.h`

Prototype

```
status_t SysNotifyUnregisterV40 (uint16_t cardNo,
   LocalID dbID, uint32_t notifyType,
   int8_t priority)
```

Parameters

→ *`cardNo`*
Number of the storage card on which the application or code resource resides.

→ *`dbID`*
Local ID of the application or code resource.

→ *`notifyType`*
The notification for which to unregister. See Chapter 4, "Notifications," on page 59.

→ *`priority`*
The priority value you passed when calling `SysNotifyRegisterV40()`.

Returns Returns `errNone` if the operation completed successfully, or one of the following otherwise:

`memErrCardNotPresent`
The *`cardNo`* parameter is non-zero.

`sysNotifyErrEntryNotFound`
The application wasn't registered to receive this notification.

`sysNotifyErrNoServer`
The notification server could not be contacted.

Comments Use this function to remove your code from the list of those that receive notifications about a particular event. This function is particularly necessary if you are writing a shared library or a separately loaded code resource that receives notifications. When the resource is unloaded, it must unregister for all of its notifications, or the system will crash when the notification is broadcast.

Compatibility This function is provided for compatibility purposes only; applications should use `SysNotifyUnregister()`—which omits the obsolete *`cardNo`* parameter and identifies the database containing the application or code resource using a `DatabaseID`—instead.

See Also `SysNotifyRegisterV40()`

Application-Defined Functions

SysNotifyProcPtr Function

Purpose Notification-handling callback function prototype.

Declared In `NotifyMgr.h`

Prototype

```
status_t (*SysNotifyProcPtr)
   (SysNotifyParamType *notifyParamsP)
```

Parameters

→ *`notifyParamsP`*
Pointer to a structure that contains the notification event that occurred and any other information about it. See `SysNotifyParamType`.

Returns Your notification handler should always return `errNone`.

Comments

NOTE: Applications should register to receive each notification as a launch code; the use of a callback function in Palm OS Cobalt is discouraged.

You pass this function as a parameter to `SysNotifyRegister()` or `SysNotifyRegisterV40()` when registering code that does not have a `PilotMain()` for a notification. See the description of

`SysNotifyRegister()` for advice on writing a notification handler.

IMPORTANT: Because the *`callbackP`* 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. Because of this, applications should generally register to receive a notification using a callback function; instead, register to receive it as a launch code.

12

Palm Types

PalmTypes.h defines a number of basic types and constants used throughout Palm OS, along with a number of macros to do byte-swapping (for endianness) and time interval conversion.

The contents of this chapter is organized as follows:

Palm Types Structures and Types

Boolean Typedef

Purpose A boolean type.

Declared In `PalmTypes.h`

Prototype `typedef unsigned char Boolean`

Comments Use the `TRUE` and `FALSE`, or `true` and `false`, constants with Boolean variables.

coord Typedef

Purpose A single floating-point coordinate type.

NOTE: This type is provided for compatibility purposes; applications should use `fcoord_t` instead.

Declared In `PalmTypes.h`

Prototype `typedef float coord`

See Also `Coord`, `fcoord_t`

Coord Typedef

Purpose A single fixed-point coordinate type, used for screen and window coordinates.

Declared In `PalmTypes.h`

Prototype `typedef int16_t Coord`

See Also `coord`

Enum16 Typedef

Purpose An enum type that can have up to 65,535 enumerated constants.

Declared In `PalmTypes.h`

Prototype `typedef uint16_t Enum16`

See Also `Enum8`, `SignedEnum16`

Enum8 Typedef

Purpose An enum type that can have up to 255 enumerated constants.

Declared In `PalmTypes.h`

Prototype `typedef uint8_t Enum8`

See Also `Enum16`, `SignedEnum8`

fcoord_t Typedef

Purpose A single floating-point coordinate type, used in conjunction with the graphics context drawing functions.

Declared In `PalmTypes.h`

Prototype `typedef float fcoord_t`

See Also `coord`, `Coord`

MemHandle Struct

Purpose A handle to a location in memory. Unlike a pointer, a handle can be used to reference memory that may be relocated.

Declared In `PalmTypes.h`

Prototype `typedef struct _opaque *MemHandle`

Fields None.

See Also `MemPtr`, `SysHandle`

MemPtr Typedef

Purpose A pointer to a location in memory.

Declared In `PalmTypes.h`

Prototype `typedef void *MemPtr`

See Also `MemHandle`

ProcPtr Typedef

Purpose Pointer to a function that returns a 32-bit integer.

Declared In `PalmTypes.h`

Prototype `int32_t (*ProcPtr) ()`

SignedEnum16 Typedef

Purpose An enum type whose enumerated constant values can range from -32,768 to +32,767.

Declared In `PalmTypes.h`

Prototype `typedef int16_t SignedEnum16`

See Also `Enum16`, `SignedEnum8`

SignedEnum8 Typedef

Purpose An enum type whose enumerated constant values can range from -128 to +127.

Declared In `PalmTypes.h`

Prototype `typedef int8_t SignedEnum8`

See Also `Enum8`, `SignedEnum16`

SysHandle Typedef

Purpose A virtual address.

Declared In `PalmTypes.h`

Prototype `typedef uint32_t SysHandle`

VAddr Typedef

Purpose A virtual address, used by the `ErrThrow()`/`ErrCatch()` exception-handling mechanism.

Declared In `PalmTypes.h`

Prototype `typedef uint32_t VAddr`

wchar16_t Typedef

Purpose A 16-bit character type.

Declared In `PalmTypes.h`

Prototype `typedef uint16_t wchar16_t`

See Also `wchar32_t`

wchar32_t Typedef

Purpose A 32-bit “wide” character type.

Declared In `PalmTypes.h`

Prototype `typedef uint32_t wchar32_t`

Comments `wchar16_t`

Palm Types Constants

Time Constants

Purpose Various multiples of one nanosecond. These constants are used by the nanosecond conversion macros documented in “Palm Types Functions and Macros” on page 238.

Declared In `PalmTypes.h`

Constants `#define P_ONE_MICROSECOND (P_ONE_NANOSECOND*1000)`
One microsecond.

`#define P_ONE_MILLISECOND (P_ONE_MICROSECOND*1000)`
One millisecond.

`#define P_ONE_NANOSECOND ((nsecs_t)1)`
One nanosecond.

`#define P_ONE_SECOND (P_ONE_MILLISECOND*1000)`
One second.

Boolean Values Enum

Purpose Defines values that can be used with Boolean variables.

Declared In `PalmTypes.h`

Constants `false`
A “false” value that can be used with Boolean variables.

`true`
A “true” value that can be used with Boolean variables.

Miscellaneous Constants

Purpose `PalmTypes.h` also defines these constants.

Declared In `PalmTypes.h`

Constants `#define FALSE (0)`

A "false" value that can be used with Boolean variables.

`#define NULL 0`

A null value that can be used with pointers.

`#define TRUE (1)`

A "true" value that can be used with Boolean variables.

Palm Types Functions and Macros

EndianSwap16 Macro

Purpose Swaps the bytes in a 16-bit value to switch between big-endian byte order and little-endian byte order.

Declared In `PalmTypes.h`

Prototype `#define EndianSwap16 (`*`n`*`)`

Parameters → *n*

The 16-bit value to be byte-swapped.

Returns Evaluates to a 16-bit value with endianness opposite that of the supplied value.

See Also `EndianSwap32`, `RsrcEndianSwap16()`

EndianSwap32 Macro

Purpose Swaps the bytes in a 32-bit value to switch between big-endian byte order and little-endian byte order.

Declared In `PalmTypes.h`

Prototype `#define EndianSwap32 (`*`n`*`)`

Parameters → *n*

The 32-bit value to be byte-swapped.

Returns Evaluates to a 32-bit value with endianness opposite that of the supplied value.

See Also `EndianSwap16`, `RsrcEndianSwap32()`

ErrConvertFrom68k Macro

Purpose Converts a 16-bit error value, of the type produced by many Palm OS functions when called through PACE, to a `status_t` value, as produced by many ARM-native operating system functions.

Declared In `PalmTypes.h`

Prototype `#define ErrConvertFrom68k (`*`x`*`)`

Parameters → *x*

The 16-bit error code to be converted.

Returns The `status_t` value that corresponds to the supplied error code.

See Also `ErrConvertTo68k`

ErrConvertTo68k Macro

Purpose Converts a `status_t` value, of the type produced by many ARM-native operating system functions, to a 16-bit error value, as produced by many operating system functions when called through PACE.

Declared In `PalmTypes.h`

Prototype `#define ErrConvertTo68k (`*`x`*`)`

Parameters → *x*

The `status_t` value to be converted.

Returns The 16-bit error code that corresponds to the supplied `status_t` value.

See Also `ErrConvertFrom68k`

P_MICROSECONDS_TO_NANOSECONDS Macro

Purpose Converts from microseconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_MICROSECONDS_TO_NANOSECONDS (us)`

Parameters → *us*

A quantity of time, in microseconds.

Returns Evaluates to the supplied amount of time, in nanoseconds.

Comments This macro is equivalent to `P_US2NS()`.

See Also `P_MILLISECONDS_TO_NANOSECONDS()`, `P_SECONDS_TO_NANOSECONDS()`, `P_NANOSECONDS_TO_MICROSECONDS()`

P_MILLISECONDS_TO_NANOSECONDS Macro

Purpose Converts from milliseconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_MILLISECONDS_TO_NANOSECONDS (ms)`

Parameters → *ms*

A quantity of time, in milliseconds.

Returns Evaluates to the supplied amount of time, in nanoseconds.

Comments This macro is equivalent to `P_MS2NS()`.

See Also `P_MICROSECONDS_TO_NANOSECONDS()`, `P_SECONDS_TO_NANOSECONDS()`, `P_NANOSECONDS_TO_MILLISECONDS()`

P_MS2NS Macro

Purpose Converts from milliseconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_MS2NS (ms)`

Parameters → *ms*

A quantity of time, in milliseconds.

Returns Evaluates to the supplied amount of time, in nanoseconds.

Comments This macro is equivalent to `P_MILLISECONDS_TO_NANOSECONDS()`.

See Also `P_S2NS()`, `P_US2NS()`, `P_NS2MS()`

P_NANOSECONDS_TO_MICROSECONDS Macro

Purpose Converts from nanoseconds to microseconds.

Declared In `PalmTypes.h`

Prototype `#define P_NANOSECONDS_TO_MICROSECONDS (ns)`

Parameters → *ms*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in microseconds.

Comments This macro is equivalent to `P_NS2US()`.

See Also `P_NANOSECONDS_TO_MILLISECONDS()`, `P_NANOSECONDS_TO_SECONDS()`, `P_MICROSECONDS_TO_NANOSECONDS()`

P_NANOSECONDS_TO_MILLISECONDS Macro

Purpose Converts from nanoseconds to milliseconds.

Declared In `PalmTypes.h`

Prototype `#define P_NANOSECONDS_TO_MILLISECONDS (ns)`

Parameters → *ms*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in milliseconds.

Comments This macro is equivalent to `P_NS2MS()`.

See Also `P_NANOSECONDS_TO_MICROSECONDS()`, `P_NANOSECONDS_TO_SECONDS()`, `P_MILLISECONDS_TO_NANOSECONDS()`

P_NANOSECONDS_TO_SECONDS Macro

Purpose Converts from nanoseconds to seconds.

Declared In `PalmTypes.h`

Prototype `#define P_NANOSECONDS_TO_SECONDS (`*`ns`*`)`

Parameters → *`ms`*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in seconds.

Comments This macro is equivalent to `P_NS2S()`.

See Also `P_NANOSECONDS_TO_MICROSECONDS()`, `P_NANOSECONDS_TO_MILLISECONDS()`, `P_SECONDS_TO_NANOSECONDS()`

P_NS2MS Macro

Purpose Converts from nanoseconds to milliseconds.

Declared In `PalmTypes.h`

Prototype `#define P_NS2MS (`*`ns`*`)`

Parameters → *`ms`*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in milliseconds.

Comments This macro is equivalent to `P_NANOSECONDS_TO_MILLISECONDS()`.

See Also `P_NS2US()`, `P_NS2S()`, `P_MS2NS()`

P_NS2S Macro

Purpose Converts from nanoseconds to seconds.

Declared In `PalmTypes.h`

Prototype `#define P_NS2S (ns)`

Parameters → *s*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in seconds..

Comments This macro is equivalent to `P_NANOSECONDS_TO_SECONDS()`.

See Also `P_NS2MS()`, `P_NS2US()`, `P_S2NS()`

P_NS2US Macro

Purpose Converts from nanoseconds to microseconds.

Declared In `PalmTypes.h`

Prototype `#define P_NS2US (ns)`

Parameters → *ms*

A quantity of time, in nanoseconds.

Returns Evaluates to the supplied amount of time, in microseconds.

Comments This macro is equivalent to `P_NANOSECONDS_TO_MICROSECONDS()`.

See Also `P_NS2MS()`, `P_NS2S()`, `P_US2NS()`

P_S2NS Macro

Purpose Converts from seconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_S2NS (s)`

Parameters → *s*

A quantity of time, in seconds.

Returns Evaluates to the supplied amount of time, in seconds..

Comments This macro is equivalent to `P_SECONDS_TO_NANOSECONDS()`.

See Also `P_MS2NS()`, `P_US2NS()`, `P_NS2S()`

P_SECONDS_TO_NANOSECONDS Macro

Purpose Converts from seconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_SECONDS_TO_NANOSECONDS (s)`

Parameters → *s*

A quantity of time, in seconds.

Returns Evaluates to the supplied amount of time, in seconds..

Comments This macro is equivalent to `P_S2NS()`.

See Also `P_MICROSECONDS_TO_NANOSECONDS()`, `P_MILLISECONDS_TO_NANOSECONDS()`, `P_NANOSECONDS_TO_SECONDS()`

P_US2NS Macro

Purpose Converts from microseconds to nanoseconds.

Declared In `PalmTypes.h`

Prototype `#define P_US2NS (us)`

Parameters → *us*

A quantity of time, in microseconds.

Returns Evaluates to the supplied amount of time, in nanoseconds.

Comments This macro is equivalent to `P_MICROSECONDS_TO_NANOSECONDS()`.

See Also `P_MS2NS()`, `P_US2NS()`, `P_NS2US()`

RsrcEndianSwap16 Macro

Purpose Swaps the bytes in a 16-bit value within a resource to switch between big-endian byte order and little-endian byte order.

Declared In `PalmTypes.h`

Prototype `#define RsrcEndianSwap16 (`*`x`*`)`

Parameters → *`x`*

The 16-bit value within the resource that is to be byte-swapped.

Returns Evaluates to a 16-bit value with endianness opposite that of the supplied value.

See Also `EndianSwap16`, `RsrcEndianSwap32()`

RsrcEndianSwap32 Macro

Purpose Swaps the bytes in a 32-bit value within a resource to switch between big-endian byte order and little-endian byte order.

Declared In `PalmTypes.h`

Prototype `#define RsrcEndianSwap32 (`*`x`*`)`

Parameters → *`x`*

The 32-bit value within the resource that is to be byte-swapped.

Returns Evaluates to a 32-bit value with endianness opposite that of the supplied value.

See Also `EndianSwap32`, `RsrcEndianSwap16()`

13

System Event Manager

The System Event Manager APIs allow you to enable the Graffiti® 2 handwriting engine and to control the device's auto-off timer.

The contents of this chapter are organized as follows:

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

For other APIs used to work with events, see Chapter 7, "Event," on page 139. For background information on the Palm OS event system, see Chapter 3, "Events and the Event Loop," on page 43.

System Event Manager Structures and Types

EvtSetAutoOffCmd Typedef

Purpose Contains one of the auto-off-timer command values defined by the `EvtSetAutoOffTag` enum.

Declared In `SysEvtMgr.h`

Prototype `typedef Enum8 EvtSetAutoOffCmd`

System Event Manager Constants

EvtSetAutoOffTag Enum

Purpose Commands used with `EvtSetAutoOffTimer()` to control the device's auto-off timer.

Declared In `SysEvtMgr.h`

Constants

`SetAtLeast`
Make sure that the device won't turn off until `timeout` seconds of idle time has passed. (This operation only changes the current value if it's less than the value you specify.)

`SetExactly`
Set the timer to turn off in `timeout` seconds.

`SetAtMost`
Make sure the device will turn before `timeout` seconds has passed. (This operation only changes the current value if it's greater than the value you specify.)

`SetDefault`
Change the default auto-off timeout to `timeout` seconds.

`ResetTimer`
Reset the auto-off timer so that the device does not turn off until at least the default seconds of idle time has passed.

System Event Manager Functions and Macros

EvtEnableGraffiti Function

Purpose Enable or disable Graffiti 2 handwriting recognition.

Declared In `SysEvtMgr.h`

Prototype `void EvtEnableGraffiti (Boolean enable)`

Parameters → *`enable`*

`true` to enable handwriting recognition, `false` to disable it.

Returns Nothing.

EvtResetAutoOffTimer Function

Purpose Reset the auto-off timer.

Declared In `SysEvtMgr.h`

Prototype `status_t EvtResetAutoOffTimer (void)`

Parameters None.

Returns Always returns `errNone`.

Comments `EvtResetAutoOffTimer` resets the auto-off timer so that the device does not turn off until at least the default amount of idle time has passed. You can use this function to ensure that the device doesn't automatically power off during a long operation without user input (for example, when there is a lot of serial port activity).

NOTE: This function requires an IPC; accordingly, it should be used sparingly.

If you need more control over the auto-off timer, consider using `EvtSetAutoOffTimer()` instead of this function.

EvtSetAutoOffTimer Function

Purpose Set the auto-off timer.

Declared In `SysEvtMgr.h`

Prototype

```
status_t EvtSetAutoOffTimer
   (EvtSetAutoOffCmd cmd, uint16_t timeout)
```

Parameters

→ *`cmd`*
One of the commands defined by the `EvtSetAutoOffTag` enum.

→ *`timeout`*
A new timeout value in seconds. If *`cmd`* is `ResetTimer`, this parameter is ignored.

Returns Always returns `errNone`.

Comments Use this function to ensure that the device doesn't automatically power off during a long operation that has no user input (for example, when there is a lot of serial port activity).

See Also `EvtResetAutoOffTimer()`

Index