PALM COMPUTING® PLATFORM™

Developing Palm OS 3.0 Applications

Part II: System Management

Navigate this online document as follows:

To see bookmarks, type:	**Command-7 (Mac OS)** **Ctrl-7 (Windows)**
To navigate, click on:	**any blue hypertext link** **any Table of Contents entry** **any Index entry** **arrows in the toolbar**

Developing Palm OS 3.0 Applications

Part II: System Management

Contact Information:

Metrowerks U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Metrowerks Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Metrowerks Mail order	Voice: 1-800-377–5416 Fax: 1-512-873–4901
3Com (Palm Computing Subsidiary) Mail Order	U.S.A.: 1-800-881-7256 Canada: 800-891-6342 elsewhere: 1-801-431-1536
Metrowerks World Wide Web	`http://www.metrowerks.com`
Palm Computing World Wide Web	`http://www.palm.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	go `Metrowerks`

Table of Contents

About This Document. 13
Palm OS SDK Documentation 13
What This Guide Contains 14
Conventions Used in This Guide 15

1 Using Palm OS System Managers 17
The Alarm Manager. 18
Alarm Manager Overview 18
Using the Alarm Manager 20
Alarm Manager Function Summary 20
The Error Manager 21
Displaying Development Errors 21
Using the Error Manager Macros 22
Understanding the Try-and-Catch Mechanism 23
Using the Try and Catch Mechanism 24
Error Manager Function Summary 25
The Feature Manager 25
The System Version Feature 26
Application-Defined Features 26
Using the Feature Manager 27
Feature Manager Function Summary 27
File Streaming Application Program Interface 28
Using the File Streaming API 28
File Streaming Data Structures 29
File Streaming Function Summary 35
The Sound Manager. 35
Using the Sound Manager 37
Sound Preferences Compatibility Information 42
Sound Manager Data Structures 46
Sound Manager Function Summary 52
The String Manager 53
String Manager Function Summary 53
The System Manager 54
System Boot and Reset 54

Power Management 55
The Microkernel 57
Application Support 58
System Manager Function Summary 63
The System Event Manager 63
Event Translation: Pen Strokes to Key Events. 64
Pen Queue Management 65
Key Queue Management 66
Auto-Off Control. 67
System Event Manager Function Summary 67
The Time Manager 68
Using Real-Time Clock Functions. 68
Using System Ticks Functions 68
Time Manager Structures 69
Time Manager Function Summary 70
Application Launcher 70

2 Palm OS System Functions 73
Alarm Manager API. 73
AlmGetAlarm 73
AlmSetAlarm 74
Functions for System Use Only. 75
Error Manager Functions 76
ErrDisplay 76
ErrDisplayFileLineMsg 77
ErrFatalDisplayIf. 78
ErrNonFatalDisplayIf. 79
ErrThrow 80
Event Manager Functions 80
EvtAddEventToQueue 80
EvtAddUniqueEventToQueue 81
EvtCopyEvent 81
EvtDequeuePenPoint 82
EvtDequeuePenStrokeInfo. 83
EvtEnableGraffiti. 83
EvtEnqueueKey 84

EvtEventAvail 85
EvtFlushKeyQueue 85
EvtFlushNextPenStroke 86
EvtFlushPenQueue 86
EvtGetEvent 87
EvtGetPen 87
EvtGetPenBtnList 88
EvtKeyQueueEmpty 88
EvtKeyQueueSize 89
EvtPenQueueSize 89
EvtProcessSoftKeyStroke 90
EvtResetAutoOffTimer 90
EvtSysEventAvail 91
EvtWakeup 91
Functions for System Use Only 92
Feature Manager Functions 93
FtrGet 93
FtrGetByIndex 94
FtrSet 95
FtrUnregister 96
Functions for System Use Only 96
Find Functions 97
FindDrawHeader 97
FindGetLineBounds 97
FindSaveMatch 98
FindStrInStr 99
Float Manager Functions 101
Using Floating Point Arithmetic 101
Using 1.0 Floating-Point Functionality 101
FplAdd 102
FplAToF 102
FplBase10Info 103
FplDiv 104
FplFloatToLong 104
FplFloatToULong 105
FplFree 105

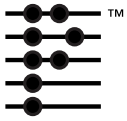
FplFToA 106
FplInit 106
FplLongToFloat 107
FplMul 107
FplSub 108
Miscellaneous System Functions 109
Crc16CalcBlock 109
MdmDial 110
MdmHangUp 111
PhoneNumberLookup 111
ResLoadForm 112
ResLoadMenu 112
System Preferences Functions 113
PrefGetAppPreferences 113
PrefGetAppPreferencesV10 114
PrefGetPreference 115
PrefGetPreferences 116
PrefOpenPreferenceDBV10 116
PrefSetAppPreferences 117
PrefSetAppPreferencesV10 118
PrefSetPreference. 119
PrefSetPreferences 119
Password Functions. 120
PwdExists. 120
PwdRemove. 120
PwdSet 121
PwdVerify. 121
String Manager Functions 122
StrAToI 122
StrCaselessCompare 122
StrCat. 123
StrChr 123
StrCompare 124
StrCopy. 124
StrDelocalizeNumber. 125
StrIToA 125

StrIToH 126
StrLen 126
StrLocalizeNumber 127
StrNCaselessCompare 127
StrNCat 128
StrNCompare 129
StrNCopy 129
StrPrintF 130
StrStr 130
StrToLower 131
StrVPrintF 131
File Streaming Functions 133
FileClearerr 133
FileClose 133
FileControl 134
FileDelete 136
FileDmRead 136
FileEOF 138
FileError 139
FileFlush 139
FileGetLastError 140
FileOpen 140
FileRead 143
FileRewind 144
FileSeek 145
FileTell 146
FileTruncate 147
FileWrite 147
Functions For System Use Only 148
File Streaming Error Codes 149
Sound Manager Functions 150
SndCreateMidiList 150
SndDoCmd 151
SndGetDefaultVolume 152
SndPlaySMF 152
SndPlaySystemSound 155
Functions for System Use Only 155

System Functions 156
SysAppLaunch 156
SysAppLauncherDialog 157
SysBatteryInfo 158
SysBatteryInfoV20 159
SysBinarySearch 160
SysBroadcastActionCode 162
SysCopyStringResource 162
SysCreateDataBaseList 163
SysCreatePanelList 164
SysCurAppDatabase 164
SysErrString 165
SysFatalAlert 165
SysFormPointerArrayToStrings 166
SysGetOSVersionString 166
SysGetRomToken 167
SysGetStackInfo 168
SysGraffitiReferenceDialog 168
SysGremlins 169
SysHandleEvent 170
SysInsertionSort 170
SysInstall 172
SysKeyboardDialog 173
SysKeyboardDialogV10 173
SysLibFind 174
SysLibLoad 175
SysQSort 176
SysRandom 177
SysReset 177
SysSetAutoOffTime 178
SysStringByIndex 178
SysTaskDelay 179
SysTicksPerSecond 179
SysUIAppSwitch 179
Functions for System Use Only 180

Time Manager Functions 189
DateAdjust 189
DateDaysToDate 189
DateSecondsToDate 190
DateToAscii 190
DateToDays 191
DateToDOWDMFormat 191
DayOfMonth 192
DayOfWeek 192
DaysInMonth 193
TimAdjust 193
TimDateTimeToSeconds 194
TimGetSeconds 194
TimGetTicks 194
TimSecondsToDateTime 195
TimSetSeconds 195
TimeToAscii 196
Functions for System Use Only 196

Index 199



About This Document

Developing Palm OS 3.0 Applications, Part II, is part of the Palm OS Software Development Kit (SDK). This introduction provides an overview of the SDK documentation, discusses what materials are included in this document, and what conventions are used.

Palm OS SDK Documentation

The following documents are part of the SDK:

Document	Description
Palm OS 3.0 Tutorial	A number of Phases step developers through using the different parts of the system. Example applications for each phase are included in the SDK.
Developing Palm OS 3.0 Applications. Part I: Interface Management	A programmer's guide and reference document that discusses all important aspects of developing an application.
Developing Palm OS 3.0 Applications. Part II. System Management.	A programmer's guide and reference document for all system managers, such as the string manager or the system event manager. See What This Guide Contains for details.

Document	Description
Developing Palm OS 3.0 Applications, Part III. Memory and Communications Management	Programmer's guide and reference document for: • Memory management; both the database manager and the memory manager. • The Palm OS communications library for serial communication. • The Palm OS network library, which provides basic network services. • The exchange manager and IR library, which provide infrared communication capabilities.
Palm OS 3.0 Cookbook.	Provides a variety of design guidelines, including localization, UI design, and optimization. Information about using CodeWarrior for Palm OS to create projects and executables.

What This Guide Contains

This section provides an overview of the chapters in this guide.

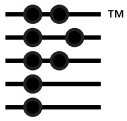
- Chapter 1, "Using Palm OS System Managers," discusses the managers that provide system functionality, including the system event manager, time manager, and error manager.
- Chapter 2, "Palm OS System Functions," provides reference-style information for each API function that allows applications to interact with the system.

Conventions Used in This Guide

This guide uses the following typographical conventions:

This style...	Is used for...
`fixed width font`	Code elements such as function, structure, field, bitfield.
`fixed width underline`	Emphasis (for code elements).
bold	Emphasis (for other elements).
blue and underlined	Hot links.
black and underlined	3.0 function names (headings only)
red and underlined	3.0 function names (in Table of Contents only)

1



Using Palm OS System Managers

In contrast to desktop computer operating systems, Palm OS consists of only one library. This library, however, contains several managers, which are groups of functions that work together to implement certain functionality. As a rule, all functions that belong to one manager use the same three-letter prefix and work together to implement a certain aspect of functionality.

In this chapter, you learn about all Palm OS managers that aren't directly responsible for interface management or memory management. As you investigate the managers more closely you'll find that some of them are mostly services provided by the system, while others contain a large number of API calls.

This chapter presents the managers in the following order:

- The Alarm Manager provides support for setting real-time alarms to perform some periodic activity or display a reminder.
- The Error Manager can be used by applications or system software for displaying unexpected runtime errors, such as those that typically show up during program development.

  Final production versions of applications or system software are not expected to use error manager.
- The Feature Manager provides information about the system software version and the optional system features and third-party extensions that are installed. An application can also use the feature manager to keep track of its own data.
- The Sound Manager lets applications and system modules control sound manager settings and play custom and predefined system sounds.

- The String Manager is a set of string manipulation functions available to applications. Use these routines instead of the standard C routines.
- The System Manager is responsible for the basic operation of the system, including booting and resetting the system, managing power, managing the microkernel, and supporting applications.
- The System Event Manager provides an interface to the low-level pen and key event queues, translates taps on silk-screened icons into key events, sends pen strokes in the Graffiti area to the Graffiti recognizer, and puts the system into low-power doze mode when there is no user activity.
- The Time Manager provides real-time clock functions and system tick functions.

The Alarm Manager

The Palm OS alarm manager provides support for setting real-time alarms, for performing some periodic activity, or for displaying a reminder. This section helps you use the alarm manager by discussing these topics:

- Alarm Manager Overview
- Using the Alarm Manager
- Alarm Manager Function Summary

Alarm Manager Overview

The alarm manager:

- Works closely with the time manager to handle real-time alarms.
- Sends launch codes to applications that set a specific time alarm to inform the application the alarm is due.
- Handles alarms by application in a two cycle operation
  - First, it notifies each application that the alarm has occurred.
  - Second, it allows each application to display some UI.
- Allows only one alarm to be set per application

However, the alarm manager

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

The following section looks in some detail at how the alarm manager and applications interact when processing an alarm.

Alarm Queue

The alarm queue contains all alarm requests. Triggered alarms are queued up until the alarm manager can send the launch code to the application that created the alarm. However, if the alarm queue becomes full, the oldest entry that has been both triggered and notified is deleted to make room for a new alarm.

Alarm Manager Processing

When an alarm is triggered, the alarm manager notifies each application that set an alarm for that alarm time via the `sysAppLaunchCmdAlarmTriggered` launch code.

After each application has processed this launch code, the alarm manager sends each application the `sysAppLaunchCmdDisplayAlarm` launch code in order for the application to display the alarm.

If a new alarm time is triggered while an older alarm is still being displayed, all applications with alarms scheduled for this second alarm time are sent the `sysAppLaunchCmdAlarmTriggered` launch code, but the display cycle is postponed until all earlier alarms have finished displaying.

Alarm Scenario

The alarm manager typically first notifies each application that an alarm has been triggered, then notifies each application to display the alarm. Here's how an application and the alarm manager typically interact when processing an alarm

1. When the alarm time is reached, the alarm manager finds the first application in the alarm queue that set an alarm for this alarm time.
2. The alarm manager sends this application the `sysAppLaunchCmdAlarmTriggered` launch code.

3. The application can now:
   - Set the next alarm.
   - Play a short sound.
   - Perform some maintenance activity.
4. The alarm manager finds in the alarm queue the next application that set an alarm and repeats steps 2 and 3.
5. This is process is repeated until no more applications are found with this alarm time.
6. The alarm manager then finds once again the first application in the alarm queue who set an alarm for this alarm time and sends this application the `sysAppLaunchCmdDisplayAlarm` launch code
7. The application can now:
   - Display a dialog box
   - Display some other type of reminder
8. The alarm manager processes the alarm queue for the next application that set an alarm for the alarm being triggered and step 6 and 7 are repeated.
9. This is process is repeated until no more applications are found with this alarm time.

Using the Alarm Manager

An applications can use the Palm OS function `AlmSetAlarm` to set and/or clear an alarm.

An application can find out its current alarm setting by using the `AlmGetAlarm` function. This function returns the alarm date and time (expressed in seconds since 1/1/1904). The return value is 0 if no active alarm exists for the application.

Alarm Manager Function Summary

The following alarm manager functions are for application use:

- AlmGetAlarm
- AlmSetAlarm

The Error Manager

The error manager can be used by applications or system software for displaying unexpected runtime errors such as those that typically show up during program development. Final versions of applications or system software won't use the error manager.

The error manager API consists of a set of functions for displaying an alert with an error message, file name, and the line number where the error occurred. If a debugger is connected, it is entered when the error occurs.

The error manager also provides a "try and catch" mechanism that applications can use for handling such runtime errors as out of memory conditions, user input errors, etc. This mechanism is closely modeled after the try / catch functionality of the recent ANSI C specification.

This section helps you understand and use the error manager, discussing the following topics:

- Displaying Development Errors
- Understanding the Try-and-Catch Mechanism
- Using the Error Manager Macros
- Error Manager Function Summary

Displaying Development Errors

The error manager provides some compiler macros that can be used in source code. These macros display a fatal alert dialog on the screen and provide buttons to reset the device or enter the debugger after the error is displayed. There are three macros: `ErrDisplay`, `ErrFatalDisplayIf`, and `ErrNonFatalDisplayIf`.

- `ErrDisplay` always displays the error message on the screen.
- `ErrFatalDisplayIf` and `ErrNonFatalDisplayIf` display the error message only if their first argument is `TRUE`.

The error manager uses the compiler define `ERROR_CHECK_LEVEL` to control the level of error messages displayed. You can set the value of the compiler define to control which level of error checking

and display is compiled into the application. Three levels of error checking are supported: none, partial, and full.

If you set ERR_CHECK_LEVEL to...	The compiler...
ERROR_CHECK_NONE (0)	Doesn't compile in any error calls.
ERROR_CHECK_PARTIAL(1)	Compiles in only ErrDisplay and ErrFatalDisplayIf calls.
ERROR_CHECK_FULL (2)	Compiles in all three calls.

During development, it makes sense to set full error checking for early development, partial error checking during alpha and beta test periods, and no error checking for the final product. At partial error checking, only fatal errors are displayed; error conditions that are only possible are ignored under the assumption that the application developer is already aware of the condition and designed the software to operate that way.

Using the Error Manager Macros

Calls to the error manager to display errors are actually compiler macros that are conditionally compiled into your program. Most of the calls take a boolean parameter, which should be set to TRUE to display the error, and a pointer to a text message to display if the condition is true.

Typically, the boolean parameter is an in-line expression that evaluates to TRUE if there is an error condition. As a result, both the expression that evaluates the error condition and the message text are left out of the compiled code when error checking is turned off. You can call ErrFatalDisplayIf, or ErrDisplay, but using ErrFatalDisplayIf makes your source code look neater.

For example, assume your source code looks like this:

```
result = DoSomething();
ErrFatalDisplayIf (result < 0, “unexpected
                    result from DoSomething”);
```

With error checking turned on, this code displays an error alert dialog if the result from `DoSomething()` is less than 0. Besides the error message itself, this alert also shows the file name and line number of the source code that called the error manager. With error checking turned off, both the expression evaluation `err < 0` and the error message text are left out of the compiled code.

The same net result can be achieved by the following code:

```
result = DoSomething();
#if ERROR_CHECK_LEVEL != ERROR_CHECK_NONE
if (result < 0)
  ErrDisplay (“unexpected result from
                DoSomething”);
#endif
```

However, this solution is longer and requires more work than simply calling `ErrFatalDisplayIf`. It also makes the source code harder to follow.

Understanding the Try-and-Catch Mechanism

The try-and-catch mechanism of the error manager is closely modeled after the ANSI C try and catch standard.

The error manager is aware of the machine state of the Palm OS device and can therefore correctly save and restore this state. The built-in try and catch of the compiler can’t be used because it’s machine dependent.

Try and catch is basically a neater way of implementing a `goto` if an error occurs. A typical way of handling errors in the middle of a routine is to go to the end of the routine as soon as an error occurs and have some general-purpose cleanup code at the end of every routine. Errors in nested routines are even trickier because the result code from every subroutine call must be checked before continuing.

When you set up a try / catch, you are providing the compiler with a place to jump to when an error occurs. You can go to that error handling routine at any time by calling `ErrThrow`. When the compiler sees the `ErrThrow` call, it performs a `goto` to your error handling

code. The greatest advantage to calling `ErrThrow`, however, is for handling errors in nested subroutine calls.

Even if `ErrThrow` is called from a nested subroutine, execution immediately goes to the same error handling code in the higher-level call. The compiler and runtime environment automatically strip off the stack frames that were pushed onto the stack during the nesting process and go to the error handling section of the higher-level call. You no longer have to check for result codes after calling every subroutine; this greatly simplifies your source code and reduces its size.

Using the Try and Catch Mechanism

The following example illustrates the possible layout for a a typical routine using the error manager's try and catch mechanism.

Listing 1.1 Try and Catch Mechanism Example

```
ErrTry {
  p = MemPtrNew(1000);
  if (!p) ErrThrow(errNoMemory);
  MemSet(p, 1000, 0);
  CreateTable(p);
  PrintTable(p);
  }

ErrCatch(err) {
  // Recover or clean up after a failure in the
  // above Try block."err" is an int
  // identifying the reason for the failure.

  // You may call ErrThrow() if you want to
  // jump out to the next Catch block.

  // The code in this Catch block doesn't
  // execute if the above Try block completes
  // without a Throw.

  if (err == errNoMemory)
    ErrDisplay("Out of Memory");
  else
```

```
    ErrDisplay("Some other error");
} ErrEndCatch
// You must structure your code exactly as
//above. You can't have an ErrTry without an
//ErrCatch { } ErrEndCatch, or vice versa.
```

Any call to ErrThrow within the ErrTry block results in control passing immediately to the ErrCatch block. Even if the subroutine CreateTable called ErrThrow, control would pass directly to the ErrCatch block. If the ErrTry block completes without calling ErrThrow, the ErrCatch block is not executed.

You can nest multiple ErrTry blocks. For example, if you wanted to perform some cleanup at the end of CreateTable in case of error,

- Put ErrTry/ErrCatch blocks in CreateTable
- Clean up in the ErrCatch block first
- Call ErrThrow to jump to the top-level ErrCatch

Error Manager Function Summary

The following error manager functions are available for application use:

- ErrDisplay
- ErrDisplayFileLineMsg
- ErrFatalDisplayIf
- ErrNonFatalDisplayIf
- ErrThrow

The Feature Manager

A **feature** is a 32-bit value that has special meaning to both the feature publisher and to users of that feature. Features can be published by the system or by applications.

Each feature is identified by a feature creator and a feature number:

- The feature creator is usually the database creator type of the application that publishes the feature.

- The feature number is any 16-bit value used to distinguish between different features of a particular creator.

Once a feature is published, it remains present until it is explicitly deleted. A feature published by an application sticks around even after the application quits.

The System Version Feature

An example for a feature is the system version. This feature is published by the system and contains a 32-bit representation of the system version. The system version has a feature creator of "psys" and a feature number of 1. Currently, the different versions of the system software have the following numbers:

0x01003001	Pilot 1000 and Pilot 5000 (Palm OS 1.0)
0x02003000	PalmPilot and PalmPilot Professional (Palm OS 2.0)
0x03003000	Palm III Connected Organizer (Palm OS 3.0)

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

Application-Defined Features

When an application adds or removes capabilities from the base system, it can create features to test for the presence or absence of those capabilities. This allows an application to be compatible with multiple versions of the system by refining its behavior, depending on which capabilities are present or not. Future hardware platforms may lack some capabilities present in the first platform, so checking the system version feature is important.

This section introduces the feature manager by discussing these topics:

- Using the Feature Manager
- Feature Manager Function Summary

Using the Feature Manager

Applications may find the feature manager useful for their own private use. For example, an application may want to publish a feature that contains a pointer to some private data it needs for processing launch codes. Because an application's global data is not generally available while it processes launch codes, using the feature manager is usually the easiest way for an application to get to its data.

To check whether a particular feature is present, call `FtrGet` and pass it the feature creator and feature number. If the feature exists, `FtrGet` returns the 32-bit value of the feature. If the feature doesn't exist, an error code is returned.

To publish a new feature or change the value of an existing one, call `FtrSet` and pass the feature creator and number, and the 32-bit value of the feature. A published feature remains available until it is explicitly removed by a call to `FtrUnregister` or until the system resets; simply quitting an application doesn't remove a feature published by that application.

Features are split into two groups: ROM-based and RAM-based. ROM-based features are stored in a separate table in ROM and can never be removed; only system-defined features are in this table. All features installed at runtime are in the RAM table. `FtrGetByIndex` accepts a parameter that specifies whether to search the ROM table or RAM table.

Call `FtrUnregister` to remove RAM-based features created at runtime by calling `FtrSet`.

You can get a complete list of all published features by calling `FtrGetByIndex` repeatedly. Passing an index value starting at 0 to `FtrGetByIndex` and incrementing repeatedly by 1 eventually returns all available features.

Feature Manager Function Summary

The following feature manager functions are available for application use:

- FtrGet
- FtrGetByIndex

- FtrSet
- FtrUnregister

File Streaming Application Program Interface

The file streaming functions in Palm OS 3.0 let you work with large blocks of data. File streams can be arbitrarily large—they are not subject to the 64k maximum size limit imposed by the memory manager on allocated objects. File streams can be used for permanent data storage; in Palm OS 3.0, their underlying implementation is a PalmOS database. You can read, write, seek to a specified offset, truncate, and do everything else you'd expect to do with a desktop-style file.

Other than backup / restore, Palm OS does not provide direct Hot Sync support for file streams, and none is planned at this time.

The use of double-buffering imposes a performance penalty on file streams that may make them unsuitable for certain applications. Record-intensive applications tend to obtain better performance from the Data Manager.

Using the File Streaming API

The File Streaming API is derived from the C programming language's `<stdio.h>` interface. Any C book that explains the `<stdio.h>` interface should serve as a suitable introduction to the concepts underlying the Palm OS File Streaming API. This section provides only a brief overview of the most commonly used file streaming functions.

The `FileOpen` function opens a file, and the `FileRead` function reads it. The semantics of `FileRead` and `FileWrite` are just like their `<stdio.h>` equivalents, the `fread` and `fwrite` functions. The other `<stdio.h>` routines have obvious analogs in the File Streaming API as well.

For example,

```
theStream = FileOpen(cardId,"KillerAppDataFile",
            'KILR', 'KILD', fileModeReadOnly,
            &err);
```

As on a desktop, the filename is the unique item. The creator ID and filetype are for informational purposes and your code may require that an opened file have the correct type and creator.

Normally, the `FileOpen` function returns an error when it attempts to open or replace an existing stream having a type and creator that do not match those specified. To suppress this error, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the `FileOpen` function.

To read data, use the `FileRead` function as in the following example:

```
FileRead(theStream, &buf, objSize, numObjs,
        &err);
```

To free the memory used to store stream data as the data is read, you can use the `FileControl` function to switch the stream to destructive read mode. This mode is useful for manipulating temporary data; for example, destructive read mode would be ideal for adding the objects in a large data stream to a database when sufficient memory for duplicating the entire file stream is not available. You can switch a stream to destructive read mode by passing the `fileOpDestructiveReadMode` selector as the value of the `op` parameter to the `FileControl` function.

The `FileDmRead` function can read data directly into a Database Manager chunk for immediate addition to a PalmOS database.

File Streaming Data Structures

This section lists enumerated types used by file streaming functions.

FileOpEnum

This data type describes the file streaming operation to perform. It is passed as the value of the `op` parameter to the `FileControl` function. Normally, you do not call the `FileControl` function yourself; it is called for you by most of the other file streaming functions or macros to perform common file streaming operations. However, you may call `FileControl` explicitly to enable specialized read modes.

Listing 1.2 FileOpEnum type definition

```
typedef enum FileOpEnum {
    fileOpNone = 0,// no-op

    fileOpDestructiveReadMode,
    // Enter destructive read mode, and rewind stream to its
    // beginning. Once in this mode, there is no turning back:
    // stream's contents after closing (or crash) are undefined.
    // Destructive read mode deletes file stream data blocks as
    // data is being read, thus freeing storage automatically.
    // You cannot call FileWrite, FileSeek or FileTruncate on a
    // stream in this mode. An exception to this rule applies to
    // streams opened in "write + append" mode and then switched
    // into destructive read mode. FileWrite appends data to this
    // stream while preserving the current file position, and
    // subsequent reads pick up where they left off (you can think
    // of this feature as a pseudo-pipe).
    // ARGUMENTS:
    // stream = open stream handle
    // valueP = NULL
    // valueLenP = NULL
    // RETURNS:
    // zero on success; fileErr... on error

    fileOpGetEOFStatus,
    // get end-of-file status (like C runtime's feof)
    // (err = fileErrEOF) indicates end of file condition
    // use FileClearerr to clear this error status
    // ARGUMENTS:
    // stream = open stream handle
```

```
// valueP = NULL
// valueLenP = NULL
// RETURNS:
// zero if _not_ end of file;
// non-zero if end of file

fileOpGetLastError,
// get error code from last operation on stream, and clear the
// last error code value. Doesn’t change status of end-of-file
// or I/O errors -- use FileClearerr to reset all error codes.
// ARGUMENTS:
// stream = open stream handle
// valueP = NULL
// valueLenP = NULL
// RETURNS:
// Error code from last file stream operation

fileOpClearError,
// clear I/O and end of file error status, and last error
// ARGUMENTS:
// stream = open stream handle
// valueP = NULL
// valueLenP = NULL
// RETURNS:
// zero on success; fileErr... on error

fileOpGetIOErrorStatus,
// get I/O error status (like C runtime's ferror)
// use FileClearerr to clear this error status
// ARGUMENTS:
// stream = open stream handle
// valueP = NULL
// valueLenP = NULL
// RETURNS:
// zero if _not_ I/O error;
// non-zero if I/O error is pending

fileOpGetCreatedStatus,
// find out whether file was created by FileOpen function
// ARGUMENTS:
```

```
    // stream = open stream handle
    // valueP = ptr to Boolean type variable
    // valueLenP = ptr to Long variable set to sizeof(Boolean)
    // RETURNS:
    // zero on success; fileErr... on error;
    // the Boolean variable will be set to
    // non-zero if the file was created.

    fileOpGetOpenDbRef,
    // Get the open database reference (handle) of the underlying
    // database that implements the stream (NULL if none); this is
    // needed for performing PalmOS-specific operations on the
    // underlying database, such as changing or getting creator
    // and type,version, backup/reset bits, etc.
    // ARGUMENTS:
    // stream = open stream handle
    // valueP = ptr to DmOpenRef type variable
    // valueLenP = ptr to Long variable set to sizeof(DmOpenRef)
    // RETURNS:
    // zero on success; fileErr... on error;
    // the DmOpenRef variable will be set to the
    // file's open db reference that may be passed
    // to Data Manager calls;
    // WARNING:
    // Do not make any changes to the data of the underlying
    // database -- doing so will corrupt the file stream.

    fileOpFlush,
    // flush any cached data to storage
    // ARGUMENTS:
    // stream = open stream handle
    // valueP = NULL
    // valueLenP = NULL
    // RETURNS:
    // zero on success; fileErr... on error;

// removed system-use-only info that appears here in FileStream.h

  } FileOpEnum;
```

FileOriginEnum

This data type describes the origin of a seek operation on a file stream. It is passed as the value of the `origin` parameter to the `FileSeek` function.

Listing 1.3 FileOriginEnum type definition

```
typedef enum FileOriginEnum {
  fileOriginBeginning = 1,
  // from the beginning (first data byte of file)
  fileOriginCurrent,
  // from the current position
  fileOriginEnd
  // from the end of file (one position beyond last data byte)
} FileOriginEnum;
```

Open Mode Constants

This section lists constants passed in the `openMode` parameter to the `FileOpen` function. These constants specify the mode in which a file stream is opened.

For each file stream, you must pass to the `FileOpen` function only one of the primary mode selectors listed in Table 1.1.

Table 1.1 Primary Open Mode Constants:

Primary Selectors (use only one)	Comment
`fileModeReadOnly`	Open for read-only access
`fileModeReadWrite`	Open / create for read / write access, discarding any previous version of stream
`fileModeUpdate`	Open / create for read / write, preserving previous version of stream if it exists
`fileModeAppend`	Open / create for read / write, always writing to the end of the stream

You can use the | operator (bitwise inclusive OR) to append to a primary mode selector one or more of the secondary mode selectors listed in Table 1.2.

Table 1.2 Secondary Open Mode Constants

Secondary Selectors (append to primary)	Comment
`fileModeDontOverwrite`	Prevents `fileModeReadWrite` from discarding an existing stream having the same name; may only be specified together with `fileModeReadWrite`
`fileModeLeaveOpen`	Leave stream open when application quits. Most applications should not use this option.
`fileModeExclusive`	No other application can open the stream until the application that opened it in this mode closes it.
`fileModeAnyTypeCreator`	Accept any type/creator when opening or replacing an existing stream. Normally, the `FileOpen` function opens only streams having the specified creator and type. Setting this option enables the `FileOpen` function to open streams having a type or creator other than those specified.
`fileModeTemporary`	Delete the stream automatically when it is closed. For more information, see Comment section of `FileOpen` function description.

File Streaming Function Summary

- FileClearerr
- FileClose
- FileControl
- FileDelete
- FileDmRead
- FileEOF
- FileError
- FileFlush
- FileGetLastError
- FileOpen
- FileRead
- FileReadLow
- FileRewind
- FileSeek
- FileTell
- FileTruncate
- FileWrite

The Sound Manager

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

The sound hardware can play only one simple tone at a time through an onboard piezoelectric speaker. Note that for a particular amplitude level, the Palm III device is slightly louder than its predecessors.

Single tones can be played by the `SndDoCmd` function and system sounds are played by the `SndPlaySystemSound` function. The

end-user can control the amplitude of alarm sounds, game sounds, and system sounds by means of the Preferences application. System-supplied sounds include the Information, Warning, Error, Start-up, Alarm, Confirmation, and Click sounds.

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

The alarm sounds used in the built-in Date Book application are SMFs stored in the System MIDI Sounds database and can be played by the `SndPlaySMF` function.

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

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

- “Adding a Standard MIDI File to a Database” on page 38
- “MIDI Record Type” on page 46
- “MIDI Record Header” on page 47

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

Previous versions of PalmOS don't support SMFs or asynchronous notes; don't use the new routines or commands when the `FtrGet` function returns a system version of less than `0x03000000`. Doing so will crash your application. For more information, see the Retrieving the System Version Number section beginning on

page 51 in the "Developing Palm OS Applications" chapter of Part I of this documentation suite.

Synchronous and Asynchronous Sound

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

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

Using the Sound Manager

Before playing custom sounds that require a volume (amplitude) setting, your code needs to discover the user's current volume settings. To do so in Palm OS 3.0, pass one of the `prefSysSoundVolume`, `prefGameSoundVolume`, or `prefAlarmSoundVolume` selectors to the `PrefGetPreference` function.

Compatibility Note

See "Sound Preferences Compatibility Information" starting on page 42 for important information regarding the correct use of sound preferences in various versions of Palm OS.

You can pass the returned amplitude information to the SndPlaySMF function as one element of a SndSmfOptionsType parameter block. Alternatively, you can pass amplitude information to the SndDoCmd function as an element of a SndCommandType parameter block.

To execute a sound manager command, pass to the SndDoCmd function a sound channel pointer (presently, only `NULL` is supported and

maps to the shared channel), a pointer to a structure of `SndCommandType`, and a flag indicating whether the command should be performed asynchronously.

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

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

The system provides no specialized API for playing game sounds or alarm sounds. When an alarm triggers, the application that set the alarm must use the standard Sound Manager API to play the sound associated with that alarm. Similarly, game sounds are implemented by the game developer using any appropriate element of the Sound Manager API. Games should observe the `prefGameSoundVolume` setting, as described in the Sound Preferences Compatibility Information section starting on page 42.

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

Adding a Standard MIDI File to a Database

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

Listing 1.4 AddSmfToDatabase

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

// returns 0 for success, nonzero for error
int AddSmfToDatabase(Handle smfH, CharPtr trackName)
{
  Err       err = 0;
  DmOpenRef dbP;
  UInt      recIndex;
  VoidHand  recH;
  Byte*     recP;
  Byte*     smfP;
  Byte      bMidiOffset;
  ULong     dwSmfSize;
  SndMidiRecHdrType recHdr;

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

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

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

  // Allocate a new record for the midi resource
  recIndex = dmMaxRecordIndex;
  recH = DmNewRecord(dbP, &recIndex, dwSmfSize + bMidiOffset);
  if ( !recH )
    return 2;

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

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

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

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

  DmCloseDatabase(dbP);

  return err;
}
```

Saving References to Standard MIDI Files

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

Retrieving a Standard MIDI File From a Database

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

To obtain a particular SMF, you need to identify the database in which it resides and the specific database record which holds the SMF data. The database record itself is always identified by record ID. The MIDI database in which it resides may be identified by name or by database ID. If you know the creator of the SMF, you can

use the SndCreateMidiList utility function to retrieve this information. Alternatively, you can use the Data Manager record API functions to iterate through MIDI database records manually in search of this information.

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

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

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

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

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

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

Finally, close the database by calling the `DmCloseDatabase` function.

Sound Preferences Compatibility Information

The sound preferences implementation and API varies slightly among versions 1.0, 2.0, and 3.0 of Palm OS. This section describes how to use sound preferences correctly for various versions of Palm OS.

Because versions 2.0 and 3.0 of Palm OS provide backward compatibility with previous sound preference mechanisms, applications written for an earlier version of the sound preferences API will get correct sound preference information from newer versions of Palm OS. However, it is strongly recommended that new applications use the latest API.

Using Sound Preferences on All Palm OS Devices

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

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

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

Using Palm OS v. 1.0 Sound Prefs

To read sound preference values in version 1.0 of Palm OS, call the `PrefGetPreferences` function to obtain the data structure shown in Listing 1.5. This `SystemPreferencesTypeV10` structure holds the current values of all system-wide preferences.You must extract from this structure the values of the `sysSoundLevel` and `alarmSoundLevel` fields. These values are the only sound preference information that Palm OS version 1.0 provides.

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

Listing 1.5 `SystemPreferencesTypeV10` data structure

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

  // International preferences
  CountryType country;// Country the device is in
  DateFormatType dateFormat;// Format to display date in
  DateFormatType longDateFormat;// Format to display date in
  Byte weekStartDay;// Sunday or Monday
  TimeFormatType timeFormat;// Format to display time in
  NumberFormatType numberFormat;// Format to display numbers in

  // system preferences
  Byte autoOffDuration;// Time period before shutting off
  SoundLevelTypeV20 sysSoundLevel;//error beeps
  SoundLevelTypeV20 alarmSoundLevel;//alarm only
  Boolean hideSecretRecords;// True to not display records with
                            // their secret bit attribute set
  Boolean deviceLocked; // Device locked until the system
                        // password is entered
  WordsysPrefFlags;// Miscellaneous system pref flags copied into
                   // the global GSysPrefFlags at boot time.
```

```
  SysBatteryKindsysBatteryKind;// The type of batteries installed.
                               // This is copied into the globals
                               // GSysbatteryKind at boot time.

  } SystemPreferencesTypeV10;
```

Using Palm OS v. 2.0 Sound Prefs

Version 2.0 of Palm OS introduces a new API for retrieving individual preference values from the system. You can pass any of the selectors `prefSysSoundLevelV20`, `prefGameSoundLevelV20`, or `prefAlarmSoundLevelV20` to the `PrefGetPreference` function to retrieve individual amplitude preference values for alarm sounds, game sounds, or for overall (system) sound amplitude. As in Palm OS 1.0, each of these settings holds values of either `slOn` (on) or `slOff` (off), as defined in the Preferences.h file. Your code must interpret the values read from these fields as an indication of whether those volumes should be on or off, then map them to appropriate amplitude values to pass to Sound Manager functions: map the `slOn` selector to the `sndMaxAmp` constant (defined in SoundMgr.h file) and map the `slOff` selector to the value `0` (zero).

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

Using Palm OS v. 3.0 Sound Prefs

Palm OS version 3.0 enhances the resolution of sound preference settings by providing discrete amplitude levels for games, alarms, and the system overall. As usual, do not set preferences yourself, but treat them as read-only values indicating the proper volume level for your application to use.

Palm OS 3.0 defines the new sound amplitude selectors `prefSysSoundVolume`, `prefGameSoundVolume`, and `prefAlarmSoundVolume` for use with the `PrefGetPreference` function. The values this function returns for these selectors are actual amplitude settings that may be passed directly to Sound Manager functions.

Compatibility Note

The amplitude selectors used in previous versions of Palm OS (all ending with the `Level` suffix, such as `prefsGameSoundLevel`) are obsoleted in version 3.0 of Palm OS and replaced by new selectors. The old selectors remain available in Palm OS 3.0 to ensure backward compatibility and are suffixed `V20` (for example, `prefsGameSoundLevelV20`).

Ensuring Sound Preferences Compatibility

For greatest compatibility with multiple versions of the sound preferences mechanism, your application should condition its sound preference code according to the version of Palm OS on which it is running. Information on Retrieving the System Version Number is available on page 51 of the "Developing Palm OS Applications" chapter of Part I of this documentation suite.

When your application is launched, it should retrieve the system version number and save the results in its global variables (or equivalent structure) for use elsewhere. If the major version number is 3 (three) or greater, then use the 3.0 mechanism for obtaining sound amplitude preferences, since this reflects the user's selection most accurately. If the major version number is 2 (two), then use the 2.0 mechanism described in Using Palm OS v. 2.0 Sound Prefs starting on page 44 of this book. If it is 1 (one), then use the 1.0 mechanism described in Using Palm OS v. 1.0 Sound Prefs starting on page 43 of this book.

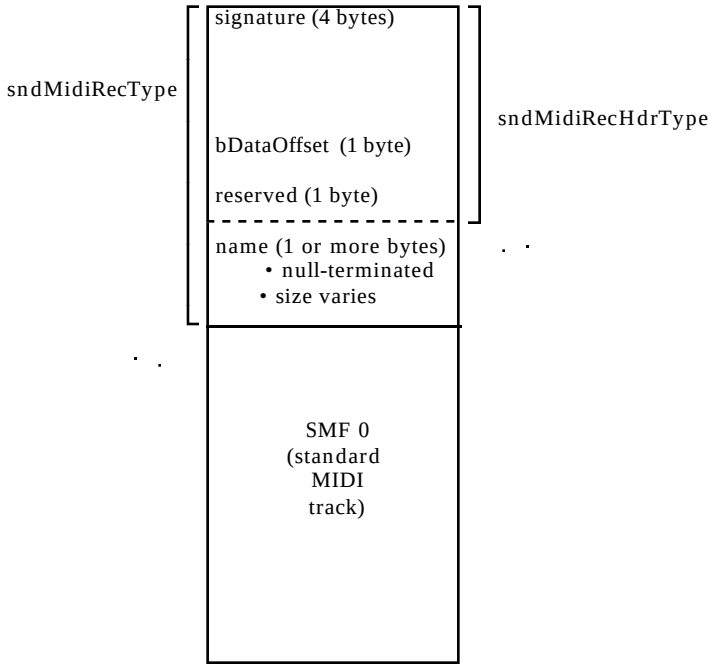
Avoid calling new API's (including new selectors) when running on older versions of Palm OS that do not implement them. In particular, note that violating any of the following conditions will cause your application to crash:

- Do not call either of the `SndPlaySMF` or `SndCreateMidiList` functions on versions of PalmOS prior to 3.0.
- Do not pass any selector other than `sndCmdFreqDurationAmp` to the `SndDoCmd` function on versions of PalmOS prior to 3.0.

Sound Manager Data Structures

This section describes the data structures that define the MIDI records and parameter blocks used by sound manager functions. Figure 1.1 depicts a Palm OS MIDI record graphically.

Figure 1.1 Palm OS Midi Record



MIDI Record Type

This variable-length header precedes the actual MIDI data in a PalmOS MIDI record. It consists of a fixed-size MIDI Record Header followed by the name of the MIDI track.

Listing 1.6 SndMidiRecType structure

```
typedef struct SndMidiRecType {
  SndMidiRecHdrType   hdr;
  // fixed-size portion of the Palm OS MIDI record header
  Char  name[1];
  // Track name: 1 or more chars including NULL terminator.
  // length of name, including NULL terminator, must not be
```

```
  // greater than sndMidiNameLength. The NULL character must
  // always be provided, even for tracks that have no name
  } SndMidiRecType;
```

MIDI Record Header

This structure defines the fixed-size portion of a Palm OS MIDI record.

Listing 1.7 SndMidiRecHdrType structure

```
typedef struct SndMidiRecHdrType {
  DWord signature;
  // set to sndMidiRecSignature
  Byte  bDataOffset;
  // offset from the beginning of the record
  // to the Standard Midi File data stream
  Byte  reserved;
  // set to zero
  } SndMidiRecHdrType;
```

SndMidiListItemType

When the SndCreateMidiList function returns TRUE, its entHP parameter holds a handle to a memory chunk containing an array of SndMidiListItemType structs.

Listing 1.8 SndMidiListItemType structure

```
typedef struct SndMidiListItemType{
  Char  name[sndMidiNameLength];
  // including NULL terminator
  ULong   uniqueRecID;
  LocalID dbID;
  UInt    cardNo;
  } SndMidiListItemType;
```

SndCommandType

This structure is passed as the value of the `cmdP` parameter to the `SndDoCmd` function. Its parameters are defined by the `SndCmdIDType` enumerated constant.

Listing 1.9 SndCommandType structure

```
typedef struct SndCommandType {
  SndCmdIDType    cmd;
  // command id
  Long  param1;
  // use varies according to value of cmd
  UInt  param2;
  // use varies according to value of cmd
  UInt  param3;
  // use varies according to value of cmd
} SndCommandType;
```

SndCmdIDType

This enumerated type defines the commands that may be specified in the `cmd` field of the `SndCommandType` struct. Each command defines its own specific use of the `param1`, `param2`, and `param3` fields.

Listing 1.10 SndCmdIDType type definition

```
typedef enum SndCmdIDType {
  sndCmdFreqDurationAmp = 1,
  // play a sound, blocking for the entire
  // duration (except for zero amplitude)
  // param1 = frequency in Hz
  // param2 = duration in milliseconds
  // param3 = amplitude (0 - sndMaxAmp);
  // if value of param3 is 0,return immediately

  // Commands added in Palm OS v3.0
  // ***IMPORTANT***
  // Please note that SndDoCmd() in Palm OS before v3.0 will
  // Fatal Error on unknown commands (anything other than
```

```
// sndCmdFreqDurationAmp).  For this reason, applications
// wishing to take advantage of these new commands while staying
// compatible with the earlier version of the OS, _must_ avoid
// using these commands when running on OS versions less than
// v3.0 (see sysFtrNumROMVersion in SystemMgr.h).
// Beginning with v3.0, SndDoCmd has been fixed to return
// sndErrBadParam when an unknown command is passed.

sndCmdNoteOn,
// play sound at specified MIDI key index
// with max duration and velocity;
// return immediately, without waiting for playback to complete.
// any other sound play request made before
// this one completes will interrupt it.
// param1 = MIDI key index (0-127)
// param2 = maximum duration in milliseconds
// param3 = velocity (0 - 127) to be interpolated as amplitude

sndCmdFrqOn,
// similar to sndCmdNoteOn except note to play
// is specified as frequency in Hz.
// play sound at specified frequency
// with max duration and velocity;
// return immediately, without waiting for playback to complete
// any other sound play request made before
// this one completes will interrupt it.
// param1 = frequency in Hz
// param2 = maximum duration in milliseconds
// param3 = amplitude (0 - sndMaxAmp)

sndCmdQuiet
// stop playback of current sound
// param1 = 0
// param2 = 0
// param3 = 0

} SndCmdIDType;
```

SndSmfOptionsType

This struct is passed as the value of the `selP` parameter to the `SndPlaySMF` function.

```
typedef struct SndSmfOptionsType {
  // dwStartMilliSec and dwEndMilliSec are used as inputs to the
  // fn for sndSmfCmdPlay and as outputs for sndSmfCmdDuration

  DWord       dwStartMilliSec;
  // position at which to begin playback, expressed as number of
  // milliseconds from beginning of track
  // 0 = "start from the beginning"

  DWord       dwEndMilliSec;
  // position at which to stop playback, expressed as number of
  // milliseconds from beginning of track
  // sndSmfPlayAllMilliSec = "play entire track";
  // the default is "play entire track"
  // if this structure is not passed in

  UInt        amplitude;
  // The amplitude and interruptible fields
  // are used only for sndSmfCmdPlay
  // relative volume: 0 - sndMaxAmp, inclusively
  // the default is sndMaxAmp if this structure
  // is not passed in; if 0, the play will be
  // skipped and the call will return immediately

  Boolean     interruptible;
  // If true, sound play will be interrupted if user interacts
  // with the controls (digitizer, buttons, etc.) even if the
  // interaction does not generate a sound command. If false,
  // playback is not interrupted; the default behavior is
  // "interruptible" if this structure is not passed in

  DWord       reserved;
  // RESERVED! -- MUST SET TO ZERO BEFORE PASSING
  } SndSmfOptionsType;
```

SndSmfChanRangeType

This struct is passed as the value of the `chanRangeP` parameter to the `SndPlaySMF` function.

Listing 1.11 SndSmfChanRangeType structure

```
typedef struct SndSmfChanRangeType {
// specifies a range of enabled channels.
// events for channels outside this range are ignored.
// if this structure is not passed,
// all channels in track are honored.
  Byte    bFirstChan;
  // first MIDI channel (0-15 decimal)
  Byte    bLastChan;
  // last MIDI channel (0-15 decimal)}
SndSmfChanRangeType;
```

Sound Callback Functions

These structures define callback functions to be executed by the `SndPlaySMF` function.

A non-null completion callback function is executed after playback of the SMF completes.

```
typedef void SndComplFuncType(void* chanP, DWord dwUserData);
typedef SndComplFuncType* SndComplFuncPtr;
```

A non-null blocking callback function is executed periodically during playback of the SMF. This function returns `TRUE` to continue playback, or `FALSE` to cancel playback. Suggested uses for this function include updating the user interface or checking for user input. You can test `sysTicksAvailable` to determine the maximum amount of time available for completion of this function.

```
typedef Boolean SndBlockingFuncType(void* chanP, DWord dwUserData,
                                    Long sysTicksAvailable);
typedef SndBlockingFuncType* SndBlockingFuncPtr;
```

Both kinds of callbacks are wrapped in a `SndCallbackInfoType` struct.

```
typedef struct SndCallbackInfoType {
  Ptr   funcP;
  // pointer to the callback function (NULL = no function)
  DWord dwUserData;
  // value to pass in dwUserData parameter of callback function
} SndCallbackInfoType;
```

The `SndSmfCallbacksType` struct is passed as the value of the `callbacksP` parameter to the `SndPlaySMF` function.

```
typedef struct SndSmfCallbacksType {
  SndCallbackInfoType completion;
  // completion callback function (see SndComplFuncType)
  SndCallbackInfoType blocking;
  // blocking hook callback function (see SndBlockingFuncType)
  SndCallbackInfoType reserved;
  // RESERVED -- SET ALL FIELDS TO ZERO BEFORE PASSING
} SndSmfCallbacksType;
```

Sound Manager Function Summary

The following sound manager functions are available for application use:

- SndCreateMidiList
- SndDoCmd
- SndGetDefaultVolume
- SndPlaySMF
- SndPlaySystemSound

The String Manager

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

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

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

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

String Manager Function Summary

The following functions are available for application use:

- StrAToI
- StrCat
- StrCaselessCompare
- StrChr
- StrCompare
- StrCopy
- StrIToA
- StrIToH
- StrLen
- StrStr
- StrToLower

The System Manager

The Palm OS system manager is responsible for the general operation of the system, including boot-up, power-up, launching applications, library management, monitoring the battery, multitasking, timing, and semaphore support. Applications need to be concerned with very few system manager API functions. Most of what the system manager does is transparent to applications and is explained here as background information only.

In this section, you learn about the following aspects of the system manager:

- System Boot and Reset — information about the different reset operations, including system reset calls
- Power Management — the three different power modes and guidelines for application developers
- The Microkernel— basic task management provided by the system
- Application Support — event processing and interapplication communication from the system's point of view
- System Manager Function Summary — list of all system manager functions available to applications

System Boot and Reset

The system manager provides support for booting the Palm OS device. Booting occurs only when the user presses the reset switch on the device (see "Palm OS Device Reset Switch" in Developing Palm OS Applications, Part I). Palm OS differs from a traditional desktop system in that it's never really turned off. Power is constantly supplied to essential subsystems and the on/off key is merely a way of bringing the device in or out of low-power mode (see Palm OS Power Modes). The obvious effect of pressing the on/off key is that the LCD turns on or off. When the user presses the power key to turn the device off, the LCD is disabled, which makes it appear as if power to the entire unit is turned off. In fact, the memory system, real-time clock, and interrupt generation circuitry are still running, though they are consuming little current.

In this version of Palm OS, there is only one user interface application running at a time. The User Interface Application Shell (UIAS) is responsible for managing the current user-interface application. The UIAS launches the current user-interface application as a subroutine and doesn't get control back until that application quits. When control returns to the UIAS, the UIAS immediately launches the next application as another subroutine. See Power Management Calls for more information.

System Reset Calls

The system calls `SysReset` to reset the device. This call does a soft reset and has the same effect as pressing the reset switch on the unit. **Normally, applications should not use this call**.

`SysReset` is used, for example, by the Sync application. When the user copies an extension onto the Palm OS device, the Sync application automatically resets the device after the sync is completed to allow the extension to install itself.

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

Power Management

This section looks at Palm OS power management, discussing the following topics:

- Palm OS Power Modes
- Guidelines for Application Developers
- Power Management Calls

Palm OS Power Modes

At any time, the Palm OS device is in one of three power modes: sleep, doze, or running. The system manager controls transitions between different power modes and provides an API for controlling some aspects of the power management.

- **Sleep mode**. If the unit appears to be off, it is actually in sleep mode and is consuming as little current as possible. At this rate, a unit could sit for almost a year on a single set of

batteries without losing the contents of memory. To enter sleep mode, the system puts as many peripherals as possible into low-power mode and sets up the hardware so that an interrupt from any hard key or the real-time clock wakes up the system.

When the system gets one of these interrupts while in sleep mode, it quickly checks that the battery is strong enough to complete the wake-up and then takes each of the peripherals, for example, the LCD, serial port, and timers, out of low-power mode.

The system reenters sleep mode when the user presses the on/off key again, when the system has been idle for the minimum auto-off time, or when the battery level reaches a critically low level.

- **Doze mode**. In doze mode, the processor is halted, but all peripherals including the LCD are powered up. The system can come out of doze mode much faster than it can come out of sleep mode since none of the peripherals need to be woken up. In fact, it takes no longer to come out of doze mode than to process an interrupt. Usually, when the system appears on, it is actually in doze mode and goes into running mode only for short periods of time to process an interrupt or respond to user input like a pen tap or key press.
- **Running mode**. Running means that the processor is executing instructions and all peripherals are powered up. A typical application puts the system into running mode only about 5% of the time.

Guidelines for Application Developers

Normally, applications don't need to be aware of power management except for a few simple guidelines. When an application calls `EvtGetEvent` to ask the system for the next event to process, the system automatically puts itself into doze mode until there is an event to process. As long as an application uses `EvtGetEvent`, power management occurs automatically. If there has been no user input for the amount of time determined by the current setting of the auto-off preference, the system automatically enters sleep mode without intervention from the application.

Applications should avoid providing their own delay loops. Instead, they should use `SysTaskDelay`, which puts the system into doze mode during the delay to conserve as much power as possible.

If an application needs to perform periodic work, it can pass a time out to `EvtGetEvent`; this forces the unit to wake up out of doze mode and to return to the application when the time out expires, even if there is no event to process. Using these mechanisms provides the longest possible battery life.

Power Management Calls

The system calls `SysSleep` to put itself immediately into low-power sleep mode. Normally, the system puts itself to sleep when there has been no user activity for the minimum auto-off time or when the user presses the power key.

The `SysSetAutoOffTime` routine changes the auto-off time value. This routine is normally used by the system only during boot, and by the Preferences application. The Preferences application saves the user preference for the auto-off time in a preferences database, and the system initializes the auto-off time to the value saved in the preferences database during boot. While the auto-off feature can be disabled entirely by calling `SysSetAutoOffTime` with a time-out of 0, doing this depletes the battery.

The current battery level and other information can be obtained through the `SysBatteryInfoV20` routine. This call returns information about the battery, including the current battery voltage in hundredths of a volt, the warning thresholds for the low-battery alerts, the battery type, and whether external power is applied to the unit. This call can also change the battery warning thresholds and battery type.

The Microkernel

Palm OS has a preemptive multitasking kernel that provides basic task management.

Most applications don't need the microkernel services because they are handled automatically by the system. This functionality is provided mainly for internal use by the system software or for certain special purpose applications.

The User Interface Application Shell (UIAS) is responsible for managing the current user-interface application, as described in System Boot and Reset.

Usually, the UIAS is the only task running. Occasionally though, an application launches another task as a part of its normal operation. One example of this is the Sync application, which launches a second task to handle the serial communication with the desktop. The Sync application creates a second task dedicated to the serial communication and gives this task a lower priority than the main user-interface task. The result is optimal performance over the serial port without a delay in response to the user-interface controls.

Normally, there is no user interaction during a sync, so that the serial communication task gets all of the processor's time. However, if the user does tap on the screen, for example, to cancel the sync, the user-interface task immediately processes the tap, since it has a higher priority. Alternatively, the Sync application could have been written to use just one task, but then it would have to periodically poll for user input during the serial communication, which would hamper performance and user-interface response time.

Application Support

The system manager provides application support in several functional areas. The following aspects of application support are discussed in this section:

- Launching and Cleanup
- Event Processing
- Interapplication Communication
- Retrieving Events
- Opening Applications Programmatically

Launching and Cleanup

Usually, applications on the Palm OS device are launched when the user presses one of the buttons on the case or selects an application icon from the application launcher screen. Alternatively, an application can programmatically launch another application by using the system manager function SysAppLaunch.

When the current user-interface application quits, the system manager cleans up by deleting any chunks in the dynamic heap(s) that the application left around and closing any databases left open.

Note, however, that applications should perform those kinds of cleanup tasks themselves.

Event Processing

The system manager provides the infrastructure for event generation and also contains the support for handling most system-related events. Hardware activity, such as taps on the digitizer and key presses, is interpreted by interrupt handlers of the system manager and converted into events that are eventually sent to the application through the `EvtGetEvent` call. In addition, many events returned by `EvtGetEvent` are system-related events that can be processed by the system manager call `SysHandleEvent`.Events in Palm OS include hardware- and software-generated events. The following table provides an overview:

Hardware-generated events	Software-generated events
Caused directly by user interaction with the device, such as tapping on the screen with the pen, or pressing a hardware button.	Generated by the system software as a side effect of a user interaction.
Include pen-downs, pen-ups (optionally including stroke data), and hard button presses.	Include events like the quit event that causes an application to exit, or keyboard events generated by the Graffiti recognizer. Applications can define software-generated events for their own use.
Typically posted by interrupt routines.	Typically posted as the result of a system call. Include application-quit events, window-enter and window-exit events, user-interface control events, etc.
• Pen-generated events are stored in the pen queue. • Hard button press events are stored in the key queue.	Stored in the software event queue.

When `EvtGetEvent` is called by the application, it first checks whether any events are in the software event queue and returns the topmost event if so.

If the software event queue is empty, `EvtGetEvent` checks the key and pen queues. The result is that all software events generated by a particular hardware event are processed before the next hardware event is processed. For example, a pen-down hardware event may trigger the system software to generate window-exit and window-enter software events. Both events are then pulled from the software event queue and processed before the next hardware event is processed.

Some event types returned by `EvtGetEvent` are not actually posted into the event queue, but are artificially generated by `EvtGetEvent` when all event queues are empty. One example is the pen-moved event, which is returned if no other events are in the queues and the pen has moved since the last time `EvtGetEvent` was called. In this way, the application is notified of low-priority events, such as pen movements, but the event queue isn't cluttered with them.

In a typical application, `SysHandleEvent` is called immediately after `EvtGetEvent`. If `EvtGetEvent` returns a pen-up event in the Graffiti writing area, `SysHandleEvent` calls the Graffiti recognizer with the pen stroke information obtained from the pen queue and uses the results of the Graffiti recognizer to post one or more keyboard events into the key queue. A similar process occurs for pen-up events detected over a silk-screened icon. `SysHandleEvent` converts the pen-up to a keyboard event with a virtual key code representing the silk-screened icon.

When an application calls `EvtGetEvent`, the event manager checks a number of system-event data structures and returns an event record to the application with information about the highest-priority event that needs processing. Events in Palm OS are stored in one of three event queues: a key queue, a pen queue, or a software event queue. The event queues are circular buffers containing event records stored in a first-in, first-out (FIFO) sequence.

Here's some additional information on hardware and software events:

- **Hardware events** are posted into their appropriate event queue by interrupt routines. The interrupt routine for handling key-

board presses immediately enqueues the keyboard event into the key queue and sets up a periodic interrupt routine to watch for auto-repeat and for key debouncing.

- **Software-generated** events include window-enter and window-exit events, application quit events, and user-interface object events like control enter, control exit, etc. These events are typically generated as a side effect of a hardware-generated event like a pen-down. Software can, however, also generate key events, usually as a result of recognizing a Graffiti stroke or a tap on a silk-screened icon.

  Software-generated events are posted into the appropriate event queue, but are not typically posted at interrupt time. Many of these events are inserted into the event queue by the various user-interface managers. Others, like key events, are posted by `SysHandleEvent` after recognizing a Graffiti stroke or a tap on a silk-screened icon.

Interapplication Communication

The system manager provides the API for interapplication communication. This API permits any application or system routine to send a **launch code** to any other application and get results back. For example, an application that is to work with the global find must support the find launch code.

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.

Predefined launch codes are listed in "Developing Palm OS Applications, Part I" and can be found in `SystemMgr.h`. All the parameters for a launch code are passed in a single parameter block, and the results are returned in the same parameter block. "How Launch Codes Control an Application" in "Developing Palm OS Applications, Part I, describes launch codes in more detail.

Retrieving Events

The `SysHandleEvent` call allows applications to correctly respond to system events like key presses, Graffiti strokes, low-battery warnings, and taps on silk-screened icons. Every application should call this routine from its event loop, usually before the application even

looks at the event. If an application needs to override any part of the default system behavior, it could selectively filter out events before calling SysHandleEvent.

Opening Applications Programmatically

The system provides several APIs for opening applications programmatically. Under most circumstances, you would use the SysUIAppSwitch routine to close your application and open a specified application. This routine notifies the system which application to launch next and feeds an application-quit event into the event queue. If and when the current application responds to the quit event and returns, the system launches the new application.

When you want to make use of another application's functionality and eventually return control of the system to your application, you can use the SysAppLaunch function to open a specified application as a subroutine of the calling application. It has numerous options, including whether to launch the application as a separate task, whether to allocate a globals world, and whether or not to give the called application its own stack. For example, you would use this function to request that the built in Address List application search its databases for a specified phone number and return the results of the search to your application. You could then call SysAppLaunch again to use the modem handle to dial the number. (In fact, this is how the built-in applications perform this task.) When calling SysAppLaunch do not set Launch Flags yourself—the SysAppLaunch function sets launch flags appropriately for you.

This routine is also used to send launch codes to applications (by telling it to use the caller's stack, no globals world, and not a separate task). Usually, applications use it only for sending launch codes to other user-interface applications. An alternative, simpler method of sending launch codes is the SysBroadcastActionCode call. This routine automatically finds all other user-interface applications and calls SysAppLaunch to send the launch code to each of them.

If your application is called to process a launch code, it is called as a subroutine from the current user-interface application. Use the routine SysCurAppDatabase to get the card number and database ID of the currently running user-interface application. This routine

doesn't return your application's database ID but the database ID of the application that initiated the launch code.

Palm OS 3.0 also provides a new application from which the end user can launch any application installed on the Palm OS device. For more information, see "Application Launcher" on page 70.

WARNING: Do not use the `SysUIAppSwitch` or `SysAppLaunch` functions to open the Application Launcher application.

System Manager Function Summary

The following system manager functions are available for application use:

- SysReset
- SysBatteryInfoV20
- SysSetAutoOffTime
- SysHandleEvent
- SysUIAppSwitch
- SysCurAppDatabase
- SysBroadcastActionCode
- SysAppLaunch

The System Event Manager

The system event manager

- Manages the low-level pen and key event queues.
- Translates taps on silk-screened icons into key events.
- Sends pen strokes in the Graffiti area to the Graffiti recognizer.
- Puts the system into low-power doze mode when there is no user activity.

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

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

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

- Event Translation: Pen Strokes to Key Events
- Pen Queue Management
- Auto-Off Control
- System Event Manager Function Summary

Event Translation: Pen Strokes to Key Events

One of the higher-level functions provided by the system event manager is conversion of pen strokes on the digitizer to key events. For example, the system event manager sends any stroke in the Graffiti area of the digitizer automatically to the Graffiti recognizer for conversion to a key event. Taps on silk-screened icons, such as the application launcher, Menu button, and Find button, are also intercepted by the system event manager and converted into the appropriate key events.

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

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

Eventually, the system returns the key event to the application as a normal result of calling `EvtGetEvent`.

Most applications rely on the following default behavior of the system event manager:

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

Pen Queue Management

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

The following table summarizes pen management.

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

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

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

Key Queue Management

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

- As a direct result of the user pressing one of the buttons on the case
- As a side effect of the user drawing a Graffiti stroke on the digitizer, which is converted in software to a key event

The following table summarizes key management:

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

The system event manager provides an API for initializing and flushing the key queue and for enqueuing and dequeuing key events. Usually, applications have no need to dequeue key events; the event manager does this automatically if it detects a key in the queue and returns a `keyDownEvent` (documented in "Developing Palm OS Applications," Part I) to the application through the `EvtGetEvent` call.

Auto-Off Control

Because the system event manager manages hardware events like pen taps and hardware button presses, it's responsible for resetting the auto-off timer on the device. Whenever the system detects a hardware event, it automatically resets the auto-off timer to 0. If an application needs to reset the auto-off timer manually, it can do so through the system event manager call `EvtResetAutoOffTimer`.

System Event Manager Function Summary

The following functions are part of the developer API to the system event manager:

- EvtAddEventToQueue
- EvtCopyEvent
- EvtDequeuePenPoint
- EvtDequeuePenStrokeInfo
- EvtEnableGraffiti
- EvtEnqueueKey
- EvtFlushKeyQueue
- EvtFlushNextPenStroke
- EvtFlushPenQueue
- EvtGetEvent
- EvtGetPen
- EvtKeyQueueEmpty
- EvtKeyQueueSize
- EvtKeyQueueEmpty
- EvtGetPenBtnList
- EvtPenQueueSize
- EvtProcessSoftKeyStroke
- EvtResetAutoOffTimer
- EvtWakeup

The Time Manager

The date and time manager (called time manager in this chapter) provides access to both the 1-second and 0.01-second timing resources on the Palm OS device.

- The 1-second timer keeps track of the real-time clock (date and time), even when the unit is in sleep mode.
- The 0.01-second timer, also referred to as the **system ticks**, can be used for finer timing tasks. This timer is not updated when the unit is in sleep mode and is reset to 0 each time the unit resets.

The basic time-manager API provides support for setting and getting the real-time clock in seconds and for getting the current system ticks value (but not for setting it). The system manager provides more advanced functionality for setting up a timer task that executes periodically or in a given number of system ticks.

This section discusses the following topics:

- Using Real-Time Clock Functions
- Using System Ticks Functions
- Time Manager Function Summary

Using Real-Time Clock Functions

The real-time clock functions of the time manager include `TimSetSeconds` and `TimGetSeconds`. Real time on the Palm OS device is measured in seconds from midnight, Jan 1, 1904. Call `TimSecondsToDateTime` and `TimDateTimeToSeconds` to convert between seconds and a structure specifying year, month, day, hour, minute, and second.

Using System Ticks Functions

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

- 100 times per second when running on the Palm OS device
- 60 times per second when running on the Macintosh under the Simulator

For tick-based timing purposes, applications should use the macro `sysTicksPerSecond`, which is conditionally compiled for different platforms. Use the function `TimGetTicks` to read the current tick count.

Although the `TimGetTicks` function could be used in a loop to implement a delay, it is recommended that applications use the `SysTaskDelay` function instead. The `SysTaskDelay` function automatically puts the unit into low-power mode during the delay. Using `TimGetTicks` in a loop consumes much more current.

Time Manager Structures

The time manager uses these structures to store information.

Listing 1.12 Time Manager Structures

```
typedef struct{
Sword second;
Sword minute;
Sword hour;
Sword day;
Sword month;
Sword year;
Sword weekDay;      //Days since Sunday (0 to 6)
}DateTimeType;
typedef DateTimeType* DateTimePTr;

typedef struct {
Byte  hours;
Byte  minutes;
}TimeType;
typedef TimeType * TimePtr;

typedef struct{
Word year :7; //years since 1904 (Mac format)
Word month:4;
Word day  :5;
}DateType;
typedef DateType * DatePtr;
```

Time Manager Function Summary

The following time manager functions are available for application use:

- DateAdjust
- DateDaysToDate
- DateSecondsToDate
- DateToAscii
- DateToDays
- DateToDOWDMFormatf
- DayOfMonth
- DayOfWeek
- DaysInMonth
- TimAdjust
- TimDateTimeToSeconds
- TimGetSeconds
- TimGetTicks
- TimSecondsToDateTime
- TimSetSeconds
- TimeToAscii

Note that two functions associated with the Date and Time object, `SelectDay` and `SelectTime` are documented in *Developing Palm OS Applications Part I.*

Application Launcher

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

Compatibility Note

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

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

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

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

Situations in which you need to open the Application Launcher programmatically are rare, but the system does provide an API for doing so. To activate the Launcher from within your application, enqueue a `keyDownEvent` that contains a `launchChr`, as shown in Listing 1.13.

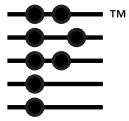
WARNING: Do not use the `SysUIAppSwitch` or `SysAppLaunch` functions to open the Application Launcher application.

Listing 1.13 Opening the Launcher

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

For information on launching other applications programmatically, see "Opening Applications Programmatically" on page 62.

2



Palm OS System Functions

Alarm Manager API

AlmGetAlarm

Purpose Return the alarm date/time in seconds since 1/1/1904 and the caller-defined alarm reference value for the given application.

Prototype

```
ULong AlmGetAlarm (  UInt cardNo,
                     LocalID dbID,
                     DWordPtr refP)
```

Parameters

-> `cardNo` Storage card number of the application.

-> `dbID` Local ID of the application.

<-> `refP` Pointer to location for the alarm's reference value.

Result Alarm seconds since 1/1/1904; if no alarm is active for the application, 0 is returned for the alarm seconds and the reference value is undefined.

AlmSetAlarm

Purpose

Set or cancel an alarm for the given application.

Prototype

```
Err AlmSetAlarm (  UInt cardNo,
                   LocalID dbID,
                   DWord ref,
                   ULong alarmSeconds,
                   Boolean quiet)
```

Parameters

-> `cardNo`	Storage card number of the application.
-> `dbID`	Local ID of the application.
-> `ref`	Caller-defined value to be passed with notifications.
-> `alarmSeconds`	Alarm date/time in seconds since 1/1/1904, or 0 to cancel the current alarm (if any).
-> `quiet`	Reserved for future upgrade (set to zero).

Result

`0`	No error.
`almErrMemory`	Insufficient memory.
`almErrFull`	Alarm table is full.

Comments

If an alarm for this application has already been set, it is replaced with the new alarm. Action code notifications are sent after the alarm is triggered and can be used by the application to set the next alarm.

Functions for System Use Only

AlmAlarmCallback

Prototype

```
void  AlmAlarmCallback (void)
```

WARNING: This function for use by system software only.

AlmCancelAll

Prototype

```
void  AlmCancelAll (Boolean enable)
```

WARNING: This function for use by system software only.

AlmDisplayAlarm

Prototype

```
void  AlmDisplayAlarm (Boolean displayOnly)
```

WARNING: This function for use by system software only.

AlmEnableNotification

Prototype

```
void AlmEnableNotificatio(Boolean enable)
```

WARNING: This function for use by system software only.

AlmInit

Prototype

```
Err  AlmInit (void)
```

WARNING: This function for use by system software only.

Error Manager Functions

ErrDisplay

Purpose Display an error alert if error checking is set to partial or full.

Prototype

```
void ErrDisplay (char* message)
```

Parameters -> `message` Error message text.

Result No return value.

Comments Call this routine to display an error message, source code filename, and line number. This routine is actually a macro that is compiled into the code only if the compiler define `ERROR_CHECK_LEVEL` is set to 1 or 2 (`ERROR_CHECK_PARTIAL` or `ERROR_CHECK_FULL`).

See Also `ErrFatalDisplayIf`, `ErrNonFatalDisplayIf`, "Using the Error Manager Macros."

ErrDisplayFileLineMsg

Purpose

Display a nonexitable dialog with an error message. Do not allow the user to continue.

Prototype

```
void ErrDisplayFileLineMsg(  CharPtr filename,
                             UInt lineno,
                             CharPtr msg)
```

Parameters

`filename`	Source code filename.
`lineno`	Line number in the source code file.
`msg`	Message to display.

Result

Never returns.

Comment

Called by `ErrFatalDisplayIf` and `ErrNonFatalDisplayIf`. This function is useful when the application is already on the device and being tested by users.

See Also

`ErrFatalDisplayIf`, `ErrNonFatalDisplayIf`, `ErrDisplay`

ErrFatalDisplayIf

Purpose

Display an error alert dialog if `condition` is `TRUE` and error checking is set to partial or full.

Prototype

```
void ErrFatalDisplayIf ( Boolean condition,
                         char* message)
```

Parameters

-> `condition` If `TRUE`, display the error.

-> `message` Error message text.

Result

No return value.

Comments

Call this routine to display a fatal error message, source code filename, and line number. The alert is displayed only if `condition` is `TRUE`. The dialog is cleared only when the user resets the system by responding to the dialog.

This routine is actually a macro that is compiled into the code if the compiler define `ERROR_CHECK_LEVEL` is set to 1 or 2 (`ERROR_CHECK_PARTIAL` or `ERROR_CHECK_FULL`).

See Also

`ErrNonFatalDisplayIf`, `ErrDisplay`, "Using the Error Manager Macros."

ErrNonFatalDisplayIf

Purpose Display an error alert dialog if `condition` is `TRUE` and error checking is set to full.

Prototype

```
void ErrNonFatalDisplayIf ( Boolean condition,
                            char* message)
```

Parameters

-> `condition` If `TRUE`, display the error.

-> `message` Error message text.

Result No return value.

Comments Call this routine to display a nonfatal error message, source code filename, and line number. The alert is displayed only if `condition` is `TRUE`. The alert dialog is cleared when the user selects to continue (or resets the system).

This routine is actually a macro that is compiled into the code only if the compiler define `ERROR_CHECK_LEVEL` is set to 2 (`ERROR_CHECK_FULL`).

See Also `ErrFatalDisplayIf`, `ErrDisplay`, "Using the Error Manager Macros."

ErrThrow

Purpose Cause a jump to the nearest Catch block.

Prototype
```
void ErrThrow (Long err)
```

Parameters
`err` Error code.

Result Never returns.

Comments Use the macros `ErrTry`, `ErrCatch`, and `ErrEndCatch` in conjunction with this function.

See Also `ErrFatalDisplayIf`, `ErrNonFatalDisplayIf`, `ErrDisplay`, "Using the Error Manager Macros."

Event Manager Functions

EvtAddEventToQueue

Purpose Add an event to the event queue.

Prototype
```
void EvtAddEventToQueue (EventPtr event)
```

Parameters
`event` Pointer to the structure that contains the event.
`error` Pointer to any error encountered by this function.

Result Returns nothing.

EvtAddUniqueEventToQueue

Purpose Look for an event in the event queue of the same event type and ID (if specified). The routine replaces it with the new event, if found.

- If no existing event is found, the new event is added.
- If an existing event is found, the routine proceeds as follows:
  - if `inPlace` is `TRUE`,the existing event is replaced with the new event
  - if `inPlace` is `FALSE`, the existing event is removed and the new event will be added to the end

Prototype

```
void EvtAddUniqueEventToQueue
( EventPtr eventP,DWord id, Boolean inPlace)
```

Parameters

`eventP`	Pointer to the structure that contains the event
`id`	ID of event. 0 means match only on the type.
`inPlace`	If `TRUE`, existing event are replaced. If `FALSE`, existing event is deleted and new event added to end of queue.

Result Returns nothing.

EvtCopyEvent

Purpose Copy an event.

Prototype

```
void EvtCopyEvent (EventPtr source, EventPtr dest)
```

Parameters

`source`	Pointer to the structure containing the event to copy.
`dest`	Pointer to the structure to copy the event to.

Result Returns nothing.

EvtDequeuePenPoint

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

Prototype

```
Err EvtDequeuePenPoint( PointType* retP)
```

Parameters

`retP` Return point.

Result Always returns 0.

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`.

See Also `EvtDequeuePenStrokeInfo`

EvtDequeuePenStrokeInfo

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

Prototype

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

Parameters

`startPtP` Start point returned here.

`startPtP` End point returned here.

Result Always returns 0.

Comments Called by the system function `EvtGetSysEvent`. 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.

See Also `EvtDequeuePenPoint`

EvtEnableGraffiti

Purpose Set Graffiti enabled or disabled.

Prototype

```
void EvtEnableGraffiti (Boolean enable)
```

Parameters

`enable` TRUE to enable Graffiti, FALSE to disable Graffiti.

Result Returns nothing.

EvtEnqueueKey

Purpose Place keys into the key queue.

Prototype

```
Err EvtEnqueueKey (  UInt ascii,
                     UInt keycode,
                     UInt modifiers)
```

Parameters

`ascii`	ASCII code of key.
`keycode`	Virtual key code of key.
`modifiers`	Modifiers for key event.

Result Returns 0 if successful, or `evtErrParamErr` if an error occurs.

Comments Called by the keyboard interrupt routine and the Graffiti and SoftKeys recognizers. Note that because both interrupt- and noninterrupt-level code can post keys into the queue, this routine disables interrupts while the queue header is being modified.

Most keys in the queue take only 1 byte if they have no modifiers and no virtual key code, and are 8-bit ASCII. If a key event in the queue has modifiers or is a non-standard ASCII code, it takes up to 7 bytes of storage and has the following format:

evtKeyStringEscape	1 byte
ASCII code	2 bytes
virtual key code	2 bytes
modifiers	2 bytes

EvtEventAvail

Purpose Return TRUE if an event is available.

Prototype
```
Boolean EvtEventAvail (void)
```

Parameters None

Result Returns TRUE if an event is available, FALSE otherwise.

EvtFlushKeyQueue

Purpose Flush all keys out of the key queue.

Prototype
```
Err EvtFlushKeyQueue (void)
```

Parameters None.

Result Always returns 0.

Comments Called by the system function `EvtSetPenQueuePtr`.

EvtFlushNextPenStroke

Purpose Flush the next stroke out of the pen queue.

Prototype
```
Err EvtFlushNextPenStroke (void)
```

Parameters None

Result Always returns 0.

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

Purpose Flush all points out of the pen queue.

Prototype
```
Err EvtFlushPenQueue (void)
```

Parameters None

Result Always returns 0.

Comments Called by the system function `EvtSetKeyQueuePtr`.

See Also `EvtPenQueueSize`

EvtGetEvent

Purpose Return the next available event.

Prototype
```
void EvtGetEvent (EventPtr event, Long timeout)
```

Parameters

`event` Pointer to the structure to hold the event returned.

`timeout` Maximum number of ticks to wait before an event is returned (-1 means wait indefinitely).

Comments Pass `timeout`= -1 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 `timeout` >= 0.

Result Returns nothing.

EvtGetPen

Purpose Return the current status of the pen.

Prototype
```
void EvtGetPen(  Sword *pScreenX,
                 Sword *pScreenY,
                 Boolean *pPenDown)
```

Parameters

`pScreenX` x location relative to display.

`pScreenY` y location relative to display.

`pPenDown` `TRUE` or `FALSE`.

Result Returns nothing.

Comments Called by various UI routines.

See Also `KeyCurrentState` (documented in *Developing Palm OS Applications, Part I*)

EvtGetPenBtnList

Purpose Return a pointer to the silk-screen button array.

Prototype

```
PenBtnInfoPtr asm
        EvtGetPenBtnList( UIntPtr numButtons)
```

Parameters

`numButtons`	Pointer to the variable to contain the number of buttons in the array.

Result Returns a pointer to the array.

Comments The array returned contains the bounds of each silk-screened button and the ASCII code and modifiers byte to generate for each button.

See Also `EvtProcessSoftKeyStroke`

EvtKeyQueueEmpty

Purpose Return `TRUE` if the key queue is currently empty.

Prototype

```
Boolean EvtKeyQueueEmpty (void)
```

Parameters None.

Result Returns `TRUE` if the key queue is currently empty, otherwise returns `FALSE`.

Comments Usually called by the key manager to determine if it should enqueue auto-repeat keys.

EvtKeyQueueSize

Purpose Return the size of the current key queue in bytes.

Prototype

```
ULong EvtKeyQueueSize (void)
```

Parameters None.

Result Returns size of queue in bytes.

Comments Called by applications that wish to see how large the current key queue is.

EvtPenQueueSize

Purpose Return the size of the current pen queue in bytes.

Prototype

```
ULong EvtPenQueueSize (void)
```

Parameters None.

Result Returns size of queue in bytes.

Comments Call this function to see how large the current pen queue is.

EvtProcessSoftKeyStroke

Purpose Translate a stroke in the system area of the digitizer and enqueue the appropriate key events in to the key queue.

Prototype

```
Err EvtProcessSoftKeyStroke( PointType* startPtP,
                             PointType* endPtP)
```

Parameters

`startPtP` Start point of stroke.

`endPtP` End point of stroke.

Result Returns 0 if recognized, -1 if not recognized.

See Also `EvtGetPenBtnList`, `GrfProcessStroke` (documented in *Developing Palm OS Applications, Part I*)

EvtResetAutoOffTimer

Purpose Reset the auto-off timer to assure that the device doesn't automatically power off during a long operation without user input (for example, serial port activity).

Prototype

```
Err EvtResetAutoOffTimer (void)
```

Parameters None.

Result Always returns 0.

Comments Called by `SerialLinkMgr`, Can be called periodically by other managers.

See Also `SysSetAutoOffTime`

EvtSysEventAvail

Purpose Return TRUE if a low-level system event (such as a pen or key event) is available.

Prototype

```
Boolean EvtSysEventAvail(Boolean ignorePenUps)
```

Parameters

`ignorePenUps` If TRUE, this routine ignores pen-up events when determining if there are any system events available.

Result Returns TRUE if a system event is available.

Comment Call `EvtEventAvail` to determine whether high-level software events are available.

EvtWakeup

Purpose Force the event manager to wake up and send a `nilEvent` to the current application. Events are documented in "*Developing Palm OS Applications, Part I*").

Prototype

```
Err EvtWakeup (void)
```

Parameters None.

Result Always returns 0.

Comments Called by interrupt routines, like the sound manager and alarm manager.

Functions for System Use Only

EvtDequeueKeyEvent

Prototype

```
Err EvtDequeueKeyEvent (EventPtr eventP)
```

WARNING: System Use Only!

EvtEnqueuePenPoint

Prototype

```
Err EvtEnqueuePenPoint (PointType* ptP)
```

WARNING: System Use Only!

EvtGetSysEvent

Prototype

```
void EvtGetSysEvent (  EventPtr eventP,
                       Long timeout)
```

WARNING: System Use Only!

EvtInitialize

Prototype

```
void EvtInitialize (void)
```

WARNING: System Use Only!

EvtSetKeyQueuePtr

Prototype

```
Err EvtSetKeyQueuePtr (Ptr keyQueueP, ULong size)
```

WARNING: System Use Only!

EvtSetPenQueuePtr

Prototype

```
Err EvtSetPenQueuePtr (Ptr penQueueP, ULong size)
```

WARNING: System Use Only!

EvtSysInit

Prototype

```
Err EvtSysInit (void)
```

WARNING: System Use Only!

Feature Manager Functions

FtrGet

Purpose

Get a feature.

Prototype

```
Err FtrGet ( DWord creator,
             UInt featureNum,
             DWordPtr valueP)
```

Parameters

`creator`	Creator type, should be same as the application that owns this feature.
`featureNum`	Feature number of the feature.
`valueP`	Value of the feature is returned here.

Result

Returns 0 if no error, or `ftrErrNoSuchFtr` or `ftrErrInternalError` if an error occurs.

Comments

The value of the feature is application-dependent.

See Also

`FtrSet`

FtrGetByIndex

Purpose

Get a feature by index.

Until the caller gets back `ftrErrNoSuchFeature`, it should pass indices for each table (ROM, RAM) starting at 0 and incrementing .

Prototype

```
Err FtrGetByIndex (  UInt index,
                     Boolean romTable,
                     DWordPtr creatorP,
                     UIntPtr numP,
                     DWordPtr valueP)
```

Parameters

`index`	Index of feature.
`romTable`	If `TRUE`, index into ROM table; otherwise, index into RAM table.
`creatorP`	Feature creator is returned here.
`numP`	Feature number is returned here.
`valueP`	Feature value is returned here.

Result

Returns 0 if no error, or `ftrErrInternalError` or `ftrErrNoSuchFeature` if an error occurs.

Comments

This routine is normally only used by shell commands. Most applications don't need it.

FtrSet

Purpose Set a feature.

Prototype

```
Err FtrSet ( DWord creator,
             UInt featureNum,
             DWord newValue)
```

Parameters

`creator`	Creator type, should be same as the application that owns this feature.
`featureNum`	Feature number of the feature.
`newValue`	New value.

Result Returns 0 if no error, or `ftrErrNoSuchFeature`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Comments The value of the feature is application-dependent.

See Also `FtrGet`

FtrUnregister

Purpose Unregister a feature.

Prototype

```
Err FtrUnregister (DWord creator,
                   UInt featureNum)
```

Parameters

`creator`	Creator type, should be same as the application that owns the creator.
`featureNum`	Feature number of the feature.

Result Returns 0 if no error, or `ftrInternalError`, `ftrErrNoSuchFeature`, `memErrChunkLocked`, `memErrInvalidParam`, or `memErrNotEnoughSpace` if an error occurs.

Functions for System Use Only

FtrInit

Prototype

```
Err FtrInit (void)
```

WARNING: This function for System use only

Find Functions

FindDrawHeader

Purpose Draw the header line that separates, by database, the list of found items.

Prototype

```
Boolean FindDrawHeader ( FindParamsPtr params,
                         CharPtr title)
```

Parameters

`params`	Handle of `FindParamsPtr`.
`title`	Description of the database (for example Memos).

Result Returns `TRUE` if Find screen is filled up. Applications should exit from the search if this occurs.

FindGetLineBounds

Purpose Returns the bounds of the next available line for displaying a match in the Find Results dialog.

Prototype

```
void FindGetLineBounds ( FindParamsPtr params,
                         RectanglePtr r)
```

Parameters

`params`	Handle of `FindParamsPtr`.
`r`	Pointer to a structure to hold the bounds of the next results line.

Result Returns nothing.

FindSaveMatch

Purpose

Saves the record and position within the record of a text search match. This information is saved so that it's possible to later navigate to the match.

Prototype

```
void FindSaveMatch     ( FindParamsPtr params,
                         UInt recordNum,
                         Word pos,
                         UInt fieldNum,
                         DWord appCustom,
                         UInt dbCardNo,
                         LocalID rdbID)
```

Parameters

`params`	Handle of `FindParamsPtr`.
`recordNum`	Record index.
`pos`	Offset of the match string from start of record.
`appCustom`	Extra data the application can save with a match.
`dbCardNo`	Card number of the database that contains the match.
`rdbID`	Local ID of the database that contains the match.

Result

Returns `TRUE` if the maximum number of displayable items has been exceeded

Comments

Called by application code when it gets a match.

FindStrInStr

Purpose

Perform a case-blind partial word search for a string in another string. This function assumes that the string to find is in lower-case characters.

Prototype

```
void FindStrInStr( CharPtr strToSearch,
                   CharPtr strToFind,
                   WordPtr posP)
```

Parameters

`strToSearch`	String to search.
`strToFind`	Converted, caseless version of the ASCII text string to be found.
`posP`	Pointer to offset in search string of the match.

Result

Returns `TRUE` if the string was found.

Comment

To convert a standard ASCII, null-terminated text string into the appropriate format for `strToFind`, use the conversion table returned by `GetCharCaselessValue` in code similar to the following:

```
CharPtr origStr;
      /* Standard null-terminated ascii string */
CharPtr strToFind;
      /* Converted string to be passed to */
      /* FindStrInStr */
BytePtr convTab;
      /* Conversion table returned from */
      /* GetCharCaselessValue*/
int i;
convTab = GetCharCaselessValue();
```

```
for (i=0; origStr[i] != 0; i++)
   {
   strToFind[i] = convTab[origStr[i]];
   }
strToFind[i] = 0;
         /* Now pass strToFind to
FindStrInStr...*/
```

Note that the `strToFind` element of the parameter block passed by the system's Find utility is preconverted, so it can be passed straight through to `FindStrInStr`, just as in the example in the tutorial.

See Also

`GetCharCaselessValue` (documented in "Developing Palm OS Applications, Part I)

Float Manager Functions

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

Using Floating Point Arithmetic

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

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

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

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

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

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

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

Floating-point functionality is identical in either method.

Using 1.0 Floating-Point Functionality

The original `Fpl` calls (documented in this section) are still available. They may be useful for applications that don't need high precision, don't want to incur the size penalty of the float library, and want to run on 1.0 devices only. To get 1.0 behavior, use the 1.0 calls (`FplAdd`, etc) and don't link in the library.

FplAdd

Purpose Add two floating-point numbers (returns a + b).

Prototype
```
FloatType FplAdd (FloatType a, FloatType b)
```

Parameters
`a, b` The floating-point numbers.

Result Returns the normalized floating-point result of the addition.

Comment Under Palm OS 2.0 and later, most applications will want to use the arithmetic symbols instead. See Using Floating Point Arithmetic.

FplAToF

Purpose Convert a zero-terminated ASCII string to a floating-point number. The string must be in the format : [-]x[.]yyyyyyyy[e[-]zz]

Prototype
```
FloatType FplAToF (char* s)
```

Parameters
`s` Pointer to the ASCII string.

Result Returns the floating-point number.

Comment The mantissa of the number is limited to 32 bits.

See Also `FplFToA`

FplBase10Info

Purpose

Extract detailed information on the base 10 form of a floating-point number: the base 10 mantissa, exponent, and sign.

Prototype

```
Err FplBase10Info (  FloatType a,
                     ULong* mantissaP,
                     Int* exponentP,
                     Int* signP)
```

Parameters

`a`	The floating-point number.
`mantissaP`	The base 10 mantissa (return value).
`exponentP`	The base 10 exponent (return value).
`signP`	The sign, 1 or -1 (return value).

Result

Returns an error code, or 0 if no error.

Comments

The mantissa is normalized so it contains at least `kMaxSignificantDigits` significant digits when printed as an integer value.

`FlpBase10Info` reports that zero is "negative"; that is, it returns a one for xSign. If this is a problem, a simple workaround is:

```
if (xMantissa == 0) {
    xSign = 0;
```

FplDiv

Purpose Divide two floating-point numbers (result = dividend / divisor).

Prototype

```
FloatType FplDiv ( FloatType dividend,
                   FloatType divisor)
```

Parameters

`dividend` Floating-point dividend.

`divisor` Floating-point divisor.

Result Returns the normalized floating-point result of the division.

Under Palm OS 2.0 and later, most applications will want to use the arithmetic symbols instead. See Using Floating Point Arithmetic.

FplFloatToLong

Purpose Convert a floating-point number to a long integer.

Prototype

```
Long FplFloatToLong (FloatType f)
```

Parameters

`f` Floating-point number to be converted.

Result Returns the long integer.

See Also `FplLongToFloat`, `FplFloatToULong`

FplFloatToULong

Purpose Convert a floating-point number to an unsigned long integer.

Prototype
```
ULong FplFloatToULong (FloatType f)
```

Parameters `f` Floating-point number to be converted.

Result Returns an unsigned long integer.

See Also FplLongToFloat, FplFloatToLong

FplFree

Purpose Release all memory allocated by the floating-point initialization.

Prototype
```
void FplFree()
```

Parameters None.

Result Returns nothing.

Comments Applications must call this routine after they've called other functions that are part of the float manager.

See Also FplInit

FplFToA

Purpose Convert a floating-point number to a zero-terminated ASCII string in exponential format : [-]x.yyyyyyyye[-]zz

Prototype

```
Err FplFToA (FloatType a, char* s)
```

Parameters

`a` Floating-point number.

`s` Pointer to buffer to contain the ASCII string.

Result Returns an error code, or 0 if no error.

See Also `FplAToF`

FplInit

Purpose Initialize the floating-point conversion routines.

Allocate space in the system heap for floating-point globals.

Initialize the `tenPowers` array in the globals area to the powers of 10 from -99 to +99 in floating-point format.

Prototype

```
Err FplInit()
```

Parameters None.

Result Returns an error code, or 0 if no error.

Comments Applications must call this routine before calling any other `fpl` function.

See Also `FplFree`

FplLongToFloat

Purpose Convert a long integer to a floating-point number.

Prototype
```
FloatType FplLongToFloat (Long x)
```

Parameters `x` A long integer.

Result Returns the floating-point number.

FplMul

Purpose Multiply two floating-point numbers.

Prototype
```
FloatType FplMul (FloatType a, FloatType b)
```

Parameters `a, b` The floating-point numbers.

Result Returns the normalized floating-point result of the multiplication.

Comment Under Palm OS 2.0 and later, most applications will want to use the arithmetic symbols instead. See Using Floating Point Arithmetic.

FplSub

Purpose Subtract two floating-point numbers (returns a - b).

Prototype

```
FloatType FplSub (FloatType a, FloatType b)
```

Parameters

`a, b` The floating-point numbers.

Result Returns the normalized floating-point result of the subtraction.

Comment Under Palm OS 2.0 and later, most applications will want to use the arithmetic symbols instead. See Using Floating Point Arithmetic.

Miscellaneous System Functions

Crc16CalcBlock

Purpose Calculate the 16-bit CRC of a data block using the table lookup method.

Prototype

```
Word Crc16CalcBlock (VoidPtr bufP,
                     UInt count,
                     Word crc)
```

Parameters

`bufP`	Pointer to the data buffer.
`count`	Number of bytes in the buffer.
`crc`	Seed crc value.

Result A 16-bit CRC for the data buffer.

MdmDial

Purpose Initialize the modem, dial the phone number and wait for result.

When executing this function, the system goes through these steps:

- Switch to the requested initial baud rate.
- If HW hand-shake is requested, enable CTS/RTS hand-shaking; otherwise, disable it.
- Reset the modem.
- Execute the setup string (if any).
- Configure the modem with required settings;
- Dial the phone number.
- Wait for CONNECT XXXXX or other response.
- If auto-baud is requested, switch to the connected baud rate.

Prototype

```
Err MdmDial (    MdmInfoPtr modemP,
                 CharPtr okDialP,
                 CharPtr setupP,
                 CharPtr phoneNumP)
```

Parameters

`modemP`	Pointer to modem info structure (filled in by caller)
`okDialP`	(NOT IMPLEMENTED) Pointer to string of chars allowed in dial string
`setupP`	Pointer to modem setup string without the AT prefix.
`phoneNumP`	Pointer to phone number string

Result 0 if successful; otherwise `mdmErrNoTone`, `mdmErrNoDCD`, `mdmErrBusy`, `mdmErrUserCan`, `mdmErrCmdError`

MdmHangUp

Purpose Hang up the modem.

Prototype

```
Err MdmHangUp (MdmInfoPtr modemP)
```

Parameters

`modemP` Pointer to modem info structure (filled in by caller)

Result 0 if successful;

Warning: This function alters configuration of the serial port (without restoring it).

PhoneNumberLookup

Purpose This routine called the Address Book application to lookup a phone number. See the `phonelookup.c` example program for more information.

Prototype

```
void PhoneNumberLookup (FieldPtr fld)
```

Parameters

`fld` Field object in which the text to match is found.

Comments When trying to match a field, this function first tries to match selected text.

- If there is some selected text, the function replaces it with the phone number if there is a match.
- If there is no selected text, the function replaces the text in which the insertion point is with the phone number if there is a match.
- If there is no match, the function displays the Address Book short list.

Result Nothing returned; it's locked.

ResLoadForm

Purpose Copy and initialize a form resource. The structures are complete except pointers updating. Pointers are stored as offsets from the beginning of the form.

Prototype

```
void* ResLoadForm (Word rscID)
```

Parameters

`rscID` The resource ID of the form.

Result The handle of the memory block that the form is in, since the form structure begins with the `WindowType` structure, this is also a `WindowHandle`.

ResLoadMenu

Purpose Copy and initialize a menu resource. The structures are complete except pointers updating. Pointers are stored as offsets from the beginning of the menu.

Prototype

```
VoidPtr ResLoadMenu (Word rscID)
```

Parameters

`rscID` The resource ID of the menu.

Result The handle of the memory block that the form is in, since the form structure begins with the `WindowType` structure this is also a `WindowHandle`.

System Preferences Functions

PrefGetAppPreferences

Purpose

Return a copy of an application's preferences. Sometimes, for variable length resources, this routine is called twice:

- Once with a `NULL` pointer and size ofk zero to find out how many bytes need to be read.
- A second time with an allocated buffer allocated of the correct size. Note that the application should always check that the return value is greater than or equal to `prefsSize`.

Prototype

```
SWord PrefGetAppPreferences (DWord creator,
                             Word id,
                             VoidPtr prefs,
                             Word *prefsSize,
                             Boolean saved)
```

Parameters

`creator`	Application creator.
`id`	ID number (lets an application have multiple preferences).
`prefs`	Pointer to a buffer to hold preferences.
`prefsSize`	Pointer to size the buffer passed.
`saved`	If `TRUE`, retrieve the saved preferences. If `FALSE`, retrieve the current preferences.

Result

Returns the constant `noPreferenceFound` if the preference resource wasn't found.

If the preference resource was found, the application should check that the value in `prefsSize` is equal or less than the return value. If it's greater than the size passed, then some bytes were not retrieved.

See Also

`PrefSetPreferences`, `PrefGetAppPreferencesV10`

PrefGetAppPreferencesV10

Purpose Return a copy of an application's preferences.

Prototype

```
Boolean PrefGetAppPreferencesV10 (ULong type,
                                  Int version,
                                  VoidPtr prefs,
                                  Word prefsSize)
```

Parameters

`type`	Application creator type.
`version`	Version number of the application.
`prefs`	Pointer to a buffer to hold preferences.
`prefsSize`	Size of the buffer passed.

Result Returns FALSE if the preference resource was not found or the preference resource contains the wrong version number.

Comments The content and format of an application preference is application-dependent.

See Also PrefSetPreferences, PrefGetAppPreferences

PrefGetPreference

Purpose

Return a system preference. Use this instead of PrefGetPreferences.

Prototype

```
DWord PrefGetPreference(
                        SystemPreferencesChoice choice)
```

Parameters

System preference choice; see Preferences.h for available options.

Comments

This function replaces the 1.0 function PrefGetPreferences. While PrefGetPreferences only let you retrieve the whole system preferences structure, this function lets you specify which preferences to retrieve. You can also choose among different preferences using an ID, or choose to access the saved or unsaved preferences.

Result

Returns the system preference.

See Also

PrefSetPreferences, PrefGetAppPreferences, PrefGetAppPreferencesV10

PrefGetPreferences

Purpose Return a copy of the system preferences.

Prototype
```
void PrefGetPreferences (SystemPreferencesPtr p)
```

Parameters p Pointer to system preferences.

Result Returns nothing. Stores the system preferences in `p`.

Comments The `p` parameter points to a memory block allocated by the caller that is filled in by this function.

This function is often called in `StartApplication` to get localized settings.

See Also `PrefSetPreferences`

PrefOpenPreferenceDBV10

Purpose Return a handle to the system preference database.

Prototype
```
DmOpenRef PrefOpenPreferenceDBV10 (void)
```

Parameters Nothing.

Result Returns the handle, or 0 if an error results.

Comments This function is for system use only in Palm OS 2.0 and later.

See Also `PrefGetPreferences`, `PrefSetPreferences`

PrefSetAppPreferences

Purpose Set an application's preferences in the preferences database.

Prototype

```
void PrefSetAppPreferences ( DWord creator,
                             Word id,
                             SWord version,
                             VoidPtr prefs,
                             Word prefsSize,
                             Boolean saved)
```

Parameters

`creator`	Application creator type.
`id`	Resource ID (usually 0).
`version`	Version number of the application.
`prefs`	Pointer to a buffer that holds preferences.
`prefsSize`	Size of the buffer passed.
`saved`	If `TRUE`, set the saved preferences. If not, set the current preferences.

Result Nothing.

See Also `PrefSetAppPreferencesV10`

PrefSetAppPreferencesV10

Purpose Save an application's preferences in the preferences database.

Prototype

```
void PrefSetAppPreferencesV10 (ULong type,
                               Int version,
                               VoidPtr prefs,
                               Word prefsSize)
```

Parameters

`type`	Application creator type.
`version`	Version number of the application.
`prefs`	Pointer to a buffer holding preferences.
`prefsSize`	Size of the buffer passed.

Result Nothing.

Comments The content and format of an application preference is application-dependent.

See Also `PrefSetAppPreferences`, `PrefGetPreferences`

PrefSetPreference

Purpose Set a system preference. Using this function instead of `PrefSetPreferences` allows you to set selected preferences without having to access the whole structure.

Prototype

```
void PrefSetPreference(
                SystemPreferencesChoice choice,
                DWord value)
```

Parameters

`choice` A SystemPreferencesChoice (see Preferences.h)

`value` Value to assign to the item in `SystemPreferencesChoice`.

Result Returns nothing. Changes the value of the system preference.

PrefSetPreferences

Purpose Set the system preferences.

Prototype

```
void PrefSetPreferences (SystemPreferencesPtr p)
```

Parameters

`p` Pointer to system preferences.

Result Returns nothing.

Comments Unless there's a reason for you to access the whole preferences structure, call `PrefSetPreference` instead.

See Also `PrefGetPreferences`

Password Functions

PwdExists

Purpose Return `TRUE` if the system password is set.

Prototype

```
Boolean PwdExists()
```

Parameters None

Result Returns `TRUE` if the system password is set.

PwdRemove

Purpose Remove the encrypted password string and recover data hidden in databases.

Prototype

```
extern void PwdRemove()
```

Parameters None

Result Returns nothing

PwdSet

Purpose Use a passed string as the new password. The password is stored in an encrypted form.

Prototype

```
void PwdSet (CharPtr oldPassword,
             CharPtr newPassword)
```

Parameters

`oldPassword`	The old password must be successfully verified or the new password isn't accepted
`newPassword`	`CharPtr` to a string to use as the password. `NULL` means no password.

Result Returns nothing

PwdVerify

Purpose Verify that the string passed matches the system password.

Prototype

```
Boolean PwdVerify (CharPtr string)
```

Parameters

`string`	String to compare to the system password. `NULL` means no current password.

Result Returns `TRUE` if the string matches the system password.

String Manager Functions

StrAToI

Purpose Convert a string to an integer.

Prototype

```
Int StrAToI (CharPtr str)
```

Parameters `str` String to convert.

Result Returns the integer.

Comments Use this function instead of the standard `atoi` routine.

StrCaselessCompare

Purpose Compare two strings with case and accent insensitivity.

Prototype

```
Int StrCaselessCompare (CharPtr s1, CharPtr s2)
```

Parameters Two string pointers.

Result Returns 0 if the two strings match, or non-zero if they don't.

Comments Use this function instead of the standard `stricmp` routine. Use it to find strings but not sort them because it ignores case and accents.

See Also `StrCompare`

StrCat

Purpose Concatenate one string to another.

Prototype
```
CharPtr StrCat (CharPtr dst, CharPtr src)
```

Parameters
`dst` Destination string pointer.
`src` Source string pointer.

Result Returns a pointer to the destination string.

Comments Use this function instead of the standard `strcat` routine.

StrChr

Purpose Look for a character within a string.

Prototype
```
CharPtr StrChr (CharPtr str, Int chr)
```

Parameters
`str` String to search.
`chr` Character to search for.

Result Returns a pointer to the first occurrence of character in `str`, or `NULL` if not found.

Comments Use this function instead of the standard `strchr` routine.

This routine does not correctly find a '\0' character.

See Also `StrStr`

StrCompare

Purpose Compare two strings.

Prototype
```
Int StrCompare (CharPtr s1, CharPtr s2)
```

Parameters `s1, s2` Two string pointers.

Result Returns 0 if the strings match.

Returns a positive number if s1 > s2.

Returns a negative number if s1 < s2.

Comments This function is case sensitive. Use it to sort strings but not to find them.

Use this function instead of the standard `strcmp` routine.

See Also `StrCaselessCompare`

StrCopy

Purpose Copy one string to another.

Prototype
```
CharPtr StrCopy (CharPtr dst, CharPtr src)
```

Parameters `s1, s2` Two string pointers.

Result Returns a pointer to the destination string.

Comments Use this function instead of the standard `strcpy` routine.

This function does not return overlapping strings.

StrDelocalizeNumber

Purpose Delocalize a number passed in as a string. Convert the number from any localized notation to US notation (decimal point and thousandth comma). The current thousand and decimal separators have to be passed in.

Prototype

```
CharPtr StrDelocalizeNumber(
                                CharPtr s,
                                Char thousandSeparator,
                                Char decimalSeparator)
```

Parameters

`s`	Pointer to the number ASCII string.
`thousandSeparator`	Current thousand separator.
`decimalSeparator`	Current decimal separator.

Result Returns a pointer to the changed number and modifies the string in s.

See Also `StrLocalizeNumber`, `LocGetNumberSeparators` (documented in "*Develping Palm OS Applications, Part I*")

StrIToA

Purpose Convert an integer to ASCII.

Prototype

```
CharPtr StrIToA (CharPtr s, Long i)
```

Parameters

`s`	String pointer to store results.
`i`	Integer to convert.

Result Returns a pointer to the result string.

See Also `StrAToI`, `StrIToH`

StrIToH

Purpose Convert an integer to hexadecimal ASCII.

Prototype
```
CharPtr StrIToH (CharPtr s, ULong i)
```

Parameters
- s — String pointer to store results.
- i — Integer to convert.

Result Returns the string pointer `s`.

See Also `StrIToA`

StrLen

Purpose Compute the length of a string.

Prototype
```
UInt StrLen (CharPtr src)
```

Parameters
- `src` — String pointer

Result Returns the length of the string.

Comments Use this function instead of the standard `strlen` routine.

StrLocalizeNumber

Purpose Convert a number (passed in as a string) to localized format, using a specified thousandSeparator and decimalSeparator.

Prototype

```
void StrLocalizeNumber(CharPtr s,
                       Char thousandSeparator,
                       Char decimalSeparator)
```

Parameters

`s`	Number ASCII string to localize
`thousandSeparator`	Localized thousand separator.
`decimalSeparator`	Localized decimal separator.

Result Returns nothing. Converts the number string in `s`.

See Also `StrDelocalizeNumber`

StrNCaselessCompare

Purpose Compares two strings out to N characters with case and accent insensitivity.

Prototype

```
Int StrNCaselessCompare( const Char* s1,
                         const Char* s2,
                         DWord n)
```

Parameters

`s1`	Pointer to first string.
`s2`	Pointer to second string.
`n`	Number of characters to compare.

Result 0 if they match, non-zero if not: positive if s1 > s2, negative if s1 < s2

See Also `StrNCompare`

StrNCat

Purpose Concatenates 1 string to another clipping the destination string to a max of N characters (including null at end).

Prototype

```
CharPtr StrNCat( CharPtr dstP,
                 const Char* srcP,
                 Word n)
```

Parameters

`dstP` Pointer to destination string.

`srcP` Pointer to source string.

`n` Maximum number of characters for `dstP`.

Result Returns a pointer to the destination string.

Comment This function differs from the standard C `strncat` function in these ways:

- `StrNCat` treats the parameter `n` as the maximum size of `dstP`. The standard C function copies `n` characters from `srcP` into `dstP`.
- `StrNCat` does not append the '\0' character to the end of the destination string if the size of the destination string is already `n`. That is, if you specify 6 as the value for `n` and the `dstP` string reaches a size of 6 characters when characters from `srcP` are added to it, `StrNCat` does not append '\0' to the `dstP` string.

StrNCompare

Purpose Compare two strings out to N characters. This function is case and accent sensitive.

Prototype

```
Int StrNCompare(const Char* s1,
                const Char* s2,
                DWord n)
```

Parameters

`s1` Pointer to first string.

`s2` Pointer to second string.

`n` Number of characters to compare.

Result Returns 0 if the strings match, non-zero if they don't match. In that case:

+ if s1 > s2

- if s1 < s2

See Also `StrNCaselessCompare`

StrNCopy

Purpose Copies up to N characters from str string to dst string. Terminates `dst` string at index N-1 if `src` string length was N-1 or less.

Prototype

```
CharPtr StrNCopy( CharPtr dstP,
                  const Char* srcP,
                  Word n)
```

Parameters

`dstP` Destination string.

`srcP` Source string.

`n` Maximum number of bytes to copy from `src` string.

Result Returns a pointer to destination string

StrPrintF

Purpose Implements a subset of the ANSI C sprintf() call.

Currently, only `%d`, `%i`, `%u`, `%x` and `%s` are implemented and don't accept field length or format specifications except for the l (long) modifier.

Prototype

```
SWord StrPrintF(CharPtr s,
                const Char* formatStr,
                ...)
```

Parameters

`s`	Destination string
`formatStr`	Format string.
`*...`	Arguments for format string.

Result Number of characters written to destination string.

See Also `StrVPrintF`

StrStr

Purpose Look for a substring within a string.

Prototype

```
CharPtr StrStr (CharPtr str, CharPtr token)
```

Parameters

`str`	String to search.
`token`	String to search for.

Result Returns a pointer to the first occurrence of `token` in `str`, or `NULL` if not found.

Comments Use this function instead of the standard `strstr` routine.

See Also `StrChr`

StrToLower

Purpose Convert all the characters in a string to lowercase.

Prototype

```
CharPtr StrToLower (CharPtr dst, CharPtr src)
```

Parameters `dst`, `src` Two string pointers.

Result Returns a pointer to the destination string.

Comments This function **doesn't** convert accented characters.

StrVPrintF

Purpose Implements a subset of the ANSI C `vsprintf()` call.

Currently, only `%d`, `%i`, `%u`, `%x` and `%s` are implemented and don't accept field length or format specifications except for the l (long) modifier.

Prototype

```
SWord StrVPrintF(  CharPtr s,
                   const Char* formatStr,
                   VoidPtr argParam)
```

Parameters

`s`	Destination string.
`formatStr`	Format string.
`argParam`	Pointer to argument list.

Result Returns the number of characters written to destination string.

Example Here's an example of how to use this call:

```
#include <stdarg.h>
void MyPrintF(CharPtr s, CharPtr formatStr, ...)
{
   va_list args;
   Char text[0x100];
   va_start(args, formatStr);
   StrVPrintF(text, formatStr, args);
   va_end(args);
   MyPutS(text);
}
```

See Also StrPrintF

File Streaming Functions

FileClearerr

Purpose Clear I/O error status, end of file error status, and last error.

Prototype
```
Err FileClearErr(FileHand stream)
```

Parameters --> `stream` Handle to open stream.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

See Also `FileGetLastError`, `FileRewind`

FileClose

Purpose Close the file stream and destroy its handle. If the stream was opened with `fileModeTemporary`, it is deleted upon closing.

Prototype
```
Err FileClose(FileHand stream)
```

Parameters --> `stream` Handle to open stream.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

FileControl

Purpose

Perform the operation specified by the `op` parameter on the `stream` file stream.

Prototype

```
Err FileControl(FileOpEnum op, FileHand stream,
                  VoidPtr valueP, LongPtr
valueLenP)
```

Parameters

`op`	The operation to perform, and its associated formal parameters, as specified by one of the following selectors:
	`fileOpDestructiveReadMode`
	`fileOpGetEOFStatus`
	`fileOpGetLastError`
	`fileOpClearError`
	`fileOpGetIOErrorStatus`
	`fileOpGetCreatedStatus`
	`fileOpGetOpenDbRef`
	`fileOpFlush`
	For details, see `FileOpEnum` on page 30.

--> `stream`	Open stream handle if required for file stream operation.
<--> `valueP`	Pointer to value or buffer, as required. This parameter is defined by the selector passed as the value of the `op` parameter. For details, see `FileOpEnum` on page 30.
<--> `valueLenP`	Pointer to value or buffer, as required. This parameter is defined by the selector passed as the value of the `op` parameter. For details, see `FileOpEnum` on page 30.

Result

Returns either a value defined by the selector passed as the argument to the `op` parameter, or an error code resulting from the requested operation. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

Comments

Normally, you do not call the `FileControl` function yourself; it is called for you by most of the other file streaming functions and macros to perform common file streaming operations. You can call `FileControl` yourself to enable specialized read modes.

Pass the `fileOpDestructiveReadMode` selector as the value of the op parameter to the `FileControl` function to enable destructive read mode. This mode deletes blocks as data are read, thus freeing storage automatically. Once in destructive read mode, you cannot re-use the file stream—the contents of the stream are undefined after it is closed or after a crash.

Writing to files opened without write access or those that are in destructive read state is not allowed; thus, you cannot call the `FileWrite`, `FileSeek`, or `FileTruncate` functions on a stream that is in destructive read mode. One exception to this rule applies to streams that were opened in "write + append" mode and then switched into destructive read state. In this case, the `FileWrite` function can append data to the stream, but it also preserves the current stream position so that subsequent reads pick up where they left off (you can think of this as a pseudo-pipe).

See Also

`FileOpEnum`, `FileClearerr`, `FileEOF`, `FileError`, `FileFlush`, `FileGetLastError`, `FileRewind`

FileDelete

Purpose Deletes the specified file stream from the specified card. Only a closed stream may be passed to this function.

Prototype

```
Err FileDelete(UInt cardNo, CharPtr nameP)
```

Parameters

`cardNo`	Card on which the file stream to delete resides. Currently, no Palm OS devices support multiple cards, so this value must be 0.
`nameP`	String that is the name of the stream to delete.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

See Also The `fileModeTemporary` argument to the `openMode` parameter of the `FileOpen` function.

FileDmRead

Purpose Read data from a file stream into a chunk, record, or resource residing in a database.

Prototype

```
Long FileDmRead(FileHand stream,
                VoidPtr startOfDmChunkP,
                Long destOffset,
                Long objSize, Long numObj,
                Err* errP)
```

Parameters

--> `stream`	Handle to open stream.
--> `startOfDmChunkP`	Pointer to beginning of chunk, record or resource residing in a database.
`destOffset`	Offset from `startOfDmChunkP` (base pointer) to the destination area (must be >= 0).

`objSize`	Size of each stream object to read.
`numObj`	Number of stream objects to read.
<--> `errP`	Pointer to variable that is to hold the error code returned by this function. Pass `NULL` to ignore. For a list of file streaming error codes, see File Streaming Error Codes beginning on page 149.

Result

The number of whole objects that were read—note that the number of objects actually read may be less than the number requested.

Comments

When the number of objects actually read is less than the number requested, you may be able to determine the cause of this result by examining the return value of the `errP` parameter or by calling the `FileGetLastError` function. If the cause is insufficient data in the stream to satisfy the full request, the current stream position is at end-of-file and the "end of file" indicator is set. If a non-NULL pointer was passed as the value of the `errP` parameter when the `FileDmRead` function was called and an error was encountered, `*errP` holds a non-zero error code when the function returns. In addition, the `FileError` and `FileEOF` functions may be used to check for I/O errors.

See Also

`FileRead`, `FileReadLow`, `FileError`, `FileEOF`

FileEOF

Purpose Get end-of-file status (err = `fileErrEOF` indicates end of file condition).

Prototype

```
Err FileEOF(FileHand stream)
```

Parameters

--> `stream` Handle to open stream.

Result 0 if *not* end of file; non-zero if end of file. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

Comments This function's behavior is similar to that of the `feof` function provided by the C programming language runtime library.

Use `FileClearerr` to clear the I/O error status.

See Also `FileClearerr`, `FileGetLastError`, `FileRewind`

FileError

Purpose Get I/O error status.

Prototype
```
Err FileError(FileHand stream)
```

Parameters

-->	`stream`	Handle to open stream.

Result 0 if no error, and non-zero if an I/O error indicator has been set for this stream. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

Comments This function's behavior is similar to that of the C programming language's `ferror` runtime function.

Use `FileClearerr` to clear the I/O error status.

See Also `FileClearerr`, `FileGetLastError`, `FileRewind`

FileFlush

Purpose Flush cached data to storage.

Prototype
```
Err FileFlush(FileHand stream)
```

Parameters

-->	`stream`	Handle to open stream.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

Comments It is not always necessary to call this function explicitly—certain operations flush the contents of a stream automatically; for example, streams are flushed when they are closed. Because this function's behavior is similar to that of the `fflush` function provided by the C programming language runtime library, you only need to call it ex-

plicitly under circumstances similar to those in which you would call `fflush` explicitly.

FileGetLastError

Purpose Get error code from last operation on file stream, and clear the last error code value (will not change end of file or I/O error status -- use `FileClearerr` to reset all error codes)

Prototype

```
Err FileGetLastErr(FileHand stream)
```

Parameters

--> `stream`	Handle to open stream.

Result Error code returned by the last file stream operation. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

See Also `FileClearerr`, `FileEOF`, `FileError`

FileOpen

Purpose Open existing file stream or create an open file stream for I/O in the mode specified by the `openMode` parameter.

Prototype

```
FileHand FileOpen (UInt cardNo, CharPtr nameP,
                   ULong type, ULong creator,
                   DWord openMode, Err* errP)
```

Parameters

`cardNo`	Card on which the file stream to open resides. Currently, no Palm OS devices support multiple cards, so this value must be 0.
--> `nameP`	Pointer to text string that is the name of the file stream to open or create. This value must be a valid name—no wildcards allowed, must not be `NULL`.

`type` — Filetype of stream to open or create. Pass `0` for wildcard, in which case `sysFileTFileStream` is used if the stream needs to be created and `fileModeTemporary` is not specified. If type is `0` and `fileModeTemporary` is specified, then `sysFileTTemp` is used for the filetype of the stream this function creates.

`creator` — Creator of stream to open or create. Pass `0` for wildcard, in which case the current application's creator ID is used for the creator of the stream this function creates.

`openMode` — Mode in which to open the file stream. You must specify only one primary mode selector. Additionally, you can use the `|` operator (bitwise inclusive OR) to append one or or more secondary mode selectors to the primary mode selector. The primary mode selectors are:

`fileModeReadOnly`
: Open for read-only access

`fileModeReadWrite`
: Open / create for read / write access, discarding any previous version of stream

`fileModeUpdate`
: Open / create for read / write, preserving previous version of stream if it exists

`fileModeAppend`
: Open / create for read / write, always writing to the end of the stream

You can use the `|` operator (bitwise inclusive OR) to append one or more of the following secondary mode selectors to the primary mode selector:

`fileModeDontOverwrite`
: Prevents `fileModeReadWrite` from discarding an existing stream having the

same name; may only be specified together with `fileModeReadWrite`

`fileModeLeaveOpen`
Leave stream open when application quits. Most applications should not use this option. See `Comments` at the end of this function description for more information.

`fileModeExclusive`
No other application can open the stream until the application that opened it in this mode closes it.

`fileModeAnyTypeCreator`
Accept any type/creator when opening or replacing an existing stream. Normally, the `FileOpen` function opens only streams having the specified creator and type. Setting this option enables the `FileOpen` function to open streams having a type or creator other than those specified.

`fileModeTemporary`
Delete the stream automatically when it is closed. See `Comments` at the end of this function description for more information.

<--> `errP` — Pointer to variable that is to hold the error code returned by this function. Pass `NULL` to ignore. For a list of file streaming error codes, see File Streaming Error Codes beginning on page 149.

Result

If successful, returns a handle to an open file stream; otherwise, returns `0`.

Comments

The `fileModeReadOnly`, `fileModeReadWrite`, `fileModeUpdate`, and `fileModeAppend` modes are mutually exclusive—pass only one of them to the `FileOpen` function!

When the `fileModeTemporary` open mode is used and the file type passed to `FileOpen` is `0`, the `FileOpen` function uses `sysFileTTemp` (defined in `SystemMgr.rh`) for the file type, as recommended. In future versions of PalmOS, this configuration will enable the automatic cleanup of undeleted temporary files after a system crash. Automatic post-crash cleanup is not implemented in current versions of Palm OS.

To open a file stream even if it has a different type and creator than specified, pass the `fileModeAnyTypeCreator` selector as a flag in the `openMode` parameter to the `FileOpen` function.

The `fileModeLeaveOpen` mode is an esoteric option that most applications should not use. It may be useful for a library that needs to open a stream from the current application's context and keep it open even after the current application quits. By default, Palm OS automatically closes all databases that were opened in a particular application's context when that application quits. The `fileModeLeaveOpen` option overrides this default behavior.

FileRead

Purpose

Reads data from a stream into a buffer. Do not use this function to read data into a chunk, record or resource residing in a database—you must use the `FileDmRead` function for such operations.

Prototype

```
Long FileRead(FileHand stream, VoidPtr bufP,
              Long objSize, Long numObj,
              Err* errP)
```

Parameters

--> `stream`	Handle to open stream.
--> `bufP`	Pointer to beginning of buffer into which data is read
`objSize`	Size of each stream object to read.
`numObj`	Number of stream objects to read.
<--> `errP`	Pointer to variable that is to hold the error code returned by this function. Pass `NULL` to ignore.

For a list of file streaming error codes, see File Streaming Error Codes beginning on page 149.

Result The number of whole objects that were read—note that the number of objects actually read may be less than the number requested.

Comments Do not use this function to read data into a chunk, record or resource residing in a database—you must use the `FileDmRead` function for such operations.

When the number of objects actually read is fewer than the number requested, you may be able to determine the cause of this result by examining the return value of the `errP` parameter or by calling the `FileGetLastError` function. If the cause is insufficient data in the stream to satisfy the full request, the current stream position is at end-of-file and the "end of file" indicator is set. If a non-NULL pointer was passed as the value of the `errP` parameter when the `FileRead` function was called and an error was encountered, `*errP` holds a non-zero error code when the function returns. In addition, the `FileError` and `FileEOF` functions may be used to check for I/O errors.

See Also `FileDmRead`

FileRewind

Purpose Reset position marker to beginning of stream and clear all error codes.

Prototype

```
Err FileRewind(FileHand stream)
```

Parameters

--> `stream` Handle to open stream.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

See Also FileSeek, FileTell, FileClearerr, FileEOF, FileError, FileGetLastError

FileSeek

Purpose Set current position within a file stream, extending the stream as necessary if it was opened with write access.

Prototype

```
Err FileSeek(FileHand stream, Long offset,
             FileOriginEnum origin)
```

Parameters

--> `stream`	Handle to open stream.
`offset`	Position to set, expressed as the number of bytes from origin. This value may be positive, negative, or `0`.
`origin`	A structure of type `FileOriginEnum`, which describes the origin of the position change (beginning, current, or end).

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

Comments Attempting to seek beyond end-of-file in a read-only stream results in an I/O error.

This function's behavior is similar to that of the `fseek` function provided by the C programming language runtime library.

See Also FileRewind, FileTell

FileTell

Purpose Get current position and, optionally, filesize.

Prototype

```
Long FileTell(FileHand stream, LongPtr fileSizeP,
              Err* errP)
```

Parameters

--> `stream`	Handle to open stream.
<-> `fileSizeP`	Pointer to variable that holds value describing size of stream in bytes when this function returns. Pass `NULL` to ignore.
<--> `errP`	Pointer to variable that is to hold the error code returned by this function. Pass `NULL` to ignore. For a list of file streaming error codes, see File Streaming Error Codes beginning on page 149.

Result If successful, returns current position, expressed as an offset in bytes from the beginning of the stream. If an error was encountered, returns `-1` as a signed long integer.

See Also `FileRewind`, `FileSeek`

FileTruncate

Purpose Truncate the file stream to a specified size; not allowed on streams open in destructive read mode or read-only mode.

Prototype

```
Err FileTruncate(FileHand stream, Long newSize)
```

Parameters

--> `stream`	Handle of open stream.
`newSize`	New size; must not exceed current stream size.

Result 0 if no error, or a `fileErr` code if an error occurs. For a complete listing of File Streaming Error Codes, see the section beginning on page 149.

See Also `FileTell`

FileWrite

Purpose Write data to a stream.

Prototype

```
Long FileWrite(FileHand stream, VoidPtr dataP,
               Long objSize, Long numObj,
               Err* errP)
```

Parameters

--> `stream`	Handle to open stream.
--> `dataP`	Pointer to buffer holding data to write.
`objSize`	Size of each stream object to write; must be ≥ `0`.
`numObj`	Number of stream objects to write.
<--> `errP`	Optional pointer to variable that holds the error code returned by this function. Pass `NULL` to ignore. For a list of file streaming error codes, see File Streaming Error Codes beginning on page 149.

Result The number of whole objects that were written—note that the number of objects actually written may be less than the number requested. Should available storage be insufficient to satisfy the entire request, as much of the requested data as possible is written to the stream, which may result in the last object in the stream being incomplete.

Comments Writing to files opened without write access or those that are in destructive read state is not allowed; thus, you cannot call the `FileWrite`, `FileSeek`, or `FileTruncate` functions on a stream that is in destructive read mode. One exception to this rule applies to streams that were opened in "write + append" mode and then switched into destructive read state. In this case, the `FileWrite` function can append data to the stream, but it also preserves the current stream position so that subsequent reads pick up where they left off (you can think of this as a pseudo-pipe).

Functions For System Use Only

FileReadLow

Purpose Low-level routine for reading data from a file stream. This function is for system use only—use the helper macros `FileRead` and `FileDmRead` instead of calling this function directly.

Prototype

```
Long FileReadLow(FileHand stream, VoidPtr baseP,
                 Long offset,
                 Boolean dataStoreBased,
                 Long objSize, Long numObj,
                 Err* errP)
```

WARNING: System Use Only!

File Streaming Error Codes

This section lists all error codes returned by the file streaming functions.

Error Code	Value	Meaning
`fileErrMemErr`	`(fileErrorClass|1)`	out of memory error
`fileErrInvalidParam`	`(fileErrorClass|2)`	invalid parameter value passed
`fileErrCorruptFile`	`(fileErrorClass|3)`	alleged stream is corrupted, invalid, or not a stream
`fileErrNotFound`	`(fileErrorClass|4)`	couldn't find the stream
`fileErrTypeCreatorMismatch`	`(fileErrorClass|5)`	type and/or creator not what was specified
`fileErrReplaceError`	`(fileErrorClass|6)`	couldn't replace existing stream
`fileErrCreateError`	`(fileErrorClass|7)`	couldn't create new stream
`fileErrOpenError`	`(fileErrorClass|8)`	generic open error
`fileErrInUse`	`(fileErrorClass|9)`	stream couldn't be opened or deleted because it is in use
`fileErrReadOnly`	`(fileErrorClass|10)`	couldn't open in write mode because existing stream is read-only
`fileErrInvalidDescriptor`	`(fileErrorClass|11)`	invalid file descriptor (`FileHandle`)
`fileErrCloseError`	`(fileErrorClass|12)`	error closing the stream
`fileErrOutOfBounds`	`(fileErrorClass|13)`	attempted operation went out of bounds of the stream
`fileErrPermissionDenied`	`(fileErrorClass|14)`	couldn't write to a stream open for read-only access
`fileErrIOError`	`(fileErrorClass|15)`	generic I/O error
`fileErrEOF`	`(fileErrorClass|16)`	end-of-file error
`fileErrNotStream`	`(fileErrorClass|17)`	attempted to open an entity that is not a stream

Sound Manager Functions

SndCreateMidiList

Purpose

Generate a list of MIDI records having a specified creator.

Prototype

```
Boolean SndCreateMidiList(ULong creator,
                          Boolean multipleDBs,
                          WordPtr wCountP,
                          Handle *entHP)
```

Parameters

-->`creator`	Creator of database in which to find MIDI records. Pass `0` for wildcard.
-->`multipleDBs`	Pass `TRUE` to search multiple databases for MIDI records. Pass FALSE to search only in the first database found that meets search criteria.
<-->`wCountP`	When the function returns, contains the number of MIDI records found.
<-->`entHP`	When the function returns, this handle holds a a memory chunk containing an array of `SndMidiListItemType` structs if MIDI records were found.

Result

Returns `FALSE` if no MIDI records were found, `TRUE` if MIDI records were found. When this function returns `TRUE`, it updates the `wCountP` parameter to hold the number of MIDI records found and updates the `entHP` parameter to hold a handle to an array of `SndMidiListItemType`structs. Each record of this type holds the name, record ID, database ID, and card number of a MIDI record.

Comments

This function is useful for displaying lists of sounds residing on the Palm device as MIDI records.

See Also

`DmFindRecordByID`, `DmOpenDatabase`, `DmQueryRecord`, `DmOpenDatabaseByTypeCreator` functions; "Rock Music" sample code.

SndDoCmd

Purpose

Send a sound manager command to a specified sound channel.

Prototype

```
Err SndDoCmd (  VoidPtr chanP,
                SndCommandPtr cmdP,
                Boolean noWait)
```

Parameters

-> `chanP`	Pointer to sound channel. Present implementation doesn't support multiple channels. Must be `NULL`.
-> `cmdP`	Pointer to a `SndCommandType` structure holding a parameter block that specifies the note to play, its duration, and amplitude.
-> `noWait`	Because asynchronous mode is not yet supported for all commands, you must pass `0` for this value. In the future, `0` = await completion (synchronous) `!0` = immediate return (asynchronous).

Note

Passing `NULL` for the channel pointer causes the command to be sent to the shared sound channel; currently, this is the only option.

Result

0	No error.
`sndErrBadParam`	Invalid parameter.
`sndErrBadChannel`	Invalid channel pointer.
`sndErrQFull`	Sound queue is full.

Comments

This function is useful for simple sound playback applications, such as playing a single note to provide user feedback. In addition to providing the same behavior it did in versions 1.0 and 2.0 of Palm OS, (specify the frequency, duration, and amplitude of a single note to be played) new command selectors provided in Palm OS 3.0 allow you to use MIDI values to specify pitch, duration, and amplitude of the note to play, and to stop the note currently being played.

See Also SndCommandType, SndPlaySMF

SndGetDefaultVolume

Purpose Return default sound volume levels cached by Sound Manager.

Prototype

```
void SndGetDefaultVolume ( UIntPtr alarmAmpP,
                           UIntPtr sysAmpP,
                           UIntPtr defAmpP)
```

Parameters

<->alarmAmpP	Pointer to storage for alarm amplitude.
<-> sysAmpP	Pointer to storage for system sound amplitude.
<-> defAmpP	Pointer to storage for master amplitude.

Result Returns nothing.

Comments Any pointer arguments may be passed as NULL. In that case, the corresponding setting is not returned.

SndPlaySMF

Purpose Performs the operation specified by the cmd parameter: play the specified standard MIDI file (SMF) or return the number of milliseconds required to play the entire SMF.

Prototype

```
Err  SndPlaySmf(void* chanP,
                SndSmfCmdEnum cmd,
                BytePtr smfP,
                SndSmfOptionsType* selP,
                SndSmfChanRangeType* chanRangeP,
                SndSmfCallbacksType* callbacksP,
                Boolean bNoWait)
```

Parameters

chanP	The sound channel used to play the sound. This value must always be NULL because current

versions of Palm OS provide only one sound channel that all applications share.

`cmd` — The operation to perform, as specified by one of the following selectors:

`sndSmfCmdPlay`
: play the selection synchronously

`sndSmfCmdDuration`
: return the duration of the entire SMF, expressed in milliseconds

`--> smfP` — Pointer to the SMF data in memory. This pointer can reference a valid `SndMidiRecType` structure followed by MIDI data, or it can point directly to the beginning of the SMF data.

`--> selP` — `NULL` or a pointer to a `SndSmfOptionsType` structure specifying options for playback volume, position in the SMF from which to begin playback, and whether playback can be interrupted by user interaction with the display. See the `SndSmfOptionsType` structure for the default behavior specified by a `NULL` value.

`--> chanRangeP` — `NULL` or a pointer to a `SndSmfChanRangeType` structure specifying a continuous range of MIDI channels 0 -15 to use for playback. If this value is `NULL`, all tracks are played.

`--> callbacksP` — `NULL` or a pointer to a `SndSmfCallbacksType` structure that holds your callback functions. Functions of type `SndBlockingFuncType` execute periodically while a note is playing, and functions of type `SndComplFuncType` execute after playback of the SMF completes. For more information, see the Sound Callback Functions section beginning on page 51.

`bNoWait` — This value is ignored. This function always finishes playing the SMF selection before returning; however, you can execute a callback function while the SMF is playing.

Result Returns `0` if no error. When an error occurs, this function returns one of the following values; for more information see the `SoundMgr.h` file included with the Palm OS 3.0 SDK:

```
// bogus value passed to this function
sndErrBadParam     (sndErrorClass | 1)
// invalid sound channel
sndErrBadChannel   (sndErrorClass | 2)
// insufficient memory
sndErrMemory       (sndErrorClass | 3)
// tried to open channel that's already open
sndErrOpen         (sndErrorClass | 4)
// can't accept more notes
sndErrQFull        (sndErrorClass | 5)
//internal use - never returned to applications
sndErrQEmpty       (sndErrorClass | 6)
// unsupported data format
sndErrFormat       (sndErrorClass | 7)
// invalid data stream
sndErrBadStream    (sndErrorClass | 8)
// play was interrupted
sndErrInterrupted  (sndErrorClass | 9)
```

Comments Although this call is synchronous, a callback function can be called while a note is playing. If the callback does not return before the number of system ticks required to play the current sound have elapsed, the next note in the SMF will not start on time.

See Also SndDoCmd, SndCreateMidiList

SndPlaySystemSound

Purpose Play a standard system sound.

Prototype
```
void  SndPlaySystemSound (SndSysBeepType beepID)
```

Parameters -> `beepID` System sound to play.

Comments The `SndSysBeepType` enum is defined in `SoundMgr.h` as follows:

```
typedef enum SndSysBeepType {
   sndInfo = 1,
   sndWarning,
   sndError,
   sndStartUp,
   sndAlarm,
   sndConfirmation,
   sndClick
   } SndSysBeepType;
```

Note that in versions of Palm OS prior to 3.0, all of these sounds were synchronous and blocking. In Palm OS 3.0, `sndAlarm` still blocks, but the rest of these system sounds are implemented asynchronously.

Result Returns nothing.

Functions for System Use Only

SndInit

Prototype
```
Err  SndInit(void)
```

WARNING: This function for use by system software only.

SndSetDefaultVolume

Prototype

```
void SndSetDefaultVolume ( UIntPtr alarmAmpP,
                           UIntPtr sysAmpP,
                           UIntPtr defAmpP)
```

WARNING: This function for use by system software only.

System Functions

SysAppLaunch

Purpose

Open an application from a specified database and card, with the appropriate launch flags—generally used to launch an application as a subroutine of the caller.

Prototype

```
Err SysAppLaunch(  UInt cardNo, LocalID dbID,
                   UInt launchFlags, Word cmd,
                   Ptr cmdPBP, DWord* resultP)
```

Parameters

`cardNo`, `dbID`	`cardNo` and `dbID` identify the application.
`launchFlags`	Set to 0.
`cmd`	Launch code.
`cmdPBP`	Launch code parameter block.
`resultP`	Pointer to what's returned by the application's `PilotMain` routine.

Result

Returns 0 if no error, or one of `sysErrParamErr`, `memErrNotEnoughSpace`, `sysErrOutOfOwnerIDs`.

Comments

Launching an application with all launch bits cleared makes the application a subroutine call from the point of view of the caller.

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`, as shown in Listing 1.13, "Opening the Launcher," on page 72.

NOTE: For important information regarding the correct use of this function, see "Opening Applications Programmatically" on page 62.

See Also `SysBroadcastActionCode`, `SysUIAppSwitch`, `SysCurAppDatabase` functions; Listing 1.13, "Opening the Launcher," on page 72.

SysAppLauncherDialog

Purpose Display the launcher popup, get a choice, ask the system to launch the selected application, clean up, and leave. If there are no applications to launch, nothing happens.

Prototype

```
void SysAppLauncherDialog()
```

Parameters None.

Result The system may be asked to launch an application.

Comments Typically, this routine is called by the system as necessary. Most applications do not need to call this function themselves.

In Palm OS version 3.0 the launcher is an application, rather than a popup. This function remains available for compatibility purposes only.

See Also "Application Launcher." starting on page 70; and the description of the `SysAppLaunch` function.

SysBatteryInfo

Purpose

Retrieve settings for the batteries. Set `set` to FALSE to retrieve battery settings. (Applications should ***not*** change any of the settings).

Warning: Use this function only to **retrieve** settings!

Prototype

```
UInt SysBatteryInfo(   Boolean set,
                       UIntPtr warnThresholdP,
                       UIntPtr criticalThresholdP,
                       UIntPtr maxTicksP,
                       SysBatteryKind* kindP,
                       Boolean* pluggedIn
                       BytePtr percentP)
```

Parameters

`set`	If FALSE, parameters with non-nil pointers are retrieved. Never set this parameter to TRUE.
`warnThresholdP`	Pointer to battery voltage warning threshold in volts*100, or nil.
`criticalThresholdP`	Pointer to the battery voltage critical threshold in volts*100, or nil.
`maxTicksP`	Pointer to the battery timeout, or nil.
`kindP`	Pointer to the battery kind, or nil.
`pluggedIn`	Pointer to `pluggedIn` return value, or nil.
`percentP`	Percentage of power remaining in the battery.

Result

Returns the current battery voltage in volts*100.

Comments

Call this function to make sure an upcoming activity won't be interrupted by a low battery warning.

`warnThresholdP` and `maxTicksP` are the battery-warning voltage threshold and time out. If the battery voltage falls below the threshold, or the timeout expires, a `lowBatteryChr` key event is

put on the queue. Normally, applications call SysHandleEvent which calls SysBatteryWarningDialog in response to this event.

criticalThresholdP is the battery voltage threshold. If battery voltage falls below this level, the system turns itself off without warning and doesn't turn on until battery voltage is above it again.

See Also SysBatteryInfoV20

SysBatteryInfoV20

Purpose

Retrieve settings for the batteries. Set set to FALSE to retrieve battery settings. (Applications should ***not*** change any of the settings).

Warning: Use this function only to **retrieve** settings!

Prototype

```
UInt SysBatteryInfo(   Boolean set,
                       UIntPtr warnThresholdP,
                       UIntPtr criticalThresholdP,
                       UIntPtr maxTicksP,
                       SysBatteryKind* kindP,
                       Boolean* pluggedIn)
```

Parameters

set	If FALSE, parameters with non-nil pointers are retrieved. Never set this parameter to TRUE.
warnThresholdP	Pointer to battery voltage warning threshold in volts*100, or nil.
criticalThresholdP	Pointer to the battery voltage critical threshold in volts*100, or nil.
maxTicksP	Pointer to the battery timeout, or nil.
kindP	Pointer to the battery kind, or nil.
pluggedIn	Pointer to pluggedIn return value, or nil.

Result Returns the current battery voltage in volts*100.

Comments Call this function to make sure an upcoming activity won't be interrupted by a low battery warning.

`warnThresholdP` and `maxTicksP` are the battery-warning voltage threshold and time out. If the battery voltage falls below the threshold, or the timeout expires, a `lowBatteryChr` key event is put on the queue. Normally, applications call `SysHandleEvent` which calls `SysBatteryWarningDialog` in response to this event.

`criticalThresholdP` is the battery voltage threshold. If battery voltage falls below this level, the system turns itself off without warning and doesn't turn on until battery voltage is above it again.

See Also `SysBatteryInfo`

SysBinarySearch

Purpose Search elements in a sorted array for the specified data according to the specified comparison function. The array must be sorted in ascending order prior to the search. Use SysInsertionSort or SysQSort to sort the array.

Prototype

```
Boolean SysBinarySearch (
      VoidPtr baseP, Int numOfElements,
      Int width, SearchFuncPtr searchF,
      const VoidPtr searchData,const Long other,
      ULongPtr position, Boolean findFirst)
```

Parameters

`baseP`	Base pointer to an array of elements
`numOfElements`	Number of elements to search, starting at 0 to `numOfElements` -1. Must be greater than 0.
`width`	Width of an element comparison function.
`searchF`	Search function.

`searchData`	Data to search for. This data is passed to the `searchF` function.
`other`	Data to be passed as the third parameter (the `other` parameter) to the comparison function.
`position`	Pointer to the position result.
`findFirst`	If set to `TRUE`, the first matching element is returned. Use this parameter if the array contains duplicate entries to ensure that the first such entry will be the one returned.

Result

Returns `TRUE` if an exact match was found. In this case, `position` points to the element number where the data was found.

Returns `FALSE` if an exact match was not found. If `FALSE` is returned, `position` points to the element number where the data should be inserted if it was to be added to the array in sorted order.

Comments

The search starts at element 0 and ends at element (`numOfElements` - 1).

The search function's (`searchF`) prototype is:

```
Int _searchF (const VoidPtr, const VoidPtr, Long other);
```

The first parameter is the data for which to search, the second parameter is a pointer to an element in the array, and the third parameter is any other necessary data.

The function returns:

- > 0 if the search data is greater than the element
- < 0 if the search data is less than the element
- 0 if the search data is the same as the element

SysBroadcastActionCode

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

Prototype
```
Err SysBroadcastActionCode (Word cmd, Ptr cmdPBP)
```

Parameters

`cmd`	Action code to send.
`cmdPBP`	Action code parameter block to send.

Result Returns 0 if no error, or one of the following errors: `sysErrParamErr, memErrNotEnoughSpace, sysErrOutOfOwnerIDs.`

Comments Launch codes are discussed in some detail in Chapter 2 of Developing Palm OS Applications, Part I.

See Also SysAppLaunch

SysCopyStringResource

Purpose Copy a resource string to a passed string.

Prototype
```
void SysCopyStringResource ( CharPtr string,
                             UInt theID)
```

Parameters

`string`	String to copy the resource string to.
`theID`	Resource string ID.

Result Stores a copy of the resource string in `string`.

SysCreateDataBaseList

Purpose

Generate a list of databases found on the memory cards matching a specific type and return the result. If `lookupName` is true then a name in a tAIN resource is used instead of the database's name and the list is sorted. Only the last version of a database is returned. Databases with multiple versions are listed only once.

Prototype

```
Boolean SysCreateDataBaseList( ULong type,
                               ULong creator,
                               WordPtr dbCount,
                               Handle *dbIDs,
                               Boolean lookupName)
```

Parameters

`type`	Type of database to find (0 for wildcard).
`creator`	Creator of database to find (0 for wildcard).
`dbCount`	Pointer to contain count of matching databases.
`dbIDs`	Pointer to handle allocated to contain the database list.
`lookupName`	Use tAIN names and sort the list.

Result

Returns `FALSE` if no databases were found, `TRUE` if databases were found. `dbCount` is updated to the number of databases found; `dbIDs` is updated to the list of matching databases found.

SysCreatePanelList

Purpose Generate a list of panels found on the memory cards and return the result. Multiple versions of a panel are listed once.

Prototype

```
Boolean SysCreatePanelList(
                WordPtr panelCount,
                Handle *panelIDs)
```

Parameters

`panelCount` Pointer to set to the number of panels.

`panelIDs` Pointer to handle containing a list of panels.

Result Returns FALSE if no panels were found, TRUE if panels were found. `panelCount` is updated to the number of panels found; `panelIDs` is updated to the IDs of panels found.

SysCurAppDatabase

Purpose Return the card number and database ID of the current application's resource database.

Prototype

```
Err SysCurAppDatabase (  UIntPtr cardNoP,
                         LocalID* dbIDP)
```

Parameters

`cardNoP` Pointer to the card number; 0 or 1.

`dbIDB` Pointer to the database ID.

Result Returns 0 if no error, or `SysErrParamErr` if an error occurs.

See Also `SysAppLaunch`, `SysUIAppSwitch`

SysErrString

Purpose Returns text to describe an error number. This routine looks up the textual description of a system error number in the appropriate List resource and creates a string that can be used to display that error.

The actual string will be of the form: "<error message> (XXXX)" where XXXX is the hexadecimal error number.

This routine looks for a resource of type 'tstl' and resource ID of (err>>8). It then grabs the string at index (err & 0x00FF) out of that resource.

Note: The first string in the resource is called index #1 by Constructor, NOT #0. For example, an error code of 0x0101 will fetch the first string in the resource.

Prototype

```
CharPtr SysErrString(  Err err,
                       CharPtr strP,
                       Word maxLen)
```

Parameters

`err`	Error number
`strP`	Pointer to space to form the string
`maxLen`	Size of strP buffer.

Result Stores the error number string.

SysFatalAlert

Purpose Display a fatal alert until the user taps a button in the alert.

Prototype

```
UInt SysFatalAlert (CharPtr msg)
```

Parameters

`msg`	Message to display in the dialog.

Result The button tapped; first button is zero.

SysFormPointerArrayToStrings

Purpose Form an array of pointers to strings in a block. Useful for setting the items of a list.

Prototype

```
VoidHand SysFormPointerArrayToStrings
                              (CharPtr c,
                              Int stringCount)
```

Parameters

`c`	Pointer to packed block of strings, each terminated by NULL.
`stringCount`	Count of strings in block.

Result Unlocked handle to allocated array of pointers to the strings in the passed block. The returned array points to the strings in the passed packed block.

SysGetOSVersionString

Purpose Return the version number of the Palm operating system.

Prototype

```
CharPtr SysGetOSVersionString()
```

Parameters None.

Result Returns a string such as “v. 3.0.”

Comments You must free the returned string using the `MemPtrFree` function.

SysGetRomToken

Return from ROM a value specified by token.

Prototype

```
Err  SysGetROMToken(Word cardNo, DWord token,
                    BytePtr *dataP, WordPtr sizeP )
```

Parameters

`cardNo` — The card on which the ROM to be queried resides. Currently, no Palm hardware provides multiple cards, so this value must be 0.

`token` — The value to retrieve, as specified by one of the following tokens:

`sysROMTokenSerial`
: The serial number of the ROM, expressed as a text string with no null terminator.

<-- `dataP` — Pointer to a text buffer that holds the requested value when the function returns.

<-- `sizeP` — The number of bytes in the `dataP` buffer.

Result

Returns the requested value if no error, or an error code if an error occurs. If this function returns an error, or if the returned pointer to the buffer is `NULL`, or if the first byte of the text buffer is `0xFF`, then no serial number is available.

Comments

This function is available only on Palm OS version 3.0 and greater. Serial numbers are available only on flash ROM-based units.

The serial number is shown to the user in the Application Launcher, along with a checksum digit you can use to validate input when your users read the ID from their device and type it in or tell it to someone else.

See Also

"Retrieving the ROM Serial Number" starting on page 51 shows how to retrieve the ROM serial number and calculate its associated checksum.

SysGetStackInfo

Purpose Return the start and end of the current thread's stack.

Prototype

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

Parameters

`startPP` Upon return, points to the start of the stack.

`endPP` Upon return, points to the end of the stack.

Result 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.

SysGraffitiReferenceDialog

Purpose Pop up the Graffiti Reference Dialog.

Prototype

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

Parameters

`referenceType` Which reference to display. See `GraffitiReference.h` for more information.

Result Nothing returned.

SysGremlins

Purpose

Query the Gremlins facility. You pass a selector for a function and parameters for that function. Gremlins performs the function call and returns the result.

Prototype

```
DWord SysGremlins(   GremlinFunctionType selector,
                     GremlinParamsType *params)
```

Parameters

`selector` The selector for a function to pass to Gremlins.

`params` Pointer to a parameter block used to pass parameters to the function specified by `selector`.

Result

Returns the result of the function performed in Gremlins.

Comments

Currently, only one selector is defined, `GremlinIsOn`, which takes no parameters. `GremlinIsOn` returns 0 if Gremlins is not running, non-zero if it is running.

Currently, non-zero values are returned only from the version of Gremlins in the Palm OS emulator. The Gremlins running in the simulator and over the serial line via the Palm Debugger return zero for `GremlinIsOn`.

Use this function if you need to alter the application's behavior when Gremlins is running. For example, the debug 3.0 ROM refuses to bring up the digitizer panel when Gremlins is running under the emulator.

SysHandleEvent

Purpose Handle defaults for system events such as hard and soft key presses.

Prototype
```
Boolean SysHandleEvent (EventPtr eventP)
```

Parameters

`eventP`	Pointer to an event.

Result Returns `TRUE` if the system handled the event.

Comments Applications should call this routine immediately after calling `EvtGetEvent` unless they want to override the default system behavior. However, overriding the default system behavior is almost never appropriate for an application.

See Also `EvtProcessSoftKeyStroke`, `KeyRates` (documented in *Developing Palm OS Applications, Part I*)

SysInsertionSort

Purpose Sort elements in an array according to the passed comparison function.

Prototype
```
void SysInsertionSort (Byte baseP,
                       Int numOfElements,
                       Int width,
                       CmpFuncPtr comparF,
                       Long other)
```

Parameters

`baseP`	Base pointer to an array of elements.
`numOfElements`	Number of elements to sort (must be at least 2).
`width`	Width of an element.
`comparF`	Comparison function (see Comments).

`other`	Other data passed to the comparison function.

Result

Returns nothing.

Comments

Only elements which are out of order move. Moved elements are moved to the end of the range of equal elements. If a large amount of elements are being sorted, try to use the quick sort (see SysQSort).

This is the insertion sort algorithm: Starting with the second element, each element is compared to the preceding element. Each element not greater than the last is inserted into sorted position within those already sorted. A binary search for the insertion point is performed. A moved element is inserted after any other equal elements.

In Palm OS 2.0 and later, `DmComparF` has 6 parameters.

These parameters allow a Palm OS application to pass more information to the system than before, most noticeably the record (and all associated information) which allows sorting by unique ID, so that the Palm OS device and the desktop always match.

The revised callback is used by new sorting routines (and can be used the same way by your application):

```
typedef Int DmComparF (  void *,
                         void *,
                         Int other,
                         SortRecordInfoPtr,
                         SortRecordInfoPtr,
                         VoidHand appInfoH);
```

As a rule, this change in the number of arguments doesn't cause problems when a 1.0 application is run on a 2.0 or later device, because the system only pulls the arguments from the stack that are there.

Note, however, that some optimized applications built with tools other than Metrowerks CodeWarrior for Palm OS may have prob-

lems as a result of the change in arguments when running on a 2.0 or later device.

The 2.0 comparison function (`comparF`) has this prototype:

```
Int comparF (VoidPtr, VoidPtr, Long other);
```

The 1.0 comparison function (`comparF`) had this prototype:

```
Int comparF (BytePtr A, BytePtr B, Long other);
```

The function returns:

- > 0 if A > B
- < 0 if A< B
- 0 if A = B

See Also SysQSort

SysInstall

Purpose Entry point for System code resource, `'CODE' #0`, in the System resource file.

Prototype

```
void SysInstall (Ptr tableP[])
```

Parameters

`tableP` Pointer to trap table.

Result Returns nothing

Comments Called by `Init()` in the ROMMain module.

SysKeyboardDialog

Purpose Pop up the system keyboard if there is a field object with the focus. The field object's text chunk is edited directly.

Prototype
```
void SysKeyboardDialog (KeyboardType kbdType)
```

Parameters
`kbdType` The keyboard type. See `keyboard.h`.

Result Returns nothing. Changes the field's text chunk.

See Also `SysKeyboardDialogV10`, `FrmSetFocus` (documented in "Developing Palm OS Applications, Part I)

SysKeyboardDialogV10

Purpose Pop up the system keyboard if there is a field object with the focus. The field object's text chunk is edited directly.

Prototype
```
void SysKeyboardDialogV10 ()
```

Parameters None.

Result Returns nothing. The field's text chunk is changed.

See Also `SysKeyboardDialog`, `FrmSetFocus` (documented in "Developing Palm OS Applications, Part I)

SysLibFind

Purpose

A utility routine to return a reference number for a library that is already loaded, given its name.

Prototype

```
Err SysLibFind (CharPtr nameP, UIntPtr refNumP)
```

Parameters

`nameP`	Pointer to the name of a loaded library.
`refNumP`	Pointer to a variable for returning the library reference number (on failure, this variable is undefined)

Result

0 if no error; otherwise: `sysErrLibNotFound` (if the library is not yet loaded), or another error returned from the library's install entry point.

Comments

Most built-in libraries (net, serial, IR) are preloaded automatically when the system is reset. Third-party libraries must be loaded before this call can succeed (use SysLibLoad). You can check if a library is already loaded by calling `SysLibFind` and checking for a 0 error return value (it will return a non-zero value if the library is not loaded).

SysLibLoad

Purpose

A utility routine to load a library given its database creator and type.

Presently, the “load” functionality is NOT supported when you use the Palm OS Simulator.

Prototype

```
Err SysLibLoad(  DWord libType,
                 DWord libCreator,
                 UIntPtr refNumP)
```

Parameters

`libType`	Type of library database.
`libCreator`	Creator of library database.
`refNumP`	Pointer to variable for returning the library reference number(on failure, `sysInvalidRefNum` is returned in this variable)

Result

0 if no error; otherwise: `sysErrLibNotFound`, `sysErrNoFreeRAM`, `sysErrNoFreeLibSlots`, or other error returned from the library's install entry point

Comments

When an application no longer needs a library that it SUCCESSFULLY loaded via `SysLibLoad`, it is responsible for unloading the library by calling `SysLibRemove` and passing it the library reference number returned by `SysLibLoad`. More information is available in the white paper on shared libraries, which you can find on the Palm developer support web site.

SysQSort

Purpose

Sort elements in an array according to the passed comparison function. Equal records can be in any position relative to each other because a quick sort tends to scramble the ordering of records. As a result, calling `SysQSort` multiple times can result in a different order if the records are not completely unique. If you don't want this behavior, use the insertion sort instead (see SysInsertionSort).

To pick the pivot point, the quick sort algorithm picks the middle of three records picked from around the middle of all records. That way, the algorithm can take advantage of partially sorted data.

These optimizations are built in:

- The routine contains its own stack to limit uncontrolled recursion. When the stack is full, an insertion sort is used because it doesn't require more stack space.
- An insertion sort is also used when the number of records is low. This avoids the overhead of a quick sort which is noticeable for small numbers of records.
- If the records seem mostly sorted, an insertion sort is performed to move only those few records that need to be moved.

Prototype

```
void SysQSort (  Byte baseP,
                 Int numOfElements,
                 Int width,
                 CmpFuncPtr comparF,
                 Long other)
```

Parameters

`baseP`	Base pointer to an array of elements.
`numOfElements`	Number of elements to sort (must be at least 2).
`width`	Width of an element.
`comparF`	Comparison function. See Comments for `SysInsertionSort`.
`other`	Other data passed to the comparison function.

Result Returns nothing.

See Also SysInsertionSort

SysRandom

Purpose Return a random number anywhere from 0 to `sysRandomMax`.

Prototype

```
Int SysRandom (ULong newSeed)
```

Parameters

`newSeed` New seed value, or 0 to use existing seed.

Result Returns a random number.

SysReset

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

Prototype

```
void SysReset (void)
```

Parameters None.

Result No return value.

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 hidden reset switch on the device.

When running an application using the simulator, this routine looks for two data files that represent the memory of card 0 and card 1. If these are found, the Palm OS memory image is created using them. If they are not found, they are created.

When running an application on the device, this routine simply looks for the memory cards at fixed locations.

SysSetAutoOffTime

Purpose Set the time out value in seconds for auto-power-off. Zero means never power off.

Prototype
```
UInt SysSetAutoOffTime (UInt seconds)
```

Parameters

`seconds`	Time out in seconds, or 0 for no time out.

Result Returns previous value of time out in seconds.

SysStringByIndex

Purpose Copy a string out of a string list resource by index. String list resources are of type 'tSTL' and contain a list of strings and a prefix string.

Warning: ResEdit always displays the items in the list as starting at 1, not 0. Consider this when creating your string list.

Prototype
```
CharPtr SysStringByIndex(  Word resID,
                           Word index,
                           CharPtr strP,
                           Word maxLen)
```

Parameters

`resID`	Resource ID of the string list.
`index`	String to get out of the list.
`strP`	Pointer to space to form the string.
`maxLen`	Size of `strP` buffer.

Result Returns a pointer to the copied string. The string returned from this call will be the prefix string appended with the designated index string. Indices are 0-based; index 0 is the first string in the resource.

SysTaskDelay

Purpose Put the processor into doze mode for the specified number of ticks.

Prototype
```
Err SysTaskDelay (Long delay)
```

Parameters `delay` Number of ticks to wait (see `SysTicksPerSecond`)

Result Returns 0 if no error.

See Also `EvtGetEvent`

SysTicksPerSecond

Purpose Return the number of ticks per second. This routine allows applications to be tolerant of changes to the ticks per second rate in the system.

Prototype
```
Word SysTicksPerSecond(void)
```

Parameters None

Result Returns the number of ticks per second.

SysUIAppSwitch

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

NOTE: For important information regarding the correct use of this function, see "Opening Applications Programmatically" on page 62.

Prototype

```
Err SysUIAppSwitch(  UInt cardNo,
                     LocalID dbID,
                     Word cmd,
                     Ptr cmdPBP)
```

Parameters

`cardNo`	Card number for the new application; currently only card 0 is valid.
`dbID`	ID of the new application.
`cmd`	Action code (launch code). See *Developing Palm OS Applications, Part I.*
`cmdPBP`	Action code (launch code) parameter block.

Result

Returns 0 if no error.

Comments

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`, as shown in Listing 1.13, "Opening the Launcher," on page 72.

See Also

`SysAppLaunch`

Functions for System Use Only

SysAppExit

Prototype

```
Err SysAppExit (SysAppInfoPtr appInfoP,
       Ptr prevGlobalsP, Ptr globalsP)
```

WARNING: System Use Only!

SysAppInfoPtr

Prototype

```
SysAppInfoPtr SysCurAppInfoP (void)
```

WARNING: System Use Only!

SysAppStartup

Prototype

```
Err    SysAppStartup ( SysAppInfoPtr appInfoPP,
       Ptr prevGlobalsP, Ptr globalsP)
```

WARNING: System Use Only!

SysBatteryDialog

Prototype

```
void SysBatteryDialog (void)
```

WARNING: System Use Only!

SysCardImageDeleted

Prototype

```
void SysCardImageDeleted (UInt cardNo)
```

WARNING: System Use Only!

SysCardImageInfo

Prototype

```
Ptr SysCardImageInfo (UInt cardNo, ULongPtr sizeP)
```

WARNING: System Use Only!

SysColdBoot

Purpose Perform a cold boot and reformat all RAM areas of both memory cards.

WARNING: System Use Only!

SysCurAppInfoP

Prototype

```
SysCurAppInfoPtr SysCurrAppInfoP (void)
```

WARNING: System Use Only!

SysDisableInts

Prototype

```
Word SysDisableInts (void)
```

WARNING: System Use Only!

SysDoze

Prototype

```
void SysDoze (Boolean onlyNMI)
```

WARNING: System Use Only!

SysEvGroupCreate

Prototype

```
Err SysEvGroupCreate(DWordPtr evIDP, DWordPtr
tagP, DWord init)
```

WARNING: System Use Only!

SysGetAppInfo

Prototype

```
SysAppInfoPtr SysGetAppInfo(
                SysAppInfoPtr *uiAppPP,
                SysAppInfoPtr *actionCodeAppPP)
```

WARNING: System Use Only!

SysEvGroupRead

Prototype

```
Err SysEvGroupRead(DWord evID, DWordPtr valueP)
```

WARNING: System Use Only!

SysEvGroupSignal

Prototype

```
Err SysEvGroupSignal(DWord evID, DWord mask, DWord
value, SDWord type)
```

WARNING: System Use Only!

SysEvGroupWait

Prototype

```
Err SysEvGroupWait(DWord evID, DWord mask, DWord
value, SDWord matchType, SDWord timeout)
```

WARNING: System Use Only!

SysGetTrapAddress

Prototype

```
VoidPtr SysGetTrapAddress (UInt trapNum)
```

WARNING: System Use Only!

SysInit

Prototype

```
void SysInit (void)
```

WARNING: System Use Only!

SysKernelInfo

Prototype

```
Err SysKernelInfo (VoidPtr paramP)
```

WARNING: System Use Only!

SysLaunchConsole

Prototype

```
Err SysLaunchConsole (void)
```

WARNING: System Use Only!

SysLibInstall

Prototype

```
Err   SysLibInstall ( SysLibEntryProcPtr libraryP,
                      UIntPtr refNumP)
```

WARNING: System Use Only!

SysLibRemove

Prototype

```
Err SysLibRemove (UInt refNum)
```

WARNING: System Use Only!

SysLibTblEntry

Prototype

```
SysLibTblEntryPtr SysLibTblEntry (UInt refNum)
```

WARNING: System Use Only!

SysMailboxCreate

Prototype

```
Err SysMailboxCreate(DWordPtr mbIDP, DWordPtr
tagP, DWord depth)
```

WARNING: System Use Only!

SysMailboxDelete

Prototype

```
Err SysMailboxDelete(DWord mbID)
```

WARNING: System Use Only!

SysMailboxFlush

Prototype

```
Err SysMailboxFlush(DWord mbID)
```

WARNING: System Use Only!

SysMailboxSend

Prototype

```
Err SysMailboxSend(DWord mbID, VoidPtr msgP, DWord
wAck)
```

WARNING: System Use Only!

SysMailboxWait

Prototype

```
Err SysMailboxWait(DWord mbID, VoidPtr msgP, DWord
priority, SDWord timeout)
```

WARNING: System Use Only!

SysNewOwnerID

Prototype

```
UInt SysNewOwnerID (void)
```

WARNING: System Use Only!

SysPowerOn

Prototype

```
void SysPowerOn (  Ptr card0P, ULong card0Size,
                   Ptr card1P, ULong card1Size,
                   DWord sysCardHeaderOffset,
                   Boolean reFormat)
```

WARNING: System Use Only!

SysRestoreStatus

Prototype

```
void SysRestoreStatus (Word status)
```

WARNING: System Use Only!

SysSetA5

Prototype

```
DWord SysSetA5 (DWord newValue)
```

WARNING: System Use Only!

SysSetTrapAddress

Prototype

```
Err SysSetTrapAddress (  UInt trapNum,
                         VoidPtr procP)
```

WARNING: System Use Only!

SysSleep

Prototype

```
void SysSleep (  Boolean untilReset,
                 Boolean emergency)
```

WARNING: System Use Only!

SysTaskResume

Prototype

```
Err SysTaskResume(DWord taskID)
```

WARNING: System Use Only!

SysTaskSuspend

Prototype

```
Err SysTaskSuspend(DWord taskID)
```

WARNING: System Use Only!

SysUILaunch

Prototype

```
void SysUILaunch (void)
```

WARNING: System Use Only!

SysTaskWait

Prototype

```
Err SysTaskWait(SDWord timeout)
```

WARNING: System Use Only!

SysTaskWaitClr

Prototype

```
Err SysTaskWaitClr(void)
```

WARNING: System Use Only!

SysTaskWake

Prototype

```
Err SysTaskWake(DWord taskID)
```

WARNING: System Use Only!

Time Manager Functions

DateAdjust

Purpose Return a new date +/- the days adjustment.

Prototype

```
void DateAdjust (DatePtr dateP, Long adjustment)
```

Parameters

`dateP`	A `DateType` structure with the date to be adjusted (see `DateTime.h`).
`adjustment`	The adjustment in number of days.

Result Changes `dateP` to contain the new date.

Comments This function is useful for advancing a day or week and not worrying about month and year wrapping.

If the time is advanced out of bounds, it is cut at the bounds surpassed.

DateDaysToDate

Purpose Return the date, given days.

Prototype

```
void DateDaysToDate (ULong days, DatePtr dateP)
```

Parameters

`days`	Days since 1/1/1904.
`dateP`	Pointer to `DateType` structure (returned).

Result Returns nothing, stores the date in `dateP`.

See Also `TimAdjust`, `DateToDays`

DateSecondsToDate

Purpose Return the date given seconds.

Prototype

```
void DateSecondsToDate ( ULong seconds,
                         DatePtr dateP)
```

Parameters

`seconds`	Seconds since 1/1/1904.
`dateP`	Pointer to `DateType` structure (returned).

Result Returns nothing; stores the date in `dateP`.

DateToAscii

Purpose Convert the time passed to an ASCII string in the passed `DateFormatType`. Handles long and short formats.

Prototype

```
void DateToAscii(  Byte months,
                   Byte days,
                   Word years,
                   DateFormatType dateFormat,
                   CharPtr pString)
```

Parameters

`months`	Months (1-12).
`days`	Days (1-31).
`years`	Years (for example 1995).
`dateFormat`	Long or short `DateFormatType`.
`pString`	Pointer to string which gets the result. Must be of length `dateStringLength` for standard formats or `longDateStrLength` for long date formats.

Result Returns nothing. Stores the result in `pString`.

See Also `TimeToAscii`, `DateToDOWDMFormat`

DateToDays

Purpose Return the date in days since 1/1/1904.

Prototype

```
ULong DateToDays (DateType date)
```

Parameters

`date`	`DateType` structure.

Result Returns the days since 1/1/1904.

See Also `TimAdjust`, `DateDaysToDate`

DateToDOWDMFormat

Purpose Convert the date passed to an ASCII string.

Prototype

```
void DateToDOWDMFormat(  Byte months,
                         Byte days,
                         Word years,
                         DateFormatType
dateFormat,
                         CharPtr pString)
```

Parameters

`months`	Month (1-12).
`days`	Day (1-31).
`years`	Years (for example 1995).
`dateFormat`	`FALSE` to use AM and PM.
`pString`	Pointer to string which gets the result. The string must be of length `timeStringLength`.

Result Returns nothing; stores ASCII string in `pString`.

See Also `DateToAscii`

DayOfMonth

Purpose Return the day of a month on which the specified date occurs (for example, `dom2ndTue`).

Prototype

```
UInt DayOfMonth (UInt month, UInt day, UInt year)
```

Parameters

`month`	Month (1-12).
`day`	Day (1-31).
`year`	Year (for example 1995).

Result Returns the day of the month as a `DayOfWeekType`, see `DateTime.h`.

DayOfWeek

Purpose Return the day of the week.

Prototype

```
UInt DayOfWeek (UInt month, UInt day, UInt year)
```

Parameters

`month`	Month (1-12).
`day`	Day (1-31).
`year`	Year (for example 1995).

Result Returns the day of the week (Sunday = 0, Monday = 1, etc.).

DaysInMonth

Purpose Return the number of days in the month.

Prototype

```
UInt DaysInMonth (UInt month, UInt year)
```

Parameters

`month`	Month (1-12).
`year`	Year (for example, 1995).

Result Returns the number of days in the month for that year.

TimAdjust

Purpose Return a new date, +/- the time adjustment.

Prototype

```
void TimAdjust(  DateTimePtr dateTimeP,
                 Long adjustment)
```

Parameters

`dateTimeP`	A `DateType` structure (see `DateTime.h`).
`adjustment`	The adjustment in seconds.

Result Returns nothing. Changes `dateTimeP` to the new date and time.

Comments This function is useful for advancing a day or week and not worrying about month and year wrapping.

If the time is advanced out of bounds it is cut at the bounds surpassed.

See Also `DateAdjust`

TimDateTimeToSeconds

Purpose Return the date and time in seconds since 1/1/1904.

Prototype
```
ULong TimDateTimeToSeconds (DateTimePtr dateTimeP)
```

Parameters `dateTimeP` A `DateType` structure (see `DateTime.h`).

Result The time in seconds since 1/1/1904.

See Also `TimSecondsToDateTime`

TimGetSeconds

Purpose Return seconds since 1/1/1904.

Prototype
```
ULong TimGetSeconds (void)
```

Parameters None.

Result Returns the number of seconds.

See Also `TimSetSeconds`

TimGetTicks

Purpose Return the tick count since the last reset. The tick count does not advance while the device is in sleep mode.

Prototype
```
ULong TimGetTicks (void)
```

Parameters None.

Result Returns the tick count.

TimSecondsToDateTime

Purpose Return the date and time, given seconds.

Prototype
```
void TimSecondsToDateTime( ULong seconds,
                           DateTimePtr dateTimeP)
```

Parameters

`seconds` Seconds to advance from 1/1/1904.

`dateTimeP` A `DateTimeType` structure that's filled by the function.

Result Returns nothing. Stores the date and time given seconds since 1/1/1904 in `dateTimeP`.

See Also `TimDateTimeToSeconds`

TimSetSeconds

Purpose Return seconds since 1/1/1904.

Prototype
```
void TimSetSeconds (ULong seconds)
```

Parameters

`seconds` Place to return the seconds since 1/1/1904.

Result Returns nothing; modifies `seconds`.

See Also `TimGetSeconds`

TimeToAscii

Purpose Convert the time passed to an ASCII string.

Prototype

```
void TimeToAscii(  Byte hours,
                   Byte minutes,
                   TimeFormatType timeFormat,
                   CharPtr pString)
```

Parameters

`hours`	Hours (0-23).
`minutes`	Minutes (0-59).
`timeFormat`	FALSE to use AM and PM.
`pString`	Pointer to string which gets the result. Must be of length `timeStringLength`.

Result Returns nothing. Stores pointer to the text of the current selection in `pString`.

See Also `DateToAscii`

Functions for System Use Only

TimGetAlarm

Prototype

```
ULong TimGetAlarm (void)
```

WARNING: System use only!

TimHandleInterrupt

Prototype

```
void TimHandleInterrupt (Boolean periodicUpdate)
```

Warning: System use only!

TimInit

Prototype

```
Err TimInit (void)
```

Warning: System use only!

TimSetAlarm

Prototype

```
ULong TimSetAlarm (ULong alarmSeconds)
```

Warning: System use only!

Index

Numerics

0.01-second timer 68
1-second timer 68

A

accented characters and StrToLower 131
adding event to event queue 80
alarm manager 18–20
    and alarm sound 19
    reminder dialog boxes 19
alarm sound 19, 36
alarms
    canceling 74
    setting 74
alerts
    SysFatalAlert 165
AlmCancelAll 75
AlmDisplayAlarm 75
AlmEnableNotification 75
almErrFull 74
almErrMemory 74
AlmGetAlarm 73
AlmInit 75
AlmSetAlarm 20, 74
application preferences 113
application-defined features 26
auto-off 57
    setting 178
    timer 67, 90
auto-repeat 61, 66

B

base 10 form of floating-point number 103
battery 57
battery conservation using modes 56
battery timeout 158, 159
battery voltage warning threshold 158, 159
booting 54
bound of next line for global find 97
buttons
    silk-screened icons 60

C

C library
    and float manager 101
    and string manager 53
canceling alarms 74
cleanup of dynamic heap 58
Click 36
code #0 resource 172
Confirmation sound 36
conserving battery using modes 56
Crc16CalcBlock 109

D

database ID
    and launch codes 63
databases
    SysCreateDataBaseList 163
date and time manager 68
DateAdjust 189
DateDaysToDate 189
DateSecondsToDate 190
dateStringLength 190
DateToAscii 190
DateToDays 191
DateToDOWDMFormat 191
DayOfMonth 192
DayOfWeek 192
DaysInMonth 193
dialog boxes (reminder) 19
digitizer
    and pen queue 65
    EvtProcessSoftKeyStroke 90
    pen stroke to key event 64
DmComparF 171
doze mode 56
    SysTaskDelay 179
dynamic heap
    cleanup 58
    reinitializing 177

E

ErrCatch 80
ErrDisplay 21, 23, 76
ErrDisplayFileLineMsg 77
ErrEndCatch 80
ErrFatalDisplayIf 21, 22, 78
ErrNonFatalDisplayIf 79
error manager 21–25
    try-and-catch mechanism 23
Error sound 36
ERROR_CHECK_FUL 76
ERROR_CHECK_FULL 79
ERROR_CHECK_LEVEL 21, 23, 76, 78, 79
ERROR_CHECK_PARTIAL 76
ErrThrow 23, 80
ErrTry 80
event processing 59
event queue
    adding event 80
events
    hard button presses 59
    hardware generated 59, 60
    software generated 59, 61
EvtAddEventToQueue 80
EvtAddUniqueEventToQueue 81
EvtCopyEvent 81
EvtDequePenStrokeInfo 65
EvtDequeuePenPoint 82
EvtDequeuePenStrokeInfo 83
EvtEnableGraffiti 83
EvtEnqueueKey 84
EvtEventAvail 85
EvtFlushKeyQueue 85
EvtFlushNextPenStroke 86
EvtFlushPenQueue 86
EvtGetEvent 59
EvtGetPen 87
EvtGetPenBtnList 88
EvtKeyQueueEmpty 88
EvtKeyQueueSize 89
EvtPenQueueSize 89
EvtProcessSoftKeyStroke 90
EvtResetAutoOffTimer 67, 90
EvtSysEventAvail 91
EvtWakeup 91

F

fatal alert 165
feature manager 25–28
features
    See functions starting with Ftr
    application-defined 26
    system version 26
FIFO queue 60
file streaming functions 35
FileClearerr 133
FileClose 133
FileControl 134
FileDelete 136
FileDmRead 136
FileEOF 138
FileError 139
FileFlush 139
FileGetLastError 140
FileOpen 140
FileRead 143
FileReadLow 148
FileRewind 144
FileSeek 145
FileTell 146
FileTruncate 147
FileWrite 147
FindDrawHeader 97
FindGetLineBounds 97
FindSaveMatch 98
FindStrInStr 99
float manager overview 101
flushing pen queue 86
FplAdd 102
FplAToF 102
FplBase10Info 103
FplDiv 104
FplFloatToLong 104
FplFloatToULong 105
FplFree 105
FplFToA 106
FplInit 106

FplLongToFloat 107
FplMul 107
FplSub 108
ftrErrInternalError 93, 94
ftrErrNoSuchFeature 94, 95, 96
ftrErrNoSuchFtr 93
FtrGet 27, 93
FtrGetByIndex 27, 94
ftrInternalError 96
FtrSet 27, 95
FtrUnregister 27, 96

G

GetCharCaselessValue
    and FindStrInStr 99
global find
    FindDrawHeader 97
    FindGetLineBounds 97
Graffiti
    enabling and disabling 83
    events 59
Graffiti recognizer 64
    EvtDequeuPenPoint 82
Graffiti Reference Dialog 168

H

hard button press events 59
hardware-generated events 59, 60
header line for global find 97

I

insertion sort 171
interrrupting Sync application 58

K

kernel 57
key debouncing 61
key events
    format 84
    from pen strokes 64
key presses 59
key queue 66
    size 89
keyboard display 173

L

launch codes 61
    and returned database ID 63
    SysBroadcastActionCode 62, 162
launcher screen 58
launching applications 58
library vs. managers 17
lists
    setting items 166
longDateStrLength 190
low-battery warning 61

M

managers
    naming convention 17
    vs. libraries 17
MdmDial 110
mdmErrBusy 110
mdmErrCmdError 110
mdmErrNoDCD 110
mdmErrNoTone 110
mdmErrUserCan 110
MdmHangUp 111
memErrChunkLocked 95, 96
memErrInvalidParam 95, 96
memErrNotEnoughSpace 95, 96, 156, 162
modem 110
modes 55
    efficient use 56
multiple preferences 113
multitasking kernel 57

N

nilEvent 91
noPreferenceFound 113

P

panel list (SysCreatePanelList) 164
password functions 120
pen
    current status 87
    strokes and key events 64
pen events 59
pen queue 65

flushing 86
size 89
PhoneNumberLookup 111
power modes 55
preferences
auto-off 57
multiple application preferences 113
PrefGetAppPreferences 113
PrefGetAppPreferencesV10 114
PrefGetPreference 115
PrefGetPreferences 116
PrefOpenPreferenceDBV10 116
PrefSetAppPreferences 117
PrefSetAppPreferencesV10 118
PrefSetPreference 119
PrefSetPreferences 119
PwdExists 120
PwdRemove 120
PwdSet 121
PwdVerify 121

Q

quitting application 59

R

real-time clock 68
reinitializing dynamic memory heap 177
reminder dialog boxes 19
reset 177
ResLoadForm 112
ResLoadMenu 112
resource database (SysCurAppDatabase) 164
response time 58
running mode 56

S

searching for string 99
searching for substring 130
silk-screen buttons
EvtGetPenBtnList 88
silk-screened icons 60
sleep mode 55
and real-time clock 68
SndCreateMidiList 150
SndDoCmd 151
sndErrBadChannel 151
sndErrBadParam 151
sndErrQFull 151
SndGetDefaultVolume 152
SndInit 155
SndPlaySMF 152
SndPlaySystemSound 155
SndSetDefaultVolume 156
SndSysBeepType 155
soft reset 177
software-generated events 59, 61
sorting array elements 171
sound manager 35–52
sound manager functions 150–156
sprintf (StrPrintF) 130
StartApplication
and PrefGetPreferences 116
Startup sound 36
StrAToI 122
StrCaselessCompare 122
StrCat 123
StrChr 123
StrCompare 124
StrCopy 124
StrDelocalizeNumber 125
string
searching 99
string manager 53
string manager functions 122–132
string resource
copying 162
StrIToA 125
StrIToH 126
StrLen 126
StrLocalizeNumber 127
StrNCaselessCompare 127
StrNCat 128
StrNCompare 129
StrNCopy 129
strokes
capturing 65
translating 90
StrPrintF 130

StrStr 130
StrToLower 131
StrVPrintF 131
substring, searching for 130
Sync application 58
SysAppLaunch 58, 156
sysAppLaunchCmdAlarmTriggered 19
sysAppLaunchCmdDisplayAlarm 19
SysAppLauncherDialog 157
SysBatteryInfo 158
SysBatteryInfoV20 159
SysBinarySearch 160
SysBroadcastActionCode 62, 162
SysCopyStringResource 162
SysCreateDataBaseList 163
SysCreatePanelList 164
SysCurAppDatabase 62, 164
sysErrLibNotFound 174, 175
sysErrNoFreeLibSlots 175
sysErrNoFreeRAM 175
sysErrOutOfOwnerID 156
sysErrOutOfOwnerIDs 162
sysErrParamErr 156, 162
SysErrString 165
SysFatalAlert 165
SysFormPointerArrayToStrings 166
SysGetAppInfo 183
SysGetOSVersionString 166
SysGetRomToken 167
SysGetStackInfo 168
SysGraffitiReferenceDialog 168
SysGremlins 169
SysHandleEvent 59, 60, 170
SysInsertionSort 170
SysInstall 172
SysKeyboardDialog 173
SysKeyboardDialogV10 173
SysLibFind 174
SysLibLoad 175
SysQSort 176
SysRandom 177
sysRandomMax 177
SysReset 177
SysSetAutoOffTime 178
SysStringByIndex 178
SysTaskDelay 179
system event manager 63–67
system events
  checking availability 91
system keyboard display 173
system ticks 68
  and Simulator 68
  on Palm OS device 68
system version feature 26
SysTicksPerSecond 179
sysTicksPerSecond 69
SysUIAppSwitch 62, 180

T

TimAdjust 193
TimDateTimeToSeconds 68, 194
time manager 68
  structures 69
TimeToAscii 196
TimGetSeconds 68, 194
TimGetTicks 69, 194
timing 69
TimSecondsToDateTime 68, 195
TimSetSeconds 68, 195
try-and-catch mechanism 23
  example 24

U

UIAS 55, 57
User Interface Application Shell 55, 57
using modes efficiently 56

V

voltage warning threshol 158, 159
vsprintf (StVPrintF) 131

W

Warning sound 36

Index