Palm OS® Programmer's Companion

Volume II Communications

Written by Greg Wilson, Jean Ostrem, Christopher Bey, Eric Shepherd, and Mark Dugger
Technical assistance from Ludovic Ferrandis, Gilles Fabre, David Fedor, Roger Flores, Steve Lemke, Bob Ebert, Ken Krugler, Paul Plaquette, Bruce Thompson, Jesse Donaldson, Tim Wiegman, Gavin Peacock, Ryan Robertson, Andy Stewart, and Waddah Kudaimi

Palm OS Programmer's Companion, Volume II: Communications
Document Number 3005-009
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

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

Table of Contents

About This Document

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

Palm OS SDK Documentation

The following documents are part of the SDK:

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

What This Volume Contains

This volume is designed for random access. That is, you can read any chapter in any order.

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

Here is an overview of this volume:

- Chapter 1, "Object Exchange." Describes how applications use the Exchange Manager to send and receive typed data objects.

- Chapter 2, "Exchange Libraries." Describes how to implement an exchange library.
- Chapter 3, "Personal Data Interchange." Describes the PDI library, which you use to exchange Personal Data Interchange (PDI) information with other devices and media
- Chapter 4, "Beaming (Infrared Communication)." Describes the two facilities for beaming, or IR communication: the exchange manager and the IR library.
- Chapter 5, "Serial Communication." Describes the serial port hardware, the serial communications architecture, the serial link protocol, and the various serial communication managers.
- Chapter 6, "Bluetooth."Describes how to use the Bluetooth APIs to access the Palm OS Bluetooth system and write Bluetooth-enabled applications.
- Chapter 7, "Network Communication." Describes the net library and Internet library and how to perform communications with networking protocols such as TCP/IP and UDP. The net library API maps very closely to the Berkeley UNIX sockets API.
- Chapter 8, "Secure Sockets Layer (SSL)." Describes the Secure Sockets Layer library. This library lets you apply SSL security to your network sockets.
- Chapter 9, "Internet and Messaging Applications." Describes the Palm.Net system and how to use the Web Clipping Application Viewer and iMessenger applications to access and send information using the wireless capabilities of the Palm VII™ device.
- Chapter 10, "Telephony Manager." Describes the component parts of the telephony API and shows how to use the telephony API in your applications.

Additional Resources

- Documentation

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

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

- Training

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

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

- Knowledge Base

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

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

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.
italic	Emphasis (for other elements).
blue and underlined	Hot links.

1

Object Exchange

The simplest form of communication for a Palm OS® application to implement is the sending and receiving of typed data objects, such as MIME data, databases, or database records.

You use the Exchange Manager to send and receive typed data objects. The Exchange Manager interface is independent of the transport mechanism. You can use IR, SMS, or any other protocol that has an Exchange Manager plug-in called an **exchange library**.

The Exchange Manager is supported in Palm OS 3.0 and higher. In Palm OS 4.0, significant updates were made.

This chapter describes how applications use the Exchange Manager to send and receive typed data objects. It covers the following topics:

This chapter does not describe how to implement an exchange library.

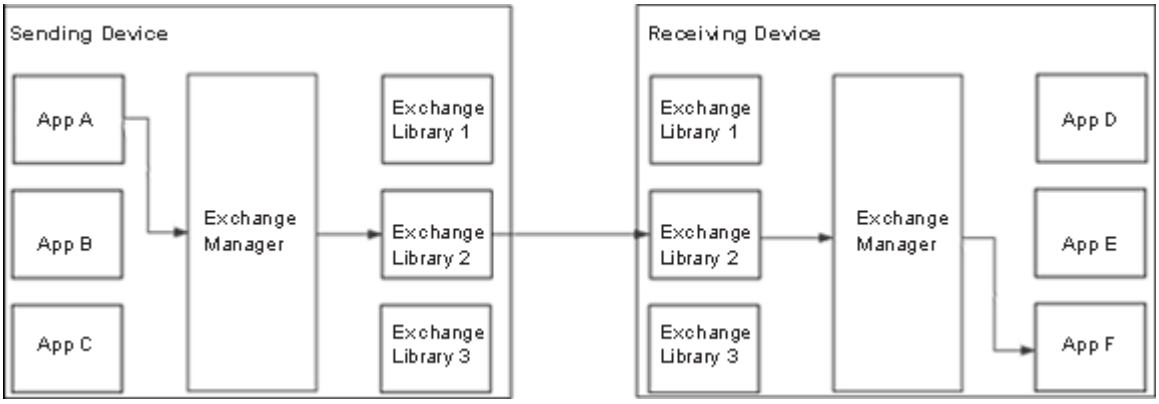
About the Exchange Manager

This section explains concepts you need to know before you can begin using the Exchange Manager. It discusses the following topics:

Exchange Libraries

The Exchange Manager works in conjunction with an exchange library. Each **exchange library** is transport-dependent and performs the actual communication with the remote device. When an application makes an Exchange Manager call, the Exchange Manager forwards the request to the appropriate exchange library. The Exchange Manager's main duty is to maintain a registry of which libraries implement each protocol and which applications receive each type of data. See Figure 1.1.

Figure 1.1 Object exchange using Exchange Manager



The list of supported exchange libraries depends on the version of Palm OS. See Table 1.1.

Table 1.1 Supported exchange libraries

Exchange Library	Minimum Palm OS Version
IR Library (IrDA)	Palm OS 3.0
Local Exchange Library	Palm OS 4.0
SMS Library (Short Messaging System)	Palm OS 4.0
Bluetooth Library[1]	Palm OS 4.0

1. The Bluetooth Library is not present in Palm OS 4.0, but is planned to be provided shortly after Palm OS 4.0 ships.

As other exchange libraries become available, users can install them on their Palm Powered™ handhelds and use the communications functionality they provide.

Note that on Palm OS 3.X the only exchange library available is the IR Library, and it is not extensible. The IR Library cannot, for example, be replaced with a different exchange library.

Typed Data Objects

The Exchange Manager sends and receives typed data objects. A **typed data object** (or **object**) is a stream of bytes plus some information about its contents. The content information includes any of: a creator ID, a MIME data type, or a filename.

The object itself can be in any format, but it's best to use a standardized data format rather than a proprietary one if you have a choice. Table 1.2 lists the standardized data formats that the built-in Palm OS applications can receive.

Table 1.2 Built-in applications and standard data types

Application	Data Type
Address Book	vCards (vcf file extension, text/x-vCard MIME type)
Datebook	vCalendars (vcs file extension, text/x-vCalendar MIME type)
Launcher	Palm OS databases (prc, pdb, oprc, and pqa file extensions, application/x-pilot and application/vnd.palm MIME types)
Memo	Plain text (txt file extension, text/plain MIME type)
ToDo	Not explicitly registered, but receives vCalendar objects from Datebook as appropriate

NOTE: The MIME type application/vnd.palm has been registered with the IANA and is preferred over the application/x-pilot MIME type.

If you want your application to receive objects, you must first register with the Exchange Manager for the type of data you want to receive. See "Registering for Data" for instructions on how to do so. You can override the built-in applications by registering for any data type listed in Table 1.2 and becoming the default application for that type. See "Setting the Default Application" for more information.

If you only want to send data, you do not have to register. Your application can send data of the types listed in Table 1.2, and the Exchange Manager ensures that the appropriate application receives it.

Initializing the Exchange Socket Structure

The Exchange Manager, exchange library, and application use an exchange socket structure (`ExgSocketType`) to communicate with

each other. This structure is passed from the application to the Exchange Manager to the exchange library and vice versa. (The use of the term "socket" in the Exchange Manager API is not related to the term "socket" as used in sockets communication programming.) When your application sends data, you must create this structure and initialize it with the appropriate information. When you receive data, this structure provides information about the connection and the incoming data.

The `ExgSocketType` structure you use must identify two important pieces of information:

- the exchange library that should do the sending (see "Identifying the Exchange Library")
- the type of data being sent (see "Identifying the Type of Data")

The socket structure defines other fields that you may use to provide other information if you want. See the description of the `ExgSocketType` structure in the *Palm OS Programmer's API Reference* for complete details.

IMPORTANT: When initializing the `ExgSocketType` structure, set all unused fields to 0.

Identifying the Exchange Library

The `ExgSocketType` structure identifies the library to be used in one of the following ways:

- a library reference number in the `libraryRef` field
- a Uniform Resource Locator (URL) in the `name` field

The Exchange Manager checks for a library reference number first. If it is 0, it checks for a URL.

When your application sends data, it must identify which exchange library to use. You only need to identify the exchange library in Palm OS 4.0 and higher. Earlier releases contain only one exchange library (for IR), so all sending is automatically done by that library. If you do not specify an exchange library on Palm OS 4.0 and higher, the IR Library is used to maintain backward compatibility.

It's more common to identify the library using a URL instead of a library reference number. The URL scheme specifies which exchange library to use. The **scheme** is the part of the URL that appears before the colon (:). For example, the scheme in the following URL is "http"

```
http://www.palmos.com
```

When you pass the preceding URL to a web browser, the scheme tells the browser to connect to the server using the HTTP protocol. Similarly, when you pass the Exchange Manager a URL, the scheme tells the Exchange Manager which exchange library to use. For example, the following URL tells the Exchange Manager to connect to a remote Palm Powered handheld using the IR Library:

```
_beam:BusinessCard.vcf
```

On Palm OS, a URL has the following format:

```
[?]scheme1[;scheme2]...:filename
```

where:

`?`
: If more than one exchange library is registered for the provided schemes, the Exchange Manager has the user select the exchange library by displaying the Send With dialog.

`scheme1``[;`*`scheme2`*`]...`
: The URL schemes that identify which exchange library should be used. If more than one exchange library is registered for the scheme, the default exchange library is selected unless the URL begins with a question mark.

  As shown, multiple schemes may be provided, separated by semicolons. Multiple schemes are only supported in conjunction with the question mark. For example, the string "?_send;_beam" has the Exchange Manager display a Send With dialog that lists all exchange libraries that support either the _send scheme or the _beam scheme.

`filename`
: The name of the file to send. Typically, this file also has an extension that is used, if necessary, to determine which application should receive the data. See "Identifying the Type of Data" for more information about the file extension.

Palm OS defines some URL prefixes that any application can use to connect with the installed exchange libraries. A URL prefix is

everything up to and including the colon character. Table 1.3 describes the prefixes.

Table 1.3 Exchange Library URL Prefixes

Exchange Library	URL Prefix
IR Library	`exgBeamPrefix`
Local Exchange Library	`exgLocalPrefix`
SMS Library	`kSmsScheme`
Any library that supports the _send scheme (user's choice)	`exgSendPrefix`
Any library that supports the _send or _beam scheme (user's choice)	`exgSendBeamPrefix`

The section "Implementing the Send Command" provides more information on using `exgSendPrefix` or `exgSendBeamPrefix`.

Identifying the Type of Data

When your application sends data, the exchange socket structure (`ExgSocketType`) identifies the type of data being sent. It can do so with one of the following values:

- A MIME type in the `type` field. This field is only used on Palm OS 4.0 and higher.
- A file extension for the file in the `name` field. That is, you might supply `MyDB.pdb` as the value of the `name` field. The part after the last period (.) is the extension.

In most cases, the data type determines which application receives the data on the remote side. (If the `target` field is specified, it determines which application receives the data instead of the data type as described below.) The Exchange Manager maintains a registry of applications and the types of data each application can receive. When the Exchange Manager receives an object, it checks the exchange socket for the data type. It checks the `type` field first, and if it is not defined or if no application is registered to receive that MIME type, it checks the `name` field for a file extension. This is discussed in more detail in the "Registering for Data" section.

Note that you may also directly specify which application should receive the data. To do so, place the creator ID in the `target` field. You do not have to specify a MIME type or file extension in this instance. When the `target` field is nonzero, the Exchange Manager checks for the existence of that application on the receiving device. If it exists, that application receives the data regardless of whether it is registered. If the target application does not exist, the Exchange Manager searches the registry as usual. Use the `target` field only if you know that you are communicating with a Palm Powered handheld and want to explicitly specify which application should receive the data.

On Palm OS 4.0 and higher, an application can register for another application's creator ID and receive all objects targeted to that creator ID. See "Setting the Default Application" for more details.

Registering for Data

In most cases, applications that want to receive data from the Exchange Manager must register for the MIME type and/or file extension that they want to receive. The function that you use to do so differs depending on which operating system versions you want to support.

On Palm OS 3.X, you call `ExgRegisterData` and pass it three parameters: your application's creator ID, a constant that identifies the type of data you want to register to receive (`exgRegExtensionID` for file extensions or `exgRegTypeID` for MIME types), and a string that lists the MIME types or file extensions. For example, on Palm OS 3.X the Beamer sample application distributed with the Palm OS SDK makes this call:

```
ExgRegisterData(beamerCreator,
  exgRegExtensionID, BitmapExt);
```

On Palm OS 4.0 and higher, `ExgRegisterData` is deprecated and replaced with `ExgRegisterDatatype`. `ExgRegisterDatatype` supports more types of data and takes more parameters. You still pass the creator ID, the type of data you want to register for, and the string that describes the specifics of what you are registering for. Palm OS 4.0 and higher supports registering for creator IDs

(`exgRegCreatorID`) or URL schemes (`exgRegSchemeID`) in addition to MIME types and file extensions; however, registering for these new data types is not as common. See "Setting the Default Application" for a case where you would register for a creator ID, and see "Requesting a URL" for a case where you would register for a URL.

In addition, you must pass two more parameters to `ExgRegisterDatatype`: a string containing descriptions of the data you are registering to receive and a flag indicating whether you want to receive the data directly if it is sent as part of another object. The descriptions that you pass in are displayed to preview the data in the exchange dialog under certain circumstances. The flag parameter is described in the "Registering to Receive Unwrapped Data" section.

For example, on Palm OS 4.0 the Beamer sample application distributed with the Palm OS SDK makes this call:

```
ExgRegisterDatatype(beamerCreator,
  exgRegExtensionID, BitmapExt, "bitmap", 0);
```

General Registration Guidelines

Follow these guidelines when registering for data:

- Register as early as possible.

  To ensure that your application can receive data at any time after it is installed, call `ExgRegisterData` or `ExgRegisterDatatype` in response to the `sysAppLaunchCmdSyncNotify` launch code. This launch code is sent to your application upon its first installation and any time the HotSync® operation modifies the application's database.

- It's best to use a standardized data format rather than a proprietary one if you have a choice.
- On Palm OS 4.0 and higher, multiple applications can register to receive the same data type. The section "Setting the Default Application" describes this further.

- When registering for file extensions, do not include the period (.) as part of the extension. Register for "TXT", for example, not ".TXT".
- Do *not* make multiple calls if you want to register for more than one MIME type or more than one file extension.

  Instead, make one call for all file extensions and one call for all MIME types. Pass a single string containing file extensions or MIME types separated by a tab (\t) character. For example, the following call registers the application for two file extensions, TXT and DOC:

```
ExgRegisterData(myCreator, exgRegExtensionID,
"TXT\tDOC", "plain text", 0);
```

- The description parameter is optional. If you implement the preview mode as described in "Displaying a Preview" later in this chapter, you do not need to provide a description. It is, however, strongly recommended that you provide one.

Setting the Default Application

Because multiple applications can register for the same data type on Palm OS 4.0 and higher, the Exchange Manager supports the concept of a default application that receives all objects of a particular data type. To set the default application, call the function `ExgSetDefaultApplication`. There is one default application per data type in the registry. Palm OS 3.X does not support having multiple applications registered for the same data types.

Suppose a device running Palm OS 4.0 receives a vCard object, and it has three applications registered to receive vCards. The Exchange Manager checks the registry to see if any of these applications is assigned as the default. If so, the default application receives all vCards (unless the exchange socket structure's `target` field is set). If none of the three applications is the default, the Exchange Manager chooses one, and that application receives all vCards.

PalmSource, Inc. strongly recommends that you allow users to choose which application is the default. To do so, you could display a panel that shows users the applications that can receive the same type of data as your application, show them which is the default, and allow them to select a different default. Use

ExgGetRegisteredApplications to get a list of all applications registered to receive the same data type as yours, and use ExgGetDefaultApplication to retrieve the current default, if any. See Listing 1.2 to see how the iMessenger example application performs this task for the `mailto` URL scheme. The full source code is distributed with the SDK.

Listing 1.1 Initializing a List of Registered Applications

```
void PrvSetMailAppsList(Int32 listSelection)
{
   ControlPtr ctl;
   ListPtr lst;
   UInt32 defaultID;

   ctl = GetObjectPtr(PrefDefaultMailTrigger);
   lst = GetObjectPtr(PrefDefaultMailList);

   // crIDs, appCnt, appNames are all global variables.
   // Get the list of creator IDs if we don't have it already.
   if(!crIDs) {
      ExgGetRegisteredApplications(&crIDs, &appCnt, &appNames, NULL,
         exgRegSchemeID, "mailto");
      if(appCnt) {
         MemHandle tmpH = SysFormPointerArrayToStrings(appNames, appCnt);
         if(tmpH)
            appNamesArray = MemHandleLock(tmpH);
         else
            return;
      }
      else
         return;
   }

   if(appNamesArray)
      LstSetListChoices(lst, appNamesArray, appCnt);
   LstSetHeight(lst, appCnt < 6 ? appCnt : 6);

   if(listSelection == -1)
   {
      UInt16 i;
      ExgGetDefaultApplication(&defaultID, exgRegSchemeID, "mailto");

      for(i=0;i<appCnt;i++) {
         if(crIDs[i] == defaultID)
            LstSetSelection(lst, i);
```

```
      }
   }
   else
      LstSetSelection(lst, listSelection);

   CtlSetLabel(ctl, appNamesArray[LstGetSelection(lst)]);
}
```

To become the default application for a data type that a built-in Palm OS application is registered to receive (see Table 1.2), you must perform some extra steps to ensure that you can receive that type of object when it is beamed from a device running Palm OS 3.X. You must register for the built-in application's creator ID and become the default application for that creator ID.

On Palm OS 3.X, the built-in applications always set their creator IDs in the `target` field when sending data, causing the data to always be sent to that application. On Palm OS 4.0 and higher, the built-in applications still register to receive the same type of data, but they do not set the `target` field when sending. This means that if your application is registered for the same data type and is the default application, it receives the data from Palm OS 4.0 and higher as expected, but if the data is sent from a device running Palm OS 3.X, you still won't receive that data because it is specifically targeted for the built-in application.

To solve this problem, the `ExgRegisterData` function in Palm OS 4.0 and higher supports registering for another application's creator ID. Listing 1.2 shows how an application that receives vCards might set the default application after allowing the user to select the default from a list, assuming the list is initialized with code similar to that in Listing 1.1.

Note that, as with all data types, your application won't receive the data targeted for the other application unless yours is the default application for that creator ID.

Listing 1.2 Setting the default application for vCards

```
UInt32 PilotMain (UInt16 cmd, void *cmdPBP, UInt16 launchFlags)
{
   ...
   // Register for vCard MIME type, extension, and Address Book's creator ID.
```

```
   // At this point, we are not the default application so we do not receive
   // vCards. We still must register upon install so that our application
   // appears in the preferences list when the user chooses the default
   // application for vCards.
   case sysAppLaunchCmdSyncNotify:
      Char addressCreatorStr[5];

      // Create a string from Address Book's creator ID.
      MemMove(addressCreatorStr, sysFileCAddress, 4);
      addressCreatorStr[4] = chrNull;

      ExgRegisterDatatype(crID, exgRegTypeID, "text/x-vCard", "vCard", 0);
      ExgRegisterDatatype(crID, exgRegExtensionID, "vcf", "vCard", 0);
      ExgRegisterDatatype(crID, exgRegCreatorID, addressCreatorStr, NULL, 0);
      ...
}

static void PrefApply (void)
{
   MemHandle h;
   FieldType *fld;
   ControlType *ctl;
   UInt16 listItem;

   // Set the default vCard app
vif(appCnt && crIDs)
   {
      UInt32 crID;
      Char addressCreatorStr[5];

      // Create a string from Address Book's creator ID.
      MemMove(addressCreatorStr, sysFileCAddress, 4);
      addressCreatorStr[4] = chrNull;

      listItem = LstGetSelection(GetObjectPtr(PrefDefaultAppList));
      crID = crIDs[listItem];
      ExgSetDefaultApplication(crID, exgRegTypeID, "text/x-vCard");
      ExgSetDefaultApplication(crID, exgRegExtensionID, "vcf");
      ExgSetDefaultApplication(crID, exgRegCreatorID, addressCreatorStr);
   }
}
```

Registering to Receive Unwrapped Data

On Palm OS 4.0 or higher, in rare circumstances, you can register to receive data that is sent enclosed in another object.

For example, suppose you have a stock quote application that wants to receive vStock objects. If the device is sent an e-mail message that has the vStock object attached, your application may want to register to receive the vStock object directly rather than having the e-mail application receive it. To do so, call `ExgRegisterDatatype` and pass the constant `exgUnwrap` as the last parameter. The flag is named `exgUnwrap` because the exchange library unwraps the received object (the e-mail message in this example) so that it can send the contained objects (the vStock object) directly.

If you want to register to receive an object when it is sent as part of another object, you probably also want to receive it when it is sent by itself. This requires two calls to `ExgRegisterDatatype`: one with the `exgUnwrap` flag set, and one without.

```
ExgRegisterDatatype(myCreator,
  exgRegExtensionID, "TXT\tDOC", "plain text",
  0);
ExgRegisterDatatype(myCreator,
  exgRegExtensionID, "TXT\tDOC", "plain text",
  exgUnwrap);
```

Thus, you might make four calls to `ExgRegisterDatatype`:

- one call to register for the file extensions
- one call to register for file extensions that are sent as part of another object
- one call to register for MIME types
- one call to register for MIME types that are sent as part of another object

As mentioned previously, it's rare for an application to register to receive unwrapped data directly. It's more common for one application (such as an e-mail application) to receive the entire compound object and then unwrap and disperse the enclosed objects using the Local Exchange Library. See "Sending and Receiving Locally" for more information.

Sending Data

This section describes how to send data using the Exchange Manager. It discusses the following topics:

Sending a Single Object

The most common use of the Exchange Manager is to send or receive a single object. To send an object, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See "Initializing the Exchange Socket Structure" for more information.
2. Call `ExgPut` to establish the connection with the exchange library.
3. Call `ExgSend` one or more times to send the data.

   In this function, you specify the number of bytes to send, and `ExgSend` returns the number of bytes that were sent. You may need to call it multiple times if data is remaining to be sent after the first and subsequent calls.
4. Call `ExgDisconnect` to end the connection.

   A zero (0) return value indicates a successful transmission. However, this doesn't necessarily mean that the receiver kept the data. If the target application for an object doesn't exist on the receiving device, the data is discarded; or the user can decide to discard any received objects.

Note that the `ExgSend` function blocks until it returns. However, most libraries provide a user interface dialog that keeps the user informed of transmission progress and allows them to cancel the operation.

The Exchange Manager automatically displays error dialogs as well, if errors occur. You must check for error codes from Exchange Manager routines, but you don't need to display an error dialog if you get one because the Exchange Manager handles this for you.

For example, Listing 1.3 shows how to send the current draw window from one Palm Powered handheld to another Palm Powered handheld. It is modified from the Beamer example application that is included in the Palm OS SDK.

Listing 1.3 Sending data using Exchange Manager

```
Err SendData(void)
{
   ExgSocketType exgSocket;
   UInt32 size = 0;
   UInt32 sizeSent = 0;
   Err err = 0;
   BitmapType *bmpP;

   // copy draw area into the bitmap
   SaveWindow();
   bmpP = PrvGetBitmap(canvasWinH, &size, &err);
   // Is there data in the field?
   if (!err && size) {
      // important to init structure to zeros...
      MemSet(&exgSocket,sizeof(exgSocket),0);
      exgSocket.description = "Beamer picture";
      exgSocket.name = "Beamer.pbm";
      exgSocket.length = size;
      err = ExgPut(&exgSocket);
      if (!err) {
         sizeSent = ExgSend(&exgSocket,bmpP,size,&err);
         ExgDisconnect(&exgSocket,err);
      }
   }
   if (bmpP) MemPtrFree(bmpP);
   return err;
}
```

Sending Multiple Objects

On Palm OS 4.0 and higher, if the exchange library supports it, you can send multiple objects in a single connection. To send multiple objects, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See "Initializing the Exchange Socket Structure" for

more information. You might also supply a value for the `count` field to specify how many objects are to be sent.

2. Call `ExgConnect` to establish the connection with the exchange library.
3. For each object, do the following:
   a. Call `ExgPut` to signal the start of a new object.
   b. Call `ExgSend` multiple times to send the data.

      In this function you specify the number of bytes to send, and `ExgSend` returns the number of bytes that were sent. You may need to call it multiple times if data is remaining to be sent after the first and subsequent calls.
4. Call `ExgDisconnect` to end the connection.

   A zero (0) return value indicates a successful transmission. However, this doesn't necessarily mean that the receiver kept the data. If the target application for an object doesn't exist on the receiving device, the data is discarded; or the user can decide to discard any beamed objects.

The `ExgConnect` call is optional. Some exchange libraries, such as the IR Library, support the sending of multiple objects but do not support `ExgConnect`. If `ExgConnect` returns an error, the first call to `ExgPut` initiates the connection. You should only continue to send objects if the first `ExgPut` call succeeds. See Listing 1.4. Libraries that support the `ExgConnect` call also support sending multiple objects without using `ExgConnect`.

Listing 1.4 Sending multiple objects

```
Boolean isConnected = false;
err = ExgConnect(&exgSocket);      //optional
if (!err)
   isConnected = true;
if (!err || err == exgErrNotSupported) {
   while (/* we have objects to send */) {
      err = ExgPut(&exgSocket);
      if (!isConnected && !err)
         isConnected = true; //auto-connected on first put.
      while (!err && (sizeSent < size))
         sizeSent += ExgSend(&exgSocket,dataP,size,&err);
      if (err)
         break;
```

```
   }
}
if (isConnected)
   ExgDisconnect(&exgSocket, err);
```

Implementing the Send Command

Starting in Palm OS 4.0, the built-in applications support a Send menu command. The purpose of this command is to allow the user to send data using any available transport mechanism.

The Exchange Manager defines a _send URL scheme. The intent is that any exchange library that supports sending is registered for the _send scheme. Currently, only the SMS Library is registered for this scheme on release ROMs. When Bluetooth support becomes available, the Bluetooth Library will be registered for this scheme. The IR Library is *not* registered for the _send scheme.

To implement the Send command in your application, construct a URL that has the prefix `exgSendPrefix`, and send the data in the normal manner. You can also use the `exgSendBeamPrefix` instead so that the user can select from all exchange libraries registered for either sending or beaming (which includes the IR Library). Both of these prefixes begin with a question mark, causing the Exchange Manager to display a dialog if it finds more than one exchange library registered for the specified schemes.

Currently on a Palm OS 4.0 release ROM, only the SMS Exchange Library supports the _send scheme, so using `exgSendPrefix` would not cause the dialog to be displayed. If the user later adds Bluetooth support, the prefix would cause the dialog to be displayed.

NOTE: On debug ROMs, the Local Exchange Library is listed as one of the possible transport mechanisms. This allows you to debug your Send command. The Local Exchange Library is not listed in the Send With dialog on release ROMs.

For an example of how to implement the Send command, see the Memo application example code distributed with the Palm OS SDK.

Receiving Data

To have your application receive data from the Exchange Manager, do the following:

1. Register for the type of data you want to receive. See "Registering for Data" for more information.
2. Handle the launch code `sysAppLaunchCmdExgAskUser` if you want to control the user confirmation dialog that is displayed. See "Controlling the Exchange Dialog" for more information.
3. Handle the launch code `sysAppLaunchCmdExgPreview` if you want to display a preview of the data to be received. See "Displaying a Preview" for more information.
4. Handle the launch code `sysAppLaunchCmdExgReceiveData` to receive the data. See "Receiving the Data" for more information.
5. If you want, handle `sysAppLaunchCmdGoTo` to display the record.

Controlling the Exchange Dialog

When the Exchange Manager receives an object and decides that your application is the target for that object, it sends your application a series of launch codes. The first launch code your application receives, in most cases, is `sysAppLaunchCmdExgAskUser`.

NOTE: In Palm OS 4.0 and higher, the Exchange Manager allows the exchange library to turn off the user confirmation dialog. In this case, your application does not receive the `sysAppLaunchCmdExgAskUser` launch code.

The Exchange Manger sends this launch code because it is about to display the exchange dialog, which asks the user to confirm the receipt of data. The launch code is your opportunity to accept the data without confirmation, reject the data without confirmation, or replace the exchange dialog.

Responding to this launch code is optional. If you don't respond, the Exchange Manager calls ExgDoDialog to display the exchange dialog.

On Palm OS 3.5 and higher, the ExgDoDialog function allows you to specify that the dialog display a category pop-up list. This pop-up list allows the user to receive the data into a certain category in the database, but the pop-up list is not shown by default. If you want the exchange dialog to display the pop-up list, you must respond to sysAppLaunchCmdExgAskUser and call ExgDoDialog yourself. Pass a pointer to an ExgDialogInfoType structure. The ExgDialogInfoType structure is defined as follows:

```
typedef struct {
  UInt16     version;
  DmOpenRef  db;
  UInt16     categoryIndex;
} ExgDialogInfoType;
```

→ *version*
Set this field to 0 to specify version 0 of this structure.

→ *db*
A pointer to an open database that defines the categories the dialog should display.

← *categoryIndex*
The index of the category in which the user wants to file the incoming data.

If db is valid, the function extracts the category information from the specified database and displays it in a pop-up list. Upon return, the categoryIndex field contains the index of the category the user selected, or dmUnfiledCategory if the user did not select a category.

If the call to ExgDoDialog is successful, your application is responsible for retaining the value returned in categoryIndex and using it to file the incoming data as a record in that category. One way to do this is to store the categoryIndex in the socket's appData field (see ExgSocketType) and then extract it from the socket in your response to the launch code

sysAppLaunchCmdExgReceiveData. See Listing 1.5 for an example.

Listing 1.5 Extracting the category from the exchange socket

```
UInt16 categoryID = (ExgSocketType *)cmdPBP->appData;

/* Receive the data, and create a new record using the
   received data. indexNew is the index of this record. */
if (category != dmUnfiledCategory){
   UInt16 attr;
   Err err;
   err = DmRecordInfo(dbP, indexNew, &attr, NULL, NULL);

   // Set the category to the one the user specified, and
   // mark the record dirty.
   if ((attr & dmRecAttrCategoryMask) != category) {
     attr &= ~dmRecAttrCategoryMask;
     attr |= category | dmRecAttrDirty;
     err = DmSetRecordInfo(dbP, indexNew, &attr, NULL);
   }
}
```

Some of the Palm OS built-in applications (Address Book, Memo, and ToDo) use this method of setting the category on data received through beaming. Refer to the example code provided in the Palm OS SDK for these applications for a more complete example of how to use `ExgDoDialog`.

When you explicitly call `ExgDoDialog`, you must set the `result` field of the `sysAppLaunchCmdExgAskUser` launch code's parameter block to either `exgAskOk` (upon success) or `exgAskCancel` (upon failure) to prevent the system from displaying the dialog a second time.

Displaying a Preview

On Palm OS 4.0 and higher, the exchange dialog contains a preview of the data to be received. The preview allows the user to see what the data is. The reason for the preview is that Palm OS 4.0 and higher supports exchange libraries other than the IR Library. When you use the IR Library to beam data to another Palm Powered handheld, the sender and the receiver must be in close contact with

one another. Other transport mechanisms do not require the devices to be within close proximity, so the user might not know that the data is being received or why. In this case, the user might need more information about the object being received, so the Exchange Manager displays information about the object in the exchange dialog. Also, some exchange libraries do not transmit information for the exchange socket's `description` field, so the Exchange Manager must provide another means of supplying the user with information about the data being received.

To display the preview, the Exchange Manager launches the receiving application with the launch code `sysAppLaunchCmdExgPreview`. Your application does not have to respond to this launch code. If it doesn't, the Exchange Manager displays the first item that it locates in the following list:

- The data's description from the exchange socket's `description` field
- The filename in the socket's `name` field
- The receiving application's description as stored in the exchange registry (you pass this description to `ExgRegisterDatatype` when registering)
- The MIME type in the socket's `type` field
- The file extension in the socket's `name` field

If you want to support a preview that is more elaborate than those in the previous list, handle the `sysAppLaunchCmdExgPreview` launch code.

The launch code's parameter block is an `ExgPreviewInfoType` structure. This structure contains the `ExgSocketType` structure, an `op` field that describes what type of preview data the Exchange Manager expects, and fields in which to return the data.

To respond to the launch code, do the following:

1. Check the `op` field in the parameter block to see what type of preview data is expected. In most cases, the preview data is a string, but a graphical display might also be requested.
2. Call `ExgAccept` to establish a connection with the exchange library.

3. Call ExgReceive one or more times to receive the data.

   In this function, you specify the number of bytes to receive and it returns the number of bytes that were received. You may need to call it multiple times if data is remaining to be received after the first and subsequent calls.

4. Place the data in the parameter block's `string` field if the `op` field specifies a string preview. If the `op` field specifies a graphical preview, draw the data into the rectangle identified by the parameter block's `bounds` field.

5. Call ExgDisconnect to end the connection.

   A zero (0) return value indicates a successful transmission.

Note that you perform essentially the same steps to preview the data as you do to receive it. The only difference is what you do with the data after you receive it. In response to `sysAppLaunchCmdExgPreview`, you pass the data back to the Exchange Manager and discard it in case the user rejects the data. In response to sysAppLaunchCmdExgReceiveData, you store the data.

For an example of handling the `sysAppLaunchCmdExgPreview` launch code, see the Address Book example application that is distributed with the Palm OS SDK. The `TransferPreview` function handles the launch code.

Receiving the Data

If the Exchange Manager receives `exgAskOk` in response to the exchange dialog or the sysAppLaunchCmdExgAskUser launch code, the next step is to launch the application with sysAppLaunchCmdExgReceiveData. This launch code tells the application to actually receive the data.

To respond to this launch code, do the following:

1. Call ExgAccept to accept the connection.

2. Call ExgReceive one or more times to receive the data.

   In this function you specify the number of bytes to receive, and `ExgReceive` returns the number of bytes that were

received. You may need to call it multiple times if data is remaining to be received after the first and subsequent calls.

Note that in the socket structure, the `length` field may not be accurate, so in your receive loop you should be flexible in handling more or less data than `length` specifies.

3. If you want your application launched again with the `sysAppLaunchCmdGoTo` launch code, place your application's creator ID in the `ExgSocketType`'s `goToCreator` field and supply the information that should be passed to the launch code in the `gotoParams` field. (The `ExgSocketType` structure is the `sysAppLaunchCmdExgReceiveData`'s parameter block.)
4. Call `ExgDisconnect` to end the connection.

   A zero (0) return value indicates a successful transmission.

After your application returns from `sysAppLaunchCmdExgReceiveData`, if the `goToCreator` specifies your application's creator ID and if the exchange library supports it, your application is launched with `sysAppLaunchCmdGoto`. In response to this launch code, your application should launch, open its database, and display the record identified by the `recordNum` field (or `matchCustom` field) in the parameter block. The Exchange Manager always does a full application launch with `sysAppLaunchCmdGoto`, so your application has access to global variables; however, if you also use this launch code to implement the global find facility, you may not have access to global variables in that instance. The example code in Listing 1.6 checks to see if globals are available, and if so, calls `StartApplication` to initialize them.

Listing 1.6 Responding to sysAppLaunchCmdGoto

```
case sysAppLaunchCmdGoto:
  if (launchFlags & sysAppLaunchFlagNewGlobals) {
     err = StartApplication();
     if (err) return err;
     GoTo(cmdPBP, true);
     EventLoop();
     StopApplication();
  } else {
```

```
        GoTo(cmdPBP, false);
}
```

On Palm OS 4.0 and higher, not all exchange libraries support using the `sysAppLaunchCmdGoto` launch code after the receipt of data.

Also note that because Palm OS 4.0 and higher supports multiple object exchange, there is no guarantee that your application is the one that is launched at the end of a receipt of data. If multiple objects are being received, it is possible for another application to receive data after yours and to set the `goToCreator` field to its own creator ID. In this case, the last application to set the field is the one that is launched.

Listing 1.7 shows a function that receives a data object and sets the `goToCreator` and `goToParams`. This code is taken from the Beamer example application that is distributed with the Palm OS SDK.

Listing 1.7 Receiving a data object

```
static Err ReceiveData(ExgSocketPtr exgSocketP)
{
   Err err;
   MemHandle dataH;
   UInt16 size;
   UInt8 *dataP;
   Int16 len;
   UInt16 dataLen = 0;

   if (exgSocketP->length)
      size = exgSocketP->length;
   else
      size = ChunkSize;
   dataH = MemHandleNew(size);
   if (!dataH) return -1;  //
   // accept will open a progress dialog and wait for your receive commands
   err = ExgAccept(exgSocketP);
   if (!err){
      dataP = MemHandleLock(dataH);
      do {
         len = ExgReceive(exgSocketP,&dataP[dataLen], size-dataLen,&err);
         if (len && !err) {
            dataLen+=len;
            // resize block when we reach the limit of this one...
```

```
            if (dataLen >= size) {
               MemHandleUnlock(dataH);
               err = MemHandleResize(dataH,size+ChunkSize);
               dataP = MemHandleLock(dataH);
               if (!err) size += ChunkSize;
            }
         }
      }
      while (len && !err);

      MemHandleUnlock(dataH);

      ExgDisconnect(exgSocketP,err); // closes transfer dialog

      if (!err) {
         exgSocketP->goToCreator = beamerCreator;
         exgSocketP->goToParams.matchCustom = (UInt32)dataH;
      }
   }
   // release memory if an error occured
   if (err) MemHandleFree(dataH);
   return err;
}
```

Sending and Receiving Databases

It's common to want to send and receive an entire database using the Exchange Manager. For example, you might want to allow your application's users to share their versions of the PDB file associated with your application by beaming that file to each other.

Sending and receiving a database involves the extra steps of flattening the database into a byte stream when sending and un-flattening it upon return.

Sending a Database

To send a database, do the following:

1. Create and initialize an `ExgSocketType` data structure with information about which library to use and the data to be sent. See "Initializing the Exchange Socket Structure" for more information.

2. Call ExgPut to establish the connection with the exchange library.
3. Call ExgDBWrite and pass it a pointer to a callback function in your application that it can use to send the database. You make the call to ExgSend in that function.
4. Call ExgDisconnect to end the connection.

The ExgDBWrite function takes as parameters the local ID and card number of the database to be sent and a pointer to a callback function. You may also pass in the name of the database as it should appear in a file list and any application-specific data you want passed to the callback function. In this case, you would pass the pointer to the exchange socket structure as the application-specific data. If you need any other data, create a structure that contains the exchange socket and pass a pointer to that structure instead.

The write callback function is called as many times as is necessary to send the data. It takes three arguments: a pointer to the data to be sent, the size of the data, and the application-specific data passed as the second argument to ExgDBWrite.

Listing 1.8 shows an example of how to send a database. The SendMe function looks up the database creator ID and card number and passes it to the SendDatabase function. The SendDatabase function creates and initializes the exchange socket structure and then passes all that information along to the ExgDBWrite function. The ExgDBWrite function locates the database in the storage heap, translates it into a stream of bytes and passes that byte stream as the first argument to the write callback function WriteDBData. WriteDBData forwards the exchange socket and the data stream to the ExgSend call, sets its size parameter to the number of bytes sent (the return value of ExgSend), and returns any error returned by ExgSend.

Listing 1.8 Sending a database

```
// Callback for ExgDBWrite to send data with Exchange Manager
Err WriteDBData(const void* dataP, ULong* sizeP, void* userDataP)
{
   Err err;

   *sizeP = ExgSend((ExgSocketPtr)userDataP, (void*)dataP, *sizeP, &err);
```

```
   return err;
}

Err SendDatabase (Word cardNo, LocalID dbID, CharPtr nameP,
CharPtr descriptionP)
{
   ExgSocketType exgSocket;
   Err err;

   // Create exgSocket structure
   MemSet(&exgSocket, sizeof(exgSocket), 0);
   exgSocket.description = descriptionP;
   exgSocket.name = nameP;

   // Start an exchange put operation
   err = ExgPut(&exgSocket);
   if (!err) {
      err = ExgDBWrite(WriteDBData, &exgSocket, NULL, dbID, cardNo);
      err = ExgDisconnect(&exgSocket, err);
   }
   return err;
}

// Sends this application
Err SendMe(void)
{
   Err err;

   // Find our app using its internal name
   LocalID dbID = DmFindDatabase(0, "Beamer");

   if (dbID)
      err = SendDatabase(0, dbID, "Beamer.prc", "Beamer application");
   else
      err = DmGetLastErr();
   return err;
}
```

Note that there is nothing about `ExgDBWrite` that is tied to the Exchange Manager, so it may be used to send a database using other transport mechanisms as well. For example, if you wanted to transfer a database from your Palm Powered handheld to your desktop PC using the serial port, you could use `ExgDBWrite` to do so.

Receiving a Database

The Launcher application receives databases with the `.prc` or `.pdb` file extension. If you want your application to be launched when the database is received, you can use a different extension and handle receiving the database within your application. For example, a book reader application might want to be launched when the user is beamed a book. In this case, the book reader application might use an extension such as `.bk` for the book databases.

You receive a database by responding to the same launch codes that you do for receiving any other data object (see "Receiving Data"); however, your response to the `sysAppLaunchCmdExgReceiveData` launch code is a little different:

1. Call `ExgAccept` to accept the connection.
2. Call `ExgDBRead` and pass it a pointer to a callback function in your application that it can use to read the database. You make the call to `ExgReceive` in that function.
3. Call `ExgDisconnect` to end the connection.

The `ExgDBRead` function takes as parameters two pointers to callback functions. The first callback function is a function that is called multiple times to read the data. The second function is used if the database to be received already exists on the device.

Requesting Data

On Palm OS 4.0 and higher, some exchange libraries allow you to request data from a remote device through a call to `ExgGet`. You can use `ExgGet` to implement two-way communications between two Palm™ devices.

This section describes how to use the Exchange Manager to request data. It covers:

- Sending a Get Request for a Single Object
- Responding to a Get Request
- Two-Way Communications
- Requesting a URL

Sending a Get Request for a Single Object

To request data from a remote device, do the following:

1. Create and initialize an exchange socket structure (`ExgSocketType`) as described in "Initializing the Exchange Socket Structure"section. The data structure should identify the exchange library and the type of data that your application wants to receive.
2. Call `ExgGet` to establish the connection and request the data.

   In response, the exchange library establishes a connection with the remote device, and upon return has data that your application should receive. If the remote device is a Palm Powered handheld, the exchange library obtains this data from an application on the remote side using the process described in the "Responding to a Get Request" section.
3. Call `ExgReceive` one or more times to receive the data.
4. Call `ExgDisconnect` to end the connection.

Responding to a Get Request

When the Exchange Manager on the remote device receives the get request, it launches the appropriate application with the launch code `sysAppLaunchCmdExgGetData`.

Your response to the `sysAppLaunchCmdExgGetData` launch code should be to send the requested data:

1. Call `ExgSend` one or more times.
2. Call `ExgDisconnect` when finished.

See the "Sending a Single Object" section for more information.

Two-Way Communications

You can use `ExgGet` and `ExgPut` in combination with the `ExgConnect` call to have your application perform two-way communication. For example, you may want to implement two-way communication in a multiuser game.

In such a situation, one device acts as a client and the other acts as a server. The client calls `ExgConnect`, which tells the exchange library that a connection is established to perform multiple

operations, such as the sending of multiple objects. The client then calls `ExgGet` or `ExgPut` repeatedly and calls `ExgDisconnect` when finished. On the server device, the appropriate application is launched for each of these requests. The server also calls `ExgDisconnect` when it is done sending or receiving each object. The swapping of client and server roles is not supported.

Remember that not all exchange libraries support `ExgConnect` and `ExgGet`. If either one of these returns an error, your application should assume that this feature is not available.

Requesting a URL

In addition to requesting data with an `ExgGet` call, you can request a URL with a `ExgRequest` call on Palm OS 4.0 and higher. The idea behind the `ExgRequest` call is to follow the model of pull technology. You could, for example, implement a web browser if you had an exchange library that supported the HTTP protocol. You could then send an `ExgRequest` call with an exchange socket containing a URL such as `http://www.palmos.com` and receive the web page in response.

The fundamental differences between `ExgRequest` and `ExgGet` are:

- `ExgRequest` does not automatically send the data back to the application that requested it. With `ExgRequest`, when the exchange library receives the requested data, it has the Exchange Manager send it to the default application for that data type.
- Applications can register for URLs sent using `ExgRequest`. `ExgRequest` first looks for an exchange library that handles the URL scheme. If it cannot find one, it looks for an application instead. If it finds an application, it launches it with the `sysAppLaunchCmdGoToURL` launch code.

  For example, the iMessenger application distributed with the Palm OS SDK registers for the `mailto` URL scheme. If another application wants to implement an e-mail command, it could do so by calling `ExgRequest` and passing an exchange socket with a URL that begins with `mailto`. In response to this command, the Exchange Manager launches

the iMessenger application, allowing the user to compose the email.

Sending and Receiving Locally

Most of this chapter has described how to use the Exchange Manager to send data to a remote device and receive data from a remote device.

You may also use the Exchange Manager to exchange data with other applications on the local device. To do so, use the Local Exchange Library. You might want to do so in the following circumstances:

- You might have an application that creates some sort of event in the Datebook application. Your users might have an application that they use in place of the built-in Datebook. To ensure that the appointment is sent to the user's chosen application, you can send that data as a vCalendar object using the Local Exchange Manager. This way, whichever application is the default in the Exchange Manager registry is the one that receives your vCalendar.
- You could use the preview feature of the Exchange Manager to have another application display data for you. As described in the "Displaying a Preview" section, an application can be launched with the `sysAppLaunchCmdExgPreview` launch code to display a preview of the data it is registered to receive. You could use this feature in your own application to display data your application does not recognize. Suppose your application has a GIF and wants to display it in a dialog. It could use the Local Exchange Library to send that GIF to a graphics application on the local device, which in response draws the preview into the bounds of a rectangle you provide.
- Your application receives compound data objects, such as e-mail messages that contain attachments intended for other applications. As described in the "Registering to Receive Unwrapped Data" section, exchange libraries can "unwrap" a compound object and deliver the objects it contains directly; however, doing so is the exception the rule.

  It's much more common for the e-mail message to be sent to the e-mail application and have the attachments delivered to

the appropriate applications only when the user requests it. In response to a user request, the e-mail application extracts the attached object and uses the Local Exchange Library to send it to the application that should receive it.

- Your application exchanges data with a remote device, and you want to debug the code that interacts with the Exchange Manager. In this case, using the Local Exchange Library causes your application to send data in loopback mode, where it is also the recipient of the data.

To use the Local Exchange Library, do the following:

1. Use a URL in the `name` field of the `ExgSocketType` structure to identify the Local Exchange Library. Begin the URL with the constant string `exgLocalPrefix`.

   The Exchange Manager only supports URLs on Palm OS 4.0 and higher. On Palm OS 3.X devices, set the `localMode` flag to 1 to interact with the Local Exchange Library instead of the IR Library.

2. If you want to suppress the exchange dialog or if you want to perform a preview operation, create and initialize an `ExgLocalSocketInfoType` structure and assign it to the socket's `socketRef` field.

```
typedef struct {
  Boolean freeOnDisconnect;
  Boolean noAsk;
  ExgPreviewInfoType *previewInfoP;
  ExgLocalOpType op;
  FileHand tempFileH;
} ExgLocalSocketInfoType;
```

where the following are parameters you might want to set:

`freeOnDisconnect`	Whether the structure is freed when the `ExgDisconnect` call is made. The default is `true`. In general, code that allocates a structure should be responsible for freeing that structure. Therefore, if you have allocated `ExgLocalSocketInfoType`, you should set this field to `false` and explicitly free the structure when you are finished with it.
`noAsk`	Set to `true` to disable the display of the exchange dialog. If you want to, for example, create a vCalendar object and send it to the datebook application in response to a user command, you probably want to set `noAsk` to `true` so that the user does not have to confirm the receipt of the data they just requested you to send.
`previewInfoP`	A pointer to an `ExgPreviewInfoType` structure, used to display a preview of the data. If you wanted to simply use another application to help display data, you would create and initialize this structure.

All other fields are set by the Local Exchange Library. If you don't create this structure, the library does it for you; therefore, you only need to create this structure if you want to supply non-default values for the `noAsk` or `previewInfoP` fields.

3. You can suppress the display of the progress dialogs that the exchange libraries typically display by setting the `noStatus` field of the `ExgSocketType` structure to `true`.
4. Send and receive data in the normal manner. See "Sending Data" and "Receiving Data" for details.

Interacting with the Launcher

On Palm OS 4.0 and higher, when you beam an application from the Launcher, other databases can be automatically beamed with it. If

the application has an associated overlay database, the overlay is beamed along with the application. You do not have to perform any extra work to allow this to happen.

Overlay database support begins in Palm OS 3.5; however, if you beam an application from the Palm OS 3.5 Launcher application, it does not beam the overlay.

In addition to beaming overlays, you can set up a record database so that the Launcher beams it along with the application database and the overlay. For example, a dictionary application might have its dictionary data in an associated database. When a user beams the dictionary application to another user, the dictionary data should be beamed along with the application itself. To allow this to happen, you set the bit `dmHdrAttrBundle` in the database's attributes, as shown here:

```
DmDatabaseInfo(cardNo, dbID, NULL, &attributes,
  NULL, NULL, NULL, NULL, NULL, NULL, NULL,
  NULL, NULL);
attributes |= dmHdrAttrBundle;
DmSetDatabaseInfo(cardNo, dbID, NULL,
  &attributes, NULL, NULL, NULL, NULL, NULL,
  NULL, NULL, NULL, NULL);
```

If you beam an application plus databases to a device running Palm OS 4.0 or higher, the user sees a single confirmation message. If you beam the application to a device running Palm OS 3.X, the device receives only the application database and displays an alert saying that it cannot receive the other databases.

Summary of Exchange Manager

Exchange Manager Functions

Sending Data

ExgSend

ExgDBWrite

ExgPut

Receiving Data

Exchange Manager Functions	
ExgReceive	ExgDBRead
ExgAccept	
Registering for Data	
ExgRegisterDatatype	ExgRegisterData
ExgSetDefaultApplication	
Requesting Data	
ExgGet	ExgRequest
Connecting and Disconnecting	
ExgDisconnect	ExgConnect
Displaying the Exchange Dialog	
ExgDoDialog	
Obtaining Registry Information	
ExgGetTargetApplication	ExgGetRegisteredTypes
ExgGetRegisteredApplications	ExgGetDefaultApplication
Querying the Exchange Library	
ExgControl	
For Exchange Library Use Only	
ExgNotifyReceive	ExgNotifyGoto
ExgNotifyPreview	

2

Exchange Libraries

This chapter describes how to implement an exchange library. It covers the following topics:

- About Exchange Libraries
- Exchange Library Components
- Implementing an Exchange Library

Prior to implementing an exchange library, you should have a clear understanding of how the Exchange Manager operates. See Chapter 1, "Object Exchange," on page 1 for an in-depth discussion on the Exchange Manager. Also see Chapter 63, "Exchange Library," on page 1409 of the *Palm OS Programmer's API Reference* for a detailed description of the functions that must be implemented in each exchange library.

About Exchange Libraries

Exchange libraries are Palm OS® shared libraries that act as "plug-ins" to the Exchange Manager. They deal with protocols and communication devices and allow Palm OS applications to import and export data objects without regard to the transport mechanism. For example, one exchange library always available to Palm Powered™ handhelds implements the IrDA protocol, IrOBEX. This allows applications to beam objects by way of infrared from one Palm Powered handheld to another.

The following can take advantage of the Exchange Library API:

- Removable storage cards
- Notification services
- Email attachments
- Web (HTTP/FTP/CTP/WAP) exchange
- HotSync® simplified import and export

Exchange Libraries, Exchange Manager, and Applications

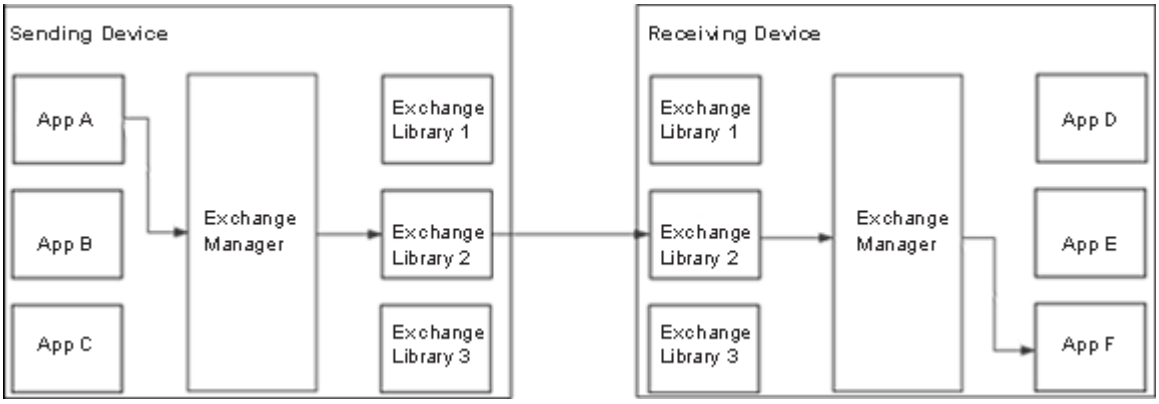
The Exchange Manager is a high-level tool for applications to use. An exchange library is a set of routines that handle the implementation specifics of a particular transport. Typically, exchange library functions are called from the Exchange Manager and are not directly accessed by applications. Applications wanting to send or receive data call the functions provided by the Exchange Manager API, many of which do little more than invoke the corresponding function in the appropriate exchange library.

Exchange libraries also make calls back into the Exchange Manager. For example, an exchange library would call `ExgNotifyReceive` to have the Exchange Manager deliver objects received by the exchange library.

No one component involved with data exchange (Exchange Manager, exchange library, or application) is complete in itself. However, applications and exchange libraries should be written so the user experiences all interaction as a single seamless interface, even though what takes place is really a complex interaction between different pieces of code.

Figure 2.1 illustrates the relationship between applications, the Exchange Manager, and the exchange libraries within two devices that are in communication.

Figure 2.1 Object exchange using Exchange Manager



The following table lists the division of responsibilities between Palm OS applications, the Exchange Manager, and the exchange libraries.

Table 2.1 Division of responsibility for data object exchange

Palm OS Application	Exchange Manager	Exchange Library
Creates, edits, and stores data	Maintains registry of exchange libraries	Sends data to or receives data from other devices
Converts data to and from the interchange formats	Maintains registry of applications that can receive data	Displays a dialog to get addressing information from user
Views or describes data	Passes send and receive requests to appropriate exchange library Displays a dialog asking if user wants to receive data	Displays status and error dialogs, possibly using the Progress Manager

Palm OS Exchange Libraries

The Exchange Manager was introduced in Palm OS 3.0 and was significantly enhanced in Palm OS 4.0. Because of this, the various exchange libraries require different versions of the OS. Table 2.2 lists the minimum OS version required by various exchange libraries.

Table 2.2 Version of Palm OS required by exchange libraries

Exchange Library	Minimum Palm OS Version
IR Library (IrDA)	Palm OS 3.0
Local Exchange Library	Palm OS 4.0

Table 2.2 Version of Palm OS required by exchange libraries

Exchange Library	Minimum Palm OS Version
SMS Library (Short Messaging System)	Palm OS 4.0
Bluetooth Library[1]	Palm OS 4.0

1. Although not present in Palm OS 4.0, Palm plans to provide a Bluetooth Library soon after Palm OS 4.0 ships.

Included with the Palm OS SDK version 4 is the HostTransfer sample exchange library which can be used as a starting point when creating your own exchange libraries.

Exchange Library Components

This section describes the components that make up an exchange library. The topics covered are:

- The Exchange Library API
- Dispatch Table

The Exchange Library API

The Palm OS Exchange Library API specifies the minimum set of functions that all exchange libraries must implement. These functions can be classified into three major categories: functions that must be included in all shared libraries, functions that establish a connection and send and receive data, and miscellaneous support functions.

Standard Shared Library Functions

Any Palm OS shared library must implement open, close, sleep, and wake functions.

- ExgLibOpen
- ExgLibClose
- ExgLibSleep
- ExgLibWake

Functions That Send and Receive Data

These functions do the work of establishing a connection and sending and receiving data.

- ExgLibAccept
- ExgLibConnect
- ExgLibDisconnect
- ExgLibGet
- ExgLibPut
- ExgLibReceive
- ExgLibRequest
- ExgLibSend

Note that each of these corresponds directly to an Exchange Manager function; in most cases the Exchange Manager simply calls the corresponding exchange library function.

Support Functions

This category consists of functions that provide information about your exchange library and that handle events.

- ExgLibControl
- ExgLibHandleEvent

Although each of the functions in these three categories must be present in every exchange library, depending on the specific requirements of the exchange library some of them can simply return `errNone` or `exgErrNotSupported`.

As with any shared library, the order in which the functions appear in the exchange library's dispatch table identifies the functions in the library. This order is specified in `ExgLib.h`. Because it's the function's position in the dispatch table and not its name that is important, the actual function names used in a given exchange library may be different from those specified in `ExgLib.h`. In fact, you'll likely want to use function names that are unique to your shared library, as the Host Transfer library does with such functions as `HostTransferLibPut`, `HostTransferLibSend`, and `HostTransferLibDisconnect`. By using function names specific to your exchange library, you can link your functions into the Mac

Simulator and debug with it. If you use the function names defined in `ExgLib.h` for your functions, you'll get a link error because the Simulator uses those names for stub functions which call your functions.

Beyond the functions listed above, additional library-specific functions must appear in the exchange library's dispatch table after `exgLibTrapLast`.

Dispatch Table

The dispatch table is a map used by the Palm OS to find the functions in the exchange library. At link time, references to the exchange library functions are resolved to a system trap by way of the `SYS_TRAP` macro. At runtime, when an exchange library function is called, a trap occurs and the trap finds the function in its library dispatch table and computes the function's offset into the code resource of the exchange library. A JMP instruction to the function's address is made, causing the function to be executed.

NOTE: The structure of the dispatch table for exchange libraries is the same as that of shared libraries. Exchange libraries must do everything shared libraries must do, plus they must register with the Exchange Manager. This gives applications access to their services by way of the Exchange Manager APIs such as `ExgPut`.

A sample dispatch table source file, `HostTransferDispatch.c`, is provided with the OS SDK. Listing 2.1 provides a sample of the dispatch table contained within this file (some parts are omitted for clarity).

Listing 2.1 HostTransferDispatch.c

```
...
void *PrvHostTransferDispatchTable(void);
...
extern Err PrvInstallHostTransferDispatcher(UInt16 refNum, SysLibTblEntryType
*entryP);
...
Err __Startup__(UInt16 refNum, SysLibTblEntryType *entryP)
{
   return PrvInstallHostTransferDispatcher(refNum, entryP);
```

```
}
...
asm void *PrvHostTransferDispatchTable(void)
{
   LEA  @Table, A0            // table ptr
   RTS                                 // exit with it

@Table:
   DC.W  @Name
   DC.W  (kOffset)       // Open
   DC.W  (kOffset+(1*4))   // Close
   DC.W  (kOffset+(2*4))   // Sleep
   DC.W  (kOffset+(3*4))   // Wake
   // Start of the exchange libary
   DC.W  (kOffset+(4*4)) // HostTransferLibHandleEvent
...
   DC.W  (kOffset+(12*4))  // HostTransferLibControl
   DC.W  (kOffset+(13*4))  // HostTransferLibRequest

@GotoOpen:
   JMP   HostTransferLibOpen
@GotoClose:
   JMP   HostTransferLibClose
@GotoSleep:
   JMP   HostTransferLibSleep
@GotoWake:
   JMP   HostTransferLibWake

@GotoHandleEvent:
   JMP   HostTransferLibHandleEvent
...
@GotoOption:
   JMP   HostTransferLibControl
@GotoCheck:
   JMP   HostTransferLibRequest

@Name:
   DC.B  HostTransferName
}
...
```

The last entry in the dispatch table is the name of the exchange library. This must match the name of the database containing the exchange library, and on the simulator, it must end with "*-crid*",

where *crid* is the creator ID. For example, the Host Transfer library uses `"HostTransfer Library-HXfr"`.

The code segment must be locked so that the dispatch table itself, and the routine addresses in it, will remain valid. The library's database is automatically protected so that it cannot be deleted.

NOTE: The system's shared library table has a slot for library globals for each loaded library. The start-up routine should at least zero this field, if not actually allocate the globals. Some libraries allocate a small structure with an `openCount` and leave the larger allocation for later, when the library is opened by way of the library's `Open()` entry point. In this case, the small structure has a reference to the larger one.

The code resource of an exchange library must start with a routine that sets up the dispatch table. This routine must be named `__Startup__`. The prototype for this function is:

```
Err __Startup__(UInt16 refNum,
SysLibTblEntryType *entryP)
```

Usually, `__Startup__` consists of a one line call in a `MyLibDispatch.c` file that calls the actual setup routine in a corresponding `MyLib.c` file. For example, in the `HostTransferDispatch.c` sample file provided with the OS SDK the following is used to install the HostTransfer dispatch table:

```
extern Err PrvInstallHostTransferDispatcher(UInt16 refNum,
SysLibTblEntryType *entryP);
...
Err __Startup__(UInt16 refNum, SysLibTblEntryType *entryP)
{
   return PrvInstallHostTransferDispatcher(refNum, entryP);
}
...
```

`__Startup__` is called to set up the dispatch table when a call to `SysLibInstall` or `SysLibLoad` is made. For example:

```
SysLibInstall(PrvInstallHostTransferDispatcher, &refNum);
```

Listing 2.2 shows how the HostTransfer dispatch table installer function is implemented. This function can be found in the OS SDK sample file `HostTransferLib.c`. The dispatch table installer function is responsible for making the system's library table entry (`entryP`) point to the dispatch table. For example:

```
entryP->dispatchTblP =
  (MemPtr *)PrvHostTransferDispatchTable();
```

The dispatch installer routine generally does a bit of initialization as well.

Listing 2.2 Host transfer dispatch table installer function

```
Err PrvInstallHostTransferDispatcher(UInt16 refNum, SysLibTblEntryType *entryP)
{
   Err err;
   HostTransferGlobalsType *gP;
   UInt32 value;
   Char macro[14];

   // Must be 4.0 or greator
   err = FtrGet(sysFtrCreator, sysFtrNumROMVersion, &value);
   if (err || value < kVersion4_0) return -1;

   // Allocate library globals and store pointer to them in the system's
   // library table
   gP = MemPtrNew(sizeof(HostTransferGlobalsType));
   ErrFatalDisplayIf(!gP, "No memory for globals");
   if (gP)
      {
      MemPtrSetOwner(gP, 0);
      MemSet(gP, sizeof(HostTransferGlobalsType), 0);
      gP->refNum = refNum;   // make self reference
      entryP->globalsP = gP;
      }

   // Install pointer to our dispatch table in system's library table
   entryP->dispatchTblP = (MemPtr *)PrvHostTransferDispatchTable();

   // Check if we're running on the simulator or emulator. On a real device,
   // there's no host so we don't register this library with the Exchange
   // Manager. In this case, we should really abort the library installation
   // altogether, but this demonstrates how exchange libs can change their
   // registration status.
```

```
#if EMULATION_LEVEL == EMULATION_NONE
   if (FtrGet('pose', 0, &value) != ftrErrNoSuchFeature)
#endif
      {
      Char description[exgTitleBufferSize + 1];
      UInt16 descriptionSize = sizeof(description);
      Err err;

      // Get the title of the library
      err = HostTransferLibControl(refNum, exgLibCtlGetTitle, &description,
         &descriptionSize);
      if (! err)
         {
         // Register this library with the Exchange Manager
         err = ExgRegisterDatatype(HostTransferCreator, exgRegSchemeID,
           kHostTransferScheme "\t" exgSendScheme, description, 0 /*flags*/);
         }
      }

   // Add a magic macro to initiate ExgRequest
   StrCopy(macro, "\x01" "0117" "0000" "0408");
   // virtualkeycode,vchrIrReceive,refnum,libEvtHookKeyMask
   macro[7] = PrvHexToAscii((refNum>>4) & 0x0f); // put refnum into string as
hex
   macro[8] = PrvHexToAscii(refNum & 0x0f);
   PrvDeleteExistingMacro(".r");
   GrfAddMacro(".r", (UInt8 *)macro, 13);

   return err;
}
```

Implementing an Exchange Library

In order to work with the Palm OS Exchange Manager, an exchange library must implement a required set of functions and must register with the Exchange Manager.

Required Functions

Exchange libraries contain functions to handle implementation specifics of a particular transport plus the functions required by the Exchange Library API. Functions required to handle transport-specific tasks or other tasks that aren't specific to the Exchange Library API are outside the scope of this document. In general,

however, other functions required by the exchange library could include tasks such as polling devices, handling interrupts, or checking for user input.

Depending on the application, the exchange library's requirements may be send only, receive only, or both. At a minimum, when sending objects, `ExgLibPut`, `ExgLibSend`, and `ExgLibDisconnect` are typically required; for receiving objects, `ExgLibAccept`, `ExgLibReceive`, and `ExgLibDisconnect` are needed. See Chapter 63, "Exchange Library," on page 1409 of the *Palm OS Programmer's API Reference* for a detailed description of each exchange library function.

Implementing ExgLibAccept

There are two situations in which an application calls the Exchange Manager's `ExgAccept` function:

- The application wants to initiate a connection to receive data, which it does in response to `sysAppLaunchCmdExgReceiveData`.
- The application wants to initiate a connection to receive a preview of the data, which it does in response to `sysAppLaunchCmdExgAskUser`.

The Exchange Manager in turn calls `ExgLibAccept`.

When previewing data, you must buffer incoming data. Your `ExgLibAccept` function should observe the preview flag and rewind the buffer, preparing for non-destructive read. When it is called again without the preview flag, it should rewind again, this time preparing for destructive read.

`ExgLibAccept` must update any progress dialogs to indicate that data is being accepted, or received, into an application.

Handling Connection Errors

`ExgLibConnect` can be used by exchange libraries as a convenient place to put code that needs to be executed prior to the first `ExgLibPut` call. Many exchange libraries don't support `ExgLibConnect`, however, instead establishing a connection in the initial call to `ExgLibPut`. If your library doesn't need to support

ExgLibConnect, your implementation of this function should simply return errNone.

If your exchange library doesn't support ExgLibConnect and an error occurs during the initial call to ExgLibPut, your implementation of ExgLibPut should clean up after itself; it should not count on ExgLibDisconnect being called. If the initial call to ExgLibPut succeeds, however, cleanup of subsequent errors can be done in ExgLibDisconnect.

If your exchange library does support ExgLibConnect and an error occurs during a call to it, ExgLibConnect should clean up after itself. Cleanup of errors that occur after a successful call to ExgLibConnect, however, can be delegated to ExgLibDisconnect.

Finally, if your exchange library supports ExgLibConnect but the application doesn't call it prior to calling ExgLibPut, the situation is as if your library didn't implement ExgLibConnect: if an error occurs during the initial call to ExgLibPut your implementation of ExgLibPut should clean up after itself, while if the initial call to ExgLibPut succeeds you can clean up after any subsequent errors in ExgLibDisconnect.

Note that you must support ExgLibConnect if your exchange library supports two-way communication as discussed in "Two-Way Communications" on page 30.

Buffering Data

Data can be sent by the exchange library as it receives it by using ExgLibSend calls or by buffering the data and sending it in response to an ExgLibDisconnect call. Buffering has some advantages. For example, the communication stack does not have to share cycles with the sending or receiving application and the communication hardware is on for the shortest possible time, conserving battery power. One drawback is that buffering requires extra storage that could be problematic if the amount of data exchanged is large.

Registering with the Exchange Manager

Exchange libraries, like applications, must register with the Exchange Manager for the object types they are to receive. Exchange libraries typically register for two URL schemes, one that is used to uniquely identify the exchange library, and one for how it is used.

For example, the IR library registers for "_irobex", which identifies the specific protocol, and for "_beam" which makes it accessible from the Beam command. The Host Transfer sample exchange library registers for "_host" and "_send". The latter registration makes it accessible from the Send command. Most exchange libraries will probably want to register for the "_send" scheme. These URL schemes all start with an underscore to avoid conflicting with standard URL schemes like "http" and "mailto". See Table 1.3 in Chapter 1, "Object Exchange," on page 1 for the supported URL schemes.

Summary of Exchange Library

Exchange Library Functions	
Handling the Connection	
ExgLibConnect	ExgLibAccept
ExgLibDisconnect	ExgLibPut
Requesting Data	
ExgLibGet	ExgLibRequest
Transferring Data	
ExgLibReceive	ExgLibSend
Querying the Exchange Library	
ExgLibControl	
Handling Events	
ExgLibHandleEvent	

Exchange Library Functions	
Required Shared Library Functions	
ExgLibOpen	ExgLibSleep
ExgLibClose	ExgLibWake

3

Personal Data Interchange

The Palm OS® provides the PDI library API for exchanging Personal Data Interchange (PDI) information with other devices and media. This chapter contains the following sections that describe how to use the Palm OS PDI library:

- About Personal Data Interchange briefly introduces the PDI standard and provides links to sources of more complete information.
- About the PDI Library describes how the Palm OS PDI library implements PDI reader and writer objects for exchanging information.
- Using the PDI Library describes how to use the functions in the PDI library.
- Using UDA for Different Media describes how you can use the Unified Data Access (UDA) Manager to access data from different media in your PDI reader or writer.
- Using a PDI Reader - An Example provides a detailed walk-through of a code segment that creates a PDI reader and then uses it to parse vCard information.
- Using a PDI Writer - An Example provides a detailed walk-through of a code segment that creates a PDI writer and then uses it to generate vCal information.

For detailed information about the PDI library data types, constants, and functions, see Chapter 88, "Personal Data Interchange Library," in *Palm OS Programmer's API Reference*.

The PDI reader and writer objects make use of the United Data Access (UDA) Manager to manage input and output data streams. "Using UDA for Different Media" on page 73 provides an overview of using the UDA Manager. The reference information for UDA

functions is in Chapter 89, "Unified Data Access Manager," on page 2355 in *Palm OS Programmer's API Reference*.

About Personal Data Interchange

Personal data interchange involves the exchange of information using a communications medium. The Palm OS PDI Library facilitates the exchange of information using standard **vObjects**, including data formatted according to **vCard** and **vCal** standards.

The vObject standards are maintained by a group known as the versit consortium, which consists of individuals from a number of companies and institutions. The best information about the PDI standards can be found at the consortium's web site:
`http://www.imc.org/pdi/`

These standards are finding increased use in a number of computers and hand-held devices that wish to exchange personal data such as business card and calendar information.

The PDI Library provides a `PdiReaderType` object for reading vObjects from an input stream, and a `PdiWriterType` object for writing vObjects to an output stream. The input streams and output streams can be connected to various data sources.

About vObjects

This section provides a brief overview of vObject standards. Two common vObject types are vCards and vCals:

- vCards are used to exchange virtual business card information electronically. Each vCard can include a large variety of personal and business information about an individual, including name, address, and telecommunications numbers.
- vCals are used to exchange virtual calendaring and scheduling information electronically. Each vCal can include:
  - vEvent objects, each of which represents a scheduled amount of time on a calendar
  - vTodo objects, each of which defines an action item or assignment

Overview of vObject Structure

This section provides a brief overview of vObject standards, including the vCard and vCal standards. Each vObject standard provides the same, basic organizational structure:

- Each vObject is a collection of one or more property definitions.
- Each property definition contains a name, a value, and an optional collection of property parameter definitions.
- Each property parameter definition contains a name and a value. Each parameter value qualifies the property definition with additional information.
- A property value can be structured to contain multiple values. The values are typically separated with commas or semicolons.

The vObject standards also allow developers to add custom extensions. All vObject readers that conform to the standard, including the `PdiReaderType` object, can read these extensions, though not all readers will act upon the information contained in them.

Each property has the following syntax:

```
PropertyName [';' Parameters] ':' PropertyValue
```

Note that property and parameter names are case insensitive.

Listing 3.1 shows a typical vCard definition.

Listing 3.1 Example of a vCard definition

```
BEGIN:VCARD
VERSION:2.1
N:Smith, John;M.;Mr.; Esq.
TEL;WORK;VOICE; MSG:+1 (408) 555-1234
TEL;CELL:+1 (408) 555-4321
TEL;WORK;FAX:+1 (408) 555-9876
ADR;WORK;PARCEL;POSTAL;DOM:Suite 101;1 Central  St.;Any
Town;NC;28654
END:VCARD
```

Each line in Listing 3.1 is a property definition, with the exception of the next to last line, which is a continuation of the ADR property definition, and begins with white space. Each property definition is delimited by a CR/LF sequence.

The BEGIN, VERSION, and END lines are examples of simple property definitions.

The N (Name) property has a structured value. The components of the name are separated by semicolons.

Each TEL (Telephone) property has parameters that qualify the kind of telephone number that is being specified.

The ADR (Address) property has parameters and a structured value.

NOTE: The vObject specifications also allow long lines of text to be **folded**. This means that wherever you can have white space in a property definition, you can insert a CR/LF followed by white space, as shown in the next to last line in Listing 3.1 When the vObject reader finds a CR/LF followed by white space, it **unfolds** the text back into one long line.

Grouping vObjects

You can specify multiple vObjects in a single vObject data stream. You can also specify a vObject as the value of a property; for example, you can include a vCard as the value of the ADR property of another vCard.

Grouping Properties

You can specify a name for a group of related properties within a vObject. The name is a single character that you use as a prefix to each property in the group.

One use of this facility is to group a comment that describes a property with the property to keep the two together. For example, the following creates a group named G that includes a vCard home telephone property with a comment property:

```
G.TEL;HOME:+1 (831) 555-1234
G.Note: This is my home office number.
```

Encodings

The default encoding for vObject properties is 7-bit. You can override this encoding for individual property values by using the `ENCODING` parameter. You can specify various encoding values, including `BASE64`, `QUOTED-PRINTABLE`, and `8-BIT`.

Character Sets

The default character set for vObject properties is ASCII. You can override the character set for individual property values by using the `CHARSET` parameter. You can specify any character set that has been registered with the Internet Assigned Numbers Authority (IANA). For example, to specify the Latin/Hebrew encoding, you would use the value `ISO-8859-8`.

Finding More Information

For a complete description of the vObject specifications, visit the versit consortium's web site:
`http://www.imc.org/pdi/`

About the PDI Library

The Palm OS PDI library is a shared library that provides objects and functions for:

- Reading vCard objects from an input data stream. The section Creating a PDI Reader describes how to create and use a PDI reader, and the section Using a PDI Reader - An Example provides an example of reading vCard data from an input stream.
- Writing vCard objects to an output data stream. The section Creating a PDI Writer describes how to create and use a PDI reader, and the section Using a PDI Writer - An Example provides an example of reading vCard data from an input stream.

The PDI library handles reading and writing objects in a number of different formats, and from or to a variety of media. For more information about specifying the media, see "Using UDA for Different Media" on page 73.

PDI Property and Parameter Types

The PDI library provides constants that you can use with the reader and writer objects to specify property information. These include the following types of constants that specify vObject standard entities:

- The **Property Name** constants represent the PDI property names. Each of the property name constants starts with the `kPdiPRN_` prefix. For example, the `kPdiPRN_ADR` constant represents the `ADR` property name. For more information, see the section Property Name Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.
- The **Property Value Field** constants represent the position of property value fields for properties with structured field values. Each of the property value field constants starts with the `kPdiPVF_` prefix. For example, the `kPdiPVF_ADR_COUNTRY` constant represents the `COUNTRY` field of an `ADR` property value. For more information, see the section Property Value Field Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.
- The **Parameter Name** constants represent the names of vObject property parameters. Each of the parameter name constants starts with the `kPdiPAN_` prefix. For example, the `kPdiPAN_Type` constant represents the `TYPE` parameter, and the `kPdiPAN_Encoding` constant represents the `ENCODING` parameter. For more information, see the section Parameter Name Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.
- The **Parameter Value** constants represent the combined name and value of parameters. Each of the parameter value constants starts with the `kPdiPAV_` prefix. For example, `kPdiPAV_ENCODING_BASE64` constant represents the `Base64` encoding. For more information, see the section Parameter Value Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.

For a complete list of all of these constants, see the `PdiConst.h` file.

The PDI Library Properties Dictionary

The PDI library features a dictionary that stores information about the properties that are considered "well-known." A well-known property is one that is defined in one of the vObject standard specifications, including the vCard and vCal standards. Both of these standards can be found online at the PDI developer's web page:
`http://www.imc.org/pdi/pdiproddev.html`

PDI readers and writers use information in the properties dictionary to determine how to read or write a certain property. Specifically, the dictionary stores information about the format of each property value; the reader uses this information to correctly parse the property value, and the writer uses this information to correctly format the written value. This information is important because some property values are structured with multiple fields, while others contain a single value field.

For example, the standard address (`ADR`) property has a structured value with seven required fields, and the fields are separated by semicolons. The dictionary stores this information, and the PDI reader then knows to read seven, semicolon-separated fields when parsing an ADR property.

By default, each PDI reader and writer uses a standard dictionary when parsing input and generating output. You can, however, override this behavior to parse or generate the value for a property in some other way. For more information, see "Reading Property Values" on page 67 and "Writing Property Values" on page 72.

You can also amend or replace the dictionary to add parsing and/or generation of customized PDI properties for your application. For more information, see "Adding Custom Extensions" on page 70.

PDI Readers

The PDI library provides the PDI reader object for reading and parsing vObject input. A PDI reader object is a structure that stores the current state of parsing through a PDI input stream.

The PDI reader parses the input stream one property at a time, starting with the Begin Object property and finishing with the End Object property.

The PdiReaderType structure stores a variety of information about the current state of parsing the input stream, including the following information about the current property:

- the encoding and character set
- the type of the current property, parameter, and property value
- the name of the current property and parameter
- the current property's value string
- a mask of the parsing events encountered for the current property

About Parsing Events

The PDI reader records each parsing event that it encounters while processing a property. For example, when it parses a `BEGIN:VCARD` property, the PDI reader records the `kPdiBeginObjectEventMask`, and when it parses a property name, the PDI reader records the `kPdiPropertyNameEventMask`.

Each event is represented by one of the Reader Event Constants, which are described in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*. The PDI reader records the event by adding (`OR`'ing) the event constant into the `events` field of the `PdiReaderType` structure.

You can determine if a specific event has occurred while parsing the current property by testing that event's constant against the `events` field in the reader structure. For example, the following statement returns `false` if the end of the input stream was reached.

```
return((reader->events & kPdiEOFEventMask)==0);
```

PDI Writers

The PDI library provides the PDI writer object for writing vObject output. A PDI writer object is a structure that stores the current state of and manages the generation of PDI data.

The PDI writer sends data to the output stream one property at a time, starting with the Begin Object property and finishing with the End Object property.

The `PdiWriterType` structure stores information about the current state of writing the output stream, including the following:

- the encoding and character set of the current property
- the mode used to write the current property value, which specifies how the property value is structured
- the number of required fields for the current property value

Format Compatibility

The PDI library can read and write data streams in the following formats:

- vCard 3.0
- vCard 2.1
- vCal 1.0
- iCalendar
- Palm format

You can use the PDI library to convert an input data stream that uses one format into an output data stream in another format. For more information, see "Specifying PDI Versions" on page 73.

Compatibility with Earlier Versions of the Palm OS

The PDI library has been designed to maintain compatibility with earlier versions of the Palm OS, which means that you can use the library functions to receive vObjects from or send vObjects to devices that use those earlier versions.

To take advantage of this compatibility, the PDI library has been built to send or receive data in different formats, one of which is the format supported by earlier versions of the Palm OS that included the `ImcUtils` implementation.

To include support for this compatibility in a PDI Reader, specify the `kPdiOpenParser` constant in your call to the `PdiReaderNew` function.

To include support for this compatibility in a PDI Writer, specify the `kPdiPalmCompatibility` option when calling the `PdiWriterNew` function.

International Considerations

The PDI library handles various character sets, including Katakana. If you specify the `CHARSET` parameter in the input stream, the PDI reader will correctly read the property value.

The PDI library included with version 4.0 of the Palm OS® understands the following character sets:

- `charEncodingAscii`
- `charEncodingISO8859_1`
- `charEncodingShiftJIS`
- `charEncodingISO2022Jp`

If you specify an unknown character set, the current character set becomes unknown, as represented by the `charEncodingUnknown` constant.

Features Not Yet Supported

The PDI library included with version 4.0 of the Palm OS does not handle the following features:

- Multi-part MIME messages are not handled.
- The XML version of vObjects is not supported.
- Applications ignore grouping. The PDI reader parses group identifiers, but ignores them. However, the name of the group most recently parsed is stored in the `groupName` field of the `PdiReaderType` object.

Using the PDI Library

This section describes how to use the functions in the PDI library to read or write PDI content. Figure 3.1 shows the typical sequences of calls that you make to read or write vObjects.

To read vObjects, you need to:

- access the PDI library
- create a PDI reader
- read each property in the input stream:
  - read the property name
  - read any parameters for the property
  - read the property value
- delete the PDI reader
- unload the PDI library

To write vObjects, you need to:

- access the PDI library
- create a PDI writer
- write each property in the input stream:
  - write the property name
  - write any parameters for the property
  - write the property value
- delete the PDI writer
- unload the PDI library

The remainder of this section describes the following operations:

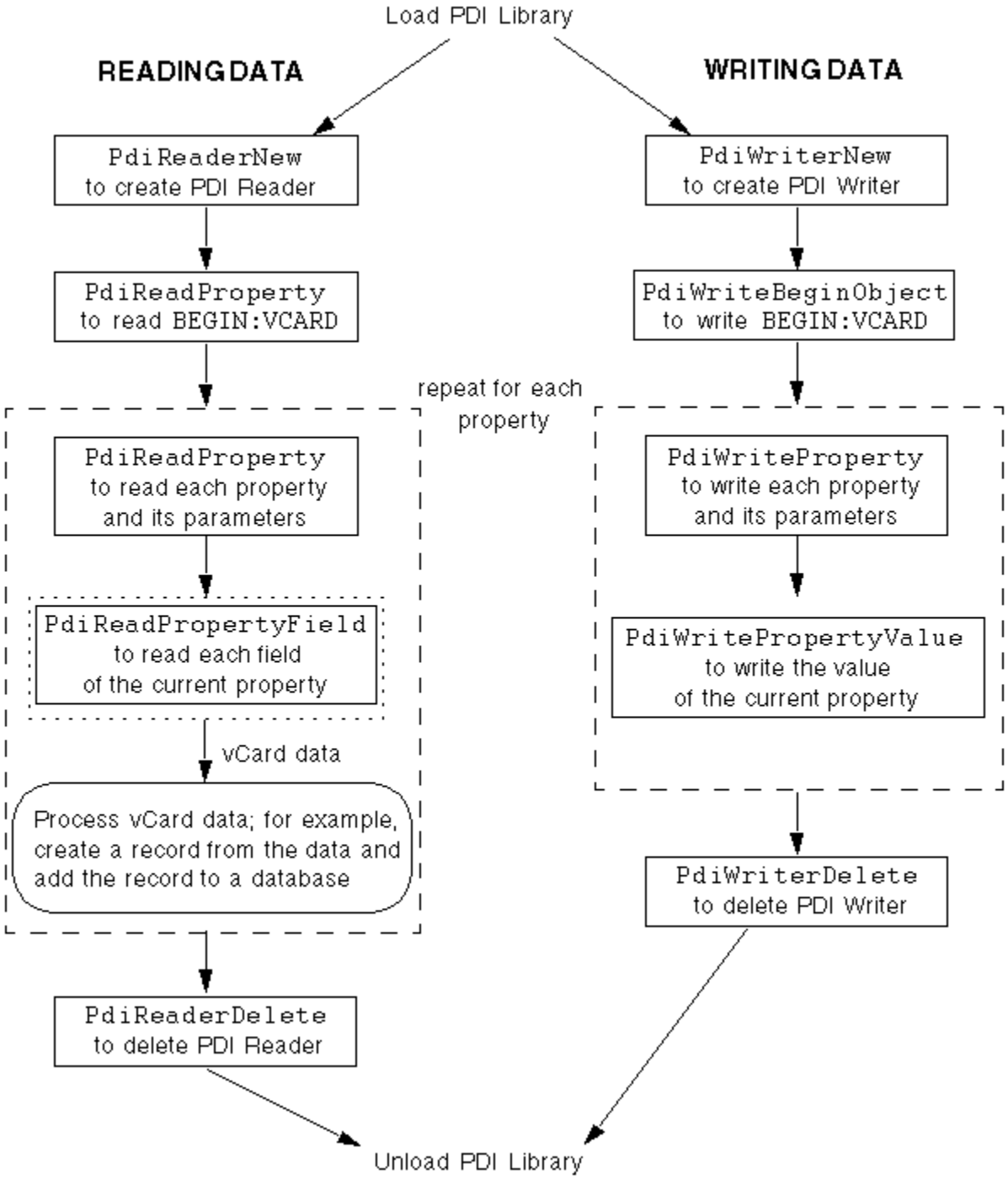
- Accessing the PDI Library
- Unloading the PDI Library
- Creating a PDI Reader
- Reading Properties
- Creating a PDI Writer
- Writing Property Values
- Specifying PDI Versions

- Using UDA for Different Media

The section "Using a PDI Reader - An Example" on page 74 provides a detailed example of creating a PDI Reader and using it to import vCard data into a database.

The section "Using a PDI Writer - An Example" on page 79 provides a detailed example of creating a PDI Writer and using it to export data from a database in vCal format.

Figure 3.1 Using the PDI library



Accessing the PDI Library

Before you can use the PDI library, you must load the library and obtain a reference number for it. Each of the functions in the library requires a reference number argument, which is used with the system code to access a shared library.

The example function `LoadPdiLibrary`, which is shown in Listing 3.2, makes sure that the PDI library is loaded and returns a reference number for it.

Listing 3.2 Loading the PDI library

```
Static Err LoadPdiLibrary(UInt16 *libRefNum)
{
   Err   error

   error = SysLibFind(kPdiLibName, librefNum);
   if (error != 0)
   {
      error = SysLibLoad(sysResTLibrary,
                              sysFileCPdiLib, libRefNum);
   }
   if (error)
   {
      ErrNonFatalDisplay(kPdiLibName "not found")
      return error;
   }
   error = PdiLibOpen(*libRefNum);
   return error;
}
```

The `LoadPdiLibrary` function first calls the `SysLibFind` function to determine if the library has already been loaded, which might be the case if your code has been called by another application that has already loaded the library. Note that the call to `SysLibFind` uses the `kPdiLibName` constant, which is defined as follows in the `PdiLib.h` file:

```
#define kPdiLibName "Pdi.lib"
```

If the library has not already been loaded, `LoadPdiLibrary` calls the `SysLibLoad` function to load the library and obtain a reference number for it.

After obtaining a reference number for the library, `LoadPdiLibrary` calls the `PdiLibOpen` function to open the loaded library.

Unloading the PDI Library

When you are done with the library, you should unload it. The example function `UnloadPdiLibrary`, which is shown in Listing 3.2, unloads the PDI library.

Listing 3.3 Unloading the PDI library

```
static void UnloadPdiLibrary(UInt16 refNum)
{
   if (PdiLibClose(refNum) == 0)
   {
      SysLibRemove(refNum);
   }
}
```

Note that the library reference number becomes invalid after you call the `SysLibRemove` function.

Creating a PDI Reader

To create a PDI reader, you need to first access the library, and then call the `PdiReaderNew` function, which is declared as follows:

```
PdiReaderType* PdiReaderNew(UInt16 libRefnum,
UDAReader *input, UInt16 optionFlags)
```

The `PdiReaderNew` parameters are:

- The library reference number, as described in "Accessing the PDI Library" on page 64.
- The Unified Data Access (UDA) input stream to use with the reader. The UDA Manager allows you to read input from various sources, including strings and the Exchange Manager. For more information, see "Using UDA for Different Media" on page 73.
- Option flags that control the parsing behavior of the reader, including its default encoding and compatibility settings.

The option flags are described in Reader and Writer Options Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.

Once you have created the reader, you can use it to parse properties from the input stream. The section "Using a PDI Reader - An Example" on page 74 provides an example of creating and using a PDI reader.

Reading Properties

To read PDI property data with a PDI reader, you need to call the data reading functions:

- `PdiReadProperty` reads a property and all of its parameters from the input stream.
- `PdiReadPropertyName` reads just the name of the next property from the input stream. You can call this function if you want to then handle the reading of the property's parameters individually.
- `PdiReadParameter` reads a single parameter and its value from the input stream.
- `PdiReadPropertyField` reads a property value field. A property value can a simple value, or it can be structured to contain multiple fields that are separated by commas or semicolons, as described in "Reading Property Values" on page 67.

The most common way to read input data is to follow these steps:

- Call `PdiReadProperty` to read the vObject Begin property. For example, if you are reading vCards, you can call `PdiReadProperty` until it reads the `kPdiPRN_BEGIN_VCARD` property from the input stream.
- Once you have found the beginning of the object, repeatedly call `PdiReadProperty` to read the next property and its parameters.
- For each property, call the `PdiReadPropertyField` function as required to read the fields of the property.
- Continue reading properties until you read the vObject End property. For vCards, you process properties until

`PdiReadProperty` reads the `kPdiPRN_END_VCARD` property from the input stream.

Examining Property Information

After calling a property-reading function, you can access fields of the `PdiReaderType` object to determine information about the current property. The current property is the one that is currently being parsed, or which has just been parsed.

For example, you can examine the `property` field of the `PdiReaderType` object to determine which type of property has just been read, or you can call the `PdiParameterPairTest` macro to determine if a certain parameter pair was present in the property definition.

Reading Property Values

Some properties have simple values and others have structured values. A structured property value has multiple fields that are separated by commas or semicolons.

For example, the following phone property definition has a simple value:

```
TEL;CELL:+1 (408) 555-4321
```

Note that the phone property contains a semicolon to separate the `CELL` parameter from the property name. Each property's value follows the colon in the definition.

The following name property definition has a structured value that contains four fields separated by semicolons:

```
N:Smith; John;M.;Mr.; Esq.
```

You must pass a parameter to the `PdiReadPropertyField` function to tell it how to process a property value. To specify how the field is formatted, use one of the Property Value Format Constants described in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.

You can specify `kPdiDefaultFields` to allow the PDI reader to determine the property value format. The reader looks up the property name in the dictionary to determine its format.

- Specify `kPdiNoFields` to have the reader parse the entire value in one operation.
- Specify `kPdiCommaFields` or `kPdiSemicolonFields` to have the reader parse a single field from the value.
- Specify `kPdiConvertComma` or `kPdiConvertSemicolon` to have the reader parse all of the fields in a value into a single value.

You can usually specify `kPdiDefaultFields` and allow the PDI Reader to use the information in the dictionary to properly parse the value. However, this might not always meet your needs, especially if your input stream contains custom properties.

Table 3.1 shows the results of using the different format constants to read the same property from the input stream. The example property is a standard address (`ADR`) property that has a structured value with seven, semicolon-delimited fields:

```
ADR:postoffice;extended;street;locale;region;postal_code;country
```

Note that since the `ADR` property is defined in the vCard standard as a structured value with seven, semicolon-delimited field, the PDI library dictionary defines its default format as `kPdiSemicolon`.

Table 3.1 Parsing a structured value with different value format types

Value format type	Description of `PdiReadPropertyField` results
`kPdiNoFields`	One call returns the entire value as a string: "postoffice;extended;street;locale;region;postal_code;country"
`kPdiSemicolon`	Each call returns a single, semicolon-delimited field from the value. For example: • the first call returns "postoffice" • the second call returns "extended" • the third call returns "street"

Table 3.1 Parsing a structured value with different value format types *(continued)*

Value format type	Description of `PdiReadPropertyField` results
`kPdiComma`	Each call returns a single, comma-delimited field from the value. For example, if the input string is "postoffice,extended,street," then: • the first call returns "postoffice" • the second call returns "extended" • the third call returns "street"
`kPdiConvertSemicolon`	One call returns the entire value as a string that has newline characters wherever a semicolon appeared in the input: "postoffice extended street locale region postal_code country"
`kPdiConvertComma`	One call returns the entire value as a string that has newline characters wherever a comma appeared in the input: "postoffice extended street locale region postal_code country"
`kPdiDefaultFields`	Same as `kPdiSemicolon`, because the PDI library dictionary defines the property value format of the `ADR` field as `kPdiSemicolon`.

Reading Value Fields One At a Time

If you are reading the fields in a structured value one at a time, and you don't know the exact number of fields, you can call

PdiReadPropertyField repeatedly until it returns a nonzero result.

For example, the following code segment from the DateTransfer.c program parses each field of the EXDATE property value fields:

Listing 3.4 Reading an undetermined number of value fields

```
while (PdiReadPropertyField(pdiRefNum, reader, &tempP,
                        kPdiResizableBuffer, kPdiSemicolonFields) == 0)
   {
               // Resize handle to hold exception
      err = MemHandleResize(exceptionListH,
sizeof(ExceptionsListType) + sizeof(DateType) * exceptionCount);
      ErrFatalDisplayIf(err != 0, "Memory full");
              // Lock exception handle
      exceptionListP = MemHandleLock(exceptionListH);
              // Calc exception ptr
      exceptionP = (DateType*)((UInt32)exceptionListP
                        + (UInt32)sizeof(UInt16)
                        + (UInt32)(sizeof(DateType) * exceptionCount));
               // Store exception into exception handle
      MatchDateTimeToken(tempP, exceptionP, NULL);
               // Increase exception count
      exceptionCount++;
               // Unlock exceptions list handle
      MemHandleUnlock(exceptionListH);
   }
```

NOTE: If you leave fields in a structured value unread, the next call to PdiReadProperty will skip over them and correctly find the beginning of the next property.

Adding Custom Extensions

The vObject standards are extensible, which means that you can add custom properties to vCards and other vObjects. The PDI library handles these custom properties; however, you must either add an entry to the library's dictionary for each custom property, or specify a constant other than kPdiDefaultFields when parsing the property's value.

Each PDI reader object and each PDI writer object can have a custom dictionary associated with it. You can configure the custom dictionary to amend or to replace the standard, built-in dictionary.

To associate a custom dictionary with a reader or writer, you need to first create the dictionary with the You can then call the `PdiDefineReaderDictionary` function to associate that dictionary with a reader object or call the `PdiDefineWriterDictionary` function to associate the dictionary with a writer object.

NOTE: For more information about the dictionary tool at http://www.palmos.com/dev/tech/kb.

Creating a PDI Writer

To create a PDI writer, you need to first access the library, and then call the `PdiWriterNew` function, which is declared as follows:

```
PdiWriterType* PdiWriterNew(UInt16 libRefnum,
UDAWriter *output, UInt8 optionFlags)
```

The `PdiWriterNew` parameters are:

- The library reference number, as described in "Accessing the PDI Library" on page 64..
- The UDA output stream to use with the writer. For more information, see "Using UDA for Different Media" on page 73.
- Option flags that control the output generation behavior of the writer, including its default encoding and compatibility settings. The option flags are described in Reader and Writer Options Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.

Once you have created the writer, you can use it to generate properties to the output stream. The section "Using a PDI Writer - An Example" on page 79 provides an example of creating and using a PDI writer.

Writing Properties

To write PDI data with a PDI writer, you need to call the data writing functions. The most commonly used functions are:

- `PdiWriteBeginObject`, which writes a vObject Begin tag to the output stream.
- `PdiWriteEndObject`, which writes a vObject End tag to the output stream.
- `PdiWriteProperty`, which writes a property to the output stream.
- `PdiWritePropertyValue`, which writes a property value to the output stream.

The most common way to write output data is to follow these steps:

- Call `PdiWriteBeginObject` to write the vObject Begin property. For example, if you are writing vCards, you call `PdiWriteBeginObject` to write the `kPdiPRN_BEGIN_VCARD` property to the output stream.
- For each property that you want to write, call `PdiWriteProperty` to write the next property and its parameters, and then call the `PdiWritePropertyValue` function to write the property's value.
- Call `PdiWriteEndObject` to write the vObject End property. For example, if you are writing vCards, you call `PdiWriteEndObject` to write the `kPdiPRN_END_VCARD` property to the output stream.

Writing Property Values

In many cases, you can simply call the `PdiWritePropertyValue` function to write a value to the output stream. If a value contains a variable number of fields, you can instead use the `PdiWritePropertyFields` to write the fields from an array. Or you can use the `PdiWritePropertyStr` to write multiple fields separated by commas or semicolons.

Specifying PDI Versions

The PDI library options constants control how the PDI reader and PDI writer operate. These options are described in Reader and Writer Options Constants in Chapter 88, "Personal Data Interchange Library," on page 2315 in *Palm OS Programmer's API Reference*.

Using UDA for Different Media

The PDI reader and writer objects use Unified Data Access (UDA) Manager objects for reading from and writing to a variety of media. The UDA data types, constants, and functions are documented in Chapter 89, "Unified Data Access Manager," on page 2355 in *Palm OS Programmer's API Reference*. This section provides an overview of using UDA objects with the PDI library.

About the UDA Library

The UDA Manager provides an abstract layer for reading, filtering, and writing data to and from different media. The UDA Manager provides three general purpose object types:

- `UDAReaderType` objects (UDA Readers) read data from an input stream.
- `UDAFilterType` objects (UDA Filters) take input from UDA Readers or UDA Filters, perform some encoding or decoding operations, and output the data to a memory buffer.
- `UDAWriterType` objects (UDA Writers) write data to a filter or an output stream.

The UDA Manager provides general purpose functions for creating these object types. In addition, the UDA Manager provides built-in object types for working with memory buffers and the Exchange Manager.

NOTE: The implementation of the UDA Manager in version 4.0 of the Palm OS does not provide built-in filter objects. These objects are planned for future versions.

Interfacing with the Exchange Manager

The UDA Manager provides two functions for interfacing with the Exchange Manager:

- The `UDAExchangeReaderNew` function creates a UDA Reader object that reads data from an Exchange Manager socket.
- The `UDAExchangeWriterNew` function creates a UDA Writer object that writes data to an Exchange Manager socket.

The Exchange Manager, which is described in Chapter 1, "Object Exchange," on page 1, provides a mechanism for reading typed data in a transport-independent manner.

When you use the UDA interface to the Exchange Manager, you add the benefits of a simple, uniform way to read and write data in a transport-independent manner. This allows you to create PDI readers and writers that can work on data that is stored on a variety of media types.

If you wish to parse PDI objects from memory, you can use an object created by the `UDAMemoryReaderNew` function instead of an Exchange Manager reader object.

The PDI Reader example in the next section reads its data from an Exchange Manger socket, using the `UDAExchangeReaderNew` function to create the reader object.

The PDI Writer example in "Using a PDI Writer - An Example" on page 79 writes its data to an Exchange Manager socket, using the `UDAExchangeWriterNew` function to create the writer object.

Using a PDI Reader - An Example

This section provides an example of reading PDI data from an input stream and storing it in a database. This example is from the `AddressTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 3.5 shows the `TransferReceiveData` function from the `AddressTransfer.c` sample program. This function controls the

reading of vCard data into the address database by performing the following operations:

- Calls the `ExgAccept` function to accept a connection from a remote device.
- Calls a local function, `PrvTransferPdiLoadLibrary`, to load an open the PDI library. The `PrvTransferPdiLoadLibrary` function is almost exactly the same as the `LoadPdiLibrary` function shown in Listing 3.2.
- Calls the `UDAExchangeReaderNew` function to create an input data stream for connection with the Exchange Manager.
- Calls the `PdiReaderNew` function to create a new PDI reader object that reads from the input stream.
- Repeatedly calls the local function `TransferImportVCard` to read vCard data and store it into the address database. This function is described in the next section, Importing vCard Data Into a Database.
- Calls the `ExgDisconnect` function to terminate the transfer and close the connection.
- Calls the `PrvTransferPdiLibUnload` function to unload the PDI library.
- Deletes the PDI reader and UDA input stream objects.

Listing 3.5 Reading a PDI input stream

```
extern Err TransferReceiveData(DmOpenRef dbP, ExgSocketPtr exgSocketP)
{
   volatile Err err;
   UInt16 pdiRefNum = sysInvalidRefNum;
   PdiReaderType* reader = NULL;
   UDAReader* stream = NULL;
   Boolean loaded;

   if ((err = ExgAccept(exgSocketP)) != 0)
      return err;
   if ((err = PrvTransferPdiLibLoad(&pdiRefNum, &loaded)))
   {
      pdiRefNum = sysInvalidRefNum;
      goto errorDisconnect;
   }
```

```
   if ((stream = UDAExchangeReaderNew(exgSocketP)) == NULL)
   {
      err = exgMemError;
      goto errorDisconnect;
   }
   if ((reader = PdiReaderNew(pdiRefNum, stream, kPdiOpenParser)) == NULL)
   {
      err = exgMemError;
      goto errorDisconnect;
   }
   reader->appData = exgSocketP;
   ErrTry
   {
      while(TransferImportVCard(dbP, pdiRefNum, reader, false, false)){};
   }
   ErrCatch(inErr)
   {
      err = inErr;
   } ErrEndCatch
   if (err == errNone && exgSocketP->goToParams.uniqueID == 0)
      err = exgErrBadData;
errorDisconnect:
   if (reader)
      PdiReaderDelete(pdiRefNum, &reader);
   if (stream)
      UDADelete(stream);
   if (pdiRefNum != sysInvalidRefNum)
      PrvTransferPdiLibUnload(pdiRefNum, loaded);
   ExgDisconnect(exgSocketP, err); // closes transfer dialog
   err = errNone; // error was reported, so don't return it
   return err;
}
```

Importing vCard Data Into a Database

The `TransferImportVCard` function imports a vCard record from an input stream. Listing 3.6 shows the basic outline of the `TransferImportVCard` function; you can review the entire function by viewing the `AddressTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 3.6 Importing vCard data into a database

```
Boolean TransferImportVCard(DmOpenRef dbP, UInt16 pdiRefNum,
PdiReaderType* reader, Boolean obeyUniqueIDs, Boolean beginAlreadyRead)
{

...    // local declarations and initialization code

   ErrTry
   {
      phoneField = firstPhoneField;
      if (!beginAlreadyRead)
      {
         PdiReadProperty(pdiRefNum, reader);
         beginAlreadyRead = reader->property == kPdiPRN_BEGIN_VCARD;
      }
      if (!beginAlreadyRead)
         ErrThrow(exgErrBadData);
      PdiEnterObject(pdiRefNum, reader);
      PdiDefineResizing(pdiRefNum, reader, 16, tableMaxTextItemSize);
      while (PdiReadProperty(pdiRefNum, reader) == 0
               && (property = reader->property) != kPdiPRN_END_VCARD)
      {
         switch(property)
         {
         case kPdiPRN_N:
         PdiReadPropertyField(pdiRefNum, reader,
                        (Char **) &newRecord.fields[name],
                           kPdiResizableBuffer, kPdiDefaultFields);
         PdiReadPropertyField(pdiRefNum, reader,
                        (Char **) &newRecord.fields[firstName],
                           kPdiResizableBuffer, kPdiDefaultFields);
      break;
         case kPdiPRN_NOTE:
            PdiDefineResizing(pdiRefNum, reader, 16,
                           noteViewMaxLength);
            PdiReadPropertyField(pdiRefNum, reader,
                        Char **) &newRecord.fields[note],
                           kPdiResizableBuffer, kPdiNoFields);
            PdiDefineResizing(pdiRefNum, reader, 16,
                           tableMaxTextItemSize);
             break;

,,,  // other cases here for other properties

      }
      } // end while
                if (newRecord.fields[name] != NULL
```

```
         && newRecord.fields[company] != NULL
         && newRecord.fields[firstName] != NULL
         && StrCompare(newRecord.fields[name],
                              newRecord.fields[company]) == 0)
      {     // if company & name fields are identical, assume company only
         MemPtrFree(newRecord.fields[name]);
         newRecord.fields[name] = NULL;
      }
AddRecord:
         err = AddrDBNewRecord(dbP, (AddrDBRecordType*) &newRecord,
                                  &indexNew);
         if (err)
            ErrThrow(exgMemError);

   ...      // handle category assignment here

   }   //end of ErrTry
   if (error == exgErrBadData)
      return false;
   if (error != errNone)
      ErrThrow(error);
   return ((reader->events & kPdiEOFEventMask) == 0);
}
```

The `TransferImportVCard` function performs the following operations:

- Calls the `PdiReadProperty` function to read the `BEGIN:VCard` property from the input stream.
- Calls the `PdiEnterObject` function to notify the PDI library that it is reading a new object from the input stream.
- Calls the `PdiDefineResizing` function to set the maximum buffer size for reading properties for the address card.
- Repeatedly calls the `PdiReadProperty` function to read properties of the address card. This repeats until `PdiReadProperty` reads the `END:VCard` property, which indicates the end of data for the address card.
- For each address card property, calls `PdiReadPropertyField` as required to read the values associated with the property. For example, when it reads the `kPdiPRN_N` name property, `AddrImportVCard` calls

`PdiReadPropertyField` twice: once to read the last name, and a second time to read the first name.

- Creates a new address record and adds it to the Address Book database.
- Deallocates memory that it has allocated and performs other cleanup operations.

Again, note that Listing 3.6 only shows the outline of this function. You can find the entire function in the `AddressTransfer.c` file.

Using a PDI Writer - An Example

This section provides an example of writing PDI data from a database record to an output stream. This example is from the `ToDoTransfer.c` file, which is located inside of the `Examples/ToDo/Src` folder.

Listing 3.7 shows an example of creating and using a PDI writer. The `ToDoSendRecordTryCatch` function controls the writing of data from the To Do database to vCal objects by performing the following operations:

- Calls a local function, `LoadPdiLibrary`, to load and open the PDI library. The `LoadPdiLibrary` function is shown in Listing 3.2.
- Calls the `PdiWriterNew` function to create a new PDI writer object that writes to the UDA output stream specified by the `media` parameter.
- Calls the `PdiWriteBeginObject` function to write the `BEGIN:VCAL` property to the output stream.
- Calls the `PdiWriteProperty` function to write the `VERSION` property, and then calls the `PdiWritePropertyValue` function to write the version value.
- Calls the `ToDoExportVCal` function to write the To Do record, as described in the next section, Exporting vCal Data From a Database.
- Calls the `PdiWriteEndObject` function to write the `END:VCAL` property to the output stream.
- Deletes the PDI writer object and unloads the PDI library.

Listing 3.7 Writing a PDI Output Stream

```
static Err ToDoSendRecordTryCatch (DmOpenRef dbP,
        Int16 recordNum, ToDoDBRecordPtr recordP, UDAWriter* media)
{
   volatile Err error = 0;
   UInt16 pdiRefNum;
   PdiWriterType* writer;

   if ((error = LoadPdiLibrary(&pdiRefNum)))
       return error;
   writer = PdiWriterNew(pdiRefNum, media, kPdiPalmCompatibility);
   if (writer)
      {
ErrTry
      {
         PdiWriteBeginObject(pdiRefNum, writer,
                                                 kPdiPRN_BEGIN_VCALENDAR);
         PdiWriteProperty(pdiRefNum, writer, kPdiPRN_VERSION);
         PdiWritePropertyValue(pdiRefNum, writer, (Char*)"1.0",
                                      kPdiWriteData);
          ToDoExportVCal(dbP, recordNum, recordP, pdiRefNum,
                              writer, true);
          PdiWriteEndObject(pdiRefNum, writer,
                                                          kPdiPRN_END_VCALENDAR)
;
          }
ErrCatch(inErr)
        {
           error = inErr;
        } ErrEndCatch
         PdiWriterDelete(pdiRefNum, &writer);
       }
   UnloadPdiLibrary(pdiRefNum);
   return error;
}
```

Exporting vCal Data From a Database

The `ToDoExportVCal` function exports a vCal record from the To Do database to an output stream. Listing 3.8 shows the basic outline of the `ToDoExportVCal` function; you can review the entire function by viewing the `ToDoTransfer.c` file, which is located inside of the `Examples/Address/Src` folder.

Listing 3.8 Exporting vCal data from a database

```
extern void ToDoExportVCal(DmOpenRef dbP, Int16 index,
ToDoDBRecordPtr recordP, UInt16 pdiRefNum, PdiWriterType* writer,
Boolean writeUniqueIDs)
{
Char *          note;
      UInt32      uid;
      Char        tempString[tempStringLengthMax];
      UInt16      attr;
...

   PdiWriteBeginObject(pdiRefNum, writer, kPdiPRN_BEGIN_VTODO);
      // Emit the Category
   PdiWriteProperty(pdiRefNum, writer, kPdiPRN_CATEGORIES);
      // ...code to create the property string (tempString)
   PdiWritePropertyValue(pdiRefNum, writer, tempString, kPdiWriteText);

      // Code to emit the record information, including the:
      //   - due date
      //   - completed flag
      //   - priority value
      //   - description text
...


      // Emit the note
   if (*note != '\0')
   {
      PdiWriteProperty(pdiRefNum, writer, kPdiPRN_ATTACH);
      PdiWritePropertyValue(pdiRefNum, writer, note, kPdiWriteText);
   }

      // Emit an unique id
   if (writeUniqueIDs)
   {
      PdiWriteProperty(pdiRefNum, writer, kPdiPRN_UID);
      // Get the record's unique id and append to the string.
      DmRecordInfo(dbP, index, NULL, &uid, NULL);
      StrIToA(tempString, uid);
      PdiWritePropertyValue(pdiRefNum, writer, tempString, kPdiWriteData);
   }

   PdiWriteEndObject(pdiRefNum, writer, kPdiPRN_END_VTODO);
}
```

The `ToDoExportVCal` function performs the following operations:

- Calls the `PdiWriteBeginObject` function to write the `BEGIN:VTODO` property to the output stream.
- Calls the `PdiWriteProperty` function to write the category information for the To Do record.
- Calls the `PdiWriteProperty` function to write other information for the To Do record, including the due date, completed flag, priority value, and description text.
- Calls the `PdiWriteProperty` function to write the note and again to write a unique ID for the note.
- Calls the `PdiWriteEndObject` function to write the `END:VTODO` property to the output stream.

Again, note that Listing 3.8 only shows the outline of this function. You can find the entire function in the `ToDoTransfer.c` file.

Summary of Personal Data Interchange

PDI Library Functions	
Library Open and Close	
PdiLibClose	PdiLibOpen
Object Creation and Deletion	
PdiReaderNew	PdiWriterNew
PdiReaderDelete	PdiWriterDelete
Property Reading	
PdiDefineResizing	PdiReadProperty
PdiEnterObject	PdiReadPropertyField
PdiParameterPairTest	PdiReadPropertyName
PdiReadParameter	
Property Writing	
PdiSetCharset	PdiWriteParameterStr
PdiSetEncoding	PdiWriteProperty
PdiWriteBeginObject	PdiWritePropertyBinaryValue
PdiWriteEndObject	PdiWritePropertyFields
PdiWriteParameter	PdiWritePropertyStr
	PdiWritePropertyValue
Property Dictionary	
PdiDefineReaderDictionary	PdiDefineWriterDictionary

Summary of Unified Data Access Manager

UDA Manager Functions

UDAControl
UDADelete
UDAEndOfReader
UDAFilterJoin
UDAInitiateWrite
UDAMoreData
UDARead
UDAWriterFlush
UDAWriterJoin

Object Creation

UDAExchangeReaderNew
UDAExchangeWriterNew
UDAMemoryReaderNew

4

Beaming (Infrared Communication)

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

- The Exchange Manager provides a high-level interface that handles all of the communication details transparently. See the "Object Exchange" chapter for more information.
- The Serial Manager provides a virtual driver that implements the IrComm protocol. To use IrComm, you specify `sysFileCVirtIrComm` as the port you want to open and use the Serial Manager APIs to send and receive data on that port. See the "Serial Communication" chapter for information on how to use the Serial Manager APIs.
- The IR Library provides a low-level, direct interface to the IR communications capabilities of the Palm OS. It is designed for applications that want more direct access to the IR capabilities than the Exchange Manager provides.

This chapter discusses the IR Library.

IR Library

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

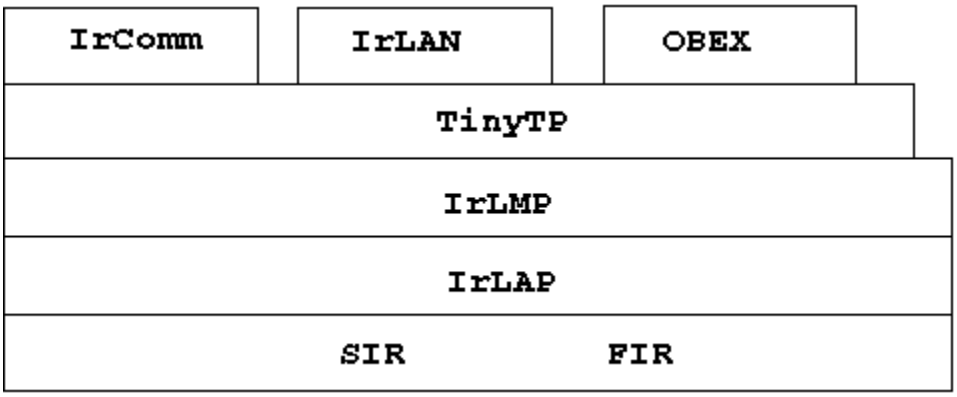
The IR support provided by the Palm OS is compliant with the IrDA specifications. IrDA (Infrared Data Association), is an industry body consisting of representatives from a number of companies involved in IR development. For a good introduction to the IrDA standards, see the IrDA web site at:

http://www.IrDA.org/

IrDA Stack

The IrDA stack comprises a number of protocol layers, of which some are required and some are optional. The complete stack looks something like Figure 4.1.

Figure 4.1 IrDA Protocol Stack



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

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

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

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

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

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

Accessing the IR Library

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

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

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

Summary of Beaming

IR Library Functions	
IrAdvanceCredit	IrIsNoProgress
IrBind	IrIsRemoteBusy
IrClose	IrLocalBusy
IrConnectIrLap	IrMaxRxSize
IrConnectReq	IrMaxTxSize
IrConnectRsp	IrOpen

IR Library Functions	
IrDataReq	IrSetConTypeLMP
IrDisconnectIrLap	IrSetConTypeTTP
IrDiscoverReq	IrSetDeviceInfo
IrIsIrLapConnected	IrTestReq
IrIsMediaBusy	IrUnbind

IR Library IAS Database Functions	
IrIAS_Add	IrIAS_GetUserString
IrIAS_GetInteger	IrIAS_GetUserStringCharSet
IrIAS_GetIntLsap	IrIAS_GetUserStringLen
IrIAS_GetObjectID	IrIAS_Next
IrIAS_GetOctetString	IrIAS_Query
IrIAS_GetOctetStringLen	IrIAS_SetDeviceName
IrIAS_GetType	IrIAS_StartResult

5

Serial Communication

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

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

- Serial Hardware describes the serial port hardware.
- Byte Ordering briefly explains the byte order used for all data.
- Serial Communications Architecture Hierarchy provides an overview of the hierarchy, including an illustration.
- The Serial Manager is responsible for byte-level serial I/O and control of the RS-232, USB, Bluetooth, and IR signals.
- The Connection Manager allows other applications to access, add, and delete connection profiles contained in the Connection preferences panel.
- The Serial Link Protocol provides an efficient mechanism for sending and receiving packets.
- The Serial Link Manager is the Palm OS implementation of the serial link protocol.

NOTE: Although the Palm OS supports Bluetooth connections, Bluetooth requires additional hardware and software that is not available as of this writing.

Serial Hardware

The Palm OS platform device serial port is used for implementing desktop PC connectivity or other external communication. The serial communication is fully interrupt-driven for receiving data. Currently, interrupt-driven transmission of data is not implemented in software, but the hardware does support it. Five external signals are used for this communication:

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

Some devices also have a configurable DTR (data terminal ready) signal. Normally, the DTR signal is always high.

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

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

Palm, Inc. publishes information designed to assist hardware developers in creating devices to interface with the serial communications port on Palm OS platform products. You can obtain this information by joining the Alliance Program and enrolling in the Plugged-In Program. For more information about this program and the serial port hardware, see the Palm™ developer web page at http://www.palm.com/developers/pluggedin/.

Byte Ordering

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

Serial Communications Architecture Hierarchy

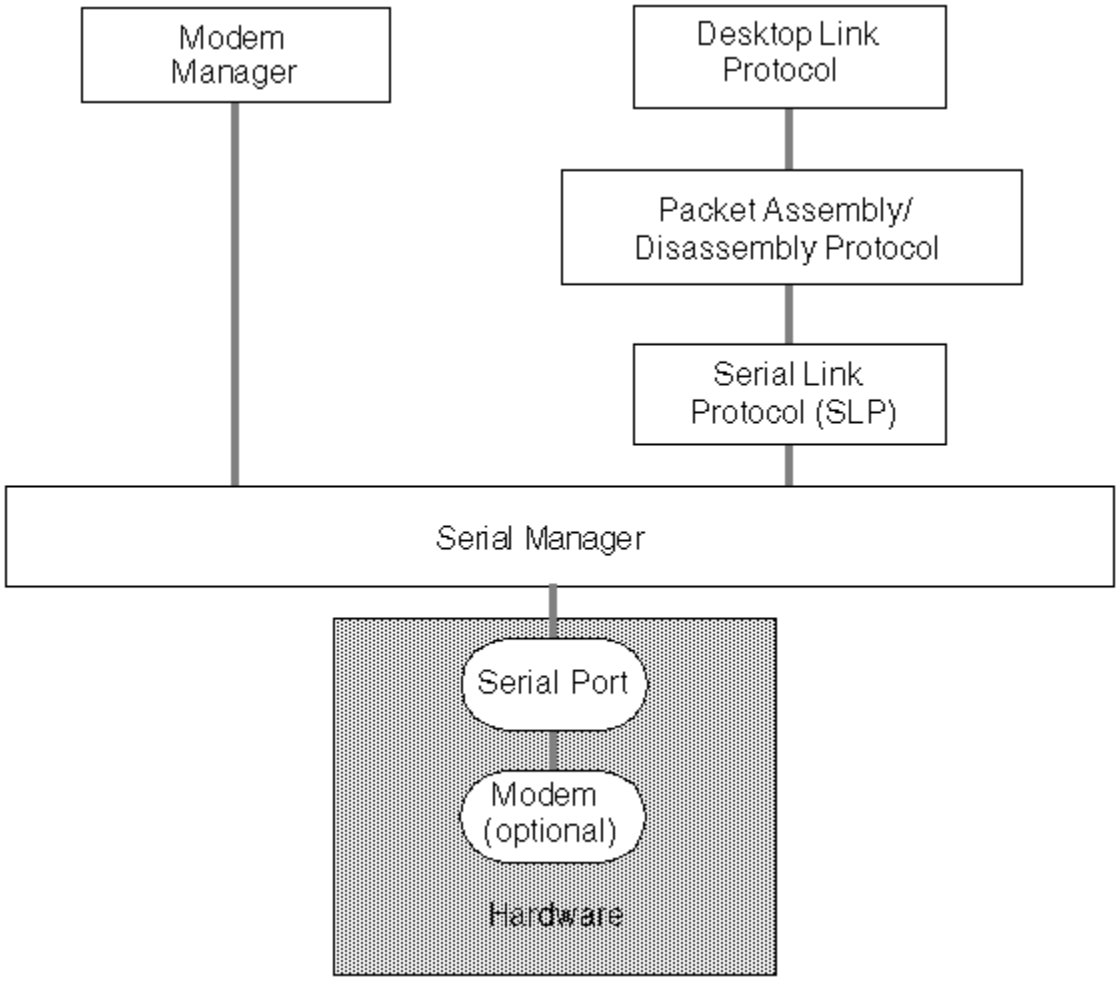
The serial communications software has multiple layers. Higher layers depend on the more primitive functionality provided by lower layers. Applications can use the functionality of all layers. The software consists of the following layers, described in more detail below:

- The Serial Manager, at the lowest layer, deals with the serial port and control of the RS-232 signals, USB signals, or IR signals, providing byte-level serial I/O. See "The Serial Manager" on page 92.
- The Modem Manager provides modem dialing capabilities.
- The Serial Link Protocol (SLP) provides best-effort packet send and receive capabilities with CRC-16. Packet delivery is left to the higher-level protocols; SLP does not guarantee it. See "The Serial Link Protocol" on page 120.
- The Packet Assembly/Disassembly Protocol (PADP) sends and receives buffered data. PADP is an efficient protocol featuring variable-size block transfers with robust error checking and automatic retries. Applications don't need access to this part of the system.
- The Desktop Link Protocol (DLP) provides remote access to Palm OS data storage and other subsystems.

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

Figure 5.1 illustrates the communications layers.

Figure 5.1 Palm OS Serial Communications Architecture



The Serial Manager

The Palm OS Serial Manager is responsible for byte-level serial I/O and control of the RS-232, IR, Bluetooth, or USB signals.

NOTE: Although the Palm OS supports Bluetooth connections, Bluetooth requires additional hardware and software that is not available as of this writing.

To ensure that the Serial Manager does not slow down processing of user events, the Serial Manager receives data asynchronously.

Sending data is performed synchronously in the current implementation.

This section describes the Serial Manager and how to write the virtual serial drivers that it can use. It covers the following topics:

- Which Serial Manager Version To Use
- Steps for Using the Serial Manager
- Opening a Port
- Closing a Port
- Configuring the Port
- Sending Data
- Receiving Data
- Serial Manager Tips and Tricks
- Writing a Virtual Device Driver

NOTE: You must check which Serial Manager is present before making any calls. See the next section for details. When in doubt, the old Serial Manager API is always available.

IMPORTANT: Virtual serial drivers are not supported in Palm OS Cobalt. Third-party developers can only create virtual serial drivers for use with devices running Palm OS 4.x and earlier (and the New Serial Manager Feature Set must be present).

Which Serial Manager Version To Use

There are several versions of the Serial Manager available. The first several releases of Palm OS had a Serial Manager that supported only a a single serial port. The API for this Serial Manager is documented in the chapter "Old Serial Manager" on page 1633 of the *Palm OS Programmer's API Reference*.

If the New Serial Manager Feature Set is present, the Serial Manager has a different set of API (described in the chapter "Serial Manager" on page 1593 of the *Palm OS Programmer's API Reference*) and can support multiple physical serial hardware devices and virtual serial

devices. Physical serial drivers manage communication with the hardware as needed, and virtual drivers manage blocks of data to be sent to some sort of block-based serial code. The detailed operation of drivers is abstracted from the main serial management code.

The newest versions of Palm OS may have an updated version of the new Serial Manager installed. Version 2 provides USB and Bluetooth virtual drivers and provides a few enhancements to the Serial Manager and virtual driver APIs.

When deciding which API to use, note the following:

- If you are writing new application code, best performance is achieved by using the new Serial Manager functions directly, if it is available. The new Serial Manager was introduced in Palm OS 3.3. If it is available on all devices in your target market, consider using new Serial Manager directly.
- The old Serial Manager API is available on all versions of Palm OS; however, it only supports RS-232 communications and low-level IrDA communications.
- The new Serial Manager API supports the IrComm protocol.
- Version 2 of the new Serial Manager supports USB and Bluetooth communication.
- If you write a virtual serial driver, you must use the new Serial Manager API. Note that virtual drivers aren't supported on Palm OS Garnet, however.

Checking the Serial Manager Version

To check whether you can use the new Serial Manager API, check for the existence of the new Serial Manager feature set by calling `FtrGet` as follows:

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

If the new Serial Manager is installed, the `value` parameter is non-zero and the returned error is zero (for no error).

To check for the existence of version 2 of the new Serial Manager, you should check both the Serial Manager version number and the Palm OS version number as follows:

```
err = FtrGet(sysFileCSerialMgr,
  sysFtrNewSerialVersion, &value);
err = FtrGet(sysFtrCreator,
  sysFtrNumROMVersion, &romVersion);
```

If the `value` parameter is 2, the `romVersion` is `0x04003000`, and both calls to `FtrGet` return 0 (for no error), version 2 of the new Serial Manager feature set is present.

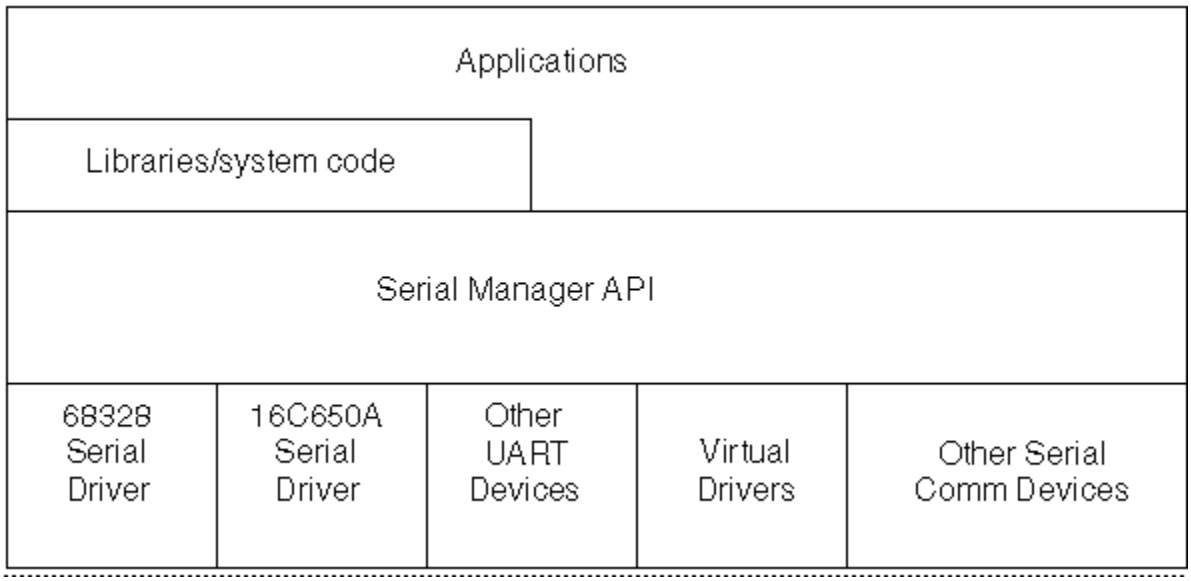
Version 2 of the new Serial Manager ships with roughly Palm OS 4.0 and higher; however, some Handspring devices that run Palm OS 3.5 have a Serial Manager that returns a version number of 2. This Serial Manager has a slightly different feature set than the Serial Manager that ships with Palm OS 4.0. It contains virtual driver operation codes and virtual driver enhancements to support USB, but it does not contain any of the public Serial Manager functions added in version 2. As well, virtual drivers aren't supported on Palm OS Garnet. Therefore, you need to check both the Serial Manager version number and the Palm OS version number before you use the version 2 Serial Manager functions.

About the New Serial Manager

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

The figure below shows the layering of communication software with the Serial Manager and hardware drivers.

Figure 5.2 Serial Communications Architecture with Serial Manager



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

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

Once a port is opened, the Serial Manager allocates a structure for maintaining the current information and settings of the particular port. The task or application that opens the port is returned a port

ID and must supply the port ID to refer to this port when other Serial Manager functions are called.

Upon closing the port, the Serial Manager deallocates the open port structure and unlocks the driver code resource to prevent memory fragmentation.

Note that applications can use the Connection Manager to obtain the proper port name and other serial port parameters that the user has stored in connection profiles for different connection types. For more information, see the section "The Connection Manager" on page 116.

Steps for Using the Serial Manager

Regardless of which version of the API you use, the main steps to perform serial communication are the same. They are:

1. Open a serial port.

   To open a port in the new Serial Manager, you specify which port to open and obtain a port ID that uniquely identifies this connection. You pass that port ID to every other Serial Manager call you make.

   Because the old Serial Manager only has one port, it uses the serial library reference number to uniquely identify the connection. Therefore, with the old Serial Manager, you must first obtain the serial library reference number and then open the port.

   See "Opening a Port" on page 98.

2. If necessary, configure the connection.

   You might need to change the baud rate or increase the size of the receive queue before you use any other Serial Manager calls. See "Configuring the Port" on page 102.

3. Send or receive data.

   See "Sending Data" on page 105 and "Receiving Data" on page 106.

4. Close the port.

   See "Closing a Port" on page 101.

The next several sections describe these steps in more detail. Where the old and new Serial Manager APIs are similar, the task is described in terms of using the new Serial Manager, and the old Serial Manager API is given in parentheses. In these cases, the only difference is in the name of the function and the ID you pass to identify the connection. Where the two APIs differ considerably, both are described.

TIP: See "Serial Manager Tips and Tricks" on page 112 for debugging information and information on how to fix common errors.

Opening a Port

The Serial Manager is installed when the device is booted. Before you can use it, however, you must enable the serial hardware by opening a port.

You open a port for the Serial Manager differently depending on which API you are using: the new Serial Manager or the old Serial Manager.

IMPORTANT: Applications that open a serial port are responsible for closing it. Opening a serial port powers up the UART and drains batteries. To conserve battery power, don't keep the port open longer than necessary.

When you attempt to open a serial port, regardless of which API you use, you must check for errors upon return:

- If `errNone` is returned, the port was opened successfully. The application can then perform its tasks and close the port when finished.
- If `serErrAlreadyOpen` is returned, the port was already open. For example, you might receive this error if the console opened the port during a previous debugging session and never closed it or, on some devices, if there is an open TCP/IP stack.
- If any other error is returned, the port was not opened, and the application must *not* close it.

Opening a Port With the New Serial Manager

To open a port using the new Serial Manager, call the SrmOpen function, specifying the port (see "Specifying the Port" on page 100) and the initial baud rate of the UART. SrmOpen returns a port ID that uniquely identifies this connection. You pass this port ID to all other Serial Manager calls.

Version 2 of the new Serial Manager supports USB and Bluetooth connections as well as RS-232 and IR connections. With the Bluetooth and USB protocols, it is often more important to specify the reason why the application is opening the port. The baud rate is unimportant as that is negotiated in USB and Bluetooth protocols. To open a USB or Bluetooth connection, use SrmExtOpen instead of SrmOpen. This function takes a SrmOpenConfigType structure, which allows you to specify the purpose of the connection instead of the baud rate.

Once the SrmOpen or SrmExtOpen call is made successfully, it indicates that the Serial Manager has successfully allocated internal structures to maintain the port and has successfully loaded the serial driver for this port.

Listing 5.1 Opening the port (new Serial Manager)

```
UInt16 portId;
Boolean serPortOpened = false;

err = SrmOpen(serPortCradlePort /* port */, 57600, /* baud */
  &portId);
if (err) {
   // display error message here.
}
//record our open status in global.
serPortOpened = true;
```

A port may be opened with either a foreground connection (SrmOpen or SrmExtOpen) or background connection (SrmOpenBackground or SrmExtOpenBackground). A foreground connection makes an active connection to the port and controls usage of the port until the connection is closed. A background connection opens the port but relinquishes control to any other task requesting a foreground connection. Background

connections are provided to support tasks (for example, a keyboard driver) that want to use a serial device to receive data only when no other task is using the port.

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

Specifying the Port

Ports must be specified using one of the following methods:

- Logical ports (see "Logical Serial Port Constants" on page 1598 of the *Palm OS Programmer's API Reference*)

  The recommended way to specify the port is to use the logical port name. Logical ports are hardware independent. Palm OS will map them to the correct physical port. It is better to use logical ports instead of physical ports.

- Physical ports (see "Physical Serial Port Constants" on page 1599 of the *Palm OS Programmer's API Reference*)

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

- Virtual ports (see "Virtual Serial Port Constants" on page 1600 of the *Palm OS Programmer's API Reference*)

  Virtual ports are associated with virtual drivers installed on the device. For example, the virtual port constant `sysFileCVirtIrComm` specifies the virtual driver that implements the IrComm protocol.

- Connection Manager (see "The Connection Manager" on page 116)

  If you want to use a particular connection profile as stored in the Connection preferences panel, use the Connection Manager to obtain the port name from the connection profile and then use that name to open the port.

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

Opening a Port with the Old Serial Manager

If you are using the old Serial Manager, there is only one port, so you always pass 0 (or the constant `serPortLocalHotSync`) to identify the port. The serial library reference number identifies the connection. To obtain the reference number, call `SysLibFind`, passing `"Serial Library"` for the library name.

The reference number remains the same within one invocation of the application. You can close and open the library as needed using the number. Between invocations, the reference number may change. Because of that, you should call `SysLibFind` each time you reopen the Serial Manager.

After the call to `SysLibFind`, use `SerOpen` to open the port. Like `SrmOpen`, you pass the baud rate along with the reference number.

Listing 5.2 Opening the port (old Serial Manager)

```
UInt16 refNum = sysInvalidRefNum;
Boolean serPortOpened = false;
Err err;

err = SysLibFind("Serial Library", &refNum);
err = SerOpen(refNum, 0 /* port is always 0*/,
  57600 /* baud */);
if (err == serErrAlreadyOpen) {
  err = SerClose(refNum);
  // display error message here.
}
//record our open status in global.
serPortOpened = true;
```

Closing a Port

Once an application is finished with the serial port, it must close the port using the `SrmClose` function (or `SerClose` function if you are using the old Serial Manager). If `SrmClose` returns no error, it indicates that the Serial Manager has successfully closed the driver and deallocated the data structures used for maintaining the port.

To conserve battery power, it is important not to leave the serial port open longer than necessary. It is generally better to close and reopen

the connection multiple times than it is to leave it open unnecessarily.

Configuring the Port

A newly opened port has the default configuration. The default port configuration is:

- A receive queue of 512 bytes
- A default CTS timeout (currently 5 seconds) set
- 1 stop bit
- 8 data bits
- Hardware handshaking on input
- Flow control enabled
- For RS-232 connections, the baud rate you specified when you opened the port.

You can change this configuration if necessary before sending or receiving data.

Increasing the Receive Queue Buffer Size

The default receive queue size is 512 bytes. If you notice a large number of hardware overruns or software overruns while running your application, consider replacing the default receive queue with a bigger one.

To use a custom receive queue, an application must:

- Allocate a memory chunk for the custom queue. This needs to be an actual memory chunk, not a global variable or an offset from the chunk.
- Call `SrmSetReceiveBuffer` (or `SerSetReceiveBuffer` in the old Serial Manager) with the new buffer and the size of the new buffer as arguments.
- Restore the default queue before closing the port. That way, any bits sent in have a place to go.
- Deallocate the custom queue after restoring the default queue. The system only deallocates the default queue.

The following code fragment illustrates replacing the default queue with a custom queue.

Listing 5.3 Replacing the receive queue

```
#define myCustomSerQueueSize 1024
void *customSerQP;
// Allocate a dynamic memory chunk for our custom receive
// queue.
customSerQP = MemPtrNew(myCustomSerQueueSize);
// Replace the default receive queue.
if (customSerQP) {
  err = SrmSetReceiveBuffer(portId, customSerQP,
     myCustomSerQueueSize);
}

// ... do Serial Manager work

// Now restore default queue and delete custom queue.
// Pass NULL for the buffer and 0 for bufSize to restore the
// default queue.
err = SrmSetReceiveBuffer(portId, NULL, 0);
if(customSerQP) {
   MemPtrFree(customSerQP);
   customSerQP = NULL;
}
```

Changing Other Configuration Settings

To change the other serial port settings, use `SrmControl` (or `SerSetSettings` in the old Serial Manager API).

Listing 5.4 configures the serial port for 19200 baud, 8 data bits, even parity, 1 stop bit, and full hardware handshake (input and output) with a CTS timeout of 0.5 seconds. The CTS timeout specifies the maximum number of system ticks the serial library will wait to send a byte when the CTS input is not asserted. The CTS timeout is ignored if `srmSettingsFlagCTSAutoM` is not set.

Listing 5.4 Changing the configuration (new Serial Manager)

```
Err err;
Int32 paramSize;
Int32 baudRate = 19200;
```

```
UInt32 flags = srmSettingsFlagBitsPerChar8 |
srmSettingsFlagParityOnM | srmSettingsFlagParityEvenM |
srmSettingsFlagStopBits1 | srmSettingsFlagRTSAutoM |
srmSettingsFlagCTSAutoM;
Int32 ctsTimeout = SysTicksPerSecond() / 2;

paramSize = sizeof(baudRate);
err = SrmControl(portId, srmCtlSetBaudRate, &baudRate,
   &paramSize);

paramSize = sizeof(flags);
err = SrmControl(portId, srmCtlSetFlags, &flags, &paramSize);

paramSize = sizeof(ctsTimeout);
err = SrmControl(portId, srmCtlSetCtsTimeout, &ctsTimeout,
   &paramSize);
```

Listing 5.5 shows how to set up the same configuration in the old Serial Manager.

Listing 5.5 Changing the configuration (old Serial Manager)

```
SerSettingsType serSettings;

serSettings.baudRate = 19200;
serSettings.flags = serSettingsFlagBitsPerChar8 |
serSettingsFlagParityOnM | serSettingsFlagParityEvenM |
serSettingsFlagStopBits1 | serSettingsFlagRTSAutoM |
serSettingsFlagCTSAutoM;
serSettings.ctsTimeout = SysTicksPerSecond() / 2;
err = SerSetSettings(refNum, &serSettings);
```

The settings remain in effect until you change them again or close the connection. As you configure the Serial Manager, note the following points:

- Set a CTS timeout if a lack of a CTS signal means a loss of connection. (Use -1 to specify no timeout.)
- If `srmSettingsFlagRTSAutoM` is not set, the RTS output will be permanently asserted. (This flag is set by default.)
- For baud rates above 19200, the use of full hardware handshaking (`srmSettingsFlagRTSAutoM | SrmSettingsFlagCTSAutoM`) is advised.

If you want to find out what the current configuration is, pass one of the `srmCtlGet...` op codes to the `SrmControl` function. For example, to find out the current baud rate, pass `srmCtlGetBaudRate`. To find out the current configuration in the old Serial Manager, use the `SerGetSettings` function.

Sending Data

To send data, use `SrmSend` (or `SerSend` in the old Serial Manager). Sending data is performed synchronously. To send data, the application only needs to have an open connection with a port that has been configured properly and then specify a buffer to send. The larger the buffer to send, the longer the send function operates before returning to the calling application. The send function returns the actual number of bytes that were placed in the UART's FIFO. This makes it possible to determine what was sent and what wasn't in case of an error.

Listing 5.6 illustrates the use of `SrmSend`.

Listing 5.6 Sending data

```
UInt32 toSend, numSent;
Err err;
Char msg[] = "logon\n";
toSend = StrLen(msg);
numSent = SrmSend(portId, msg, toSend, &err);
if (err == serErrTimeOut) {
  //cts timeout detected
}
```

If `SrmSend` returns an error, or if you simply want to ensure that all data has been sent, you can use any of the following functions:

- Use `SrmSendWait` (`SerSendWait` in the old Serial Manager) if you need to wait for all data to leave the device before performing other actions. The `SrmSend` function returns when it has loaded the last byte into the FIFO. The `SrmSendWait` function does not return until the FIFO empties. Like `SrmSend`, the `SrmSendWait` call can timeout if CTS handshaking is on and the CTS timeout value is reached. Note that the old Serial Manager version of this call, `SerSendWait`, takes a timeout parameter, but this

parameter is ignored. The new Serial Manager call simply takes the port ID.

- Use `SrmSendCheck` (or `SerSendCheck`) to determine how many bytes are left in the FIFO. Note that not all serial devices support this feature.

  If the hardware does not provide an exact reading, the function returns an approximate number: 8 means full, 4 means approximately half-full. If the function returns 0, the queue is empty.

- The `SrmSendFlush` (or `SerSendFlush`) function can be used to flush remaining bytes in the FIFO that have not been sent.

Receiving Data

Receiving data is a more involved process because it depends on the receiving application actually listening for data from the port.

To receive data, an application must do the following:

- Ensure that the code does not loop indefinitely waiting for data from the receive queue.

  The most common way to do this is to pass a timeout value to `EvtGetEvent`.

  Virtual devices often run in the same thread as applications. If you don't specify a timeout for the event loop, it can prevent the virtual device and other serial related code from properly handling received data.

  If your code is outside of an event loop, you can use the `EvtEventAvail` function to see if the system has an event it needs to process, and if so, call `SysHandleEvent`.

- To avoid having the system go to sleep while it's waiting to receive data, an application should call `EvtResetAutoOffTimer` periodically (or call `EvtSetAutoOffTimer`). For example, the Serial Link Manager automatically calls `EvtResetAutoOffTimer` each time a new packet is received.

TIP: For many applications, the auto-off feature presents no problem. Use `EvtResetAutoOffTimer` with discretion; applications that use it drain the battery.

- To receive the data, call `SrmReceive` (or `SerReceive`). Pass a buffer, the number of bytes you want to receive, and the inter-byte timeout in system ticks. This call blocks until all the requested data have been received or an error occurs. This function returns the number of bytes actually received. (The error is returned in the last parameter that you pass to the function.)
- If you want to wait until a certain amount of data is available before you receive it, call `SrmReceiveWait` (or `SerReceiveWait`) before you call `SrmReceive`. Specify the number of bytes to wait for, which must be less than the current receive buffer size, and the amount of time to wait in system ticks. If `SrmReceiveWait` returns `errNone`, it means that the receive queue contains the specified number of bytes. If it returns anything other than `errNone`, that number of bytes is not available.

  `SrmReceiveWait` is useful, for example, if you are receiving data packets. You can use `SrmReceiveWait` to wait until an entire packet is available and then read that packet.
- It's common to want to receive data only when the system is idle. In this case, have your event loop respond to the `nilEvent`, which is generated whenever `EvtGetEvent` times out and another event is not available. In response to this event, call `SrmReceiveCheck` (or `SerReceiveCheck`). Unlike `SrmReceiveWait`, `SrmReceiveCheck` does not block awaiting input. Instead, it immediately returns the number of bytes currently in the receive queue. If there is data in the receive queue, call `SrmReceive` to receive it. If the queue has no data, your event handler can simply return and allow the system to perform other tasks.
- Check for and handle error conditions returned by any of the receive function calls as described in "Handling Errors" on page 108.

IMPORTANT: Always check for line errors. Due to unpredictable conditions, there is no guaranteed of success. If a line error occurs, all other Serial Manager calls fail until you clear the error.

For example code that shows how to receive data, see "Receive Data Example" on page 110.

In the new Serial Manager, you can directly access the receive queue using `SrmReceiveWindowOpen`, and `SrmReceiveWindowClose`. These functions allow fast access to the buffer to reduce buffer copying. These functions are not supported on systems where the new Serial Manager feature set is not present.

Handling Errors

If an error occurs on the line, all of the receive functions return the error condition `serErrLineErr`. This error will continue to be returned until you explicitly clear the error condition and continue.

To clear line errors, call `SrmClearErr` (or `SerClearErr`).

If you want more information about the error, call `SrmGetStatus` (or `SerGetStatus`) before you clear the line.

Listing 5.7 checks whether a framing or parity error have returned and clears the line errors.

Listing 5.7 Handling line errors (new Serial Manager)

```
void HandleSerReceiveErr(UInt16 portId, Err err) {
   UInt32 lineStatus;
   UInt16 lineErrs;

   if (err == serErrLineErr) {
      SrmGetStatus(portId, &lineStatus, &lineErrs);
      // test for framing or parity error.
      if (lineErrs & serLineErrorFraming |
serLineErrorParity)
      {
            //framing or parity error occurred. Do something.
      }
       SrmClearErr(portId);
   }
}
```

Listing 5.8 performs the same tasks using the old Serial Manager. Note that the `SerGetStatus` call looks a little different from the `SrmGetStatus` call.

Listing 5.8 Handling line errors (old Serial Manager)

```
void HandleSerReceiveErr(UInt16 refNum, Err err) {
   UInt16 lineErrs;
   Boolean ctsOn, dsrOn;

   if (err == serErrLineErr) {
      lineErrs = SerGetStatus(refNum, &ctsOn, &dsrOn);
      // test for framing or parity error.
      if (lineErrs & serLineErrorFraming |
serLineErrorParity)
      {
            //framing or parity error occurred. Do something.
      }
       SerClearErr(refNum);
   }
}
```

TIP: See “Common Errors” on page 113 for some common causes of line errors and how to fix them.

In some cases, you may want to discard any received data when an error occurs. For example, if your protocol is packet driven and you detect data corruption, you should flush the buffer before you continue. To do so, call `SrmReceiveFlush` (or `SerReceiveFlush`). This function flushes any bytes in the receive queue and then calls `SrmClearErr` for you.

`SrmReceiveFlush` takes a timeout value as a parameter. If you specify a timeout, it waits that period of time for any other data to be received in the queue and flushes it as well. If you pass 0 for the timeout, it simply flushes the data currently in the queue, clears the line errors, and returns. The flush timeout has to be large enough to flush out the noise but not so large that it flushes part of the next packet.

Receive Data Example

Listing 5.9 shows how to receive large blocks of data using the Serial Manager.

Listing 5.9 Receiving Data Using the Serial Manager

```
#include <PalmOS.h> // all the system toolbox headers
#include <SerialMgr.h>
#define k2KBytes 2048
/***********************************************************
*
* FUNCTION: RcvSerialData
*
* DESCRIPTION: An example of how to receive a large chunk of data
* from the Serial Manager. This function is useful if the app
* knows it must receive all this data before moving on. The
* YourDrainEventQueue() function is a chance for the application
* to call EvtGetEvent and handle other application events.
* Receiving data whenever it's available during idle events
* might be done differently than this sample.
*
* PARAMETERS:
* thePort -> valid portID for an open serial port.
* rcvDataP -> pointer to a buffer to put the received data.
* bufSize <-> pointer to the size of rcvBuffer and returns
*   the number of bytes read.
*
***********************************************************/
Err RcvSerialData(UInt16 thePort, UInt8 *rcvDataP, UInt32 *bufSizeP)
```

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

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

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

   // Now loop while getting blocks of data and filling the buffer.
   do {
      // Is the max size larger then the number of bytes left?
      if (bytesLeft < maxRcvBlkSize)
         // Yes, so change the rcv block amount.
    maxRcvBlkSize = bytesLeft;
      // Try to receive as much data as possible,
      // but wait only 1/5 second for it.
      bytesRcvd = SrmReceive(thePort,  rcvDataP, maxRcvBlkSize, waitTime,
         &error);
      // Remember the total number of bytes received.
      totalRcvBytes += bytesRcvd;
      // Figure how many bytes are left to receive.
      bytesLeft -= bytesRcvd;
      rcvDataP += bytesRcvd; // Advance the rcvDataP.
      // If there was a timeout and no data came through...
      if ((error == serErrTimeOut) && (bytesRcvd == 0))
         goto ReceiveError; // ...bail out and report the error.
      // If there's some other error, bail out.
      if ((error) && (error != serErrTimeOut))
         goto ReceiveError;
```

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

   ReceiveError:
      // Clearing the receive buffer can also be done right before
      // the port is to be closed.
      // Set back the default buffer when we're done.
      SrmSetReceiveBuffer(thePort, 0L, 0);

   Exit:
      MemPtrFree(newRcvBuffer); // Free the space.
      *bufSizeP = totalRcvBytes;
      return error;
}
```

Serial Manager Tips and Tricks

The following tips and tricks help you debug your serial application and help avoid errors in the first place.

Debugging Tips

The following are some tips to help you track down errors while debugging.

- Debug first using the Palm OS Emulator. Debug on the device last.

  The Palm OS Emulator supports all Serial Manager functions and lets you test applications that use the Serial Manager. You can use the desktop computer's serial port to connect to outside devices. For more information on how to set up and use the emulator to debug serial communications, see the emulator documentation.

- Track communication errors and the amount of data sent and received.

  In your debug build, maintain individual counts for the amount of data transferred and for each communication error of interest. This includes timeouts and retries for reliable protocols.

- Use an easily recognizable start-of-frame signature. This helps during debugging of packet-based protocols.
- Implement developer back doors for debugging.

  Implement a mechanism to trigger one or more debugging features at runtime without recompiling. For example, you may want to create a back door to disable the receive timeout on one side to prevent it from timing out while you are debugging the other side. Another back door might print some debugging information to the display. For example, your application might look for a pen down event in the upper right corner of the digitizer while the page-up key is being pressed to trigger one of your back doors.
- Use the HotSync® log for debug-time error logging on the device.

  You may use `DlkSetLogEntry` to write your debugging messages to the HotSync log on the device. The HotSync log will accept up to 2KB of text. You may then switch to the HotSync application to view the log.

NOTE: Restrict writing to the HotSync log to debugging. Users will not appreciate having your debugging messages in their HotSync log.

- If you have a protocol analyzer, use it to examine the data that is actually sent and received.

Common Errors

Even if you're careful, errors may crop up. Here are some frequently encountered problems and their solutions.

- Nothing is being received

  Check for a broken or incorrectly wired connection and make sure the expected handshaking signals are received.
- Garbage is received

  Check that baud rate, word length, and/or parity agree.

- Baud rate mismatch

  If the two sides disagree on the baud rate, it may either show up as a framing error, or the number of received characters will be different from the number that was sent.

- Parity error

  Parity errors indicate that the data has been damaged. They can also mean that the sender and receiver have not been configured to use the same parity or word length.

- Word-length mismatch

  Word-length mismatches may show up as a framing error.

- Framing error

  Framing errors indicate a mismatch in the number of bits and are reported when the stop bit is not received when it is expected. This could indicate damaged data, but frequently it signals a disagreement in common baud rate, word length, or parity setting.

- Hardware overrun

  The Serial Manager's receive interrupt service routine cannot keep up with incoming data. Enable full hardware handshaking (see "Configuring the Port" on page 102).

- Software overrun

  The application is not reading incoming data fast enough. Read data more frequently, or replace the default receive queue with a larger one. (see "Configuring the Port" on page 102).

Writing a Virtual Device Driver

If the new Serial Manager feature set is present, and the Palm OS version is less than 5.0, the Serial Manager supports the ability to add virtual device drivers to the system. Virtual serial device drivers transmit and receive data in blocks instead of a byte at a time.

A virtual driver is a code resource (ID=0) that is independently compiled and installed on a Palm device. Virtual driver .prc files are of file type 'vdrv' and their creator type is chosen by the developer (and must be registered with PalmSource, Inc. in the creator ID

database). When the Serial Manager is installed, it searches the Database Manager for code resources of the 'vdrv' type and then calls the driver's entry point function to get information about the features and capabilities of this virtual device. Unlike physical serial device drivers, virtual device drivers send and receive data in blocks instead of transferring one byte at a time. Their purpose is to abstract a level of communication protocol away from serial devices without forcing applications to work through a different API than the Serial Manager that may already be used for normal RS-232 serial communication.

NOTE: Creator types with all lowercase letters are reserved by PalmSource, Inc. For more information about assigning and registering creator types, see "Assigning a Database Type and Creator ID" on page 15 of the *Palm OS Programmer's Companion*, vol. I.

Virtual Driver Functions

There are six functions that each virtual driver must minimally support in order to work with the Serial Manager. These functions are briefly described in this section. For details on the exact operations each function must perform, see the function descriptions in the *Palm OS Programmer's API Reference*.

NOTE: Virtual serial ports are not supported on Palm OS Garnet.

The functions a virtual driver must implement include:

- `DrvEntryPointProcPtr` must be the first function defined in a virtual driver code resource and must be marked as the `__Startup__` function of the code resource. When the code resource is loaded, the Serial Manager jumps to the beginning of the code resource and begins execution at `DrvEntryPoint`. This function is called at system restart, when the Serial Manager is building a database of installed drivers and their capabilities, and when a virtual port is opened.

- The VdrvOpenProcPtr and VdrvOpenProcV4Ptr function is responsible for initializing the virtual device to begin communication.
- The VdrvCloseProcPtr function must handle all activities needed to close the virtual device.
- VdrvControlProcPtr extends the SrmControl function to the level of the virtual device.
- VdrvStatusProcPtr returns a bitfield that describes the current state of the virtual device.
- VdrvWriteProcPtr writes a block of bytes to the virtual device.
- The optional VdrvControlCustomProcPtr function can handle any custom control codes defined specifically for this virtual driver.

Note that there is no virtual read function in the current implementation. Virtual devices must save received data by using the functions provided in the DrvrRcvQType when they are notified that data is available using some callback mechanism.

For an example of how to implement a virtual serial driver, download the CryptoDrvr example from the Palm OS Developer Knowledge Base.

The Connection Manager

The Connection Manager allows applications to access, add, and delete connection profiles contained in the Connection preferences panel. Earlier releases of the Palm OS have a Modem preferences panel. The Connection panel replaces the Modem panel. This change was made as more connection choices (serial cable, IR, modem, network and so on) became available to users.

The Connection Manager was introduced at the same time as the Connection panel to manage connection profiles that save preferences for various connection types. A connection profile includes information on the hardware port to be used for a particular connection, the port details (speed, flow control, modem initialization string), and any other pertinent information.

The Connection Manager is not available on all Palm devices. You must ensure that it is present before you can make Connection Manager calls. If the New Serial Manager Feature Set is present, then at least the basic version of the Connection Manager is available. If the Connection Manager Feature Set is present, then an expanded version of the Connection Manager is available. This expanded Connection Manager allows profiles that specify communications with mobile phones and profiles that specify communications with Bluetooth devices. It is also more extensible, allowing you to create your own profile parameters if necessary.

NOTE: Although the Connection Manager supports Bluetooth connections, Bluetooth requires additional hardware and software that is not available as of this writing.

The basic version of the Connection Manager provides functions that list the saved connection profiles (`CncGetProfileList`), return details for a specific profile (`CncGetProfileInfo`), add a profile (`CncAddProfile`), and delete a profile (`CncDeleteProfile`).

When you create a profile with the basic Connection Manager, each profile parameter is passed as a parameter to the `CncAddProfile` function. Similarly, when you request profile information, each profile parameter is passed in an output parameter to `CncGetProfileInfo`.

Because the newer, expanded Connection Manager supports more types of connections than the basic Connection Manager, it also supports many more types of profile parameters. For this reason, you now retrieve profile information one parameter at a time using `CncProfileSettingGet`. In the new API, constants specify the predefined profile parameters. (See "General Profile Parameters" on page 1324 of the *Palm OS Programmer's API Reference*.) For example, to retrieve the connection's port, you use code similar to that shown in Listing 5.10.

Listing 5.10 Retrieving port information

```
UInt16 dataSize;
UInt32 portCreator;
```

```
dataSize = kCncParamPortSize;
err = CncProfileSettingGet(profileID, kCncParamPort,
   &portCreator, &dataSize);
```

To create a profile, you first must obtain a unique profile ID and then set the profile parameters one by one as shown in Listing 5.11. Note that Listing 5.11 uses `CncProfileOpenDB` to open the Connection Manager profile database and `CncProfileCloseDB` to close it. These are not required calls. If you don't explicitly open and close the database, each Connection Manager function opens the database, performs its work, and then closes the database. By calling `CncProfileOpenDB` in front of a series of Connection Manager calls and calling `CncProfileCloseDB` at the end, you save the overhead of having each function open and close the database.

Listing 5.11 Creating a connection profile

```
// Open the Connection Manager profile database;
err = CncProfileOpenDB();
// obtain new profile ID.
err = CncProfileCreate(&profileId);

if (!err) {
   // Create a name for the profile.
   err = CncProfileSettingSet(profileId, kCncParamName,
         myProfileName, StrLen(myProfileName)+1);

   // Set some other required parameters.
   port = serPortLocalHotSync;
   err = CncProfileSettingSet(profileId, kCncParamPort,
      &port, kCncParamPortSize);
   baud = 57600;
   err = CncProfileSettingSet(profileId, kCncParamBaud,
      &baud, kCncParamBaudSize);
   deviceKind = kCncDeviceKindSerial;
   err = CncProfileSettingSet(profileId, kCncParamDeviceKind,
      &deviceKind, kCncParamDeviceKindSize);
}

// close the profile database.
err = CncProfileCloseDB();
```

The expanded Connection Manager API also allows you to create profile parameters that are unique to your type of connection. You

can do so with the `CncDefineParamID` macro. See its description in the *Palm OS Programmer's API Reference* for more information.

The Serial Link Protocol

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

SLP Packet Structures

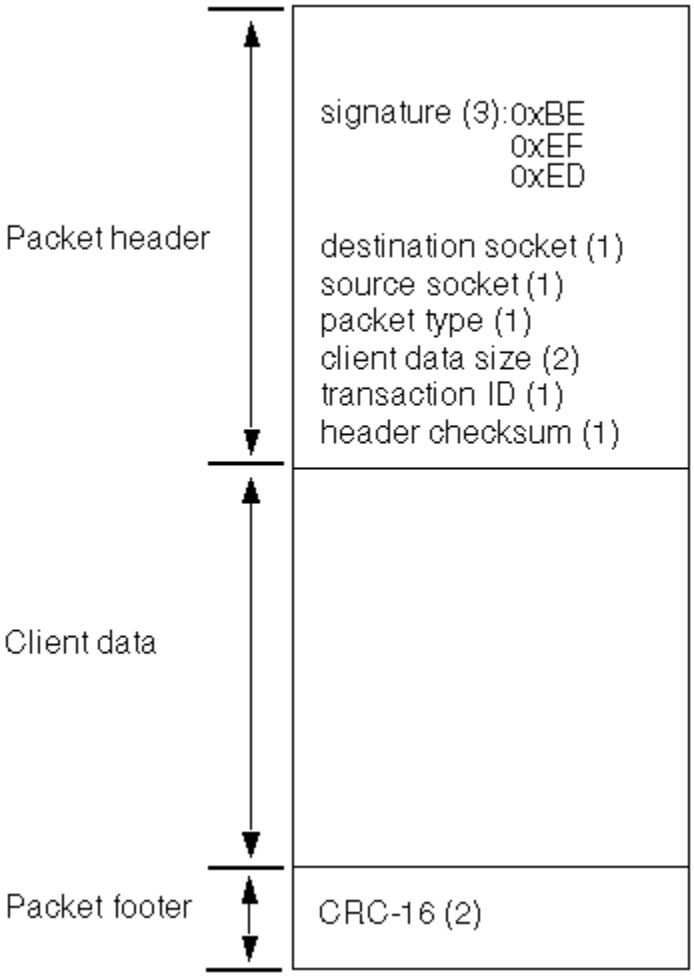
The following sections describe:

- SLP Packet Format
- Packet Type Assignment
- Socket ID Assignment
- Transaction ID Assignment

SLP Packet Format

Each SLP packet consists of a packet header, client data of variable size, and a packet footer, as shown in Figure 5.3.

Figure 5.3 Structure of a Serial Link Packet



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

Packet Type Assignment

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

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

Socket ID Assignment

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

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

Transaction ID Assignment

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

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

Transmitting an SLP Packet

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

Transmission of an SLP packet consists of these steps:

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

Receiving an SLP Packet

Receiving an SLP packet consists of these steps:

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

The Serial Link Manager

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

Serial link manager provides the mechanisms for managing multiple client sockets, sending packets, and receiving packets both synchronously and asynchronously. It also provides support for the Remote Debugger and Remote Procedure Calls (RPC).

Using the Serial Link Manager

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

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

To use the serial link manager socket services, open a Serial Link socket by calling `SlkOpenSocket`. Pass a reference number or port ID (for the Serial Manager) of an opened and initialized communications library (see `SlkClose`), a pointer to a memory location for returning the socket ID, and a Boolean indicating whether the socket is static or dynamic. If a static socket is being opened, the memory location for the socket ID must contain the desired socket number. If opening a dynamic socket, the new socket ID is returned in the passed memory location. Sharing of sockets is not supported. Success is indicated by an error code of 0 (zero). For information about static and dynamic socket IDs, see "Socket ID Assignment" on page 122.

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

To obtain the communications library reference number for a particular socket, call `SlkSocketRefNum`. The socket must already

be open. To obtain the port ID for a socket, if you are using the Serial Manager, call `SlkSocketPortID`.

To set the interbyte packet receive timeout for a particular socket, call `SlkSocketSetTimeout`.

To flush the receive stream for a particular socket, call `SlkFlushSocket`, passing the socket number and the interbyte timeout.

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

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

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

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

To send a packet, call `SlkSendPacket`, passing a pointer to the packet header (`SlkPktHeaderType`) and a pointer to an array of `SlkWriteDataType` structures. `SlkSendPacket` stuffs the signature, client data size, and the checksum fields of the packet

header. The caller must fill in all other packet header fields. If the transaction ID field is set to 0 (zero), the serial link manager automatically generates and stuffs a new non-zero transaction ID. The array of `SlkWriteDataType` structures enables the caller to specify the client data part of the packet as a list of noncontiguous blocks. The end of list is indicated by an array element with the `size` field set to 0 (zero). Listing 5.12 incorporates the processes described in this section.

Listing 5.12 Sending a Serial Link Packet

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

// Initialize packet body
...

// Compose the packet header. Let Serial Link Manager
// set the transId.
sendHdr.dest = slkSocketDLP;
sendHdr.src = slkSocketDLP;
sendHdr.type = slkPktTypeSystem;
sendHdr.transId = 0;

// Specify packet body
writeList[0].size = sizeof(body);     //first data block size
writeList[0].dataP = body;    //first data block pointer
writeList[1].size = 0;    //no more data blocks

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

Listing 5.13 Generating a New Transaction ID

```
//
// Example: Generating a new transaction ID given the
// previous transaction ID. Can start with any seed value.
//

UInt8 NextTransactionID (UInt8 previousTransactionID)
{
   UInt8  nextTransactionID;

   // Generate a new transaction id, avoid the
   // reserved values (0x00 and 0xFF)
   if ( previousTransactionID >= (UInt8)0xFE )
      nextTransactionID = 1;    // wrap around
   else
      nextTransactionID = previousTransactionID + 1;
      // increment

   return nextTransactionID;
}
```

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

Summary of Serial Communications

New and Old Serial Manager Functions

Opening and Closing the Port

SrmOpen	SrmExtOpen
SrmOpenBackground	SrmExtOpenBackground
SerOpen	SrmClose
SerClose	

New and Old Serial Manager Functions	
Receiving Data	
SrmReceive	SerReceive
SrmReceiveCheck	SerReceiveCheck
SrmReceiveFlush	SerReceiveFlush
SrmReceiveWait	SerReceiveWait
SrmReceiveWindowClose	
SrmReceiveWindowOpen	
Sending Data	
SrmSend	SerSend
SrmSendCheck	SerSendFlush
SrmSendFlush	SerSendWait
SrmSendWait	
Configuring the Port	
SrmSetReceiveBuffer	SerControl
SrmControl	SerSetReceiveBuffer
SrmCustomControl	SerSetSettings
	SerGetSettings
Error Checking	
SrmClearErr	SerClearErr
SrmGetStatus	SerGetStatus
Obtaining Device Information	
SrmGetDeviceCount	SrmGetDeviceInfo
Implementing a Wakeup Handler	
SrmPrimeWakeupHandler	SrmSetWakeupHandler

Virtual Driver Functions

DrvEntryPointProcPtr
GetSizeProcPtr
GetSpaceProcPtr
VdrvControlProcPtr
VdrvOpenProcPtr
VdrvOpenProcV4Ptr
VdrvStatusProcPtr
VdrvWriteProcPtr
VdrvControlCustomProcPtr
WriteBlockProcPtr
WriteByteProcPtr
SignalCheckPtr

Connection Manager Functions

Basic Connection Manager Functions

CncAddProfile
CncDeleteProfile
CncGetProfileInfo
CncGetProfileList

Extended Connection Manager Functions

CncProfileCreate
CncProfileDelete
CncProfileGetCurrent
CncProfileGetIDFromIndex
CncProfileGetIDFromName
CncProfileGetIndex
CncProfileOpenDB
CncProfileSetCurrent
CncProfileSettingGet
CncProfileSettingSet
CncGetParamType
CncGetSystemFlagBitnum
CncGetTrueParamID
CncIsFixedLengthParamType
CncIsSystemFlags
CncIsSystemRange
CncIsThirdPartiesRange
CncIsVariableLengthParamType
CncProfileCloseDB
CncProfileCount

Serial Link Manager Functions

SlkClose
SlkCloseSocket
SlkFlushSocket
SlkOpen
SlkOpenSocket
SlkReceivePacket
SlkSendPacket
SlkSetSocketListener
SlkSocketPortID
SlkSocketSetTimeout

6

Bluetooth

The Bluetooth APIs provide developers a way to access the Palm OS Bluetooth system and write Bluetooth-enabled applications. In addition to enabling Bluetooth development, the Palm OS Bluetooth system also provides:

- a user interface for device discovery and connection
- a user interface for passkey entry
- a modified Palm Connection Panel to support Bluetooth
- serial port emulation using the Bluetooth Virtual Serial Driver
- object exchange support using the Bluetooth Exchange Library

This documentation covers how to use the Palm OS Bluetooth APIs but does not provide the basic understanding of Bluetooth concepts and protocols that you need to write Bluetooth code. For more information about Bluetooth, refer to the *Specification of the Bluetooth System*, available at the Bluetooth Special Interest Group website at `www.bluetooth.com`. There are also several third-party books that you may wish to consult for helpful Bluetooth information.

Palm OS Bluetooth System

The Palm OS Bluetooth system enables a Palm Powered handheld to:

- access the internet through LAN access points and cell phones
- exchange objects such as business cards and appointments over Bluetooth
- perform HotSync operations over Bluetooth

- communicate with other handhelds for multi-user applications like games and various collaborative applications
- send SMS messages and manage your phone's internal phone book.

The Palm OS Bluetooth system designers focused their efforts on the user, recognizing that on the Palm OS technical interoperability is simply not enough. The user cares about the overall experience. The user's "Bluetooth learning curve" should be short. And, as always, simplicity is key.

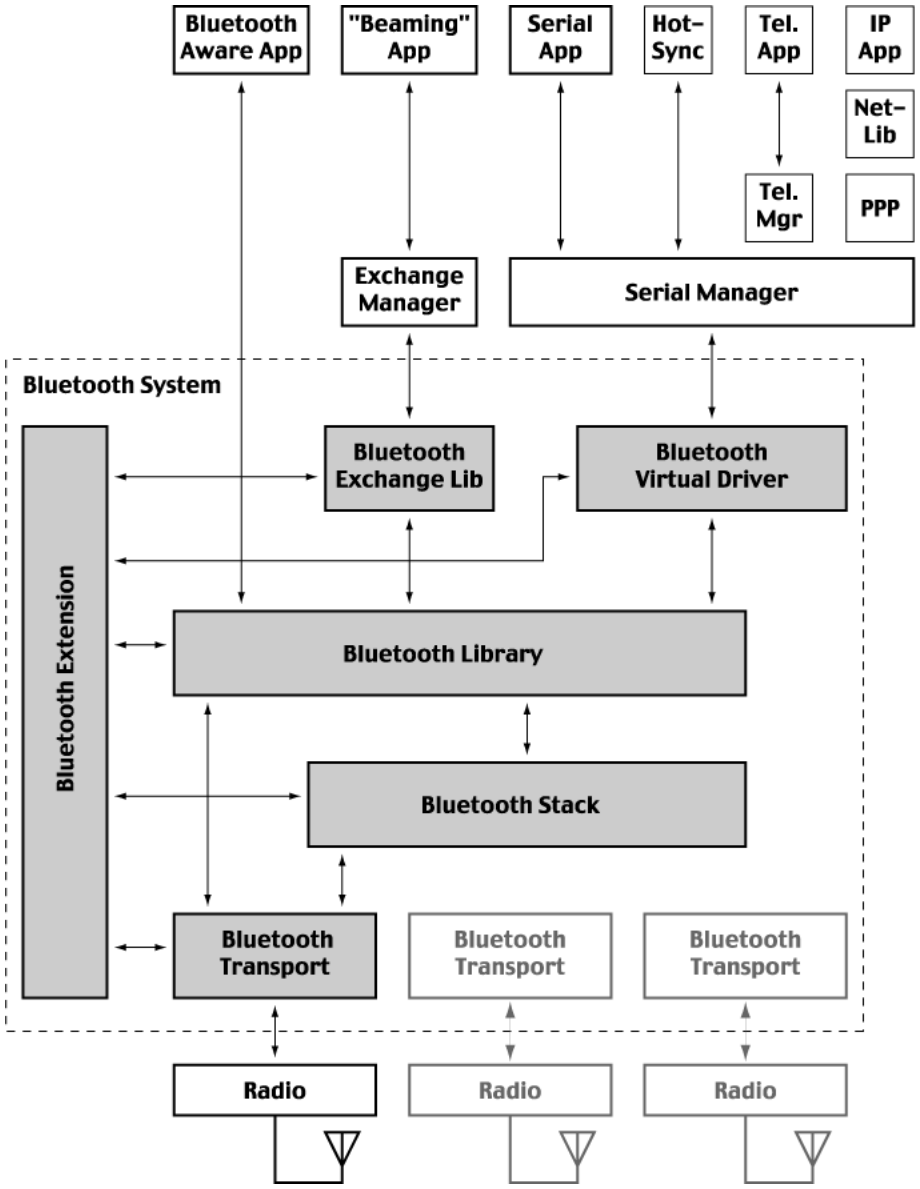
Bluetooth System Components

The Palm OS Bluetooth system contains the following components:

- Bluetooth Library
- Bluetooth Virtual Serial Driver
- Bluetooth Exchange Library
- Bluetooth Stack
- Bluetooth Transports
- Bluetooth Extension

Figure 6.1 shows these components and their relationship with each other.

Figure 6.1 Overall Palm OS Bluetooth architecture



Bluetooth Library

The Bluetooth Library is a shared library that provides an API for developers to develop Bluetooth applications. The API provides functions in the following areas:

- Managing remote devices, piconets, and ACL links

- Communicating using the L2CAP and RFCOMM protocols
- Advertising services and querying for remote services using SDP
- Maintaining a list of trusted devices

Bluetooth Virtual Serial Driver

The Bluetooth Virtual Serial Driver allows applications to use the Palm OS New Serial Manager with Bluetooth's RFCOMM protocol as the serial link. As shown in Figure 6.1, the Bluetooth Serial Driver communicates with the rest of the Bluetooth system through the Bluetooth Library. The Bluetooth Virtual Serial Driver is used by PPP, HotSync, and Telephony.

Bluetooth Exchange Library

The Bluetooth Exchange Library allows applications to use the Palm OS Exchange Manager with Bluetooth as the link. As shown in Figure 6.1, the Bluetooth Exchange Library communicates with the rest of the Bluetooth system through the Bluetooth Library. RFCOMM is used as the sole transport mechanism for the Exchange Manager.

Bluetooth Stack

The Bluetooth Stack is a shared library that implements the various protocols of the Bluetooth specification. Palm OS developers don't need to access the Bluetooth Stack directly.

Bluetooth Transports

Bluetooth Transports are shared libraries that act as device drivers for different radios. Palm OS developers cannot access the Bluetooth Transports.

Bluetooth Extension

The Bluetooth extension oversees and coordinates the multiple libraries of the Bluetooth system. Palm OS developers cannot access the Bluetooth extension.

Implementation Overview

The Bluetooth system is a collection of PRCs that can reside in either RAM or ROM. A minimum of 4Mb of RAM is required (256k heap). Incorporation into actual devices is up to the handheld manufacturers.

The Bluetooth system runs in the UI thread, except when it is used by the virtual serial driver.

Profiles

Table 6.1 lists the profiles supported by the Palm OS Bluetooth system.

Table 6.1 Supported Bluetooth profiles

Profile	Description
Generic Access	Describes the use of the lower layers of the Bluetooth protocol stack (LC and LMP), security-related alternatives, and the higher layers: L2CAP, RFCOMM, and OBEX.
Service Discovery Application	Defines the protocols and procedures used by a service discovery application on a device to locate services in other Bluetooth-enabled devices using the Service Discovery Protocol (SDP).

Table 6.1 Supported Bluetooth profiles *(continued)*

Profile	Description
Serial Port	Defines the protocols and procedures used by devices using Bluetooth for RS-232 (or similar) serial cable emulation. The scenario covered by this profile deals with legacy applications using Bluetooth as a cable replacement through a virtual serial port abstraction (which in itself is operating system-dependent).
Dial-up Networking	Defines the protocols and procedures used by devices implementing the "Internet Bridge" usage model. This profile covers the usage of a cellular phone or modem both to receive data calls and to connect to a dial-up Internet access server or other dial-up service.
LAN Access Point	Defines LAN access using PPP over RFCOMM.

Table 6.1 Supported Bluetooth profiles *(continued)*

Profile	Description
Generic Object Exchange	Defines the protocols and procedures used by the applications providing the usage models that need object exchange capabilities.
Object Push	Defines the requirements for the protocols and procedures used by applications providing the object push usage model. This profile makes use of the generic object exchange profile to define the interoperability requirements for the protocols needed by applications.

The following profiles are *not* supported by the Palm OS Bluetooth system:

- Cordless Telephony
- Intercom
- Headset
- Fax
- File Transfer
- Synchronization

Note that although the Bluetooth system does not support the Bluetooth Synchronization profile, it implements HotSync operations over Bluetooth using the Serial Port profile. Also note that network HotSync operations use PPP.

The Bluetooth system can dial and control voice calls on a Bluetooth-enabled phone as if it were connected through a serial cable. It does this using AT modem commands and not the Cordless Telephony profile.

Usage Scenarios

Bluetooth-enabled Palm Powered handhelds are able to communicate with a variety of remote Bluetooth devices. The Bluetooth system uses the profiles defined by the Bluetooth specification in order to support the following usage scenarios:

Table 6.2 Profiles required by various usage scenarios

Feature	Handheld Connects With	Required Profiles						
		Generic Access	Service Discovery	Serial Port	Dial-up Networking	LAN Access	Generic Obj. Exchange	Object Push
Email and Web Clipping	Cell phone	X	X	X	X			
	Access point	X	X	X		X		
	Desktop computer	X	X	X		X		
HotSync	Cell phone	X	X	X	X			
	Access point	X	X	X		X		
	Desktop computer	X	X	X				
SMS and Mobile Handset Management	Cell phone	X	X	X				
Beaming	Many devices	X	X	X			X	X

Authentication and Encryption

The Bluetooth system handles the generation, utilization, and storage of authentication and encryption keys at the OS level.

The Bluetooth system doesn't support Authorization. Access concerns beyond authentication are left up to the individual application, as in a standard networking environment.

The Bluetooth system supports security modes 1 and 2: the "non-secure" and "service-level enforced security" modes. Security mode 3—"link-level enforced security"—isn't supported by the Bluetooth system.

Device Discovery

In a system of Bluetooth devices, ad-hoc networks are established between the devices. The "inquiry" procedure is used to discover Bluetooth devices within range. The specification defines two inquiry modes, "General" and "Limited." The General mode, which is supported by the Bluetooth system, is used by devices that need to discover devices that are made discoverable continuously or for no specific condition. Limited mode, on the other hand, is used to devices that need to discover devices that are made discoverable for only a limited period of time, during temporary conditions, or for a specific event. The Bluetooth system doesn't support the Limited inquiry mode.

Piconet Support

There are two main scenarios in which a piconet can be created, and the Bluetooth system supports both:

- Master performs inquiry, sees a number of devices, and proceeds to contact each of them. A variant is that the master later performs inquiry to find additional slaves. This should be useful for a game server where the master establishes the connection to each Palm device that wants to participate in a game.
- Master sits in page scan mode, and when a device connects to it, a master/slave switch is performed. The LAN access profile uses this approach for multi-point LAN access devices. The Bluetooth system handles the master/slave negotiation automatically.

The Bluetooth system places existing connections in hold mode while new links are established. In the first scenario outlined above, hold times for each connection are determined based upon a list of

the devices that the user has selected to participate in the piconet. The Bluetooth system performs the following for each device on the list:

1. Establish an ACL connection to the device.
2. Place the device in hold mode for a period of time that is a function of the total number of devices that are to participate in the piconet.
3. Delay for a set period of time to allow the slave to enter hold mode.

After all connections have been established, each of the slave hold timers should expire, and the piconet should be operational.

Radio Power Management

The extended battery life of Palm Powered handhelds is considered to be a key competitive advantage by many Palm Powered handheld manufacturers. The Bluetooth system helps preserve battery life by taking advantage of the Bluetooth power efficiency modes (hold, park, and sniff) and the internal power management functionality built into the Bluetooth radio chipset.

Applications don't explicitly put the radio into the sniff, park, or standby modes. Instead, power management is under the control of the Bluetooth system. When participating in a piconet, the Bluetooth system honors requests from the other members of the piconet to enter any of the defined power-saving modes.

Some Palm OS devices support Bluetooth sleep and wake scheduling. This allows the device to be configured to be awakened by incoming connection attempts only during certain time periods.

You can determine if the device supports sleep and wake scheduling by checking for the `sysFtrBtSupportsScheduledWakeup` feature. Use this `FtrGet()` call:

```
err = FtrGet(sysFtrCreator,
sysFtrBtSupportsScheduledWakeup, &value);
```

If Bluetooth sleep and wake scheduling is available, the `value` parameter will be non-zero and the returned error should also be zero (for no error).

Developing Bluetooth-Enabled Applications

The Palm OS exposes Bluetooth through multiple interfaces, allowing you to choose the interface that is best suited for the task at hand. Bluetooth development is supported through the Serial Manager using the Virtual Serial Driver, which is discussed in "Bluetooth Virtual Serial Driver" on page 149. Object transfer is supported through the Exchange Manager using the Bluetooth Exchange Library, which is discussed in "Bluetooth Exchange Library Support" on page 154. Finally, you can program directly with the Bluetooth Library APIs, which is the subject of this section.

Regardless of which approach you take, your applications should check if the Bluetooth system is running on the handheld before using any Bluetooth APIs. To do so, use the following code:

```
UInt32 btVersion;

// Make sure Bluetooth components are installed
// This check also ensures Palm OS 4.0 or greater
if (FtrGet(btLibFeatureCreator, btLibFeatureVersion,
   &btVersion) != errNone)
{
   // Alert the user if it's the active application
   if ((launchFlags & sysAppLaunchFlagNewGlobals) &&
      (launchFlags & sysAppLaunchFlagUIApp))
      FrmAlert (MissingBtComponentsAlert);
   return sysErrRomIncompatible;
}
```

Once your application has determined that Bluetooth support is available, it needs to load the Bluetooth library. This can be done with code like this:

```
UInt16 btLibRefNum;
Err error = errNone;

if (SysLibFind(btLibName, &btLibRefNum)) {
   error = SysLibLoad(sysFileTLibrary, sysFileCBtLib,
```

```
            &btLibRefNum);
}
```

Your application is then ready to open and use the Bluetooth library; it will use the `btLibRefNum` value to reference the library.

Overview of the Bluetooth Library

From a programmer's perspective, the functions of the Bluetooth library fall into four areas: management, sockets, security, and utility.

- The management functions deal with the radio and baseband parts of the Bluetooth specification. You use them to find nearby devices and establish ACL links.
- The socket functions enable communication with L2CAP, RFCOMM, and SDP.
- The security functions manage a set of trusted devices—devices that do not have to authenticate when they create a secure connection with the handheld.
- The utility functions perform useful data conversions.

Management

Three basic management tasks common among Bluetooth applications are finding the Bluetooth devices in range, establishing ACL links, and working with piconets. However, in order for your code to use any of the functions that do these operations, you need to create a management callback function.

Management Callback Function

Most management calls are asynchronous. In other words, they start an operation and return before the operation actually completes. When the operation completes, the Bluetooth Library notifies the application by way of a callback function. Such a notification is called a **management event**.

In some cases, a management function fails before starting the asynchronous operation. In this case, the callback function does not get called. You can tell whether the callback function will be called or not by looking at the management function's return code:

`btLibErrNoError`
: The operation has completed and the callback function will not be called.

`btLibErrPending`
: The operation has started successfully and the callback function will be called,

`any other error code`
: The operation failed and the callback function will not be called.

The management callback function has two parameters: a management event structure, which contains all the information about the event that has occurred, and a reference context, an optional `UInt32` you can use to establish the context of the event. The callback function needs to provide the code that handles the events generated as a result of the operations you perform.

The callback function should not perform any heavyweight processing; doing so prevents the Bluetooth stack from running. You can defer processing by generating a custom system event in the callback function and responding to the event with your event handling code. For some operations, you must defer the processing. For example, the callback function cannot close the Bluetooth library in response to a `btLibManagementEventAclDisconnect` event.

As a simple example, consider the task of finding nearby devices, discussed in the next section. The callback function must respond to four events: `btLibManagementEventInquiryResult`, `btLibManagementEventInquiryComplete`, `btLibManagementEventInquiryCanceled`, and `btLibManagementEventRadioState`. The following code is a skeleton of the callback function you need:

```
void MyManagementCallback (BtLibManagementEventType *eventP,
  UInt32 refcon) {
  switch (eventP->event) {
    case btLibManagementEventInquiryResult :
      // A device has been found. Save it in a list
      break;
    case btLibManagementEventInquiryComplete :
      // The inquiry has finished
      break;
```

```
    case btLibManagementEventInquiryCanceled :
      // The inquiry has been canceled
      break;
    case btLibManagementEventRadioState :
      // The radio state has changed
      break;
    default :
      // Unknown event
      break;
  }
```

To tell the Bluetooth Library to use your callback function, call `BtLibRegisterManagementNotification`. You should always unregister your callback before closing the Bluetooth Library.

For a list of management events, see "Management Callback Events" in Chapter 82, "Bluetooth Library: Management."

Opening the Library

To open the Bluetooth library, use the `BtLibOpen` function. At this time, the Bluetooth library starts the radio initialization process. When initialization successfully finishes, the Bluetooth library generates a `btLibManagementEventRadioState` event with a status of `btLibErrRadioInitialized`. You must wait for the initialization to complete successfully before calling any Bluetooth library function involving the radio.

The exceptions to this rule are the discovery functions, `BtLibDiscoverMultipleDevices` and `BtLibDiscoverSingleDevice`, which handle the radio initialization events automatically and can be called directly after the Bluetooth library is opened.

Finding Nearby Devices

There are two ways to find Bluetooth devices that are within range:

- Use the `BtLibDiscoverMultipleDevices` and `BtLibDiscoverSingleDevice` functions to find nearby devices. These functions bring up a user interface that allows the user to choose one or more devices.
- Perform a device inquiry using `BtLibStartInquiry`. This is more difficult to do than using one of the discovery functions, but provides more flexibility.

When you call BtLibStartInquiry, the Bluetooth Library searches for all devices in range. Whenever it finds a device, it generates a btLibManagementEventInquiryResult event. When the inquiry has completed, a btLibManagementEventInquiryComplete event is generated. To cancel the inquiry, call BtLibCancelInquiry. The btLibManagementEventInquiryCanceled event is generated when the cancellation succeeds.

Creating ACL Links

Once you have the device address of a remote device, you can attempt to create an ACL link to it using the BtLibLinkConnect function. This causes the btLibManagementEventAclConnectOutbound event to be generated, and the status code within that event indicates whether or not the link was successfully established.

To disconnect a link, use the BtLibLinkDisconnect function. This causes the btLibManagementEventAclDisconnect event to be generated. Note that the same event is generated when the remote device initiates the disconnection.

Your program must also respond to btLibManagementEventAclConnectInbound events that indicate that a remote device has established a link with the handheld. You can disconnect an inbound link with the BtLibLinkDisconnect function.

Working With Piconets

Bluetooth supports up to seven slaves in a piconet. The Bluetooth Library provides simplified APIs to create and destroy piconets.

Note that the Bluetooth 1.1 specification suggests that the upper software layers place slaves in hold or park mode while new connections are established. This isn't well–defined in the specification, and is difficult to do because of timing. The Bluetooth Library expects the radio baseband to handle piconet timing.

To create a piconet, the "master" calls BtLibPiconetCreate. Slaves can then discover the master and join the piconet, or the master can discover and connect to the slaves. The master stops

advertising once the limit of seven slaves has been reached. Note that any device should be capable of acting as a slave.

The piconet can be locked to prevent additional slaves from joining. The master can still discover and add slaves, however. With the piconet locked, there is a bandwidth improvement of approximately 10%.

In the Bluetooth Library, the following functions support the management of piconets:

- `BtLibPiconetCreate`: create a piconet or reconfigure an existing piconet so the local device is the master.
- `BtLibPiconetDestroy`: destroy the piconet by disconnecting links to all devices and removing all restrictions on whether the local device is a master or a slave.
- `BtLibPiconetLockInbound`: prevent remote devices from creating ACL links into the piconet.
- `BtLibPiconetUnlockInbound`: allow additional slaves to create ACL links into the piconet.

Remember the following limitations of piconets: Slave-to-slave communication is not permitted. The master cannot "broadcast" to slaves.

Sockets

The Bluetooth Library uses the concept of sockets to manage communication between Bluetooth devices. A socket represents a bidirectional packet-based link to a remote device. Sockets run over ACL connections. The Bluetooth library can accommodate up to 16 simultaneous sockets.

Three types of sockets are supported by the Bluetooth Library. L2CAP and RFCOMM sockets establish data channels and send and receive arbitrary data over those channels. SDP sockets allow you to query remote devices about the services those devices provide.

To send a packet of data over an L2CAP or RFCOMM socket, use the `BtLibSocketSend` function. The send buffer must not change until the send completes. In other words, you must not modify the buffer, free the buffer, or use a local variable for the buffer. The Bluetooth Library notifies you when the send completes by

generating a btLibSocketEventSendComplete event. You can only have one outstanding packet on each socket.

The btLibSocketEventData event indicates data has been received.

L2CAP

L2CAP sockets don't allow for flow control.

Establishing Inbound L2CAP Connections

To set up for inbound L2CAP connections, you call the following:

1. BtLibSocketCreate: create an L2CAP socket.
2. BtLibSocketListen: set up an L2CAP socket as a listener.
3. BtLibSdpServiceRecordCreate: allocate a memory chunk that represents an SDP service record.
4. BtLibSdpServiceRecordSetAttributesForSocket: initialize an SDP memory record so it can represent the newly-created L2CAP listener socket as a service
5. BtLibSdpServiceRecordStartAdvertising: make an SDP memory record representing a local SDP service record visible to remote devices.

When you get a btLibSocketEventConnectRequest event, you need to respond with a call to BtLibSocketRespondToConnection. You then receive a btLibSocketEventConnectedInbound event with an inbound socket with which you can send and receive data.

The listening socket remains open and will notify you of further connection attempts. In other words, you can use a single L2CAP listening socket to spawn several inbound sockets. You cannot close the listening socket until after you close its inbound sockets.

Establishing Outbound L2CAP Connections

To establish an outbound L2CAP connection, you first establish an ACL link to the remote device. Then you call:

1. BtLibSocketCreate: create an SDP socket.
2. BtLibSdpGetPSMByUuid: get an available L2CAP PSM using SDP.

3. BtLibSocketClose: close the SDP socket.
4. BtLibSocketCreate: create an L2CAP socket.
5. BtLibSocketConnect: create an outbound L2CAP connection.

RFCOMM

RFCOMM emulates a serial connection. It is used by the Bluetooth Virtual Serial Driver and the Bluetooth Exchange Library.

When using RFCOMM, you can only have one inbound connection per listener socket. Flow control uses a "credit" system: you need to advance a credit to the far end before you can receive a data packet.

RFCOMM defines the notions of **server** and **client**. A server uses SDP to advertise its existence and listens for inbound connections. A client creates an outbound RFCOMM connection to a server.

Establishing Inbound RFCOMM Connections

To set up for inbound RFCOMM connections, call the following:

1. BtLibSocketCreate: create an RFCOMM socket.
2. BtLibSocketListen: set up the RFCOMM socket as a listener.
3. BtLibSdpServiceRecordCreate: allocate a memory chunk that represents an SDP service record.
4. BtLibSdpServiceRecordSetAttributesForSocket: initialize an SDP memory record so it can represent the newly-created RFCOMM listener socket as a service
5. BtLibSdpServiceRecordStartAdvertising: make the SDP memory record representing your local SDP service record visible to remote devices.

When you get a btLibSocketEventConnectRequest event, you need to respond with a call to BtLibSocketRespondToConnection. You then receive a btLibSocketEventConnectedInbound event with an inbound socket with which you can send and receive data. To send data, use the BtLibSocketSend function. The btLibSocketEventData event indicates data has been received.

The listening socket will not notify you of further connection attempts. In other words, a single RFCOMM listening socket can only spawn a single inbound RFCOMM socket. You cannot close the listening socket until after you close its inbound socket.

Establishing Outbound RFCOMM Connections

To establish an outbound RFCOMM connection, you first establish an ACL link to the remote device. Then you call:

1. `BtLibSocketCreate`: create an SDP socket.
2. `BtLibSdpGetServerChannelByUuid`: get an available RFCOMM server channel using SDP.
3. `BtLibSocketCreate`: create an RFCOMM socket.
4. `BtLibSocketConnect`: Create an outbound RFCOMM connection.

Bluetooth Virtual Serial Driver

The Bluetooth system implements the serial port profile with a Virtual Serial Driver. This driver has the following characteristics:

- Opens a background thread for the Bluetooth stack.
- Supports only one current active serial channel (point-to-point connection) at a time.
- Is opened explicitly as either a client or a server.
- Is utilized, as a client, by the following Palm OS components:
  - PPP
  - HotSync
  - Telephony
- If opened as a server, advertises a list of services (UUIDs) for remote clients to query.
- If opened as a client, creates the necessary baseband and RFCOMM connections, based upon information passed in by the opener.

An RFCOMM-based virtual serial port is far less symmetrical than a physical serial port. In a traditional serial port, there is no need to establish the underlying transport. When establishing a Bluetooth

serial port, however, there are roles for a client and a server device on three different stack levels—ACL, L2CAP, and RFCOMM—as well as responsibilities for registering with and querying SDP.

Opening the Serial Port

You can use the new `SrmExtOpen` function to open a Bluetooth serial port, passing Bluetooth-specific parameters in a custom info block. You should first verify that the port being opened is in fact a Bluetooth serial port, since other types of ports may use the block for other purposes.

For the benefit of certain legacy applications, the driver also supports being opened by the old `SrmOpen` function. In that case, the driver presumes the role of client, performs device discovery with user interaction, and looks for remote channel advertising the Serial Port Service Class.

To open an RFCOMM virtual serial port with `SrmExtOpen`, you need to create a `BtVdOpenParams` structure. This structure is declared as follows:

```
typedef struct {
    BtVdRole role;
    union {
        BtVdOpenParamsClient client;
        BtVdOpenParamsServer server;
    } u;
    Boolean authenticate;
    Boolean encrypt;
} BtVdOpenParams;
```

How you populate this structure depends on whether you are opening the serial port as a client or as a server. A client initiates baseband and RFCOMM connections, while a server waits for incoming baseband and RFCOMM connections.

If you are acting as a client, set the structure's `role` member to `btVdClient` and fill in the `client` member of the union as described in "Opening the Port as a Client" on page 151. If you are acting as a server, set the structure's `role` member to `btVdServer` and fill in the `server` member of the union as described in "Opening the Port as a Server" on page 152.

Irrespective of whether you are acting as a client or as a server, set `authenticate` to `true` if you require link authentication. Similarly, set `encrypt` to `true` if you require link encryption. Link encryption requires link authentication.

Opening the Port as a Client

When playing the client role you must specify the address of the remote Bluetooth device and the service to connect to on the remote device. You do this by filling out the `client` member of the union in the `BtVdOpenParams` structure. This member is declared to be a `BtVdOpenParamsClient` structure, which looks like this:

```
typedef struct {
    BtLibDeviceAddressType remoteDevAddr;
    BtVdClientMethod method;
    union {
        BtLibRfCommServerIdType channelId;
        BtVdUuidList uuidList;
    } u;
} BtVdOpenParamsClient;
```

If you know the address of the Bluetooth device to which you want to connect, supply the address in `remoteDevAddr`. Otherwise, supply an address value of all zeros; this will cause a Bluetooth device discovery operation to be initiated, allowing the handheld user to choose the device to connect to.

When connecting to a remote service using RFCOMM, you have two basic options:

- use SDP to look for one or more UUIDs
- connect to a specific RFCOMM channel ID

You normally specify the service to which to connect by providing a list of one or more service class UUIDs. Simply set the structure's `method` member to `btVdUseUuidList` and supply a list of UUIDs using the `uuidList` member of the structure's union. This will trigger a series of SDP queries, searching for each of the specified service classes. The first service class that is found on the remote device will be used. If it suits your application, specify an empty list of service class UUIDs (set the list count of zero); this

causes an SDP query to be made for a default Palm-specific service class UUID (953D4FBC-8DA3-11D5-AA62-0030657C543C).

The result of a successful SDP query is the RFCOMM server channel to which to connect on the remote device. To facilitate testing, you can bypass SDP querying and directly specify the remote RFCOMM server channel ID. Simply set the structure's `method` member to `btVdUseChannelId` and set the `channelId` member of the structure's union to the server channel ID.

The call to `SrmExtOpen` blocks until the RFCOMM connection is established or it is determined that the connection cannot be established. The driver displays a progress dialog, giving the user the opportunity to cancel the connection attempt.

`SrmExtOpen` returns zero if and only if the RFCOMM connection was successfully established.

Opening the Port as a Server

Relative to the process of opening the serial port as a client, opening the port as a server is pretty simple. When playing the server role, you need only specify the UUID of the service you wish to advertise. Optionally, you can also specify a user-readable name for that service.

Specify the service UUID and user-readable name by filling out the `server` member of the union in the `BtVdOpenParams` structure. This member is a `BtVdOpenParamsServer` structure, which is declared as follows:

```
typedef struct {
    BtLibSdpUuidType uuid;
    const Char *name;
} BtVdOpenParamsServer;
```

As a convenience, you can specify a null UUID (all binary zeros), in which case the default Palm-specific service class UUID will be advertised. (953D4FBC-8DA3-11D5-AA62-0030657C543C).

The call to `SrmExtOpen` returns immediately, without waiting for an incoming RFCOMM connection. To wait for incoming data, periodically call either `SrmReceive`, `SrmReceiveWait`, or `SrmReceiveCheck`.

Example

The following code excerpt illustrates a call to `SrmExtOpen` that, acting as a client, creates an RFCOMM virtual serial port and connects to a known RFCOMM channel on a remote device:

```
Err err;
SrmOpenConfigType config;
BtVdOpenParams btParams;
UInt16 btPortId;

config.function = 0;                                // must be zero
config.drvrDataP = (MemPtr)&btParams;
config.drvrDataSize = sizeof(BtVdOpenParams);

btParams.role = btVdClient;                         // we are the client side
btParams.u.client.remoteDevAddr.address[0] = ...; // remote device addr byte 1
    ...
btParams.u.client.remoteDevAddr.address[5] = ...; // remote device addr byte 6
btParams.u.client.method = btVdUseChannelId;
btParams.u.client.u.channelId = 0x53;

err = SrmExtOpen(
    sysFileCVirtRfComm, // type of port == RFCOMM
    &config,            // port configuration params
    sizeof(config),     // size of port config params
    &btPortId           // receives the id of this virtual serial port instance
);
```

Note that this code excerpt will not compile as-is; the remote Bluetooth device address has not been properly specified.

Palm-to-Palm Communication

Most applications act as clients only. However, in the case of Palm-to-Palm applications, they may need to act as both clients and servers. In this case, the virtual serial driver should initially be configured as a server to advertise its services to the other remote device. At this point, both devices are acting as servers, advertising their services. When a user-initiated action causes one of the devices to reopen the virtual serial driver as a client, it can then discover the remote device and its advertised service—advertised through a predefined, agreed-upon UUID—so that a channel can be opened between the two devices.

How Palm OS Uses the Bluetooth Virtual Serial Driver

Within the Palm OS, HotSync, PPP, and the Telephony Manager can all use Bluetooth, although they can only act as clients.

For these clients, the user performs device discovery and device pairing, if appropriate, from within the Connection Panel. These clients can then consult the Connection Panel to determine the address of the remote device, the link key, and the service class to look for on the remote device. For example, if PPP is using the connection profile that indicates that the remote device is a phone or a modem, it looks for the Dialup Networking Service Class UUID. But if the profile indicates that the remote device is a PC or a LAN access point, it looks for the LAN Access Using PPP Service Class UUID.

Bluetooth Exchange Library Support

Accompanying the Bluetooth Library is the Bluetooth Exchange Library, a shared library that allows applications to support Bluetooth using the standard Exchange Manager APIs. The Bluetooth Exchange Library conforms to the Object Push and Generic Object Exchange profiles.

For more information about the Exchange Manager, see the "Object Exchange" chapter of the *Palm OS Programmer's Companion*.

Detecting the Bluetooth Exchange Library

To check for the presence of the Bluetooth Exchange Library, you use `FtrGet`:

```
err = FtrGet(btexgFtrCreator,
btexgFtrNumVersion, &btExgLibVersion);
```

If the Bluetooth Exchange Library is present, `FtrGet` returns `errNone`. In this case, the value pointed to by `btExgLibVersion` contains the version number of the Bluetooth Exchange Library. The format of the version number is `0xMMmfsbbb`, where `MM` is the major version, `m` is the minor version, `f` is the bug fix level, `s` is the

stage, and `bbb` is the build number. Stage 3 indicates a release version of the library. Stage 2 indicates a beta release, stage 1 indicates an alpha release, and stage 0 indicates a development release. So, for example, a value of `0x01013000` would correspond to the released version 1.01 of the Bluetooth Exchange Library.

Using the Exchange Manager With Bluetooth

Using the Exchange Manager with Bluetooth is almost exactly like using it with IRDA and SMS. The differences are as follows:

- The URL you use when you send an object has some special fields specific to Bluetooth.
- Your application may want to know the URL of the device or devices with which it is communicating. The Exchange Manager provides a way to get this information.
- The `ExgGet` and `ExgRequest` functions are not supported with Bluetooth.

These differences are discussed further in the following sections.

Bluetooth Exchange URLs

If you send objects using the Bluetooth Exchange Library and use a URL, you can send the objects to single or multiple devices at the same time depending on the way the URL is formed. A Bluetooth Exchange Library URL can have one of the following forms:

`_btobex:`*`filename`*

`_btobex://`*`filename`*

`_btobex://?_multi/`*`filename`*
: Performs a device inquiry, presents the available devices to the user, and allows the user to choose one or more devices. Sends the object to all selected devices.

`_btobex://?_single/`*`filename`*
: Performs a device inquiry, presents the available devices to the user, and allows the user to choose only one device. Sends the object to that device.

`_btobex://`*`address1`*`[,`*`address2`*`, ...]/`*`filename`*
: Sends the object to the device(s) with the specified Bluetooth device address(es). The addresses are in the form "`xx:xx:xx:xx:xx:xx`".

Do not combine these URL forms. Doing so may give unintended results.

Obtaining the URL of a Remote Device

For some applications you need to know the URL that addresses the remote device from which you receive data. This is especially useful for games. You can get the URL after calling `ExgAccept` using a `ExgControl` function code as shown in the following code:

```
ExgCtlGetURLType getUrl;
UInt16 getUrlLen;

// First get the size of the URL
getUrl.socketP = exgSocketP;
getUrl.URLP = NULL;
getUrl.URLSize = 0;
getUrlLen =  sizeof(getUrl);
ExgControl(exgSocketP, exgLibCtlGetURL, &getUrl, &getUrlLen);

// Now get the URL
getUrl.URLP = MemPtrNew(getUrl.URLSize);
ExgControl(exgSocketP, exgLibCtlGetURL, &getUrl, &getUrlLen);

// getUrl.URLP points to a null-terminated URL string
// describing the remote device, for example,
// "_btobex://01:23:45:67:89:ab/"
...
// Free the URL after you're done with it
MemPtrFree(getUrl.URLP);
```

ExgGet and ExgRequest

The Bluetooth Exchange Library does not support the pull functions provided by `ExgGet` and `ExgRequest`. If you want to perform these functions, you must use the general Bluetooth Library APIs. See "Developing Bluetooth-Enabled Applications".

7

Network Communication

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

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

Net Library

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

The basic network services provided by the net library include:

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

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

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

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

About the Net Library

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

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

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

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

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

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

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

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

Constraints

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

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

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

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

Palm OS Garnet versions 5.4 and higher no longer impose a maximum stack size for the Net Library nor a maximum number of active, simultaneous network sockets. Individual manufacturers of Palm OS devices decide on the appropriate stack size and number of active sockets.

The Programmer's Interface

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

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

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

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

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

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

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

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

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

The macro for `socket` is:

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

Net Library Usage Steps

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

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

1. Obtain the net library's reference number.

   Because the net library is a system library, all net library calls take the library's reference number as the first parameter. For this reason, your first step is to obtain the reference number

and save it. See "Obtaining the Net Library's Reference Number."

Set up for using Berkeley sockets API.

You can either use the net library's native API or the Berkeley sockets API for the majority of what you do with the net library. If you're already familiar with Berkeley sockets API, you'll probably want to use it instead of the native API. If so, follow the steps in "Setting Up Berkeley Socket API."

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

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

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

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

4. Make calls to access the network.

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

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

   Closing the net library frees up the resources. See "Closing the Net Library."

Obtaining the Net Library's Reference Number

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

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

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

Setting Up Berkeley Socket API

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

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

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

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

Setup and Configuration Calls

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

In rare instances, an application might need to perform setup and configuration itself. For example, some applications might allow

users to select a particular "service" before trying to establish a connection. Such applications present a pick list of service names and allow the user to select a service name. This functionality is provided via the Network preferences panel. The panel provides launch codes (defined in `SystemMgr.h`) that allow an application to present a list of possible service names to let the end user pick one. The preferences panel then makes the necessary net library setup and configuration calls to set up for that particular service.

Usually, the setup and configuration calls are made while the library is closed. A subset of the calls can also be issued while the library is open and will have real-time effects on the behavior of the library. Chapter 66, "Net Library," in *Palm OS Programmer's API Reference*, describes the behavior of each call in more detail.

Settings for Interface Selection

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

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

- `NetLibIFAttach` attaches an interface to the library so that it will be used when and if the library is open.
- `NetLibIFDetach` detaches an interface from the library.
- `NetLibIFGet` returns an interface's creator and instance number.

Unlike most net library functions, these functions can be called while the library is open or closed. If the library is open, the specific interface is attached or detached in real time. If the library is closed, the information is saved in the active configuration. See "Network Configurations" on page 168 for more information about configurations.

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

addresses of two variables where the creator and instance number are returned.

The creator is one of the following values:

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

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

Interface Specific Settings

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

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

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

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

Setting an Interface's Login Script

The `netIFSettingLoginScript` setting is used to store the login script for an interface. The login script is generated either from the script that the user enters in the Network preferences panel or from a script file that is downloaded onto the handheld during a HotSync® operation. The format of the script is rigid; if a syntactically incorrect login script is presented to the net library, the results are unpredictable. The basic format is a series of null-terminated command lines followed by a null byte at the end of the script. Each command line has the format:

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

where the command byte is the first character in the line and there is 1 and only 1 space between the command byte and the parameter string. Table 7.1 lists the possible commands.

Table 7.1 Login Script Commands

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

1. See "Extending the Network Login Script Support."

The parameter string to the send (`s`) command can contain the escape sequences shown in Table 7.2.

Table 7.2 Send Command Escape Sequences

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

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

```
set serviceName
set userID
set password
set phoneNumber
set connection
set ipAddr
```

```
set dynamicIP
set primaryDNS
set secondaryDNS
set queryDNS
set closewait
set inactivityTimeout
set establishmentTimeout
set protocol
waitfor
transmit
getip
delay
prompt
waitforprompt
plugin "pluginname:cmd[:arg]"
```

The `plugin` command is a Palm OS-specific extension used to perform a command defined in a plugin. See "Extending the Network Login Script Support" for more information on plugins.

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

General Settings

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

Network Configurations

Palm OS 3.2 and later supports network configurations. A **configuration** captures a particular way the user can connect to the internet. The net library maintains an array of configurations. When the net library is opened, the network settings for the connection are supplied by one of the configurations in the array.

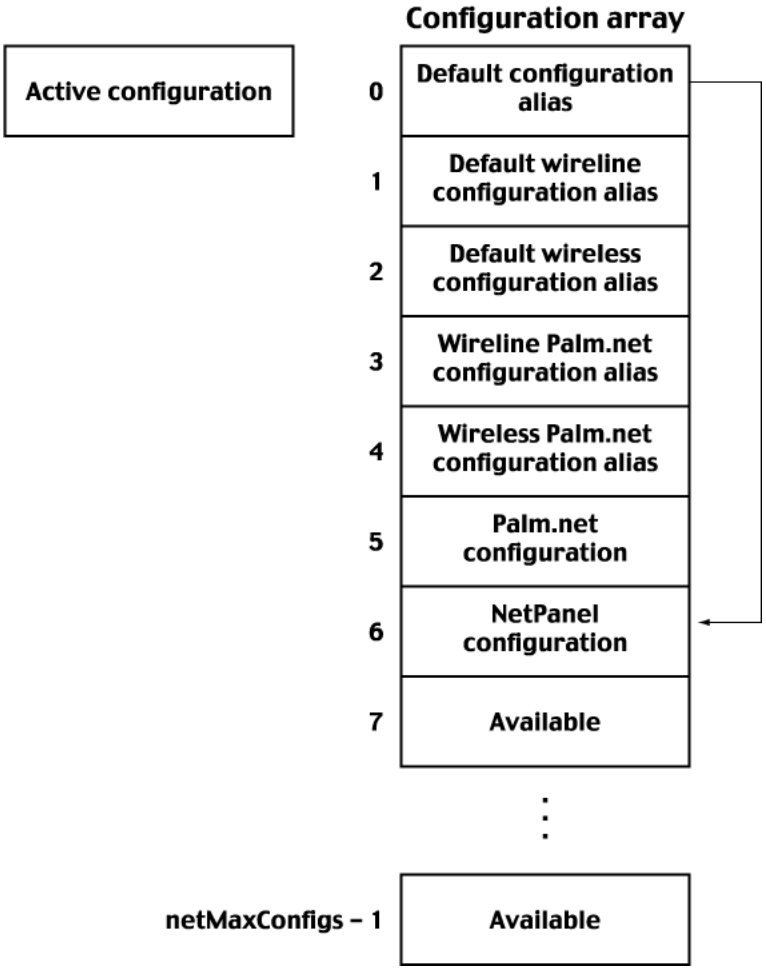
A configuration contains the following information:

- the network interfaces used by the net library. When the net library is opened, it brings up these network interfaces.

- general net library settings. These are the settings accessed by the `NetLibSettingGet` and `NetLibSettingSet` functions.[1]

The configuration array is unchanged after a soft reset. It is erased and reinitialized after a hard reset. See Figure 7.1 on page 169 for a diagram of the configuration array.

Figure 7.1 Configuration architecture



1. Except the trace settings. These settings are global—they are not different for different configurations.

A configuration can be an **alias**. An alias does not contain any configuration information. Instead, it points to another configuration. The net library defines an alias for each of the following configurations:

- A default configuration for general use
- Wireline configuration for general use
- Wireless configuration for general use
- Wireline configuration for use with the Palm.net proxy server
- Wireless configuration for use with the Palm.net proxy server

You can specify an alias anywhere in the API you would specify a configuration.

An example of an alias is the first configuration in the configuration array, called the **default configuration**. When you call `NetLibOpen` to open the net library, the net library gets its connection settings from this configuration. The default configuration typically points to the sixth configuration, called "`NetPanel`" that actually contains the settings. Therefore, when you call `NetLibOpen`, the net library gets its settings from the "`NetPanel`" configuration. See "Opening the Net Library."

The net library maintains another special configuration called the **active configuration**. This configuration is a copy of the configuration used when the net library was last opened. When you attach or detach network interfaces or modify general net library settings, you modify the active configuration. Changing the active configuration this way does not affect any stored configurations.

NOTE: You cannot open the net library according to the settings in the active configuration. You need to save the active configuration before you can use its settings to open the net library.

The net library provides functions to manage configurations. These functions have names beginning with `NetLibConfig`.

- `NetLibConfigList` returns a list of all configurations by name. You could use this to display a list of available

configurations to your users and allow them to choose which one should be used.

- NetLibConfigIndexFromName obtains a configuration's index number from its name. Most configuration functions use the index number to refer to an configuration instead of its name.
- NetLibConfigAliasGet obtains the value of a configuration alias.
- NetLibConfigAliasSet sets the configuration alias to point to a specific configuration.
- NetLibConfigSaveAs defines a new configuration and saves it by name.
- NetLibConfigDelete deletes a configuration from the list.

WARNING! The Network Panel may interfere with your configuration. When the user exits after making a modification to a service in the Network Panel, the Network Panel overwrites the "NetPanel" configuration and resets the default configuration alias to point to the "NetPanel" configuration.

Suppose your application requires the use of wireless communications. It could obtain access to the user's default wireless setup and use it to initialize the net library in the following way (the constant netCfgNameDefWireless defines the name of the default wireless configuration alias):

```
UInt16 configIndex, ifErr;
Err err;

err = NetLibConfigIndexFromName(ref,
   netCfgNameDefWireless, &configIndex);
if (!err)
   err = NetLibOpenConfig(ref, configIndex, 0,
     &ifErr);
```

Listing 7.1 shows another example of using the configuration functions. It demonstrates how to create a configuration that uses a custom network interface. The code also points the default configuration alias to the new configuration so NetLibOpen will open the library according to the settings in the new configuration.

Listing 7.1 Creating a configuration

```
#define myNetIFCreator '....' // Set this value

Err CreateMyConfig () {
   Err err;
   UInt16 instance;
   UInt32 creator;
   UInt16 netLibRefNum;
   UInt16 index;
   NetConfigNameType myConfigName = { "..." }; // Set this too

   // Find the reference number of the Net Library
   err = SysLibFind("Net.lib",&netLibRefNum);
   if (err) return err;

   // Activate the default configuration
   err = NetLibConfigMakeActive(netLibRefNum,0);
   if (err) return err;

   // Detach all network interfaces
   while (true) {
      err = NetLibIFGet(netLibRefNum,0,&creator,&instance);
      if (err) break;
      err = NetLibIFDetach(netLibRefNum,creator,instance,1000L);
      if (err) return err;
   }

   // Attach the custom network interface
   err = NetLibIFAttach(netLibRefNum,myNetIFCreator,0,1000L);
   if (err) return err;

   // Save the configuration so you can use it to open the Net Library
   err = NetLibConfigSaveAs(netLibRefNum,&myConfigName);
   if (err) return err;

   // Get the index of the new configuration
   err = NetLibConfigIndexFromName(netLibRefNum,&myConfigName,&index);
   if (err) return err;

   // Point the default configuration alias to the new configuration
   err = NetLibConfigAliasSet(netLibRefNum,0,index);
   return err;
}
```

Opening the Net Library

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

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

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

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

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

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

```
UInt16 index, ifInstance;
UInt32 ifCreator;
Err err;
```

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

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

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

On Palm OS 3.2 or later, you can open the net library with a specific network configuration (see "Network Configurations") with the function `NetLibOpenConfig`. Typically, you'd specify one of the configuration aliases. For example, your application might require a wireline network, so you would open the net library with the configuration `netCfgNameDefWireline` to specify the user's default wireline connection. On Palm OS 3.2 or later, `NetLibOpen` simply calls `NetLibOpenConfig` specifying the user's default configuration.

Closing the Net Library

Before an application quits, or if it no longer needs to do network I/O, it should call `NetLibClose`.

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

`NetLibClose` simply decrements the open count. The `false` parameter specifies that if the open count has reached 0, the net library should not immediately close. Instead, `NetLibClose`

schedules a timer to shut down the net library unless another NetLibOpen is issued before the timer expires. When the net library's open count is 0 but its timer hasn't yet expired, it's referred to as being in the **close-wait state**.

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

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

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

- Set NetLibClose's second parameter to true. This parameter specifies whether the library should close immediately or not.
- Call NetLibFinishCloseWait. This function checks the net library to see if it's in the close-wait state and if so, performs an immediate close.

Version Checking

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

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

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

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

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

For example:

V1.1.2b3 would be encoded as 0x01122003

V2.0a2 would be encoded as 0x02001002

V1.0.1 would be encoded as 0x01013000

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

Network I/O and Utility Calls

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

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

- You can open a maximum of four sockets at once in the net library. This is to keep net library's memory requirements to a minimum.
- When you try to send a large block of data, the net library automatically buffers only a portion of that block because of the limited available dynamic memory. The function call returns the number of bytes of data that it actually transmitted. You must check the return value and if there's more data to send, call the function again until the transmission is finished.

- If you expect to also receive data during a large transmission, you should send a smaller block, then read back whatever is available to read before sending the next block. In this way, the amount of memory in the dynamic heap that must be used to buffer data waiting to send out and data waiting to be read back in by the application is kept to a minimum.

For more information, see the following:

- The next section, "Berkeley Sockets API Functions," provides tables that list the supported Berkeley sockets calls, the corresponding native net library call, and gives a brief description of what each call does.
- Chapter 66, "Net Library," of the *Palm OS Programmer's API Reference* provides detailed descriptions of each net library call. Where applicable, it gives the equivalent sockets API call for each net library native call.
- The `NetSample` example application in the `Palm OS Examples` directory shows how to use the Berkeley sockets API in Palm OS applications.

Berkeley Sockets API Functions

This section provides tables that list the functions in the Berkeley sockets API that are supported by the net library. In some cases, the calls have limited functionality from what's found in a full implementation of the sockets API and these limitations are described here.

Socket Functions

Berkeley Sockets Function	Net Library Function	Description
accept	`NetLibSocketAccept`	Accepts a connection from a stream-based socket.
`bind`	`NetLibSocketBind`	Binds a socket to a local address.
`close`	`NetLibSocketClose`	Closes a socket.

Berkeley Sockets Function	Net Library Function	Description
`connect`	`NetLibSocketConnect`	Connects a socket to a remote endpoint to establish a connection.
`fcntl`	`NetLibSocketOptionSet` `NetLibSocketOptionGet (...,netSocketOptSockNonBlocking,...)`	Supported only for socket `refnums` and the only commands it supports are `F_SETFL` and `F_GETFL`. The commands can be used to put a socket into non-blocking mode by setting the `FNDELAY` flag in the argument parameter appropriately — all other flags are ignored. The `F_SETFL`, `F_GETFL`, and `FNDELAY` constants are defined in `<unix/unix_fcntl.h>`.
`getpeername`	`NetLibSocketAddr`	Gets the remote socket address for a connection.
`getsockname`	`NetLibSocketAddr`	Gets the local socket address of a connection.
`getsockopt`	`NetLibSocketOptionGet`	Gets a socket's control options. Only the following options are implemented: • `TCP_NODELAY` Allows the application to disable the TCP output buffering algorithm so that TCP sends small packets as soon as possible. This constant is defined in `<unix/netinet_tcp.h>`.

Berkeley Sockets Function	Net Library Function	Description
		• `TCP_MAXSEG` Get the TCP maximum segment size. This constant is defined in `<unix/netinet_tcp.h>`. • `SO_KEEPALIVE` Enables periodic transmission of probe segments when there is no data exchanged on a connection. If the remote endpoint doesn't respond, the connection is considered broken, and `so_error` is set to `ETIMEOUT`. • `SO_LINGER` Specifies what to do with the unsent data when a socket is closed. It uses the `linger` structure defined in `<unix/sys_socket.h>`. • `SO_ERROR` Returns the current value of the variable `so_error`, defined in `<unix/sys_socketvar.h>` • `SO_TYPE` Returns the socket type to the caller.
`listen`	`NetLibSocketListen`	Sets up the socket to listen for incoming connection requests. The queue size is quietly limited to 1. (Higher values are ignored.)

Berkeley Sockets Function	Net Library Function	Description
`read`, `recv`, `recvmsg`, `recvfrom`	`NetLibReceive` `NetLibReceivePB`	Read data from a socket. The `recv`, `recvmsg`, and `recvfrom` calls support the `MSG_PEEK` flag but not the `MSG_OOB` or `MSG_DONTROUTE` flags.
`select`	`NetLibSelect`	Allows the application to block on multiple I/O events. The system will wake up the application process when any of the multiple I/O events occurs. This function uses the `timeval` structure defined in `<unix/sys_time.h>` and the `fd_set` structure defined in `sys/types.h`. Also associated with this function are the following four macros defined in `<unix/sys_types.h>`: • `FD_ZERO` • `FD_SET` • `FD_CLR` • `FD_ISSET` Besides socket descriptors, this function also works with the "stdin" descriptor, `sysFileDescStdIn`. This descriptor is marked as ready for input whenever a user or system event is available in the event queue. This includes any event that would be returned by `EvtGetEvent`. No other descriptors besides `sysFileDescStdIn` and socket `refnums` are allowed.

Berkeley Sockets Function	Net Library Function	Description
`send`, `sendmsg`, `sendto`	`NetLibSend` `NetLibSendPB`	These functions write data to a socket. These calls, unlike the `recv` calls, *do* support the `MSG_OOB` flag. The `MSG_PEEK` flag is not applicable and the `MSG_DONTROUTE` flag is not supported.
`setsockopt`	`NetLibSocketOptionSet`	This function sets control options of a socket. Only the following options are allowed: • `TCP_NODELAY` • `SO_KEEPALIVE` • `SO_LINGER`
`shutdown`	`NetLibSocketShutdown`	Similar to `close()`; however, it gives the caller more control over a full-duplex connection.
`socket`	`NetLibSocketOpen`	Creates a socket for communication.The only valid address family is `AF_INET`. The only valid socket types are `SOCK_STREAM`, `SOCK_DGRAM`, and in Palm OS version 3.0 and higher, `SOCK_RAW`. The protocol parameter should be set to 0.
`write`	`NetLibSend`	Writes data to a socket.

Supported Network Utility Functions

Berkeley Sockets Function	Net Library Function	Description
`getdomainname`	`NetLibSocketOptionGet(.., netSettingDomainName, ...)`	Returns the domain name of the local host.
`gethostbyaddr`	`NetLibGetHostByAddr`	Looks up host information given the host's IP address. It returns a `hostent` structure, as defined in `<netdb.h>`.
`gethostbyname`	`NetLibGetHostByName`	Looks up host information given the host's name. It returns a `hostent` structure which is defined in `<netdb.h>`.
`gethostname`	`NetLibSettingGet(.., netSettingHostName, ...)`	Returns the name of the local host.
`getservbyname`	`NetLibGetServByName`	Returns a `servent` structure, defined in `<netdb.h>` given a service name.
`gettimeofday`	glue code using `TimGetSeconds`	Returns the current date and time.
`setdomainname`	`NetLibSettingSet(.., netSettingDomainName, ...)`	Sets the domain name of the local host.
`sethostname`	`NetLibSettingSet(.., netSettingHostName, ...)`	Sets the name of the local host.
`settimeofday`	glue code using `TimSetSeconds`	Sets the current date and time.

Supported Byte Ordering Macros

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

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

Supported Network Address Conversion Functions

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

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

Berkeley Sockets Function	Net Library Function	Description
`inet_netof`	glue code	Returns the network number part of an IP address.
`inet_ntoa`	`NetLibAddrINToA`	Converts an IP address from 32-bit format to dotted decimal format.

Extending the Network Login Script Support

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

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

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

The sections below describe the basics of how to write a login script plugin. For more detailed information on the API you use to write a plugin, see the chapter "Script Plugin" on page 1555 in the *Palm OS Programmer's API Reference*.

Writing the Login Script Plugin

To write a login script plugin, you create a project like you normally would; however, specify `'scpt'` as the database type instead of

`'appl'`. (If you're using Metrowerks CodeWarrior, you specify the database type in the PalmRez post linker panel.)

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

- `scptLaunchCmdListCmds` to inform the Network preferences panel of the commands your plugin implements.
- `scptLaunchCmdExecuteCmd` to execute one of your commands.

Responding to scptLaunchCmdListCmds

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

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

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

Responding to scptLaunchCmdExecuteCmd

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

The `scptLaunchCmdExecuteCmd` parameter block is a structure of type `PluginExecCmdType`. It contains:

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

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

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

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

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

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

Listing 7.2 Simple Script Plugin Command

```
#define  pluginSecondCmd "Send Uname"

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

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

   dataBuffer = MemPtrNew(pluginMaxLenTxtStringArg+1);
   if (!dataBuffer) {
      return failure;
   }
   MemSet(dataBuffer,pluginMaxLenTxtStringArg+1,0);

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

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

      dataSize = StrLen((Char*)dataBufferP);

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

   return error;
   }
}
```

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

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

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

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

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

Socket Notices

Palm OS Garnet version 5.4 introduces a mechanism that allows an application to respond when a socket changes state—for instance, when a socket is closed or when the socket receives TCP data. This mechanism is called a **socket notice**. Your application registers for a socket notice by calling `NetLibSocketOptionSet()` and passing

in the *condition* (the socket state changes you are interested in) and the *notice type* (the means by which Palm OS communicates with your application—for instance, by sending a notification).

NOTE: In Palm OS Garnet version 5.4, socket notices can communicate with your application *only* through notifications. No other means of receiving socket notices is currently supported.

Socket Notifications

The short-hand term for "socket notices that send notifications" is **socket notifications**. This section describes how to use socket notifications in your application:

1. Define a socket notification constant—the **notify type**. This constant must be unique to your application.
2. Register for the notification by calling `SysNotifyRegister()` and passing in your socket notification constant and any data your application will need to handle the notification. The other parameters you pass in are determined by whether you are registering to receive a notification through a `sysAppLaunchCmdNotify` launch code or a callback function.
3. Prime the socket notice system by calling `NetLibSocketOptionSet()`. Among other parameters, you pass in a pointer to an option value whose type is `NetSocketNoticeType`. This structure tells Net Library the following information:
   a. Notice type. Use the constant `netSocketNoticeNotify` to identify a socket notice that uses notifications. This constant is defined in `NoticeTypeEnum`)
   b. Notify type. This is the socket notification constant you defined for your application.
   c. Conditions. These are flags that indicate which socket changes you are interested in. If you are interested in all socket conditions, use the value `0xFFFFFFFF`. See "Socket Notice Trigger Conditions" on page 1468 of *Palm OS Programmer's API Reference*.

4. Handle the notification.
   a. Cast the incoming parameter block as a `SysNotifyNetSocketType`. (For details on this type, see "Socket Notification Specific Data" on page 87 of *Palm OS Programmer's API Reference*).
   b. Find out what condition(s) triggered the notification. Do this by checking the `condition` field of the `SysNotifyNetSocketType` structure.
   c. Perform the appropriate actions.
5. Re-prime the socket notice system in order to receive the next notification. Normally, you do that by calling `NetLibSocketOptionSet()` in your notification handler.

Notice vs. Notification

The terms **notices** and **notifications** refer to related, but distinct things. A **socket notice** is net library functionality that communicates to an application when a socket condition has changed. There are various means by which this communication might take place, not all of which are currently implemented. Currently, net library causes the system to send a **notification** to your application. In future, however, Net Library might be able to post an event to the event queue or to send a message to a specified mailbox ID.

Unsupported Code in NetMgr.h

If you are perusing `NetMgr.h`, you may encounter code that is currently not supported but that may be supported in future versions of Palm OS. For the present, you may ignore such code, including:

- Most of the `NoticeTypeEnum` definition. Only `netSocketNoticeNotify` can currently be used. The other enum constants—such as `netSocketNoticeEvent` and `netSocketNoticeCallback`—are not available in Palm OS Garnet version 5.4.

NOTE: The *socket notice* callback is not supported in Palm OS Garnet version 5.4. Do not confuse it with the *notification* callback, which is implemented as indicated in "Socket Notifications" on page 189.

- The `NetSocketNoticeEventType`, `NetSocketNoticeMailboxType`, `NetSocketNoticeCallbackPtr` definitions. These are not currently supported.
- Most of the `notice` union in `NetSocketNoticeType`. Only the `notify` structure of that union is supported. The `event`, `mailbox`, `callback`, and `wake` structures are not supported.

Related Sections

In order to make socket notifications work, you will need to understand how regular notifications work. For that purpose, consult the following:

- "Notifications"section in Chapter 2, "Application Startup and Stop," of *Palm OS Programmer's Companion*, vol. I
- Chapter 3, "Notifications," in *Palm OS Programmer's API Reference*.
- Chapter 43, "Notification Manager," in *Palm OS Programmer's API Reference*.

Internet Library

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

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

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

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

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

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

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

System Requirements

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

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

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

When the Internet library is present and running, it requires an estimated additional 1 KB of RAM, beyond the net library. More

additional memory is used for the security library, if that is used (when accessing secure sites), and for opening a cache database, if that is used.

Initialization and Setup

Before using the Internet library, an application must call `SysLibFind` to obtain a library reference number, as follows:

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

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

`INetLibOpen` also opens the net library for the application. In addition, the application can tell `INetLibOpen` the type of network service it prefers: wireline or wireless. `INetLibOpen` queries the available network interfaces and attaches the appropriate one(s) for the desired type of service. When the application calls `INetLibClose`, the previous interface configuration is restored. For more information on configurations, see the section "Internet Library Network Configurations" on page 197.

The Internet library gets some of its default behavior from the system preferences database, and some of these preference settings are made by the user via the Wireless preferences panel. The preferences set by this panel include the proxy server to use and a setting that determines whether or not the user is warned when the device ID is sent. Other settings stored in the preferences database come from Internet library network configurations (see "Internet Library Network Configurations" on page 197). All these settings can be queried and/or overridden by each application through the `INetLibSettingGet` and `INetLibSettingSet` calls. However, any changes made by an application are not stored into the system preferences, but only take effect while that `inetH` handle is open.

Accessing Web Pages

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

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

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

If an application requires finer control over the operation, it can replace the call to `INetLibURLOpen` with other lower-level Internet library calls (`INetLibSockOpen`, `INetLibSockSettingSet`, etc.) that are described in the section "Using the Low Level Calls" on page 196.

Asynchronous Operation

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

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

Using `INetLibGetEvent` is the preferred way of performing network I/O since it maximizes battery life and user-interface responsiveness.

With `INetLibGetEvent`, the process of accessing a web page becomes only slightly more complicated. Instead of calling `INetLibSockRead` immediately after `INetLibURLOpen`, the application should instead return to its event loop and wait for the next event. When it gets a network event that says data is ready at the socket, then it should call `INetLibSockRead`.

There are two types of network events that `INetLibGetEvent` can return in addition to the standard user-interface events. The first event is a status change event (`inetSockStatusChangeEvent`). This event indicates that the status of a socket has changed and the application may want to update its user interface. For example, when calling `INetLibURLOpen` to access an HTTP server, the status on the socket goes from "finding host," to "connecting with host," to "waiting for data," to "reading data," etc. The event structure associated with an event of this type contains both the socket handle and the new status so that the application can update the user interface accordingly.

The second type of event that `INetLibGetEvent` can return is a data-ready event (`inetSockReadyEvent`). This event is returned when data is ready at the socket for reading. This event tells the application that it can call `INetLibSockRead` and be assured that it will not block while waiting for data to arrive.

The general flow of an application that uses the Internet library is to open a URL using `INetLibURLOpen`, in response to a user command. Then it repeatedly calls `INetLibGetEvent` to process events from both the user interface and the newly created socket returned by `INetLibURLOpen`. In response to `inetSockStatusChangeEvent` events, the application should update the user interface to show the user the current status, such as finding host, connecting to host, reading data, etc. In response to `inetSockReadyEvent` events, the application should read data from the socket using `INetLibSockRead`. Finally, when all available data has been read (`INetLibSockRead` returns 0 bytes read), the application should close the socket using `INetLibSockClose`.

Finally, the convenience call INetLibSockStatus is provided so that an application can query the status of a socket handle. This call never blocks on network I/O so it is safe to call at any time. It not only returns the current status of the socket but also whether or not it is ready for reading and/or writing. It essentially returns the same information as conveyed via the events inetSockReadyEvent and inetSockStatusChangeEvent. Applications that don't use INetLibGetEvent could repeatedly poll INetLibSockStatus to check for status changes and readiness for I/O, though polling is not recommended.

Using the Low Level Calls

Applications that need finer control than INetLibURLOpen provides can use the lower level calls of the Internet library. These include INetLibSockOpen, INetLibSockConnect, INetLibSockSettingSet, INetLibSockHTTPReqCreate, INetLibSockHTTPAttrGet, INetLibSockHTTPAttrSet, and INetLibSockHTTPReqSend.

A single call to INetLibURLOpen for an HTTP resource is essentially equivalent to this sequence: INetLibSockOpen, INetLibSockConnect, INetLibSockHTTPReqCreate, and INetLibSockHTTPReqSend. These four calls provide the capability for the application to access non-standard ports on the server (if allowed), to modify the default HTTP request headers, and to perform HTTP PUT and POST operations. The only calls here that actually perform network I/O are INetLibSockConnect, which establishes a TCP connection with the remote host, and INetLibSockHTTPReqSend, which sends the HTTP request to the server.

INetLibSockHTTPAttrSet is provided so that the application can add or modify the default HTTP request headers that INetLibSockHTTPReqCreate creates.

INetLibSockSettingSet allows an application finer control over the socket settings.

Finally, the routine IINetLibURLCrack is provided as a convenient utility for breaking a URL into its component parts.

Cache Overview

The Internet library maintains a cache database of documents that have been downloaded. This is an LRU (Least Recently Used) cache; that is, the least recently used items are flushed when the cache fills. Whether or not a retrieved page is cached is determined by a flag (`inetOpenURLFlagKeepInCache`) set in the socket or by `INetLibURLOpen`. Another flag (`inetOpenURLFlagLookInCache`) determines if the Internet library should check the cache first when retrieving a URL.

The same cache database can be used by any application using the Internet library, so that every application can share the same pool of prefetched documents. Alternately, an application can use a different cache database. The cache database to use is specified in the `INetLibOpen` call.

Generally, a cached item is stored in one or more database records in the same format as it arrives from the server.

In the cache used by the Web Clipping Application Viewer application, each record includes a field that contains the "master" URL of the item. This field is set to the URL of the active PQA, so all pages linked from one PQA have the same master URL. This facilitates finding all pages in a hierarchy to build a history list.

The Internet library maintains a list of items in the cache. You can retrieve items in this list, or iterate over the whole list, by calling `INetLibCacheList`. You can retrieve a cached document directly by using `INetLibCacheGetObject`.

You can check if a URL is cached by calling `INetLibURLGetInfo`.

Internet Library Network Configurations

The Internet library supports network configurations. A **configuration** is a specific set of values for several of the Internet library settings (from the `INetSettingEnum` type).

The Internet library keeps a list of available configurations and aliases to them. There are three built-in configurations:

- A wireless configuration that uses the Palm.Net wireless system and the Palm Web Clipping Proxy server.

- A wireline configuration that uses the wireline network configuration specified in the Network preferences panel and the Palm Web Clipping Proxy server.
- A generic configuration that uses the wireline network configuration specified in the Network preferences panel and no proxy server.

You can also define your own configuration by modifying an existing one and saving it under a different name.

The Internet library also defines several **configuration aliases** (see "Configuration Aliases" on page 1892 in the *Palm OS Programmer's API Reference*). An alias is a configuration name that simply points to another configuration. You can specify an alias anywhere in the API you would specify a configuration. This facilitates easy re-assignment of the built-in configurations and eliminates having duplicate settings. You assign an alias by using `INetLibConfigAliasSet` and can retrieve an alias by using `INetLibConfigAliasGet`.

For example, to change the default configuration used by the Internet library for a particular kind of connection, you can set up the appropriate values for a connection, save the configuration, and then set the Internet library's default alias configuration to point to your custom configuration. When an application specifies which configuration it wants to use, if it specifies the alias, it will use the custom settings.

If you use configurations at all, it will probably be to specify a specific configuration when opening the Internet library via `INetLibOpen`. The Internet library also contains an API to allow you to manipulate configurations in your application, but doing so is rare. You can list the available configurations (`INetLibConfigList`), get a configuration index (`INetLibConfigIndexFromName`), select (`INetLibConfigMakeActive`) the Internet library network configuration you would prefer to use (wireless, wireline, etc.), rename existing configurations (`INetLibConfigRename`), and delete configurations (`INetLibConfigDelete`).

The configuration functions are provided primarily for use by Preferences panels while editing and saving configurations. The general procedure is to make the configuration active that you want

to edit, set the settings appropriately, then save the configuration using `INetLibConfigSaveAs`. Note that configuration changes are not saved after the Internet library is closed, unless you call `INetLibConfigSaveAs`.

Summary of Network Communication

Net Library Functions

Library Open and Close

NetLibClose
NetLibConnectionRefresh
NetLibFinishCloseWait
NetLibOpen
NetLibOpenCount

Socket Creation and Deletion

NetLibSocketClose
NetLibSocketOpen

Socket Options

NetLibSocketOptionGet
NetLibSocketOptionSet

Socket Connections

NetLibSocketAccept
NetLibSocketAddr
NetLibSocketBind
NetLibSocketConnect
NetLibSocketListen
NetLibSocketShutdown

Send and Receive Routines

NetLibDmReceive
NetLibReceive
NetLibReceivePB
NetLibSend
NetLibSendPB

Utilities

NetHToNL
NetHToNS
NetLibAddrAToIN
NetLibAddrINToA
NetLibGetHostByAddr
NetLibGetHostByName
NetLibGetMailExchangeByName
NetLibGetServByName
NetLibMaster
NetLibSelect
NetLibTracePrintF
NetLibTracePutS
NetNToHL
NetNToHS

Net Library Functions

Setup

NetLibIFAttach
NetLibIFDetach
NetLibIFDown
NetLibIFGet
NetLibIFSettingGet
NetLibIFSettingSet
NetLibIFUp
NetLibSettingGet
NetLibSettingSet

Network Utilities

NetUReadN
NetUTCPOpen
NetUWriteN

Internet Library Functions

Library Open and Close

INetLibClose
INetLibOpen

Settings

INetLibSettingGet
INetLibSettingSet

Event Management

INetLibGetEvent

High-Level Socket Calls

INetLibSockClose
INetLibSockRead
INetLibURLOpen

Low-Level Socket Calls

INetLibSockConnect
INetLibSockOpen
INetLibSockSettingGet
INetLibSockSettingSet
INetLibSockStatus

Internet Library Functions	
HTTP Interface	
INetLibSockHTTPAttrGet	INetLibSockHTTPReqCreate
INetLibSockHTTPAttrSet	INetLibSockHTTPReqSend
Utilities	
INetLibCheckAntennaState	INetLibURLsAdd
IINetLibURLCrack	INetLibWiCmd
INetLibURLGetInfo	
Cache Interface	
INetLibCacheGetObject	INetLibCacheList
Configuration	
INetLibConfigAliasGet	INetLibConfigList
INetLibConfigAliasSet	INetLibConfigMakeActive
INetLibConfigDelete	INetLibConfigRename
INetLibConfigIndexFromName	INetLibConfigSaveAs

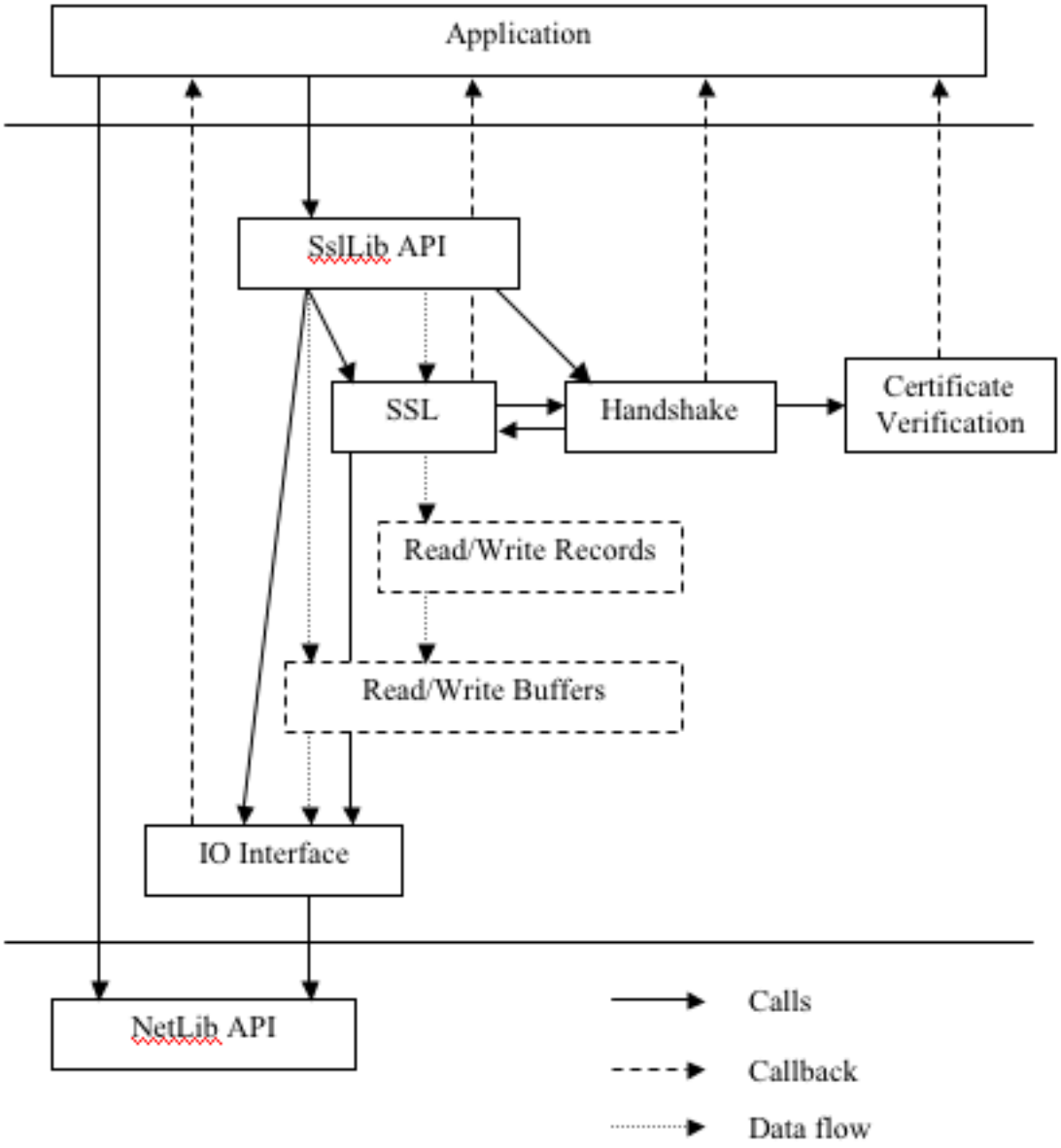
8

Secure Sockets Layer (SSL)

SSL Library Architecture

The SslLib library is an implementation of the SSL protocol for use under Palm OS. The API implements an interface that can be used to perform SSL and non-SSL network I/O. Figure 8.1 is intended to help show the relationship between the different components of SslLib and how they interact with the user's application.

Figure 8.1 SSL Library architecture



In this diagram, the following items are labeled.

- Application – This is the user's application that will be using the SslLib library to secure its network connections.

- NetLib API – This is the Palm OS Net Library API. This box represents calls into that library.
- SslLib API – This is the Palm OS SslLib API. This box represents calls into that library via it's public interfaces.
- SSL – The SSL protocol which is under the SslLib API. This represents the code that performs the SSL encapsulation of the application's data.
- Handshake – The SSL protocol, during the initial connection, performs a message exchange with the remote SSL server. This box represents the part of the SSL protocol that implements this exchange.
- Certificate Verification – As part of the SSL handshake, certificates need to be verified. This box represents the logic that performs the certificate verification.
- Read/Write Records – The SSL protocol sends and receives SSL records. This box represents the data structures used to keep track of the last record read and the next record to be written.
- Read/Write Buffers – SslLib buffers incoming and outgoing data. This box represents the data structures used to hold this data.
- IO Interface – This is the code that sends data from a write buffer to the network, or the code that reads data from the network and puts it in the read buffer.

The application will call NetLib directly to configure and establish a network connection (a `NetSocketRef`). Once the `NetSocketRef` has been configured, it is passed into SslLib by associating the socket with an SslContext (`SslContextSet_Socket()`). When a read or write call is made to SslLib, depending on the mode of operation the SslContext is configured to operate in (`SslContextSet_Mode()`), either the data bytes will be directly sent, or they will under go SSL processing to encrypt and MAC the data. The diagram shows how the data bytes always go via the SslContext's read/write buffers. These buffers are used to store bytes waiting to be sent to NetLib and any extra bytes read from NetLib that have not yet been processed. The SSL protocol initially enters a handshake state, where the security parameters to use to

encrypt and MAC the application's data bytes are determined. As part of this process, some certificates need to be verified.

The callback arrows indicate where the application can register to receive notification of activity in those relevant subsystems. The IO Interface can return via the info callback (`SslContextSet_InfoCallback()`) information about the calls to NetLib. The SSL box callback indicates the notification of SSL Protocol Alerts that are received (via the info callback). The handshake callback arrow indicates the calls to the info callback when-ever the SSL handshake protocol changes state (`SslContextGet_HsState()`). The information returned from these three access points is mostly of interest for debugging reasons. The final callback, the Verify callback (`SslContextSet_VerifyCallback()`) is often used to modify the policies regarding certificates.

Attributes

The SslLib library uses two main structures to hold information: the SslLib structure and the SslContext. The SslContext is used to hold all information associated with a single SSL network connection. It contains various flags that govern how the SSL protocol will operate, and also contains a read buffer and a write buffer where SSL protocol packets are assembled and disassembled. As part of the SSL handshake, various structures are created . These include the security parameters associated with the particular connection and the certificate from the SSL server that is on the other end of the network connection. Quite a large number of these attributes can be retrieved for debugging and informational reasons. Others can be set by the application to modify the behavior of the SSL protocol. The SslLib can be though of as a template for many of these options. The SslLib can have many of its attributes set, and then when an SslContext is created using the SslLib, these attributes are inherited directly. These values are copied into the SslContext, so subsequent changes to the SslLib's attributes will not modify any existing SslContext's.

Attribututes can be broken into two main classes; integer values, and pointer values. The integer values are numbers that can be set or retrieved via the `SslLibGetLong()`, `SslLibSetLong()`,

SslContextGetLong() and SslContextSetLong() calls. These functions are not normally called directly; instead, applications typically employ those macros declared in SslLibMac.h. The pointer-based attributes are similarly set or retrieved using macros; those macros evaluate to calls to SslLibGetPtr(), SslLibSetPtr(), SslContextGetPtr() and SslContextSetPtr(). Whenever an attribute is passed in via a pointer, the type of the pointer is defined by the attribute being used. The object that the pointer is pointing to is always copied into the SslLib or SslContext, so the data element that is passed in does not need to be preserved. There are some exceptions to this rule. Pointer-based attributes that are retrived from an SslLib or an SslContext will always be references to objects held inside the SslLib or SslContext. If the application wishes these values to be preserved, it should copy them into local storage.

The attributes can be grouped into several categories: some will always be used, some will be regularly used and will profoundly modify the behavior of some of SslLib core functions. Some are to help debugging, and some are used to configure more subtle protocol specific internal configuration parameters. The following sections detail each attribute, grouping them by these categories.

Always-Used Attributes

AutoFlush

This attribute affects the behavior of SslSend() and SslWrite(). When enabled, these functions will attempt to immediately send the supplied data bytes to the network. If the application performs 200 one-byte SslWrite() calls, this will generate 200 network packets, each about 80 bytes in size (assuming TCP over Ethernet), for a total of 16,000 bytes. If this data was buffered, it would have been sent in a single packet of about 280 bytes. When buffering, there is an additional advantage in that the write calls will not generate errors unless the buffer fills. This can be used to simplify routines that package data for transmission. It is very important to remember to use the SslFlush() call when AutoFlush is disabled. SslFlush() will write any data that is in the SslContext's write buffer. If an application does not flush this data to the network, the server application at the other end will not reply, so the application

will probably deadlock, awaiting a response from the server that will never come because the client has not yet sent its data to the server.

The internal logic in SslLib is as follows:

```
Int32 SslWrite(...) {
write_data_to_output_buffer(...);
if (ssl->autoflush)
   flush_output_buffer(...);
}
```

Auto-flush is enabled by default.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_AutoFlush()`

SslLib Write: `SslLibSet_AutoFlush()`

SslContext Read: `SslContextGet_AutoFlush()`

SslContext Write: `SslContextSet_AutoFlush()`

CipherSuites

This attribute is used to specify the SSL cipher suites that the SSL protocol will attempt to use. The pointer refers to an array of UInt8 bytes that specify the SSLv3 cipher suite values, in the order desired, to be sent to the SSL Server. The first two bytes, in network byte order, contain the number of bytes that follow. Following these two bytes are values selected from "Cipher Suites" on page 2207. Note that each `sslCs_RSA...` #define is two bytes long.

This value is inherited from the SslLib when an SslContext is created. Setting `CipherSuites` with a value of `NULL` will restore the use of the default cipher suite list. The default cipher suites list (including the size bytes) is:

```
{0x00, 0x08, sslCs_RSA_RC4_128_MD5, sslCs_RSA_RC4_128_SHA1,
sslCs_RSA_RC4_56_SHA1, sslCs_RSA_RC4_40_MD5}
```

To ensure that an application only uses strong encryption, it should make the following call:

```
static UInt8 cipherSuites[]={
   0x00,0x04,      /* Number of following bytes
                      (each value is two bytes) */
   sslCs_RSA_RC4_128_MD5,
   sslCs_RSA_RC4_128_SHA1
};

SslLibSet_CipherSuites(theLibRef, lib, cipherSuites);
/* To change the cipher suite for an existing SslContext */
SslContextSet_CipherSuites(theLibRef, lib, cipherSuites);
```

Use the following macros to read and write this attribute:

SslLib Read: SslLibGet_CipherSuites()

SslLib Write: SslLibSet_CipherSuites()

SslContext Read: SslContextGet_CipherSuites()

SslContext Write: SslContextSet_CipherSuites()

Error

When a fatal error occurs while using an SslContext, the internal error attribute is set to the error value. The application can retrieve this error value and change it if it desires. Normally an application will not change this value, but once the error attribute is set, the SslLib network APIs will continue to return this error (unless the error is a non-fatal error) until either an SSL Reset is performed on the SslContext or the error is cleared, at which point the Error attribute will be zero. A SSL Reset can be performed with SslContextSet_Mode():

```
SslContextSet_Mode(theLibRef, ssl,SslContextGet_Mode(ssl));
```

Note that `SslErrIo` is a non-fatal error.

Use the following macros to read and write this attribute:

SslContext Read: SslContextGet_Error()

SslContext Write: SslContextSet_Error()

Mode

This attribute is used to turn the SSL protocol on or off. It applies to the SslContext, and when set to `sslModeClear`, causes the SSL protocol to be bypassed. This can be useful for an application since it can be written to use the SslLib API, and still perform normal non-SSL data transfers via that API. This will let an application take advantage of the buffering provided in an SslContext so that it can perform buffer reads and buffer writes to the network. When an SslContext has its Mode attribute changed, an SSL Reset occurs. This clears any SSL state information and sets the SslContext back to a state ready to establish a new SSL connection. The SSL Session information is not cleared. This means that an application can start in `sslModeClear`, and then switch to `sslModeSslClient`. If the application switches back to `sslModeClear`, and again over to `sslModeSslClient`, a new handshake will be performed.

The `SslModeSsl` is a subset value of `sslModeSslClient`. In a future release of SslLib, the server side of the SSL protocol may be supported in which case `sslModeSslServer` would be added.

An application can do the following in order to determine if the SSL protocol is being used:

```
If (SslContextGet_Mode(theLibRef, ssl) & sslModeSsl)
   /* SSL protocol enabled */
else
   /* Using cleartext */
```

A comparison with `sslModeSslClient` could be used to determine if the client or server side of the protocol is being used for that particular SslContext.

The `sslModeFlush` flag is special. When supplied to `SslContextSet_Mode()`, it causes any data in the internal data buffers to be cleared. This is normally required when reusing an SslContext for a new connection. If an application is using an SslContext for cleartext, and then wants to enable SSL on the same connection, this flag should not be used.

By default, the `mode` attribute is set to `sslModeSslClient`.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_Mode()`

SslLib Write: `SslLibSet_Mode()`

SslContext Read: `SslContextGet_Mode()`

SslContext Write: `SslContextSet_Mode()`

"Mode Attribute Values" on page 2203 lists the values that this attribute can have.

RbufSize

The read and write buffers are used in the SslContext to buffer incoming and outgoing data. When these values are set for an SslLib, SslContexts that are created against the SslLib will inherit the SslLib's values.

The write buffer size is the maximum number of bytes that can be buffered before a network write operation is performed. The number of application data bytes that can be buffered is less than this number when in SSL mode—approximately 30 bytes less due to SSL record overheads. If the application writes a 16 kb block of data and the write buffer is about 1 kb in size, about 16 network packets will be sent.

The read buffer is a little different from the write buffer in that it may be automatically increased is size depending on other configuration information. The SSLv3 protocol supports SSL Records up to 16 Kbytes in size. Depending on the encryption cipher being used, the protocol may need to decrypt the record in a single operation. In this case the read buffer will be increased in size to buffer the incoming record. See the ReadStreaming option for advanced usage of the read buffer to decrease latency of data availability for the application.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_RbufSize()`

SslLib Write: `SslLibSet_RbufSize()`

SslContext Read: `SslContextGet_RbufSize()`

SslContext Write: `SslContextSet_RbufSize()`

The read buffer's default size is 2048 bytes. You can change the size of the read buffer to any value from 0 to 16384 bytes.

Socket

This call is used to specify the NetLib socket that the SslContext should use to perform its network I/O operations. An SslContext is unable to perform any network operation until the application creates and supplies a suitable `NetSocketRef`. The SslLib library does not perform any NetLib operations on the supplied `NetSocketRef` other than `NetLibSend()` and `NetLibReceive()`. All socket creation and shutdown operations must be performed by the application.

Use the following macros to read and write this attribute:

SslContext Read: `SslContextGet_Socket()`

SslContext Write: `SslContextSet_Socket()`

VerifyCallback

The callback function is used to assist with certificate verification. See `SslCallbackFunc()` (documented on page 2241) for more details on the `SslCallback` structure and its usages, specifically when used to assist in certificate verification.

When a new Verify callback is specified, the application passes in a pointer to an `SslCallback` structure. This structure is copied into an internal `SslCallback` structure. The callback and data fields are preserved. When the Verify callback structure is copied into an SslLib, or copied into an SslContext, the callback function is called with a command of `sslCmdNew`. When the parent SslLib or SslContext is destroyed, a `sslCmdFree` command is issued.. If a SSL Reset is performed, a `sslCmdReset` command is issued. Outside of these situations, the callback will be called during the certificate verification process as outlined in the documentation for the `SslCallbackFunc()` function.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_VerifyCallback()`

SslLib Write: `SslLibSet_VerifyCallback()`

SslContext Read: `SslContextGet_VerifyCallback()`

SslContext Write: `SslContextSet_VerifyCallback()`

WbufSize

The read and write buffers are used in the SslContext to buffer incoming and outgoing data. When these values are set for an SslLib, SslContexts that are created against the SslLib will inherit the SslLib's values.

The write buffer size is the maximum number of bytes that can be buffered before a network write operation is performed. The number of application data bytes that can be buffered is less than this number when in SSL mode—approximately 30 bytes less due to SSL record overheads. If the application writes a 16 kb block of data and the write buffer is about 1 kb in size, about 16 network packets will be sent.

The read buffer is a little different from the write buffer in that it may be automatically increased is size depending on other configuration information. The SSLv3 protocol supports SSL Records up to 16 Kbytes in size. Depending on the encryption cipher being used, the protocol may need to decrypt the record in a single operation. In this case the read buffer will be increased in size to buffer the incoming record. See the ReadStreaming option for advanced usage of the read buffer to decrease latency of data availability for the application.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_WbufSize()`

SslLib Write: `SslLibSet_WbufSize()`

SslContext Read: `SslContextGet_WbufSize()`

SslContext Write: `SslContextSet_WbufSize()`

The write buffer's default size is 1024 bytes. You can change the size of the write buffer to any value from 0 to 16384 bytes.

Debugging and Informational Attributes

AppInt32

The `AppInt32` attribute is a 32-bit integer value that the application can read or write as it sees fit. It is present so the application can attach an arbitrary value to an SslLib or a SslContext. `SslLibDestroy()` and `SslContextDestroy()` do not modify

this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_AppInt32()`

SslLib Write: `SslLibGet_AppInt32()`

SslContext Read: `SslContextGet_AppInt32()`

SslContext Write: `SslContextSet_AppInt32()`

AppPtr

The `AppPtr` attribute is a pointer value that the application can read or write as it sees fit. It is present so the application can attach an arbitrary pointer to an SslLib or a SslContext. `SslLibDestroy()` and `SslContextDestroy()` do not modify this attribute, so if the data pointed to by this attribute needs to be disposed of, the application must do this itself. The value of the `AppPtr` attribute is `NULL` by default.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_AppPtr()`

SslLib Write: `SslLibGet_AppPtr()`

SslContext Read: `SslContextGet_AppPtr()`

SslContext Write: `SslContextSet_AppPtr()`

CipherSuite

Pass a pointer to a uint8_t pointer in order to retrieve this attribute. The returned value points to two bytes which identify the cipher suite being used by the current connection. Possible values for the cipher suites are:

`0x00, 0x00`
No cipher suite

`0x00, 0x04`
sslCs_RSA_RC4_128_MD5

`0x00, 0x05`
sslCs_RSA_RC4_128_SHA1

`0x00, 0x64`
sslCs_RSA_RC4_56_SHA1

`0x00, 0x03`
sslCs_RSA_RC4_40_MD5

Also see the `CipherSuites` attribute for instructions on specifying which cipher suites SslLib should advertise as available for use when it initially connects to the SSL server.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_CipherSuite()`

CipherSuiteInfo

This function differs from most others in that the application passes in a structure to be populated from the SslContext. Normally the SslContext returns a pointer to an internal data structure. This call returns the information relevant to the current cipher suite.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_CipherSuiteInfo()`

ClientCertRequest

The SSL protocol allows the SSL server to request a certificate from the client. This attribute will be set if the server requested a client certificate.

SslContext Read: `SslContextGet_ClientCertRequest()`

Compat

Turn on compatibility with incorrect SSL protocol implementations. These bugs will not normally be encountered while using the SSL protocol, but if desired, it is worth enabling the compatibility in case old buggy servers are being accessed.

See "Compatibility Flags" on page 2204 for the defined constants that correspond to the compatibility flags. By default, none of these compatibility flags are set.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_Compat()`

SslLib Write: `SslLibSet_Compat()`

SslContext Read: `SslContextGet_Compat()`

SslContext Write: `SslContextSet_Compat()`

HsState

This attribute is the state that the SSL protocol is currently in. Possible values are defined under "SSL Protocol States" on page 2210. This information is generally only of use during debugging. See the SSL protocol specification for clarification on what the values mean.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_HsState()`

InfoCallback

This callback is called when various situations occur during the usage of an SslContext. It is primarily intended for debugging and feedback purposes. If the callback returns a non-zero value, this error will be returned back out to the SslLib API. The callback will be called with a command argument of `sslCmdInfo`.

A single Info callback is used for notification of four different types of events. The `InfoInterest` attribute controls which of these events will invoke the Info callback.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_InfoCallback()`

SslLib Write: `SslLibSet_InfoCallback()`

SslContext Read: `SslContextGet_InfoCallback()`

SslContext Write: `SslContextSet_InfoCallback()`

InfoInterest

This value is used to specify the events for which the `InfoCallback` will be called. The value is the logical ORing of the `sslFlgInfo...` values listed under "InfoInterest Values" on page 2209. The `sslFlgInfoIo` value controls the notification of the four different Info Callbacks. By default, the `InfoInterest` attribute value is zero.

Use the following macros to read and write this attribute:

SslLib Read: SslLibGet_InfoInterest()

SslLib Write: SslLibSet_InfoInterest()

SslContext Read: SslContextGet_InfoInterest()

SslContext Write: SslContextSet_InfoInterest()

ioFlags

Since we will normally be using TCP connections with SSL, this attribute is more included for completeness rather than utility. Read about this flags value in the NetLibSend() and NetLibReceive() documentation.

NOTE: The `netIOFlagOutOfBand` and `netIOFlagPeek` values are not valid and will be silently removed.

Use the following macros to read and write this attribute:

SslContext Read: SslContextGet_IoFlags()

SslContext Write: SslContextSet_IoFlags()

ioStruct

The SslContext's internal SslSocket structure.

Use the following macros to read and write this attribute:

SslContext Read: SslContextGet_IoStruct()

SslContext Write: SslContextSet_IoStruct()

ioTimeout

The SslContext contains internally a default timeout value to pass to NetLib calls. When a call is made into the SslLib API which does not specify a timeout, this internal value is used. If the API call has a timeout value, it overrides this internal value.

By default, the SslContext's internal timeout value is 10 seconds.

Use the following macros to read and write this attribute:

SslContext Read: SslContextGet_IoTimeout()

SslContext Write: SslContextSet_IoTimeout()

LastAlert

The alert values are received from the server and are either fatal or non-fatal. Non-fatal alerts have a value of the form `0x01XX`, while fatal alerts have the form `0x02XX`. SslLib will fail on fatal alerts and continue on non-fatal alerts. See "SSL Server Alerts" on page 2212 for the complete list of alerts.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_LastAlert()`

LastApi

This attribute is the last SslLib API call that was made. `sslLastApiRead` is set if `SslRead()`, `SslPeek()` or `SslReceive()` was called. `sslLastApiWrite` is set if `SslWrite()` or `SslSend()` was called. This attribute can be useful in event driven programs.

See "LastApi Attribute Values" on page 2209 for the set of values that this attribute can have.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_LastApi()`

LastIo

This function can be called to determine the last network operation. If SslLib, while performing a network operation, encounters an error, the error value will be returned to the application. Since most of the SslLib API I/O functions can cause an SSL handshake to be performed, it is often not possible to know if the reason that a `SslSend()` returned `netErrWouldBlock` is because the send operation failed or a receive operation failed (because a SSL Handshake was being performed). This attribute allows the application to determine which I/O operation was being called if an network error is returned. If the application is using `NetLibSelect()`, this attribute is very important. This attribute returns the last network operation performed. This means that `sslLastIoNone` will only be returned if the SslContext has not performed any I/O operations since its last reset.

See "LastIO Attribute Values" on page 2210 for the set of values that this attribute can have.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_LastIo()`

PeerCert

If the certificate supplied by the other end of the SSL connection is available, the certificate is returned. The returned pointer references a data structure which is internal to the SslContext and will be disposed of by the SslContext. If a new connection is established with the SslContext, previously returned `PeerCert` pointers will become invalid. If the application wishes to preserve the certificate for an extended period, it should make a local copy.

The `SslExtendedItems` structure is described in "The SslExtendedItems Structure" on page 2247.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_PeerCert()`

PeerCommonName

This call will return a pointer to an `SslExtendedItems` structure which contains the common name for the server's certificate. If using SSL in an https context, the client application should ensure that the common name contained in the servers certificate matches the URL requested. This function facilitates this functionality. The pointer returned refers to a data structure from inside the peer certificate; the offset field in the returned value is relative to the value returned by `SslContextGet_PeerCert()`.

The following code shows how to access the common name from within the `SslExtendedItems` structure (see "The SslExtendedItems Structure" on page 2247 for a description of this structure):

```
SslExtendedItems *cert;
SslExtendedItem *commonName;
uint16_t length;
uint8_t *bytes;

SslContextGet_PeerCert(theLibRef, ssl, &cert);
if (cert == NULL) goto err;
SslContextGet_PeerCommonName(theLibRef, ssl,&commonName);
length=commonName->len;
```

```
bytes=((Int8 *)cert)+commonName->offset;
// bytes now points to the common name, and length contains
// the length of the common name string.
```

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_PeerCommonName()`

ProtocolVersion

The version of the SSL protocol to use. There are 3 versions of the SSL protocol. SSLv2 which is deprecated due to security flaws, SSLv3 which is the most widely deployed, and TLSv1, or SSLv3.1. SslLib implements only SSLv3 at this point in time, so modification of this value is not a good idea. By default this attribute is set to `sslVersionSSLv3`.

See "Protocol Versions" on page 2204 for the defined constants that correspond to the SSL protocol versions.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_ProtocolVersion()`

SslLib Write: `SslLibSet_ProtocolVersion()`

SslContext Read: `SslContextGet_ProtocolVersion()`

SslContext Write: `SslContextSet_ProtocolVersion()`

SessionReused

The SSL protocol has the capability to re-establish a secure connection with a truncated handshake. This can be performed if both end-points have communicated previously and share an SSL Session. An SSL Session is a collection of security attributes that are normally determined as part of the SSL Handshake. If the SSL handshake was able to perform a truncated handshake by re-using the SSL session values in the SslContext, this attribute will have a non-zero value. See the `SslSession` attribute.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_SessionReused()`

SslSession

This attribute is either the `SslSession` currently being used, or the `SslSession` for this SslContext to use to establish its next connection. The `SslSession` holds all the security information associated with a particular SSL connection. If an SslContext is configured to use the same `SslSession` as a previous connection to the same server, the SSL protocol can perform a truncated handshake that involves less network traffic and a smaller CPU load on the server.

If a new connection is performed on the SslContext, or another call is made to retrieve the `SslSession`, any previously returned `SslSession` pointers will become invalid. If the program wants to keep the `SslSession` for an extended period, it should make a local copy.

Use the following macros to read and write this attribute:

SslContext Read: `SslContextGet_SslSession()`

SslContext Write: `SslContextSet_SslSession()`

SslVerify

During certificate verification, an `SslVerify` structure (see "The SslVerify Structure" on page 2244 for a definition of this structure) is used in the SslContext to preserve state. The application can retrieve this structure to help it resolve any problems that SslLib may have encounterd during certificate verifcation.

When a certificate is being verified and a verification error occurs, if the application has registered a `VerifyCallback` the callback will be called with an *`argv`* value pointing to the `SslVerify` structure. If there is no callback, or if the callback still reports an error, SslLib will return the error back to the application. The application can then decide to look at the certificate verification state (by calling `SslContextGet_SslVerify()`) and, if it determines that the error is not fatal, clear the error and re-call the SslLib API that just returned the error.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_SslVerify()`

Advanced Protocol Attributes

The following attributes are not normally used. They give access to various internal aspects of the SSL protocol and or SslLib.

BufferedReuse

The SSL protocol is capable of performing a truncated handshake if both endpoints share an SslSession from a previous connection. The truncated handshake finishes with SslLib sending a SSL handshake message to the SSL server. If the application then sends a message, say a URL, under some network stacks a significant delay can be incurred as the TCP protocol waits for a response from the SSL server's TCP stack. This option, if enabled, will buffer the last message in an SslSession-reused handshake instead of sending it over the network. The application *must* send data before it tries to read any, or more to the point, it must make sure the data is flushed, ether by having `AutoFlush` enabled, or by explicitly calling `SslFlush()`. There are security implications in that a "man in the middle" attack would only be detected once the first data bytes are read from the server. This would mean an attacker could have read all the bytes in the first message sent to the server. For this reason this option should not be normally used. By default, this attribute is set to zero, disabling the buffered reuse option.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_BufferedReuse()`

SslLib Write: `SslLibSet_BufferedReuse()`

SslContext Read: `SslContextGet_BufferedReuse()`

SslContext Write: `SslContextSet_BufferedReuse()`

DontSendShutdown

During the SSL protocol shutdown sequence, the two SSL endpoints swap shutdown messages. This can incur a time penalty since extra messages need to be exchanged over the network. If DontSendShutdown is set, then a `SslClose()` will not send a shutdown message to the server. If `DontWaitForShutdown` is set, then SslLib will not wait for a shutdown message in `SslClose()`. To perform a correct SSL shutdown, these options should not be on.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_DontSendShutdown()`

SslLib Write: `SslLibSet_DontSendShutdown()`

SslContext Read: `SslContextGet_DontSendShutdown()`

SslContext Write: `SslContextSet_DontSendShutdown()`

DontWaitForShutdown

During the SSL protocol shutdown sequence, the two SSL endpoints swap shutdown messages. This can incur a time penalty since extra messages need to be exchanged over the network. If DontSendShutdown is set, then a `SslClose()` will not send a shutdown message to the server. If DontWaitForShutdown is set, then SslLib will not wait for a shutdown message in `SslClose()`. To perform a correct SSL shutdown, these options should not be on.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_DontWaitForShutdown()`

SslLib Write: `SslLibSet_DontWaitForShutdown()`

SslContext Read: `SslContextGet_DontWaitForShutdown()`

SslContext Write: `SslContextSet_DontWaitForShutdown()`

ReadBufPending

This attribute is the number of data bytes that are currently buffered for reading from the SslContext. This number of bytes also include bytes used for encoding SSL records. This attribute is mostly for debugging purposes.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_ReadBufPending()`

ReadOutstanding

This attribute is the number of bytes in the current record that have not been read from the network. If this value is 0, then all bytes that have been read from the network have had their MAC checked. If it is not 0, then the last bytes that have been read have not had their MAC value checked yet. See the Streaming and ReadStreaming attributes to see why this value can be useful.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_ReadOutstanding()`

ReadRecPending

Unlike ReadBufPending, this attribute is the number of application data bytes that are buffered, awaiting the application to read. If this number of bytes is 0, then the next `SslRead()` or `SslReceive()` will cause a `NetLibReceive()` call.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_ReadRecPending()`

ReadStreaming

The SSL protocol exchanges records between its endpoints. A SSL record can contain up to 16K bytes of data. This record is encrypted and protected with a cryptographic checksum call a MAC. If the network is very low speed (300 baud modem), it can be desirable to allow data to be returned to the application from the SSL connection before the full record has been downloaded. If the ReadStreaming flag is on, this protocol modification is enabled. There are security implications behind this modification. The record MAC is used to ensure that the data bytes downloaded have not been modified. If the application has been sent a 16K record, and it is read-streaming and only processing 300 bytes at a time, those bytes could be corrupted or forged without SslLib notifiying the application of this error until the last bytes of the 16K of data is sent. This attribute can be useful if the application is displaying or saving the downloaded data and does not want to be stuck in a `SslRead()` for an extended period of time. Remember that if read-streaming is turned on, the data may be invalid and you will only receive notification when the last bytes are read from the record.

This attribute has a default value of zero. A non-zero value indicates that the SSL protocol should be modified.

Use the following macros to read and write this attribute:

SslLib Read: `SslLibGet_ReadStreaming()`

SslLib Write: `SslLibSet_ReadStreaming()`

SslContext Read: `SslContextGet_ReadStreaming()`

SslContext Write: `SslContextSet_ReadStreaming()`

Streaming

This attribute returns 1 if the current SslContext is doing read-streaming. Just because the ReadStreaming attribute is set, that does not mean the SslLib will use read-streaming.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_Streaming()`

WriteBufPending

This attribute returns the number of bytes in the SslContext's write buffer waiting to be sent to the remote machine. This value will normally be zero unless AutoFlush is disabled and/or non-blocking I/O is being used. A `SslFlush()` will attempt to write these bytes to the network.

Use the following macro to read this attribute:

SslContext Read: `SslContextGet_WriteBufPending()`

Sample Code

The following is a simple example that demonstrates the usage of some of the SslLib libraries functions by way of listing subroutines that could be used by an application utilizing the SSL protocol.

```
#include <SslLib.h>

/* We will perform the initial SslLib setup.  The SslLib would be
 * created with reasonable default values, which can be modified.
 * Quite a few of these values are
 * 'inherited' during SslContext creation.
 */
```

```
Err InitaliseSSL(libRet)
SslLib **libRet;
   {
   SslLib *lib;
   Int16 err;
   lnt32 lvar;

   /* Create the structure */
   if ((err=SslLibCreate(theLibRef, &lib)) != 0)
      return(err);

   /* Make sure we use the SSL protocol by default and increase
 * the write buffer size */
   SslLibSet_Mode(theLibRef, lib,sslModeSslClient);
   SslLibSet_WbufSize(theLibRef, lib,1024*8);

   *libRet=lib;
   return(0);
   }

/* This function would be called to create an sslContext from an open socket
 */
Err CreateSslConnection(SslLib *lib,NetSocketRef socket,SslContext **sslRet)
   {
   SslContext ssl=NULL;
   Int16 err;

   /* We first create a new SslContext.
    * This context will inherit various internal configuration
    * details from the SslLib.
    */
   if ((err=SslContextCreate(theLibRef, lib,&ssl)) != 0)
      return(err);

   /* We now specify the socket to use for IO */
   SslContextSet_Socket(theLibRef, ssl,socket);

   /* At this point we could specify the SSL Mode of operation to use,
    * but since we already specified this for the SslLib, we do not
    * need to do it again.
    */
   //SslContextSet_Mode(theLibRef, ssl,sslModeSslClient);

   /* For this example, we will perform the SSL handshake now. */
   err=SslOpen(theLibRef, ssl,0,30*SysTicksPerSecond());

   *sslRet=ssl;
   return(Err);
```

```
   }

/* Shutdown the SSL protocol and return the socket */
Err CloseSslConnection(SslContext ssl,NetSocketRef *retSock)
   {
   NetSocketRef socket;
   SslSession *sslSession;
   MemHandle ssHandle;

   /* We will perform a full SSL protocol shutdown.  We could have
    * set a flag against the SslContext earlier, or even against the
    * SslLib to specify the shutdown behavior.
    */
   err=SslClose(theLibRef, ssl,0,10*SysTicksPerSecond());

   /* We have now closed the SSL protocol, but the socket is still open
    * and the SslContext still has SslSession information that
    * other connections to the same site may want to use.
    * In this case we ask for a reference to the SslSession.
    * Since this structure is variable in size, once we have a
    * reference to it, we can duplicate it if we want to keep it.
    */
   SslContextGet_SslSession(theLibRef, ssl,&sslSession);

   ssHandle=MemHandleNew(sslSession->length);
   Memcpy(MemHandleLock(ssHandle),sslSession,sslSession->length)
   MemHandleUnlock(ssHandle);
   /* We now have a handle to a SslSession that we can store
    * for later use with a new connection.  We would need to store
    * this SslSession with the relevant hostname/url information
    * to ensure we try to reuse it on only the relevant SSL server.
    * This mapping is application/protocol-specific (urls for https).
    */

   /* We will return the Socket */
   *retSock=SslContextGet_Socket(theLibRef, ssl);

   /* Throw away the SslContext structure */
   SslContextDestroy(theLibRef, ssl);
   return(0);
   }

Err HTTPS_call(SslContext ssl,char *send,Uint16 len,char *reply,Uint32 *outlen)
   {

   Err err;
   Int16 ret;
```

```
   /* We will send the 'send' data, and then wait for the response */
   ret=SslSend(theLibRef, ssl,send,len,0,NULL,0,60*SysTicksPerSecond(),&err)
   if (ret <= 0) goto end;

   ret=SslReceive(theLibRef, ssl, reply, *outlen, 0, NULL, 0,
      60*SysTicksPerSecond(), &err);
   if (ret < 0) goto end;
   *outlen=ret;
end:
   return(err);
   }
```

9

Internet and Messaging Applications

This chapter provides an overview of wireless Internet access with the Palm OS® and describes the programmatic interfaces to the Web Clipping Application Viewer and email applications.

NOTE: You cannot use the features described in this chapter with a version of the Palm OS earlier than version 3.2.

WARNING! The Web Clipping Application Viewer is only present on a limited set of handhelds produced by Palm, Inc. Applications can programmatically detect the presence of the Web Clipping Application Viewer; see "System Version Checking" on page 236 for details.

This chapter begins with a brief discussion of Internet access on Palm Powered™ handhelds and then provides a brief overview of how web clipping applications work in the following sections:

For more information about web clipping applications, displaying HTML pages on Palm Powered handhelds, and the Palm.Net system, see the *Web Clipping Developer's Guide*.

This chapter also describes how to programmatically access the Web Clipping Application Viewer (the **Viewer**) and IMessenger applications in the following sections:

- Sending Email Messages
- Using Wireless Capabilities in Your Applications

For more information about programmatic access to the Internet, see Chapter 79, "Internet Library," in the *Palm OS Programmer's API Reference*.

Internet Access on Palm Powered Handhelds

Starting with version 3.2, the Palm OS added support for wireless Internet access and messaging. Version 3.5 of the Palm OS extended those capabilities, and along with the Mobile Internet Kit, extended wireless access capabilities to other Palm Powered handhelds. Version 4.0 extends the wireless communications features even further, adding numerous additional messaging, telephony, and web access capabilities.

Two of the fundamental communications capabilities that Palm OS users can take advantage of are:

- sending and receiving email communications
- viewing and interacting with the Internet

Users can access these capabilities with the built-in wireless antenna on the Palm VII™ family of devices, or with a PPP connection that uses a wireline or a wireless modem and cell phone on other Palm Powered handhelds.

Overview of Web Clipping Architecture

PalmSource, Inc. invented web clippings to make it possible for users to easily access information on the Internet with a small screen and low connection bandwidth. Web clipping technology allows users to extract and receive specific information from a web page, much like clipping a specific article out of a newspaper.

Numerous web sites are now enabled for web clipping, which means that the site's content is available in web clipping format. In the typical scenario, a web clipping application running on a Palm Powered handheld sends a query to the web site. The web site

responds to the query by sending a clipping back to the handheld, and the web clipping application displays the returned clipping.

You create web clipping applications by compiling standard HTML 3.2 pages with Palm's Web Clipping Application Builder tool, which generates a `.pqa` that operates like a mini-web site and can execute on Palm Powered handhelds. These `.pqa` files are actually databases that are run by the Viewer, which communicates with the Internet.

The Viewer sends requests for information to the Internet via a Palm Proxy server, which converts the compressed format used by the Viewer into standard format and relays the request to the destination server. The server sends information back to the proxy server in standard format, and the proxy server converts the information into compressed format and relays it to the user's device, on which the Viewer displays it.

NOTE: The Palm OS automatically launches the Viewer when the user taps on a web clipping application icon in the Applications Launcher. The Viewer is not visible to users as an application in the Launcher.

For a more complete description of how web clipping works, including a discussion of the Palm Proxy Servers, see the *Web Clipping Developer's Guide*.

About Web Clipping Applications

A web clipping application might contain hyperlinks or an HTML form that displays information and allows the sending or requesting of information. The information can be stored locally, on the handheld device, or remotely, on a web site that can be accessed on the Internet. A web clipping application can also be a self-contained web site.

Figure 9.1 shows a typical web clipping application. This application contains a number of links; when the user taps on any of these links, the HTTP(S) request is sent to the Internet, and the web site sends back a web clipping page that is displayed on the handheld screen.

Figure 9.1 Typical web clipping application

For complete information about creating and building web clipping applications, including reference information about the HTML language features and extensions you can use in these applications, see the *Web Clipping Developer's Guide*.

Using the Viewer to Display Information

The Viewer program is a Palm OS program that displays web clipping applications, interacts with the Palm Proxy Server, and displays web clippings sent back from the Internet. You can also use the Viewer to display HTML content that you have created by launching the Viewer from your Palm OS applications.

Before using the features described in this section, you should verify that the wireless access features are available in the system on which your application is running. For more information, see "System Version Checking" on page 236.

To launch the Viewer and display a web clipping page, use the launch code `sysAppLaunchCmdOpenDB`. Pass the database ID and card number for the `.pqa` that you want to display.

To launch the Viewer and display a specific URL, use the launch code `sysAppLaunchCmdGoToURL`. Pass a pointer to the URL string as a parameter to this launch code. Listing 9.1 shows an example.

NOTE: The Viewer was previously known as the Clipper; thus you see "Clipper" used in various constant names.

Listing 9.1 Launching Viewer with a URL

```
Err GoToURL(Char* origurl)
{         // parameter is ptr to URL string
   Err err;
   Char* url;
   DmSearchStateType searchState;
   UInt16* cardNo;
   LocalID* dbID;

         // make a copy of the URL, since the OS will free
         // the parameter once Viewer quits
   url = MemPtrNew(StrLen(origurl)+1);
   if (!url) return sysErrNoFreeRAM;
   StrCopy(url, origurl);
   MemPtrSetOwner(url, 0);

         // find Viewer and launch it
   err = DmGetNextDatabaseByTypeCreator (true, &searchState,
sysFileTApplication, sysFileCClipper, true, &cardNo, &dbID);
   if (err) { // Viewer is not present
      FrmAlert(NoClipperAlert);
      MemPtrFree(url);
   }
   else
      err =
SysUIAppSwitch(cardNo,dbID,sysAppLaunchCmdGoToURL,url);

   return err; // ErrNone (0) means no error
}
```

IMPORTANT: When programmatically launching an application that connects to the Internet, remember that many Palm users pay for their wireless transmissions on a per-byte basis, and that web sites that are not designed to be Palm-friendly can result in increased airtime charges. For more information about Palm-friendly web pages, see the *Web Clipping Developer's Guide*.

Sending Email Messages

You can send email messages from your Palm OS applications in three different ways:

- Use the standard `mailto` URL in the Viewer.
- Use the `sysAppLaunchCmdAddRecord` launch code to launch the email program with its editor open.
- Use the `sysAppLaunchCmdAddRecord` launch code to silently add the email item to the outbox.

Each of these message-sending methods is described in this section.

Before using the features described in this section, you should verify that the wireless access are available in the system on which your application is running. For more information, see "System Version Checking" on page 236.

Registering an Email Application

The standard, default email application on the Palm OS is either the Palm `iMessenger` program (for the Palm VII device family) or MultiMail (for other Palm Powered handhelds). Starting with version 4.0 of the Palm OS, you can register additional email handling applications by calling the Exchange Manager from within a Palm OS application:

```
ExgRegisterDatatype(  CRID,            // ID of registering app
                      exgReg           // URL scheme registry
                      "mailto",        // the scheme to associate
                      "Email URL",     // description
                      0 );             // any flags
```

When you register a new email-handling application, the Exchange Manager makes that application the default handler for email messages.

For more information about the Palm Exchange Manager, see Chapter 62, "Exchange Manager," on page 1355 and the *Palm OS Programmer's API Reference* book.

Sending Mail from the Viewer

You can send an email message with the Viewer just like you do in standard HTML pages, by using the `mailto:` tag. For example:

```
<a href="mailto:us@company.com">Email us</a>
```

When the Viewer encounters the `mailto` tag, it calls the Exchange Manager to handle sending the email message. The Exchange Manager calls the default email application.

Launching the Email Application for Editing

Use the `sysAppLaunchCmdAddRecord` launch code to launch the iMessenger email program with its editor open (optionally filling in some of the fields via the passed parameter block). This allows the user to edit the email message. To make the email program display the message in its editor, set the `edit` field in the parameter block to `true`.

NOTE: The `sysAppLaunchCmdAddRecord` method of launching an email program is only guaranteed to work with the iMessenger email program.

Adding an Email to the Outbox

Use the `sysAppLaunchCmdAddRecord` launch code to silently add an item (the email message) to the default email program's outbox database. You must pass all the needed information in the parameter block. To prevent the email program from displaying the message in its editor, set the `edit` field in the parameter block to `false`.

When launched via the `sysAppLaunchCmdAddRecord` launch code, the email application returns an error code, or `errNone` if there was no error.

To send a launch code to the default email application, you need obtain its database ID. You can use `DmGetNextDatabaseByTypeCreator` and pass the constant `sysFileCMessaging` for the `creator` parameter.

Note that adding an item to the email outbox does not actually send the message over the radio. It simply stores the message in the outbox until the user later opens the email application and chooses to send queued messages. This always gives the user control over when the radio is used.

Using Wireless Capabilities in Your Applications

This section provides information about system-level features that you may need to use with your Palm OS applications that access wireless communications capabilities. The following topics are covered:

- System Version Checking
- Wireless keyDownEvent Key Codes
- Including Over-the-Air Characters

System Version Checking

Before using any special features of the operating system for wireless communications, you must ensure that your application is running on a device that supports the wireless internet access features of the Palm OS.

NOTE: In some Palm Powered handhelds, the web clipping components are not built into the operating system, but are installed as separate components.

You can check that this feature set is implemented by checking for the existence of the Viewer and iMessenger™ applications. Here's an example of how to check for the Viewer:

```
DmSearchStateType searchState;
UInt cardNo;
LocalID dbID;

err = DmGetNextDatabaseByTypeCreator(true, &searchState,
 sysFileTApplication, sysFileCClipper, true, &cardNo, &dbID);
```

If Viewer is not present, the `DmGetNextDatabaseByTypeCreator` function returns an error. To check for iMessenger, you can use the creator type `sysFileCMessaging`.

For more information on checking system compatibility, see Appendix B, "Compatibility Guide."

Wireless keyDownEvent Key Codes

Versions 3.2 and later of the Palm OS provide special `keyDownEvent` virtual key codes to support the wireless capabilities of the Palm VII family of devices. These include:

- `vchrHardAntenna`, which signals that the user has raised the antenna, activating the radio
- `vchrRadioCoverageOK`, which signals that the unit is within radio coverage following a coverage check
- `vchrRadioCoverageFail`, which signals that the unit is outside radio coverage following a coverage check, and thus cannot communicate with the Palm.Net system

Virtual key codes are passed in the `chr` field of a `keyDownEvent` data block, with the `commandKeyMask` bit set in the `modifiers` field, as described in the section "keyDownEvent" on page 60 of the *Palm OS Programmer's API Reference*.

Normally, you ignore these events in your application event handler, and let the system event handler handle them. For example, the `vchrHardAntenna` event causes the system to invoke the Launcher and switch to the Palm.Net category. If you want to do something different in your application, you must trap and handle the event in your application event handler.

Alternatively, if you want your application to have control over the antenna (avoiding having the system switch to the Launcher on a `vchrHardAntenna` event), you can open the Internet library when your application starts, by calling `INetLibOpen`. You need to open the Internet library with the default or wireless configuration. When your application exits, you must close the Internet library by calling `INetLibClose`. For more information about using the Internet library, see Chapter 7, "Network Communication."

Including Over-the-Air Characters

One of the overriding user interface design goals of the Palm wireless communications system is to always give the user control when making a wireless transaction, partly because of the costs associated with doing so.

You can use the Palm over-the-air characters in your user interface buttons to help the user recognize a wireless transaction. Palm provides two different characters: one for standard transactions, and another for secure transactions, as shown in Figure 9.2.

Figure 9.2 Over the Air Characters

If your application includes a button that causes data to be transmitted when tapped, end the button text with the "Over-the-air" character (`chrOta`). This alerts the user that tapping the button will cause data transmission and incur possible airtime charges.

If your application includes a button that causes data to be transmitted securely when tapped, end the button text with the "Over-the-air-secure" character (`chrOtaSecure`). This alerts the user that tapping the button will cause secure data transmission and incur possible airtime charges.

Note that the Viewer application automatically adds these special characters when rendering remote hyperlinks or buttons. You only need to explicitly add these characters if you are building an application that doesn't use this capability of the Viewer.

10

Telephony Manager

You can use the Palm OS® Telephony Manager to communicate between a handheld and a phone or to access telephony services in an application intended for a smartphone. This chapter contains the following sections that describe how to use the Palm OS Telephony API:

- Telephony Service Types describes the component parts of the telephony API.
- Using the Telephony API describes how to use the telephony API in your applications.

For detailed information about the Telephony Manager data types, constants, and functions, see the following chapters in the *Palm OS Programmer's API Reference*:

- Chapter 73, "Telephony Basic Services."
- Chapter 74, "Telephony Security and Configuration."
- Chapter 75, "Telephony Network."
- Chapter 76, "Telephony Calls."
- Chapter 77, "Telephony SMS."
- Chapter 78, "Telephony Phone Book."

The "Telephony Basic Services" chapter describes the basic services and provides a map to the other functions.

Telephony Service Types

The telephony API organizes functions within sets called **service sets**. Each service set contains a related set of functions that may or may not be available on a particular mobile device or network. You can use the `TelIs<ServiceSet>Available` macro to determine if a service set is supported in the current environment, and you can use the `TelIs<FunctionName>Supported` macro to determine if a specific function is supported in the current environment.

NOTE: Sometimes a service set is supported, but not all of the functions in that service set are supported. See Testing the Telephony Environment for more information.

Each function in the telephony API is prefixed with `Tel`; each telephony service set adds an addition 3 characters to the prefix. Table 10.1 describes the telephony service sets.

Table 10.1 Telephony API service sets

Service set	Functionality	Service prefix
Basic	Basic functions that are always available	`Tel`
Configuration	Services that allow you to configure phones, including SMS configuration	`TelCfg`
Data calls	Data call handling	`TelDtc`
Emergency calls	Emergency call handling	`TelEmc`
Information	Functions to retrieve information about the current phone	`TelInf`
Network	Functions the provide network-oriented services, including authorized networks, current network, signal level, and search mode information	`TelNwk`
OEM	Functions that allow hardware manufacturers to extend the Telephony Manager. Each manufacturer can provide a specific set of OEM functions for a particular device	`TelOem`
Phone book	Functions to access the phone's SIM and address book, including the ability to create, view, and delete phone book entries	`TelPhb`
Power	Power supply level functions	`TelPow`
Security	Functions that provide PIN code management and related services for phone and SIM security-related features	`TelSty`

Table 10.1 Telephony API service sets *(continued)*

Service set	Functionality	Service prefix
Short Message Service	Services to handle Short Message Service (SMS) and to enable the reading, sending, and deleting of short messages	`TelSms`
Sound	Phone sound management, including the playing of key tones and muting.	`TelSnd`
Speech calls	Functions to handle the sending and receiving of speech calls. This service also includes functions that handle DTMF	`TelSpc`

Using the Telephony API

This section provides examples excerpted from the Phone Book Application (`PhBkApp`) sample program, which provides the following capabilities:

- creates, modifies, and deletes entries on a phone, using the SIM and built-in storage on the phone device
- imports entries from the Address Book application
- exports entries to the Address Book application

The `PhBkApp` program opens and accesses the Telephony Manager library and makes a number of calls into the library. It provides an excellent example of using telephony services in your applications.

Accessing the Telephony Manager Library

Before you can use the Telephony Manager library, you must load the library and obtain a reference number for it. Each of the functions in the library requires a reference number argument, which is used with the system code to access a shared library.

Each of the functions in the library also requires an application attachment identifier, which you can obtain by calling the `TelOpen` function.

The example function `LoadTelMgrLibrary`, which is shown in Listing 10.1, makes sure that the Telephony Manager library is

loaded, obtains an application attachment identifier, and returns a reference number for it.

Listing 10.1 Loading the Telephony Manager library

```
Err LoadTelMgrLibrary(UInt16 *telRefNumP, UInt16 *telAppIdP)
{
   Err err;

   err = SysLibFind(kTelMgrLibName, telRefNumP);
   if (err != errNone)
   {
      err = SysLibLoad(kTelMgrDatabaseType,
                  kTelMgrDatabaseCreator, telRefNumP);
      if (err)
         return err;
   }

   err = TelOpen(*telRefNumP, kTelMgrVersion, telAppIdP);
   return err;
}
```

The `LoadTelMgrLibrary` function first calls the `SysLibFind` function to determine if the library has already been loaded, which might be the case if your code has been called by another application that has already loaded the library.

If the library has not already been loaded, `LoadTelMgrLibrary` calls the `SysLibLoad` function to load the library and obtain a reference number for it.

After obtaining a reference number for the library, `LoadTelMgrLibrary` calls the `TelOpen` function to open the loaded library. `TelOpen` opens the Telephony library using the currently selected Connection Manager profile. If you need to use a specific profile, call `TelOpenProfile` instead.

Note that if you are writing an application for a handheld to communicate with a phone, you do not need to establish a network connection between the two. After the Telephony library is successfully opened, each call to the Telephony Manager opens a connection to the phone, performs the necessary operation, and then closes the connection. If you are going to make several calls in succession, use `TelOpenPhoneConnection` to open the connection

to the phone and leave it open. Then use `TelClosePhoneConnection` when you are done.

Closing the Telephony Manager Library

When you are done with the library, you should close it by calling the `TelClose` function, which releases any resources associated with your use of the Telephony Manager.

As shown in Listing 10.2, you must test the return value of the `TelClose` function; if the result is not `telErrLibStillInUse`, you must unload the shared library by calling the `SysLibRemove` function.

Listing 10.2 Closing the Telephony Manager library

```
Err UnloadTelMgrLibrary(UInt16 telRefNum, UInt16 telAppId)
{
   if ((TelClose(telRefNum, telAppId)!= telErrLibStillInUse))
      SysLibRemove(telRefNum);

   return errNone;
}
```

Testing the Telephony Environment

Before running your application, you need to verify that the environment in which it is running (the Palm Powered™ handheld and the telephone device) supports the facilities that your application needs. The Telephony Manager allows you to determine if a specific service set is available, and also allows you to determine if a specific function call is supported.

The code excerpt in Listing 10.3 shows how the `PhBkApp` program verifies that the environment supports the capabilities that it needs, which include all of the phone book-related features of the Telephony Manager. The `PhBkApp` program first tests for the availability of the phone book services, and then determines if several specific functions are supported. Note that the `PhBkApp` refuses to run if any of the capabilities it is using are not available.

Listing 10.3 Testing for the presence of specific capabilities

```
err = TelIsPhbServiceAvailable(gDataP->refNum, gDataP->appId, NULL);
   // Test if phone book capabilities are present
if (err != errNone)
   return err;

   // Check that this phone supports adding entry services
err = TelIsPhbAddEntrySupported(gDataP->refNum, gDataP->appId, NULL);
if (err != errNone)
   return err;

   // Check that this phone supports selecting a phone book
err = TelIsPhbSelectPhonebookSupported(gDataP->refNum, gDataP->appId, NULL);
if (err != errNone)
   return err;

   // Check that this phone supports getting entries
err = TelIsPhbGetEntriesSupported(gDataP->refNum, gDataP->appId, NULL);
if (err != errNone)
   return err;

   // Check that this phone supports getting entry count
err = TelIsPhbGetEntryCountSupported(gDataP->refNum, gDataP->appId, NULL);
if (err != errNone)
   return err;

// Check that this phone supports deleting an entry
err = TelIsPhbDeleteEntrySupported(gDataP->refNum, gDataP->appId, NULL);
return err;
```

For a complete list of the service availability macros, see `TelIs<ServiceSet>Available` in Chapter 73, "Telephony Basic Services," in *Palm OS Programmer's API Reference*.

For more information about determining if a specific function is supported, see `TelIs<FunctionName>Supported` in Chapter 73, "Telephony Basic Services," in *Palm OS Programmer's API Reference*.

Using Synchronous and Asynchronous Calls

Almost all of the telephony API functions can be called either synchronously or asynchronously. If you call a function asynchronously, your application receives an event to notify it that

the function has completed; the event that you receive contains status and other information returned by the function.

This section provides a simple example of calling the `TelPhbAddEntry` function both synchronously and asynchronously to illustrate the difference.

When you call a function synchronously, you need to test the result value returned by the function to determine if the call was successful. For example, the code in Listing 10.4 calls the `TelPhAddEntry` function synchronously.

Listing 10.4 Calling a function synchronously

```
err = TelPhbAddEntry(gTelRefNum, gTelAppID, &gEntry,
   NULL);
printf("Result of adding entry is %d", err);
```

To call the same function asynchronously, you must do the following (see Listing 10.5):

- Pass a pointer to an unsigned integer as the last argument to the call instead of passing `NULL`.

  An asynchronous function call returns immediately. Upon return, the last argument contains an ID to identify this particular operation (the transaction ID).

- In your application's main event loop, use `TelGetEvent` instead of `EvtGetEvent` to get the next event.

  `TelGetEvent` checks both the Telephony Manager's asynchronous reply queue and the system event queue. If a telephony event is available, it returns that. If not, it returns the first event in the system event queue.

  If you're operating outside of the event loop and are only interested in telephony events, you can call `TelGetTelephonyEvent` instead. `TelGetTelephonyEvent` returns only telephony events.

- Check for the `kTelephonyEvent` event type.

  When the function call completes, `TelGetEvent` returns a `TelEventType` structure with the event type set to `kTelephonyEvent`. If the event type is not equal to this event, the normal event loop should process it.

- Check the `functionId` and optionally the `transId` field of the `TelEventType` structure.

  The `functionId` field is a constant that tells you which Telephony Manager call has completed. The `transId` field tells you which instance of the call completed in case you have made two or more asynchronous calls to the same function.

Listing 10.5 Calling a function asynchronously

```
   err = TelPhbAddEntry(gTelRefNum, gTelAppID, &gEntry,
      &transId);
...
}


static void ProcessTelephonyEvent(TelEventType *eventP)
{
   switch(eventP->functionId) {
   ...
      case kTelPhbAddEntryMessage:
         printf("Result of adding entry is %d",
                 eventP->returnCode);
         break;
   ...
}

static void AppEventLoop(void)
{
   UInt16 error;
   EventType event;

   do {
      TelGetEvent(gTelRefNum, gTelAppId,
         (TelEventType *)&event, evtWaitForever);

      if (event->eType == kTelephonyEvent) {
         ProcessTelephonyEvent((TelEventType *)&event);
      } else {
      if (! SysHandleEvent(&event))
         if (! MenuHandleEvent(0, &event, &error))
            if (! AppHandleEvent(&event))
               FrmDispatchEvent(&event);
```

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

Registering for Notifications

If you need to receive events from the Telephony Manager when your application is not the current application, you should register for the `kTelTelephonyNotification`. For example, if you are writing an application that should handle incoming phone calls or receive SMS messages, it is likely that your application will not be the active application when the call or SMS is received. For these types of events, the Telephony Manager posts a `kTelTelephonyNotification`.

If you've registered for the notification, your application receives a `sysAppLaunchCmdNotify` launch code for every telephony event. The `notifyDetailsP` field of this notification's parameter block points to a `TelNotificationType` structure. The `notificationId` field of this structure identifies which specific telephony event has occurred.

Listing 10.6 shows how to register for and receive incoming SMS message notifications.

Listing 10.6 Receiving an SMS message

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP,
   UInt16 launchFlags)
{
   Err error;
   switch (cmd) {
      //Register for the notification when we are first
      //installed and upon each system reset.
      case sysAppLaunchCmdSyncNotify:
      case sysAppLaunchCmdReset:
         error = SysCurrentAppDatabase(&gCardNo, &gAppID);
         if (error) return error;
         SysNotifyRegister(gCarNo, gAppID,
            kTelephonyNotification, NULL,
            sysNotifyNormalPriority, NULL);
      break;

      //Once registered, we receive incoming SMS message
      //notifications as a sysAppLaunchCmdNotify launch code
```

```
      case sysAppLaunchCmdNotify:
         if((SysNotifyParamType *)cmdPBP->notifyType ==
            kTelephonyNotification) && ((SysNotifyParamType*)
            cmdPBP->notifyDetailsP->notificationId ==
            kTelSmsLaunchCmdIncomingMessage))
               ProcessIncomingSms(cmdPBP->notifyDetailsP);
   }
}
```

Using Data Structures With Variably-sized Fields

Many of the telephony functions use data structures that have variably-sized buffer fields. For example, the `TelPhbGetEntry()` function uses the `TelPhbEntryType` structure, which contains two such fields.

```
typedef struct _TelPhbEntryType
{
   UInt16   phoneIndex;
   Char*    fullName;
   UInt8    fullNameSize;
   Char*    dialNumber;
   UInt8    dialNumberSize;
} TelPhbEntryType;
```

The `fullName` and `dialNumber` buffers are variable-sized strings that you allocate in the heap. When you initialize one of these structures to pass to the `TelPhbGetEntry()` function, you must preallocate the buffers and store the allocated size in the corresponding size fields.

Listing 10.7 initializes a `TelPhbEntryType` data structure and passes it to the `TelPhbGetEntry` function to retrieve an entry from the phone book.

Listing 10.7 Initializing a TelPhbEntrypType structure

```
#define maxNameSize         45
#define maxNumSize          20
TelPhbEntryType myEntry;
UInt16 theIndex = 1;
```

```
myEntry.phoneIndex =            theIndex;
myEntry.fullName =              MemPtrNew(maxNameSize);
myEntry.fullNameSize =          maxNameSize;
myEntry.dialNumber =            MemPtrNew(maxNumSize);
myEntry.dialNumberSize =        maxNumSize;

err = TelPhbGetEntry(gPrefs->telRefNum, gPrefsP->telAppId,
                           &myEntry, NULL);
```

Note that you can call the `TelPhbGetEntryMaxSizes()` function to retrieve the maximum name size (in addition to other information) instead of hardcoding it, as done in the above example.

Upon return from the function, the buffer fields are filled in, and the size fields contain the actual number of bytes that were stored into the buffer fields.

If the allocated size of a buffer is not large enough to contain the entire value, the command function does the following:

- Returns the `telErrBufferSize` error.
- Fills the buffer with as much data as it can, and truncates the data that does not fit. If the data ends with a null terminator and is truncated, the null terminator is retained.
- Sets the value of the size field to the actual size required to contain all of the data.

Note that for string buffers, the size includes the byte required for the null terminator character.

NOTE: When you call a function asynchronously, the `telErrBufferSize` error is returned in the `returnCode` field of the event you receive upon completion of the function's execution.

Also, when you call a function asynchronously, it is your responsibility to ensure that any data structure used by the function remains in memory until you receive the completion event.

Summary of Telephony Manager

Telephony Manager Functions

Basic Functions

TelCancel
TelClose
TelClosePhoneConnection
TelGetCallState
TelGetEvent
TelGetTelephonyEvent
TelInfGetInformation
TelIsCfgServiceAvailable
TelIsDtcServiceAvailable
TelIsEmcServiceAvailable
TelIsInfServiceAvailable
TelIsNwkServiceAvailable
TelIsOemServiceAvailable
TelIsPhbServiceAvailable
TelIsPhoneConnected
TelIsPowServiceAvailable
TelIsSmsServiceAvailable
TelIsSndServiceAvailable
TelIsSpcServiceAvailable
TelIsStyServiceAvailable
TelMatchPhoneDriver
TelOemCall
TelOpen
TelOpenProfile
TelOpenPhoneConnection
TelSendCommandString
TelIs<FunctionName>Supported
TelIs<ServiceSet>Available

Data Calls

TelDtcCallNumber
TelDtcCloseLine
TelDtcReceiveData
TelDtcSendData

Emergency Calls

TelEmcCall
TelEmcCloseLine
TelEmcGetNumber
TelEmcGetNumberCount
TelEmcSelectNumber
TelEmcSetNumber

Telephony Manager Functions

Network Interface

TelNwkGetLocation
TelNwkGetNetworkName
TelNwkGetNetworks
TelNwkGetNetworkType
TelNwkGetSearchMode
TelNwkGetSelectedNetwork
TelNwkGetSignalLevel
TelNwkSelectNetwork
TelNwkSetSearchMode

Phone Book

TelPhbAddEntry
TelPhbDeleteEntry
TelPhbGetAvailablePhonebooks
TelPhbGetEntries
TelPhbGetEntry
TelPhbGetEntryCount
TelPhbGetEntryMaxSizes
TelPhbGetSelectedPhonebook
TelPhbSelectPhonebook

Power Management

TelPowGetBatteryStatus
TelPowGetPowerLevel
TelPowSetPhonePower

Security

TelStyChangeAuthenticationType
TelStyEnterAuthenticationCode
TelStyGetAuthenticationState

Short Message Services

TelCfgGetSmsCenter
TelCfgSetSmsCenter
TelSmsDeleteMessage
TelSmsGetAvailableStorage
TelSmsGetDataMaxSize
TelSmsReadMessages
TelSmsReadReport
TelSmsReadReports
TelSmsReadSubmittedMessage
TelSmsReadSubmittedMessages

Telephony Manager Functions	
TelSmsGetMessageCount	TelSmsSelectStorage
TelSmsGetSelectedStorage	TelSmsSendManualAcknowledge
TelSmsGetUniquePartId	TelSmsSendMessage
TelSmsReadMessage	
Sound	
TelSndMute	TelSndStopKeyTone
TelSndPlayKeyTone	
Speech Calls	
TelSpcAcceptCall	TelSpcRejectCall
TelSpcCallNumber	TelSpcRetrieveHeldLine
TelSpcCloseLine	TelSpcSelectLine
TelSpcConference	TelSpcSendBurstDTMF
TelSpcGetCallerNumber	TelSpcStartContinuousDTMF
TelSpcHoldLine	TelSpcStopContinuousDTMF
TelSpcPlayDTMF	

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W