Low-Level Communications

Exploring Palm OS®

Written by Eric Shepherd
Edited by Susan Salituro
Engineering contributions by Rene Portier, Justin Morey, Kamran Khan, Tom Keel, David Schlesinger, Eric Lapuyade

Exploring Palm OS: Low-Level Communications
Document Number 3116-005
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

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

Table of Contents

Part III: Bluetooth

Part IV: Networking and Sockets

Part V: WiFi

17 WiFi Reference 347

Part VI: IOS STDIO

About This Document

This book covers the portions of Palm OS® that make it possible to develop applications that make use of telecommunication technologies such as networking, infrared, Bluetooth, and serial connectivity.

The primary focus of this book is the lower-level aspects of communication. If your application needs to perform higher-level functions, such as exchanging typed data objects or exchanging standard vObjects, you should instead refer to the book *Exploring Palm OS: High-Level Communications*.

Intended Audience

You should read this book if you want to write Palm OS applications that use networking, Bluetooth, IrDA, or serial communications to transmit and receive data between a Palm OS device and either another Palm OS device or a peripheral device.

The APIs described in this book are only needed if your application will perform communications of this nature. You should read *Exploring Palm OS: Programming Basics* before this book, in order to gain the necessary background in Palm OS programming. Read this book when you find that you need to enable your application with communications functionality.

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/

Part I Serial Communication

Palm OS provides a complete architecture for accessing and manipulating devices using a serial interface.

1

Introduction to Serial Communications

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

- The Serial Manager is responsible for byte-level serial I/O and control of the RS-232, USB, Bluetooth, and IR signals.
- The Serial Link Protocol provides an efficient mechanism for sending and receiving packets.
- Serial Link Manager is the Palm OS implementation of the serial link protocol.

Serial Communications Overview

Serial communications in Palm OS are provided through the same driver architecture as all other forms of communication. Serial drivers operate as part of the I/O Subsystem.

Serial Communications Components

There are, however, some additional components built on top of this architecture that provide additional services:

- The Serial Manager API provides a simplified mechanism for performing serial communications, and also provides source code compatibility for applications originally written for previous versions of Palm OS. See "The Serial Manager" on page 5.

- 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 21.
- 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).

Byte Ordering

It is important to be aware that the ARM processor uses little-endian byte ordering, where the 68K series of processors used by older Palm OS devices used big-endian byte ordering. This may be an issue when transmitting data using a serial connection, and your application is responsible for coping with byte order differences.

2

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. Under Palm OS Cobalt and later versions of the operating system, the Serial Manager is implemented as a STREAMS driver and a compatibility library that lets you continue to use the Serial Manager API.

IMPORTANT: The Palm OS Cobalt Serial Manager implements the API formerly known as the "New Serial Manager;" functions with names beginning with "`Srm`" are supported. The Old Serial Manager functions—those that begin with "`Ser`"—are not supported under Palm OS Cobalt, version 6, and later.

Because the Serial Manager is now based on the Palm OS Cobalt I/O architecture, the concept of "virtual serial drivers" is no longer supported; instead, the Serial Manager works with the I/O Subsystem and the Connection Manager to support communication through any communications interface. The legacy Serial Manager ports, such as `serCradlePort`, `sysFileCVirtIrComm`, `sysFileCVirtRfComm`, and so on are still supported.

To ensure that the Serial Manager does not slow down processing of user events, the Serial Manager receives data asynchronously. The Serial Manager API, however, executes synchronously; if a Serial Manager function blocks during execution, this does not affect the system's ability to keep receiving data.

The Serial Manager functions that send data return as soon as they have handed off all the data to the lower-level IOS STREAMS write queue. The actual transmission of the data will be handled later, asynchronously.

This chapter describes the Serial Manager. It covers the following topics:

- Steps for Using the Serial Manager
- Opening a Port

- Closing a Port
- Configuring the Port
- Sending Data
- Receiving Data
- Serial Manager Tips and Tricks

About the Serial Manager

The Serial Manager provides an interface to communications devices. These communications devices can include a serial port, cradle port, infrared port, USB, Bluetooth, and other devices that are accessible through the Connection Manager. This API provides a degree of compatibility with software written for previous versions of Palm OS.

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 closes the underlying IOS connection.

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 book *Exploring Palm OS: High-Level Communications* for information on the Connection Manager.

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 , 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.

   See “Opening a Port” on page 7.

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 9.

3. Send or receive data.

   See "Sending Data" on page 11 and "Receiving Data" on page 12.

4. Close the port.

   See "Closing a Port" on page 9.

The next several sections describe these steps in more detail.

TIP: See "Serial Manager Tips and Tricks" on page 18 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.

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

When you attempt to open a serial port, 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. This error is returned if one of the underlying drivers involved in the connection is already in use; for example, if an active PPP session is currently using the UART.
- If any error is returned, the port was not opened, and the application must *not* close it.

Opening a Port

To open a port , call the SrmOpen() function, specifying the port (see "Specifying the Port" on page 9) and the initial baud rate of the serial interface. SrmOpen returns a port ID that uniquely identifies this connection. You pass this port ID to all other Serial Manager calls.

The 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 2.1 Opening the port

```
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;
```

Specifying the Port

Ports are specified using a hardware-independent port ID. Palm OS will map them to the correct physical port by locating the appropriate port using the Connection Manager.

See Chapter 4, "Port Constants," on page 32 for a list of port IDs you can use when opening a serial connection.

Closing a Port

Once an application is finished with the serial port, it must close the port using the `SrmClose()` function. 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
- CTS/RTS hardware flow control with a 5-second timeout on CTS low
- 1 stop bit
- 8 data bits
- 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.

Using a Custom Receive Queue

The default receive queue size is 512 bytes. If you wish to use a different size of buffer, you can do so by using a custom receive queue.

To use a custom receive queue, an application must:

- Allocate memory for the custom queue; this memory can be allocated using `malloc()`, or can be either a local or global variable. Be aware that the memory must remain in place as long as the buffer is in use.
- Call `SrmSetReceiveBuffer()` 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 2.2 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()` .

Listing 2.3 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 2.3 Changing the configuration

```
status_t 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);
```

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`.

Sending Data

To send data, use `SrmSend()`. 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 2.4 illustrates the use of `SrmSend()`.

Listing 2.4 Sending data

```
UInt32 toSend, numSent;
status_t 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 `SrmSendCheck()` 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()` function can be used to flush remaining bytes in the FIFO that have not been sent.

Under Palm OS Cobalt, the `SrmSendWait()` function no longer waits to ensure that the data has been sent. There is no longer any way to ensure that the data has actually been transmitted. This function’s use is discouraged.

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()` or `IOSPoll()` in your event loop.

  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()`. 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()` 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 milliseconds. 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()` . 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 14.

IMPORTANT: Always check for line errors. Due to unpredictable conditions, there is no guarantee 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 15.

You can directly access the receive queue using the `SrmReceiveWindowOpen()` and `SrmReceiveWindowClose()` functions. These functions allow fast access to the buffer to reduce buffer copying.

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()`.

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

Listing 2.5 checks whether a framing or parity error has been returned and clears the line errors.

Listing 2.5 Handling line errors

```
void HandleSerReceiveErr(UInt16 portId, status_t 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);
   }
}
```

TIP: See "Common Errors" on page 19 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()`. 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 2.6 shows how to receive large blocks of data using the Serial Manager.

Listing 2.6 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.
*
************************************************************/
status_t RcvSerialData(UInt16 thePort, UInt8 *rcvDataP, UInt32 *bufSizeP)
{
UInt32 bytesLeft, maxRcvBlkSize, bytesRcvd, waitTime, totalRcvBytes = 0;
UInt8 *newRcvBuffer;
UInt16 dataLen = sizeof(UInt32);
status_t* 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 = 200;

   // 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 Simulator. Debug on the device last.

  The Simulator 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 Simulator 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 9).

- Software overrun

  The application is not reading incoming data fast enough. Read data more frequently, or use hardware flow control. (see "Configuring the Port" on page 9).

3

The Serial Link Protocol

The Serial Link Protocol

The Serial Link Protocol (SLP) provides an efficient packet send and receive mechanism that is used by the Palm OS® 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

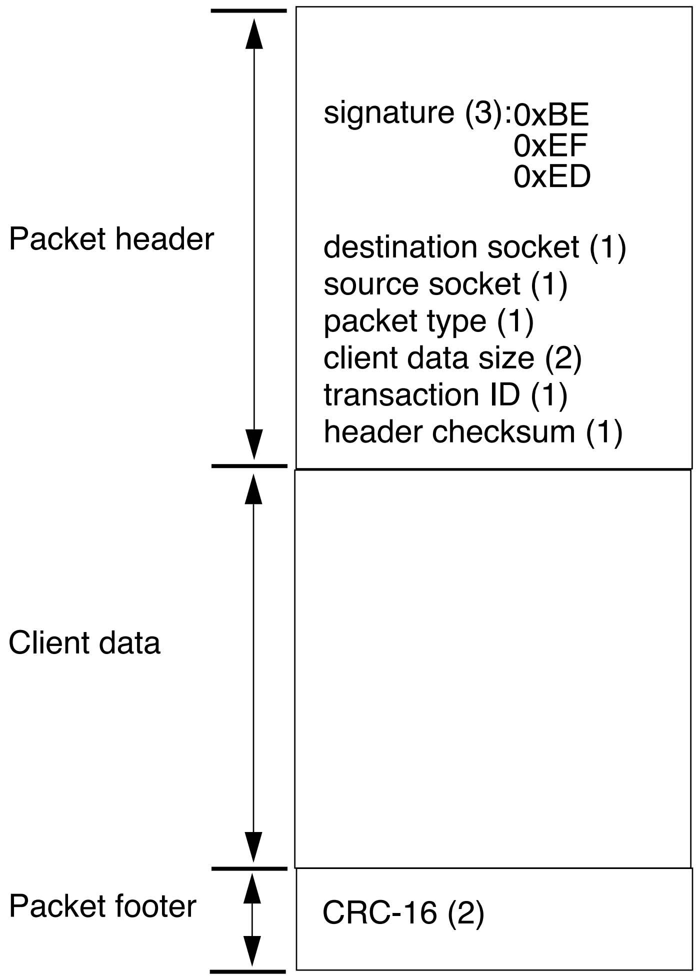
The following sections describe:

- 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 3.1.

Figure 3.1 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.

The 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 23.

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 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 3.1 incorporates the processes described in this section.

Listing 3.1 Sending a Serial Link Packet

```
status_t                     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 3.2 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.

4

Serial Manager Reference

This chapter provides reference material for the Serial Manager API:

- Serial Manager Data Structures
- Serial Manager Constants
- Serial Manager Functions
- Serial Manager Application-Defined Functions

The header file `SerialMgr.h` declares the Serial Manager API. The file `SystemResources.h` defines some serial port constants.

Serial Manager Data Structures

DeviceInfoType Typedef

Purpose The `DeviceInfoType` structure defines information about a serial device. This structure is returned by the `SrmGetDeviceInfo()` function.

Prototype

```
typedef struct DeviceInfoType {
   uint32_t serDevCreator;
   uint32_t serDevFtrInfo;
   uint32_t serDevMaxBaudRate;
   uint32_t serDevHandshakeBaud;
   char *serDevPortInfoStr;
   uint8_t reserved[8];
} DeviceInfoType;
typedef DeviceInfoType *DeviceInfoPtr;
```

Fields

`serDevCreator`
: Four-character creator ID for serial driver.

`serDevFtrInfo`
Flags defining features of this serial hardware. See Serial Capabilities Constants for a description of these flags.

`serDevMaxBaudRate`
Maximum baud rate for this device.

`serDevHandshakeBaud`
Hardware handshaking is recommended for baud rates over this rate.

`serDevPortInfoStr`
Description of serial hardware device or virtual device.

SrmOpenConfigType Struct

Purpose
The `SrmOpenConfigType` structure specifies parameters for opening a serial port. This structure is passed as a parameter to `SrmExtOpen()`.

Prototype

```
typedef struct SrmOpenConfigType {
   uint32_t baud;
   uint32_t function;
   MemPtr drvrDataP;
   uint16_t drvrDataSize;
   uint16_t sysReserved0;
   uint32_t sysReserved1;
   uint32_t sysReserved2;
} SrmOpenConfigType;
```

Fields

`baud`
Baud rate at which to open the connection. Serial drivers that do not require baud rates ignore this field.

`function`
Reserved for system use.

`drvrDataP`
Pointer to a driver-specific data block.

`drvrDataSize`
The size of the data block pointed to by `drvrDataP`.

`sysReserved0`
Reserved for future use.

`sysReserved1`
Reserved for future use.

`sysReserved2`
Reserved for future use.

Comments The `function` field, which was used under Palm OS® 5.x and earlier, is now reserved for system use.

SrmRfCommOpenParamsType Struct

Purpose Specifies open parameters for opening an RFCOMM port.

Prototype

```
typedef struct SrmRfcommOpenParamsType {
   BtLibDeviceAddressType btAddr;
   uint16_t sysReserved0;
   char *serviceClassIDName;
} SrmRfcommOpenParamsType
```

Fields `btAddr`
The address of the Bluetooth device to connect to. If a null address (00:00:00:00:00:00) is specified, a Bluetooth discovery operation is performed at connect time.

`sysReserved0`
Reserved for system use.

`serviceClassIDName`
A string describing the service class ID to connect to. This is usually set to “serial_port” for a standard RFCOMM connection.

NOTE: This behavior has changed since Palm OS Garnet. 68K applications remain binary compatible, but source code compatibility is broken for newly-written applications. In addition, it is no longer possible to use the Serial Manager to open an RFCOMM port in server mode; use the IOS API instead.

Serial Manager Constants

Port Constants

Purpose When you specify the port to open in the `SrmOpen()` or `SrmExtOpen()` call, you can specify one of these ports to select a standard interface instead of using the Connection Manager to select the interface you want to use.

Constants

`serPortLocalHotSync`
The physical HotSync® port. The Serial Manager automatically detects whether this port is USB or RS-232.

`serPortCradlePort`
Cradle port. The Serial Manager automatically detects whether this port is USB or RS-232. Most applications should specify this as the port.

`serPortIrPort`
The IR port. This is a raw IrDA port with no protocol support.

`serPortConsolePort`
The debug console port, either USB or RS-232. USB is preferred where both are available.

`serPortCradleRS232Port`
Port for the RS-232 cradle. Specify this port if you want to ensure that your application uses RS-232 communications only.

`serPortCradleUSBPort`
Port for the USB cradle. This port may only be used by the HotSync application.

`sysFileCVirtIrComm`
Serial communications over infrared (IrComm). Retained for compatibility with previous versions of Palm OS.

`sysFileCVirtRfComm`
Serial communications over Bluetooth (RFCOMM). Retained for compatibility with previous versions of Palm OS. This port must be used by calling `SrmExtOpen()`, with `drvrDataP` pointing to an `SrmRfcommOpenParamsType` structure.

Serial Capabilities Constants

Purpose The serial capabilities constant flags describe serial hardware capabilities. These flags are set in the `serDevFtrInfo` field of the `DeviceInfoType` structure.

Constants

`serDevCradlePort`
Serial hardware controls RS-232 serial from cradle connector of the device.

`serDevRS232Serial`
Serial hardware has RS-232 line drivers.

`serDevIRDACapable`
Serial hardware has IR line drivers and generates IrDA mode serial signals.

`serDevConsolePort`
Serial device is the default console port.

`serDevUSBCapable`
Serial hardware controls USB serial from cradle connector of the device.

`serDevHotsyncCapable`
Serial device can be used for HotSync.

Serial Settings Constants

Purpose The serial settings constants identify bit flags that correspond to various serial hardware settings. Use `SrmControl()` with the op code `srmCtlSetFlags` to control which settings are used.

Constants

`srmSettingsFlagStopBitsM`
Mask for stop bits field

`srmSettingsFlagStopBits1`
1 stop bit (default)

`srmSettingsFlagStopBits2`
2 stop bits

`srmSettingsFlagParityOnM`
Mask for parity on

`srmSettingsFlagParityEvenM`
Mask for parity even

`srmSettingsFlagXonXoffM`
Mask for Xon/Xoff flow control (not implemented)

`srmSettingsFlagRTSAutoM`
Mask for RTS receive flow control. This is the default.

`srmSettingsFlagCTSAutoM`
Mask for CTS transmit flow control

`srmSettingsFlagBitsPerCharM`
Mask for bits per character

`srmSettingsFlagBitsPerChar5`
5 bits per character

`srmSettingsFlagBitsPerChar6`
6 bits per character

`srmSettingsFlagBitsPerChar7`
7 bits per character

`srmSettingsFlagBitsPerChar8`
8 bits per character (default)

`srmSettingsFlagFlowControlIn`
Protect the receive buffer from software overruns. When this flag and `srmSettingsFlagRTSAutoM` are set, which is the default case, it causes the Serial Manager to assert RTS to prevent the transmitting device from continuing to send data when the receive buffer is full. Once the application receives data from the buffer, RTS is de-asserted to allow data reception to resume.

Note that this feature effectively prevents software overrun line errors but may also cause CTS timeouts on the transmitting device if the RTS line is asserted longer than the defined CTS timeout value.

SrmCtlEnum Enum

Purpose The `SrmCtlEnum` enumerated type specifies a serial control operation. Specify one of these enumerated types for the *op* parameter to the `SrmControl()` call.

Constants `srmCtlSetBaudRate`
Sets the current baud rate for the serial hardware.

`srmCtlGetBaudRate`
: Gets the current baud rate for the serial hardware.

`srmCtlSetFlags`
: Sets the current flag settings for the serial hardware. Specify flags from the set described in Serial Settings Constants.

`srmCtlGetFlags`
: Gets the current flag settings for the serial hardware.

`srmCtlSetCtsTimeout`
: Sets the current CTS timeout value for hardware handshaking.

`srmCtlGetCtsTimeout`
: Gets the current CTS timeout value for hardware handshaking.

`srmCtlIrDAEnable`
: Enable IrDA connection on this serial port.ioctl

`srmCtlIrDADisable`
: Disable IrDA connection on this serial port.

`srmCtlRxEnable`
: Enable receiver (for IrDA).

`srmCtlRxDisable`
: Disable receiver (for IrDA).

`srmCtlEmuSetBlockingHook`
: Set a blocking hook routine for emulation mode only. Not supported on the actual device.

`srmCtlSystemReserved`
: Reserves op codes between 0x7000 and 0x8000 for system use.

NOTE: Palm OS Cobalt no longer supports custom opcodes. If you need the added flexibility, you should use IOS STDIO calls directly.

Status Constants

Purpose The status constants identify bit flags that correspond to the status of serial signals. They can be returned by the `SrmGetStatus()` function.

Constants

`srmStatusCtsOn`
CTS line is active.

`srmStatusRtsOn`
RTS line is active.

`srmStatusDsrOn`
DSR line is active.

`srmStatusBreakSigOn`
Break signal is active.

`srmStatusDtrOn`
DTR is active.

`srmStatusDcdOn`
DCD is active.

`srmStatusRingOn`
Ring detected.

NOTE: You can set most of these signals by using `IOSIoctl()` calls.

Line Error Constants

Purpose The line error constants identify bit flags that correspond to the line errors that may occur on the port. They can be returned by the `SrmGetStatus()` function.

Constants

`serLineErrorParity`
Parity error

`serLineErrorHWOverrun`
Hardware overrun

`serLineErrorFraming`
Framing error

`serLineErrorBreak`
Break signal asserted

`serLineErrorHShake`
Line handshake error

`serLineErrorSWOverrun`
Software overrun

`serLineErrorCarrierLost`
Carrier detect signal dropped

Serial Manager Functions

SrmClearErr Function

Clears the port of any line errors.

Declared In `SerialMgr.h`

Prototype `status_t SrmClearErr (uint16_t portId)`

Parameters → *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrNotSupported`
The port is not the foreground port.

SrmClose Function

Purpose Closes a serial port and makes it available to other applications, regardless of whether the port is a foreground or background port.

Declared In `SerialMgr.h`

Prototype `status_t SrmClose (uint16_t portId)`

Parameters → *`portId`*
Port ID for port to be closed.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The serial port is not open.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments If a foreground port is being closed and a background port exists, the background will have access to the port as long as another foreground port is not opened.

If a foreground port is being closed and a yielded port exists, the yielded port will have access to the port as long as it does not yield to the opening of another foreground port. If there are both a yielded port and a background port for the foreground port being closed, the yielded port takes precedence over the background port.

See Also `SrmOpen()`

SrmControl Function

Purpose Performs a serial control function.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmControl (uint16_t portId,
   uint16_t op, void *valueP,
   uint16_t *valueLenP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *op*
Control operation to perform. Specify one of the `SrmCtlEnum` enumerated types.

→ *valueP*
Pointer to a value to use for the operation. See Comments for details.

↔ *valueLenP*
Pointer to the size of *valueP*. See Comments for details.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadParam`
An invalid op code was specified.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The serial port is not open.

`serErrNoDevicesAvail`
No serial devices could be found.

`serErrNotSupported`
The specified op code is not supported in the current configuration.

Comments Table 4.1 shows what to pass for the *valueP* and *valueLenP* parameters for each of the operation codes. Control codes not listed do not use these parameters. See `SrmCtlEnum` for a complete list of control codes.

Table 4.1 SrmControl Parameters

Operation Code	Parameters
`srmCtlSetBaudRate`	-> *valueP* = Pointer to `int32_t` (baud rate) -> *valueLenP* = Pointer to `sizeof(int32_t)`
`srmCtlGetBaudRate`	<- *valueP* = Pointer to `int32_t` (baud rate) <- *valueLenP* = Pointer to `int16_t`
`srmCtlSetFlags`	-> *valueP* = Pointer to `Uint32` (bitfield; see Serial Settings Constants) -> *valueLenP* = Pointer to `sizeof(uint32_t)`
`srmCtlGetFlags`	<- *valueP* = Pointer to `uint32_t` (bitfield) <- *valueLenP* = Pointer to `int16_t`
`srmCtlSetCtsTimeout`	-> *valueP* = Pointer to `int32_t` (timeout value) -> *valueLenP* = Pointer to `sizeof(int32_t)`
`srmCtlGetCtsTimeout`	<- *valueP* = Pointer to `int32_t` (timeout value) <- *valueLenP* = Pointer to `int16_t`
`srmCtlUserDef`	<-> *valueP* = Pointer passed to the serial driver <-> *valueLenP* = Pointer to `sizeof(int32_t)` For a serial driver, these pointers are passed to the driver's control function and they contain that functions return values (if any) upon return.

`srmCtlSetCtsTimeout`	-> *valueP* = Pointer to `int32_t` (timeout value) -> *valueLenP* = Pointer to `sizeof(int32_t)`
`srmCtlGetCtsTimeout`	<- *valueP* = Pointer to `int32_t` (timeout value) <- *valueLenP* = Pointer to `int16_t`
`srmCtlUserDef`	<-> *valueP* = Pointer passed to the serial driver <-> *valueLenP* = Pointer to `sizeof(int32_t)` For a serial driver, these pointers are passed to the driver's control function and they contain that functions return values (if any) upon return.

SrmExtOpen Function

Purpose Opens a foreground port connection with the specified configuration.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmExtOpen (uint32_t port,
   SrmOpenConfigType *configP,
   uint16_t configSize, uint16_t *newPortIdP)
```

Parameters

→ *port*
The four-character port name (such as 'ircm' or 'u328') or logical port number to be opened. (See Port Constants.)

→ *configP*
Pointer to the configuration structure specifying the serial port's properties. See `SrmOpenConfigType`.

→ *configSize*
The size of the configuration structure pointed to by *configP*.

← *newPortIdP*
Contains the port ID to be passed to other Serial Manager functions.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
The *port* parameter does not specify a valid port.

`serErrBadParam`
The *configP* parameter is `NULL`.

`serErrAlreadyOpen`
The Serial Manager already has a port open.

`memErrNotEnoughSpace`
There was not enough memory available to open the port.

Comments Do not keep the port open any longer than necessary. An open serial port consumes more energy from the device's batteries.

The values specified in the *configP* parameter depend on the type of connection being made. For RS-232 connections, you specify the baud rate but not a purpose. For USB connections, you specify a purpose but not a baud rate.

When opening the RFCOMM (`'rfcm'`) port, you should specify in the *configP*`->drvrDataP` field a pointer to an `SrmRfcommOpenParamsType` structure.

A newly opened port has its line errors cleared, the default CTS timeout set (specified by the constant `srmDefaultCTSTimeout`), a 512-byte receive queue allocated, 1 stop bit, 8 bits per character, RTS enabled, and flow control enabled. To increase the receive queue size, use `SrmSetReceiveBuffer()`. To change the other serial port settings, use `SrmControl()`.

See Also `SrmOpen()`

SrmGetDeviceCount Function

Purpose Returns the number of available serial devices.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmGetDeviceCount
    (uint16_t *numOfDevicesP)
```

Parameters

← *numOfDevicesP*
Pointer to address where the number of serial devices is returned.

Returns

`errNone`
No error.

See Also `SrmGetDeviceInfo()`

SrmGetDeviceInfo Function

Purpose Returns information about a serial device.

Declared In `SerialMgr.h`

Prototype
```
status_t SrmGetDeviceInfo (uint32_t deviceID,
   DeviceInfoType *deviceInfoP)
```

Parameters

→ *`deviceID`*
ID of serial device to get information for. You can pass a zero-based index (0, 1, 2, ...), a valid port ID returned from `SrmOpen()` or `SrmExtOpen()`, or a 4-character port name (such as 'u328', 'u650', or 'ircm'). See Port Constants.

← *`deviceInfoP`*
Pointer to a `DeviceInfoType` structure where information about the device is returned.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNoDevicesAvail`
The Serial Manager cannot find any serial devices.

See Also `SrmGetDeviceCount()`

SrmGetStatus Function

Purpose Returns status information about the serial hardware.

Declared In `SerialMgr.h`

Prototype
```
status_t SrmGetStatus (uint16_t portId,
   uint32_t *statusFieldP, uint16_t *lineErrsP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

← *statusFieldP*
Pointer to address where hardware status information for the port is returned. This is a 32-bit field using the flags described in Status Constants.

← *lineErrsP*
Pointer to address where the number of line errors for the port is returned. The line error flags are described in Line Error Constants.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotSupported`
The port is a yielded port.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments Typically, `SrmGetStatus()` is called to retrieve the line errors for the port if some of the send and receive functions return a `serErrLineErr` error code.

SrmOpen Function

Purpose Opens a foreground port connection with the specified port name or logical port number.

Declared In `SerialMgr.h`

Prototype
```
status_t SrmOpen (uint32_t port, uint32_t baud,
   uint16_t *newPortIdP)
```

Parameters

→ *port*
: The four-character port name or logical port number to be opened. See Port Constants for more information.

→ *baud*
: Initial baud rate of port.

← *newPortIdP*
: Contains the port ID to be passed to other Serial Manager functions.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrAlreadyOpen`
: This port already has an installed foreground owner.

`serErrBadPort`
: This port doesn't exist.

`memErrNotEnoughSpace`
: There was not enough memory available to open the port.

Comments Only one application or task may have access to a particular serial port at any time.

Do not keep the port open any longer than necessary. An open serial port consumes more energy from the device's batteries.

SrmPrimeWakeupHandler Function

Purpose Sets the number of received bytes that triggers a call to the wakeup handler function.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmPrimeWakeupHandler (uint16_t portId,
    uint16_t minBytes)
```

Parameters

→ *portId*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *minBytes*
Number of bytes that must be received before wakeup handler is called. Typically, this is set to 1.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments This function primes a wakeup handler installed by `SrmSetWakeupHandler()`.

See Also `SrmSetWakeupHandler()`, `WakeupHandlerProcPtr()`

SrmReceive Function

Purpose Receives a specified number of bytes.

Declared In `SerialMgr.h`

Prototype

```
uint32_t SrmReceive (uint16_t portId,
   void *rcvBufP, uint32_t count,
   int32_t timeout, status_t *errP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

← *rcvBufP*
Pointer to buffer where received data is to be returned.

→ *count*
Length of data buffer (in bytes). This specifies the number of bytes to receive.

→ *timeout*
The amount of time (in milliseconds) that the Serial Manager waits to receive the requested block of data. At the end of the timeout, data received up to that time is returned.

← *errP*
Error code.

Returns Number of bytes of data actually received.

Comments

IMPORTANT: Note that in versions of Palm OS prior to 6.0, the timeout was specified in ticks. It is now specified in milliseconds.

The following error codes can be returned in *errP*:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrTimeOut`
Unable to receive data within the specified timeout period.

`serErrConfigurationFailed`
The port needs time to configure, and the configuration has failed.

`serErrNotSupported`
The port is not the foreground port.

`serErrConfigurationFailed`
The port could not configure itself.

`serErrLineErr`
A line error occurred during the receipt of data. Use `SrmGetStatus()` to obtain the exact line error.

`serErrNoDevicesAvail`
No serial devices could be found.

See Also `SrmReceiveCheck()`, `SrmReceiveFlush()`, `SrmReceiveWait()`

SrmReceiveCheck Function

Purpose Checks the receive FIFO and returns the number of bytes in the serial receive queue.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmReceiveCheck (uint16_t portId,
   uint32_t *numBytesP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

← *numBytesP*
Number of bytes in the receive queue.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrLineErr`
: A line error has occurred. Use `SrmGetStatus()` to obtain the exact line error.

See Also `SrmReceive()`, `SrmReceiveFlush()`, `SrmReceiveWait()`

SrmReceiveFlush Function

Purpose Flushes the receive FIFOs.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmReceiveFlush (uint16_t portId,
   int32_t timeout)
```

Parameters

→ *`portId`*
: Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *`timeout`*
: Timeout value, in milliseconds.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrNotSupported`
: The port is not the foreground port.

`serErrNoDevicesAvail`
: No serial devices could be found.

Comments

> **IMPORTANT:** Note that in versions of Palm OS prior to 6.0, the timeout was specified in ticks. It is now specified in milliseconds.

The *`timeout`* value forces this function to wait a period of microseconds after flushing the port to see if more data shows up to be flushed. If more data arrives within the timeout period, the port is flushed again and the timeout counter is reset and waits again. The function only exits after no more bytes are received by the port for the full timeout period since the last flush of the port. To avoid this waiting behavior, specify 0 for the timeout period.

Any errors on the line are cleared before this function returns.

See Also `SrmReceive`, `SrmReceiveCheck`, `SrmReceiveWait`

SrmReceiveWait Function

Purpose Waits until some number of bytes of data have arrived into the serial receive queue, then returns.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmReceiveWait (uint16_t portId,
   uint32_t bytes, int32_t timeout)
```

Parameters

→ *`portID`*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *`bytes`*
Number of bytes to wait for.

→ *`timeout`*
Timeout value, in microseconds.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrTimeOut`
Unable to receive data within the specified timeout period.

`serErrNotSupported`
The port is not the foreground port.

`serErrBadParam`
The bytes parameter exceeds the size of the receive queue. Use `SrmSetReceiveBuffer()` to increase the size of the receive queue.

`serErrLineErr`
A line error occurred during the receipt of data. Use `SrmGetStatus()` to obtain the exact line error.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments

IMPORTANT: Note that in versions of Palm OS prior to 6.0, the timeout was specified in ticks. It is now specified in milliseconds.

If this function returns no error, the application can either check the number of bytes currently in the receive queue (using `SrmReceiveCheck()`) or it can just specify a buffer and receive the data by calling `SrmReceive()`.

Do not call `SerReceiveWait()` from within a wakeup handler. If you do, the `serErrTimeOut` error is returned.

See Also

`SrmReceive()`, `SrmReceiveCheck()`, `SrmReceiveFlush()`

SrmReceiveWindowClose Function

Purpose Closes direct access to the Serial Manager's receive queue.

Declared In `SerialMgr.h`

Prototype
```
status_t SrmReceiveWindowClose (uint16_t portId,
   uint32_t bytesPulled)
```

Parameters

→ *portId*
: Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *bytesPulled*
: Number of bytes the application read from the receive queue.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrNotSupported`
: The port is not the foreground port.

`serErrNoDevicesAvail`
: No serial devices could be found.

Comments Call this function when the application has read as many bytes as it needs out of the receive queue or it has read all the available bytes.

See Also `SrmReceiveWindowOpen()`

SrmReceiveWindowOpen Function

Purpose Provides direct access to the Serial Manager's receive queue.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmReceiveWindowOpen (uint16_t portId,
   UInt8 **bufPP, uint32_t *sizeP)
```

Parameters

→ *portId*
: Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

← *bufPP*
: Pointer to a pointer to the receive buffer.

← *sizeP*
: Available bytes in buffer.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrNotSupported`
: The port is not the foreground port.

`serErrLineErr`
: The data in the queue contains line errors.

`serErrNoDevicesAvail`
: No serial devices could be found.

Comments This function lets applications directly access the Serial Manager's receive queue to eliminate buffer copying by the Serial Manager. This access is a "back door" route to the received data. After retrieving data from the buffer, the application must call `SrmReceiveWindowClose()`.

Applications that want to empty the receive buffer entirely should call the `SrmReceiveWindowOpen()` and

`SrmReceiveWindowClose()` functions repeatedly until the buffer size returned is 0.

IMPORTANT: Once an application calls `SrmReceiveWindowOpen()`, it should not attempt to receive data via the normal method of calling `SrmReceive()` or `SrmReceiveWait()`, as these functions interfere with direct access to the receive queue.

See Also `SrmReceiveWindowClose()`

SrmSend Function

Purpose Sends a block of data out the specified port.

Declared In `SerialMgr.h`

Prototype

```
uint32_t SrmSend (uint16_t portId,
   const void *bufP, uint32_t count,
   status_t *errP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *bufp*
Pointer to data to send.

→ *count*
Length of data buffer, in bytes.

← *errP*
Error code. See the Comments section for details.

Returns Number of bytes of data actually sent.

Comments When `SrmSend()` returns, you should check the value returned in the *errP* parameter. If `errNone`, then the entire data buffer was sent. If not `errNone`, then the result equals the number of bytes sent before the error occurred. The possible error values are:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrTimeOut`
Unable to send data within the specified CTS timeout period.

`serErrNoDevicesAvail`
No serial devices could be found.

`serErrConfigurationFailed`
The port configuration has failed.

`serErrNotSupported`
The specified port is not the foreground port.

See Also `SrmSendCheck()`, `SrmSendFlush()`, `SrmSendWait()`

SrmSendCheck Function

Purpose Checks the transmit FIFO and returns the number of bytes left to be sent.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmSendCheck (uint16_t portId,
   uint32_t *numBytesP)
```

Parameters

→ *portID*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

← *numBytesP*
Number of bytes left in the FIFO queue.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrNotSupported`
: This feature not supported by the hardware.

`serErrNoDevicesAvail`
: No serial devices could be found.

Comments Not all serial devices support this feature.

See Also `SrmSend()`, `SrmSendFlush()`, `SrmSendWait()`

SrmSendFlush Function

Purpose Flushes the transmit FIFO.

Declared In `SerialMgr.h`

Prototype `status_t SrmSendFlush (uint16_t portId)`

Parameters → *`portID`*
: Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: The port is not open.

`serErrNotSupported`
: The port is not the foreground port.

`serErrNoDevicesAvail`
No serial devices could be found.

See Also `SrmSend()`, `SrmSendCheck()`, `SrmSendWait()`

SrmSendWait Function

Purpose Waits until all previous data has been sent from the transmit FIFO, then returns.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmSendWait (uint16_t portId)
```

Parameters → *`portID`*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrTimeOut`
Unable to send data within the CTS timeout period.

`serErrNotSupported`
The port is not the foreground port.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments Consider calling this function if your software needs to detect when all data has been transmitted by `SrmSend()`. The `SrmSend()` function blocks until all data has been transmitted or a timeout occurs. A subsequent call to `SrmSendWait()` blocks until all data queued up for transmission has been transmitted or until another CTS timeout occurs (if CTS handshaking is enabled).

See Also `SrmSend()`, `SrmSendCheck()`, `SrmSendFlush()`

SrmSetReceiveBuffer Function

Purpose Installs a new buffer into the Serial Manager's receive queue.

Declared In `SerialMgr.h`

Prototype

```
status_t SrmSetReceiveBuffer (uint16_t portId,
    void *bufP, uint16_t bufSize)
```

Parameters

→ *portID*
: Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *bufP*
: Pointer to new receive buffer. Ignored if *bufSize* is `NULL`.

→ *bufSize*
: Size of new receive buffer in bytes. To remove this buffer and allocate a new default buffer (512 bytes), specify 0.

Returns This function returns the following error codes:

`errNone`
: No error.

`serErrBadPort`
: This port doesn't exist.

`serErrNotOpen`
: This port is not open.

`memErrNotEnoughSpace`
: Not enough memory to allocate default buffer.

`serErrNoDevicesAvail`
: No serial devices could be found.

Comments The buffer that you pass to this function must remain allocated while you have the serial port open. Before you close the serial port, you must restore the default queue by calling `SrmSetReceiveBuffer()` with `NULL` as the *`bufP`* and 0 as the *`bufSize`* parameter.

IMPORTANT: Applications must install the default buffer before closing the port (or disposing of the new receive queue).

SrmSetWakeupHandler Function

Purpose Installs a wakeup handler.

Declared In `SerialMgr.h`

Prototype
```
status_t SrmSetWakeupHandler (uint16_t portId,
   WakeupHandlerProcPtr procP, uint32_t refCon)
```

Parameters

→ *`portID`*
Port ID returned from `SrmOpen()` or `SrmExtOpen()`.

→ *`procP`*
Pointer to a `WakeupHandlerProcPtr()` function. Specify `NULL` to remove a handler.

→ *`refCon`*
User-defined data that is passed to the wakeup handler function. This can any 32-bit value, including a pointer.

Returns This function returns the following error codes:

`errNone`
No error.

`serErrBadPort`
This port doesn't exist.

`serErrNotOpen`
The port is not open.

`serErrNoDevicesAvail`
No serial devices could be found.

Comments The wakeup handler is a function in your application that you want to be called whenever there is data ready to be received on the specified port.

The wakeup handler function will not become active until it is primed with a number of bytes that is greater than 0, by the `SrmPrimeWakeupHandler()` function. Every time a wakeup handler is called, it must be re-primed (using `SrmPrimeWakeupHandler()`) in order to be called again.

See Also `SrmPrimeWakeupHandler()`, `WakeupHandlerProcPtr()`

Serial Manager Application-Defined Functions

WakeupHandlerProcPtr Function

Purpose Called after some number of bytes are received by the Serial Manager's interrupt function.

Declared In `SerialMgr.h`

Prototype `void (*WakeupHandlerProcPtr)(uint32_t refCon)`

Parameters → *refCon*

User-defined data passed from the `SrmSetWakeupHandler()` function.

Returns Returns nothing.

Comments This handler function is installed by calling `SrmSetWakeupHandler()`. The number of bytes after which it is called is specified by `SrmPrimeWakeupHandler()`.

Under Palm OS Cobalt, the wakeup handler is called from a thread in the application's process. Because of this, it's possible that the handler can be called while the application is already calling a Serial Manager function.

If your application manages synchronization between the wakeup handler and its main thread, it can call Serial Manager functions within the wakeup handler. However, if your needs are complex, or you want to maximize performance, you may benefit from using the IOS API instead of the Serial Manager.

Two common implementations of wakeup handlers include:

- Calling `EvtWakeup()`, which causes any pending `EvtGetEvent` call to return and then sends a `nilEvent` to the current application.
- Using `SrmReceiveWindowOpen()` and `SrmReceiveWindowClose()` to gain direct access to the receive queue without blocking.

See Also `SrmPrimeWakeupHandler()`, `SrmSetWakeupHandler()`

5

Serial Link Manager

This chapter provides reference material for the Serial Link Manager API. The header file `SerialLinkMgr.h` declares the Serial Link Manager API. For more information on the Serial Link Manager, see Chapter 3, "The Serial Link Protocol," on page 21.

This API is defined in the header file `SerialLinkMgr.h`.

Serial Link Manager Functions

SlkClose Function

Purpose Close down the Serial Link Manager.

Prototype `status_t SlkClose (void)`

Parameters None.

Returns

`errNone`
: No error.

`slkErrNotOpen`
: The Serial Link Manager was not open.

Comments When the open count reaches zero, this routine frees resources allocated by Serial Link Manager.

SlkCloseSocket Function

Purpose Closes a socket previously opened with `SlkOpenSocket()`.

The caller is responsible for closing the communications library used by this socket, if necessary.

Prototype

```
status_t SlkCloseSocket (UInt16 socket)
```

Parameters

→ *`socket`*
The socket ID to close.

Returns

`errNone`
No error.

`slkErrSocketNotOpen`
The socket was not open.

Comments `SlkCloseSocket()` frees system resources the Serial Link Manager allocated for the socket. It does not free resources allocated and passed by the client, such as the buffers passed to `SlkSetSocketListener()`; this is the client's responsibility. The caller is also responsible for closing the communications library used by this socket.

See Also SlkOpenSocket()

SlkFlushSocket Function

Purpose Flush the receive queue of the communications library associated with the given socket.

Prototype

```
status_t SlkFlushSocket (UInt16 socket,
    Int32 timeout)
```

Parameters

→ *`socket`*
Socket ID.

→ *`timeout`*
Interbyte timeout in system milliseconds.

Returns `errNone`
No error.

`slkErrSocketNotOpen`
The socket wasn't open.

SlkOpen Function

Purpose Initialize the Serial Link Manager.

Prototype

```
status_t SlkOpen (void)
```

Parameters None.

Returns `errNone`
No error.

`slkErrAlreadyOpen`
No error.

Comments Initializes the Serial Link Manager, allocating necessary resources. Return codes of 0 (zero) and `slkErrAlreadyOpen` both indicate success. Any other return code indicates failure. The `slkErrAlreadyOpen` function informs the client that someone else is also using the Serial Link Manager. If the Serial Link Manager was successfully opened by the client, the client needs to call `SlkClose()` when it finishes using the Serial Link Manager.

SlkOpenSocket Function

Purpose Open a serial link socket and associate it with a communications library. The socket may be a known static socket or a dynamically assigned socket.

Prototype

```
status_t SlkOpenSocket (UInt16 portID,
    UInt16 *socketP, Boolean staticSocket)
```

Parameters → *portID*
Comm library reference number for socket.

↔ *socketP*
Pointer to location for returning the socket ID. If *staticSocket* is `true`, then on entry this contains the desired static socket number to open.

→ *staticSocket*
If `true`, *socketP* contains the desired static socket number to open. If `false`, any free socket number is assigned dynamically and opened.

Returns `errNone`
No error.

`slkErrOutOfSockets`
No more sockets can be opened.

Comments The communications library must already be initialized and opened (see `SrmOpen()`). When finished using the socket, the caller must call `SlkCloseSocket()` to free system resources allocated for the socket. For information about well-known static socket IDs, see Chapter 3, "The Serial Link Protocol," on page 21.

SlkReceivePacket Function

Purpose Receive and validate a packet for a particular socket or for any socket. Check for format and checksum errors.

Prototype
```
status_t SlkReceivePacket (UInt16 socket,
   Boolean andOtherSockets,
   SlkPktHeaderPtr headerP, void *bodyP,
   UInt16 bodySize, Int32 timeout)
```

Parameters → *socket*
The socket ID.

→ *andOtherSockets*
If `true`, ignore destination in packet header.

↔ *headerP*
Pointer to the packet header buffer (size of `SlkPktHeaderType`). Note that the header is in big-endian byte order.

↔ *bodyP*
Pointer to the packet client data buffer.

→ *bodySize*
Size of the client data buffer (maximum client data size which can be accommodated).

→ *timeout*
Maximum number of system ticks to wait for beginning of a packet; -1 means wait forever.

Returns `errNone`
No error.

`slkErrSocketNotOpen`
The socket was not open.

`slkErrTimeOut`
Timed out waiting for a packet.

`slkErrWrongDestSocket`
The packet being received had an unexpected destination.

`slkErrChecksum`
Invalid header checksum or packet CRC-16.

`slkErrBuffer`
Client data buffer was too small for packet's client data.

If *andOtherSockets* is `false`, this routine returns with an error code unless it gets a packet for the specific socket.

If *andOtherSockets* is `true`, this routine returns successfully if it sees any incoming packet from the communications library used by `socket`.

Comments You may request to receive a packet for the passed socket ID only, or for any open socket which 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. If a packet is received for a socket with a registered socket listener, it will be dispatched via its socket listener procedure. On success, the packet header buffer and packet client data buffer is filled in with the actual size of the packet's client data in the packet header's `bodySize` field.

SlkSendPacket Function

Purpose Send a serial link packet via the serial output driver.

Prototype

```
status_t SlkSendPacket (SlkPktHeaderPtr headerP,
    SlkWriteDataPtr writeList)
```

Parameters

↔ *headerP*
: Pointer to the packet header structure with client information filled in (see Comments).

→ *writeList*
: List of packet client data blocks (see Comments).

Returns

`errNone`
: No error.

`slkErrSocketNotOpen`
: The socket was not open.

`slkErrTimeOut`
: Handshake timeout.

Comments `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). This call blocks until the entire packet is sent out or until an error occurs.

SlkSetSocketListener Function

Purpose Register a socket listener for a particular socket.

Prototype

```
status_t SlkSetSocketListener (UInt16 socket,
   SlkSocketListenPtr socketP)
```

Parameters

→ *`socket`*
Socket ID.

→ *`socketP`*
Pointer to a `SlkSocketListenType` structure.

Returns

`errNone`
No error.

`slkErrBadParam`
Invalid parameter.

`slkErrSocketNotOpen`
The socket was not open.

Comments Called by applications to set up a socket listener.

Since the Serial Link Manager does not make a copy of the `SlkSocketListenType` structure, but instead saves the passed pointer to it, the structure

- must **not** be an automatic variable (that is, local variable allocated on the stack)
- may be a global variable in an application
- may be 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: the packet header buffer (size of `SlkPktHeaderType`), and the packet body (client data) buffer. The packet body buffer must be large enough for the largest expected client data size. Both buffers may 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.

NOTE: The application is responsible for freeing the `SlkSocketListenType` structure or the allocated buffers when the socket is closed. The Serial Link Manager doesn't do it.

SlkSocketPortID Function

Purpose Get the port ID associated with a particular socket; for use with the new serial manager.

Prototype

```
status_tSlkSocketPortID (UInt16 socket,
   UInt16 *portIDP)
```

Parameters

→ *socket*
The socket ID.

↔ *portIDP*
Pointer to location for returning the port ID.

Returns

`errNone`
No error.

`slkErrSocketNotOpen`
The socket was not open.

SlkSocketSetTimeout Function

Purpose Set the interbyte packet receive-timeout for a particular socket.

Declared In `SerialLinkMgr.h`

Prototype
```
status_t SlkSocketSetTimeout (UInt16 socket,
    Int32 timeout)
```

Parameters
→ *socket*
Socket ID.

→ *timeout*
Interbyte packet receive-timeout in system ticks.

Returns
`errNone`
No error.

`slkErrSocketNotOpen`
The socket was not open.

Part II
Infrared Communication (Beaming)

Palm OS® provides a robust infrared communication architecture using the IrDA standard. This part of Exploring Palm OS: Low-Level Communications covers making use of infrared communication in your applications.

6

Introduction to Infrared Communication (Beaming)

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 Chapter 4, "Object Exchange," on page 105 of *Exploring Palm OS: High-Level Communications* 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 Chapter 2, "The Serial Manager," on page 5 for information on how to use the Serial Manager APIs.
- The Sockets API lets you use the same functions you would use for other communications methods to perform IR communications.

This chapter focuses on using the Sockets API for beaming.

IMPORTANT: Versions of Palm OS prior to 6.0 offered a separate library, called IRLib, for performing infrared communications. This library has been deprecated and should not be used when creating new applications.

The IR support provided by 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/

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, and 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.

7

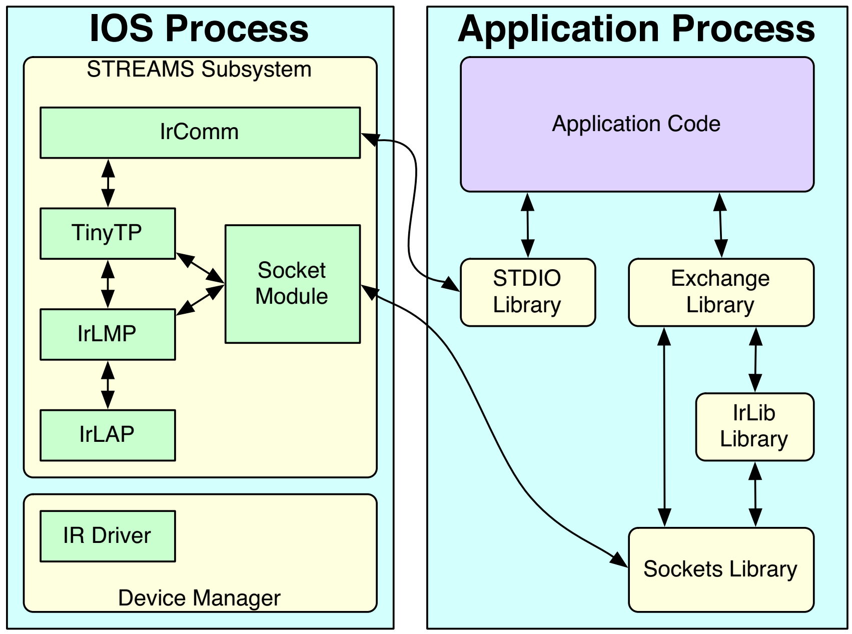
The IrDA Protocol Stack

The IrDA protocol stack serves primarily as a transport for the Exchange Manager, which uses the **Infrared Object Exchange Protocol** (IrOBEX) to transfer data objects between devices. IrOBEX is built on top of the **TinyTP** protocol.

Additionally, the IrDA protocol stack's IrComm module provides a serial interface that allows the transfer of data over the infrared media. This interface is implemented as a STREAMS module, which lets it be used both by new applications and "legacy" applications that have no specific knowledge of the underlying infrared media. As an example, HotSync® uses IrComm to enable synchronizing with an IrDA-equipped PC.

Figure 7.1 on page 78 shows the organization of the IrDA protocol modules and how they interface with the rest of Palm OS®.

Figure 7.1 The IrDA protocol stack and how it interfaces with the rest of the system



The IrDA protocol stack is available to applications through the standard sockets API. Some additional IrDA-specific functionality is provided through a new shared library, IrDALib.

NOTE: The IrLib provided by Palm OS Garnet and earlier versions of Palm OS has been deprecated, and is no longer available to native ARM applications. However, a compatibility library exists to allow 68k "legacy" applications to continue to use IrLib. New software must, however, be written to the new IrDALib API.

8

Using the IrDA Protocols

The IrLAP Protocol Layer

IrLAP, the **Infrared Link Access Protocol**, lies at the bottom of the IrDA protocol stack. It provides a reliable, sequenced exchange of frames between two IrDA-capable devices, as well as a process for detecting (or "discovering") other nearby IrDA devices.

Palm OS® Cobaltdoes not allow applications to directly interface with the IrLAP protocol layer. IrLAP connections and discoveries are managed entirely by the IrLMP protocol.

The IrLMP Protocol Layer

The IrLMP (**Infrared Link Management Protocol**) layer sits just above IrLAP in the IrDA protocol stack. It serves as a multiplexer on an IrLAP connection, allowing multiple concurrent "conversations" between a pair of connected devices. Each conversation consists of a sequenced stream of reliably-delivered messages; the messages are guaranteed to be delivered to applications in the same order in which they were sent. Message boundaries are preserved.

In addition, IrLMP provides an exclusive mode, in which a single conversation can take full control of the underlying IrLAP connection, locking out all others. This is useful for applications that require a reduced-latency connection. Exclusive mode can be controlled using the `SO_IREXCLUSIVE` `setsockopt()` command.

The IrLMP Sequenced Packet Interface

The IrLMP sequenced packet interface does not provide any segmentation or reassembly of large messages, so the maximum size of incoming and outgoing messages is limited to the data sizes negotiated during the establishment of the IrLAP connection. IrLMP also provides no end-to-end flow control; when data is sent through an IrLMP socket faster than it can be consumed at the other end, messages will be discarded by the receiver's IrLMP protocol, thereby being lost to the receiving application.

Listing 8.1 demonstrates how to create and use IrDA socket connections. This example creates a socket and listens for an incoming connection. Once a connection is initiated, the code creates a new socket for sending data to the remote device. Once that's been done, the code makes the first socket idle by calling `setsockopt()` with the `SO_IRIDLE` command, gives the new socket exclusive control with the `SO_IREXCLUSIVE` command, and transmits data.

Once the data has been transmitted, the data transfer socket is closed and the control socket is reactivated.

Listing 8.1 Creating and using IrDA socket connections

```
int s1, s2, s3;
struct sockaddr_irda addr;
int mtu, len;
char *buf;
int zero = 0;
int one = 1;

// create an IrLMP socket, and wait for someone to call us
s1 = socket(AF_IRDA, SOCK_SEQPACKET, IRPROTO_LMP);

memset(&addr, 0, sizeof(addr));
addr.sir_family = AF_IRDA;
addr.sir_lsap = 0x69;

bind(s1, (struct sockaddr *)&addr, sizeof(addr));
listen(s1, 1);
s2 = accept(s1, NULL, NULL);

// initiate another IrLMP socket connection to our caller
s3 = socket(AF_IRDA, SOCK_SEQPACKET, IRPROTO_LMP);

addr.sir_lsap = 0x12;
addr.sir_addr = IRADDR_ANY;
connect(s3, (struct sockaddr *)&addr, sizeof(addr));
```

```
// mark first connection idle
setsockopt(s2, SOL_SOCKET, SO_IRIDLE, (const char *)&one, sizeof(one));

// take exclusive control of IrLAP connection
setsockopt(s3, SOL_SOCKET, SO_IREXCLUSIVE, (const char *)&one, sizeof(one));

// send a maximum-sized message
len = sizeof(mtu);
getsockopt(s3, SOL_SOCKET, SO_IRMTU, (const char *)&mtu, &len);
buf = malloc(mtu);
memset(buf, 0xff, mtu);
send(s3, buf, mtu, 0);

// close second connection, thereby relinquishing exclusive control
close(s3);

// the first connection can now become active
setsockopt(s2, SOL_SOCKET, SO_IRIDLE, (const char *)&zero, sizeof(zero));
```

The IrLMP Datagram Interface

IrLMP provides a datagram interface for connectionless data exchange. Datagram messages are sent unreliably; it is not possible for the sending application to be certain that any other device has received a sent message. Additionally, datagram messages are always broadcast, so they may be received by any and all IrDA-capable devices within range of the sending device.

Listing 8.2 demonstrates how to broadcast a datagram and wait for a reply.

Listing 8.2 Broadcasting a datagram message

```
int s;
sockaddr_irda addr;
char buf[40];
int len;

// create IrLMP datagram socket
s = socket(AF_IRDA, SOCK_DGRAM, 0);

// bind to IrLMP's connectionless SAP
memset(&addr, 0, sizeof(addr));
addr.sir_family = AF_IRDA;
addr.sir_lsap = irLsapUnitdata;
```

```
if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) != 0) {
   // another application beat us to it
   return;
}

// broadcast datagram
memset(&buf, 0x69, sizeof(buf));
addr.sir_addr = IRADDR_BROADCAST;
sendto(s, buf, sizeof(buf), 0, (struct sockaddr *)&addr, sizeof(addr));

// listen for response
len = sizeof(addr);
recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *)&addr, &len);
```

Discovering IrDA Devices

Before you can establish a connection to an IrDA device, you have to find it. This is done by discovering the available devices via the `IrDADiscoverDevices()` function in the IrDALib shared library. The sample code in Listing 8.3 demonstrates this process.

Listing 8.3 Discovery of IrDA devices

```
   int s;
   uint32_t nLogs;
   IrLmpDeviceInfoType logs[2];
   Boolean cached;
   int i, j;
   struct sockaddr_irda addr;

   // perform discovery
   nLogs = 2;
   IrDADiscoverDevices(&nLogs, logs, &cached));

   printf("discovery found %d %sdevices:\n", nLogs, (cached ? "cached " : ""));
   for (i = 0; i < nLogs; i++) {
      printf("   %08x ", logs[i].deviceAddr);
      switch (logs[i].method) {
      case kIirLmpSniffing:
         printf("sniffed");
         break;

      case kIirLmpActiveDiscovery:
         printf("discovered");
         break;
```

```
    case kIirLmpPassiveDiscovery:
      printf("found us");
      break;
    }

    printf(" %d bytes of device info: ", logs[i].infoLen);
    for (j = 0; j < logs[i].infoLen; j++)
      printf("%c", logs[i].deviceInfo[j]);
    printf("\n");
  }

  if (nLogs == 0) {
    printf("found no devices - aborting test\n");
    return;
  }

  // open irlmp socket
  s = socket(AF_IRDA, SOCK_STREAM, 0);

  // connect to first device discovered
  memset(&addr, 0, sizeof(addr));
  addr.sir_family = AF_IRDA;
  strcpy(addr.sir_name, "OBEX");
  addr.sir_addr = logs[0].deviceAddr;

  connect(s, (struct sockaddr *)&addr, sizeof(addr));
```

This code discovers the available devices, printing out information about them, then connects to the first device discovered by creating a new socket and connecting to the socket using OBEX on a STREAM based socket.

The TinyTP Protocol Layer

The TinyTP Sequenced Packet Interface

The **Tiny Transport Protocol** (TinyTP) sits on top of IrLMP in the IrDA protocol stack. It builds upon the functionality of the IrLMP sequenced packet interface by providing segmentation and reassembly of large messages, as well as end-to-end flow control for individual IrLMP connections.

The code in Listing 8.4 shows how to create a socket to listen for a sequenced TinyTP connection, enable automatic reassembly of incoming messages, determine the connection's MTU, and mark the control socket as idle. Once this code is done executing, it's time to transfer data as seen in Listing 8.1.

Listing 8.4 Setting up a TinyTP sequenced packet connection

```
int s1, s2;
struct sockaddr_irda addr;
int mru, mtu, len;
int one = 1;

// create sequenced TTP socket
s1 = socket(AF_IRDA, SOCK_SEQPACKET, 0);

// bind to well-known lsap
memset(&addr, 0, sizeof(addr));
addr.sir_family = AF_IRDA;
addr.sir_lsap = irLsapAny;
strncpy(addr.sir_name, "IrTest", sizeof(addr.sir_name));
bind(s1, (struct sockaddr *)&addr, sizeof(addr));

// enable automatic re-assembly of incoming messages
mru = 2345;
setsockopt(s1, SO_IRMRU, (const char *)&mru, sizeof(mru));

// wait for someone to connect to us
listen(s1, 1);
s2 = accept(s1, NULL, NULL);

// retrieve connection MTU
len = sizeof(mtu);
getsockopt(s2, SOL_SOCKET, SO_IRMTU, (const char *)&mtu, &len);

// mark connection idle
setsockopt(s2, SOL_SOCKET, SO_IRIDLE, (const char *)&one, sizeof(one));
```

The TinyTP Stream Interface

In addition to the sequenced packet interface, TinyTP provides a stream interface, in which it manages all aspects of segmentation and reassembly for the application. Applications can ignore the TinyTP MTU negotiated for the connection and send messages of any length. The TinyTP stream interface will take care of segmenting them as necessary, so attempting to send a message larger than the MTU will not result in failure.

NOTE: When using the TinyTP stream interface, message boundaries are not preserved end-to-end. The TinyTP protocol module may break messages into multiple pieces, or combine multiple small messages into a single larger message.

This is the easiest way to send data, as you can see from Listing 8.5. All you do is open the connection and send and receive data on it. No worrying about packet sizes or sequencing; it just works.

Listing 8.5 Sending and receiving data using a TinyTP stream

```
int s;
struct sockaddr_irda addr;
int mtu, len;
char *buf;
int i;
char c;

// create TinyTP stream socket
s = socket(AF_IRDA, SOCK_STREAM, IRPROTO_TTP);

// connect to anyone
memset(&addr, 0, sizeof(addr));
addr.sir_family = AF_IRDA;
strncpy(addr.sir_name, "IrTest", sizeof(addr.sir_name));
addr.sir_addr = IRADDR_BROADCAST;
connect(s, (struct sockaddr *)&addr, sizeof(addr));

// retrieve connection MTU
len = sizeof(mtu);
getsockopt(s, SOL_SOCKET, SO_IRMTU, (const char *)&mtu, &len);

// send lots of data
buf = malloc(mtu * 2);
memset(buf, 0x69, mtu * 2);
send(s, buf, mtu * 2, 0);

// read response
for (i = 0; i < 200; i++)
  recv(s, &c, sizeof(c), 0);
```

Getting and Providing Information About IrDA Services

Every device with an IrDA protocol stack includes the **Information Access Service** (IAS), which consists of a directory listing all of the IrDA services that the device offers, as well as server software that allows other devices to access this directory. The IAS query interface provides a method for discovering services offered by a remote device, by querying its IAS directory server.

Structure of the IAS Database

Entries in the IAS database consist of IAS objects, each of which describes a single service offered by the device. These objects consist of sets of attributes, typed name-value pairs containing specific information about the service. Additionally, each object contains a class name, which is a string that describes the object's type. The type indicates what attributes the object includes.

Getting Information about IrDA Services

The `IASGetValueByClass()` function lets an application ask a remote device for attribute values with a given name belonging to a given class. The `connect()` and `bind()` functions automatically handle some aspects of IAS and provide a simplified interface to it, as shown in "The IrLMP Protocol Layer" on page 79.

Listing 8.6 Querying IAS

```
IASQueryType query;
uint8_t buffer[256];

// look for an IrOBEX server on any remote device (the class and attribute
// names are well-known, specified by the IrOBEX standard document)
query.addr = IRADDR_BROADCAST;
query.className = "OBEX";
query.attribName = "IrDA:TinyTP:LsapSel";
query.resultBuf = buffer;
query.resultBufLen = sizeof(buffer);
if (IASQueryValueByClass(&query) == errNone &&
    query.attribCount > 0 &&
    query.attribValues[0]->attribType == kIASiasAttribIntegerAttrib)
```

```
{
  printf("Found IrOBEX server on device %08lx, at LSAP %04x\n",
       query.addr, query.attribValues[0]->value.integer);
}
```

Providing Information About Offered IrDA Services

To advertise the existence of an IrDA service, an application needs to add the service to its IAS directory. This can be done using the `bind()` function, which automatically creates the IAS entry, or manually as shown in Listing 8.7.

The `IASRegisterObject()` function associates an IAS service entry with an IrDA socket. This is generally used by a server application to inform remote devices of the location and type of services the application offers. The function returns an object ID that uniquely identifies the object. This object ID can be used to unregister the service later through a call to `IASUnregisterObject()`.

An easier method of associating a service name with a server socket is provided by the `IASRegisterService()` function. This function creates an IAS entry of a service class specified when calling it, containing a single entry specifying the LSAP address of the specified socket. This attribute's name will correctly reflect the type of socket. For example, if a TinyTP socket is specified, the attribute's name will be "`IrDA:TinyTP:LsapSel`".

Once registered, a service entry remains in the device's IAS database until either the socket with which it is associated is unbound, or the `IASUnregisterObject()` function is called with its ID.

Listing 8.7 Registering a service with IAS

```
#define N_ATTRIBS 2
IASObjectType obj;
const char *names[N_ATTRIBS]  = {
  "IrDA:IrLMP:InstanceName",
  "IrDA:IrLMP:LsapSel"
};
IASAttribValueType *values[N_ATTRIBS];
```

```
uint8_t buf1[10], buf2[20];

// fill out attribute values
values[0] = (IASAttribValueType *)buf1;
values[0]->attribType = iasAttribInteger;
values[0]->value.integer = <listener socket's LSAP>;

values[1] = (IASAttribValueType *)buf2;
values[1]->attribType = kIASiasAttribUserStringAttrib;
values[1]->value.userString.charSet = iasAttribUserString;
strcpy(values[1]->value.userString.chars, "Bar");

// advertise service on our listener socket
obj.className = "Foo";
obj.attribCount = N_ATTRIBS;
obj.attribNames = names;
obj.attribValues = values;
if (IASRegisterObject(<listener socket>, &obj, false) == errNone) {
   // our object is now in this device's IAS information base, and will
   // remain there until <listener socket> is closed
}
```

This code creates an array of IAS attribute structures (of type `IASAttribValueType`) and fills each of them out with information describing the attribute. It then sets up an IAS object (of type `IASObjectType`) and calls `IASRegisterObject()` to register the service.

9

IrDA Reference

IrDA Constants

IASAttribTypeType Enum

Purpose Define IAS attribute types.

Declared In `IAS.h`

Constants

Constant	Definition
`iasAttribMissing`	The attribute is missing.
`iasAttribInteger`	The attribute is an integer.
`iasAttribOctetString`	The attribute is a string of bytes.
`iasAttribUserString`	The attribute is a user string.

IASCharSetType Enum

Purpose Define character sets supported by IAS.

Declared In `IAS.h`

Constants

Constant	Definition
iasCharSetASCII	ASCII.
iasCharSetISO8859_1	ISO 8859-1.
iasCharSetISO8859_2	ISO 8859-2.
iasCharSetISO8859_3	ISO 8859-3.

Constant	Definition
`isaCharSetISO8859_4`	ISO 8859-4.
`isaCharSetISO8859_5`	ISO 8859-5.
`isaCharSetISO8859_6`	ISO 8859-6.
`isaCharSetISO8859_7`	ISO 8859-7.
`isaCharSetISO8859_8`	ISO 8859-8.
`isaCharSetISO8859_9`	ISO 8859-9.
`isaCharSetUnicode`	Unicode.

IrDA Protocol Identifier Constants

Purpose Define IrDA protocols when using the Sockets API.

Declared In `IrDA.h`

Constants

Constant	Definition
IRPROTO_LAP	The IrLAP protocol.
IRPROTO_LMP	The IrLMP protocol.
IRPROTO_TTP	The TinyTP protocol.

IrDA Socket Address Family Constant

Purpose Defines the address family used by IrDA socket connections.

Declared In `posix/sys/socket.h`

Constants `AF_IRDA`

Comments When creating a socket for use in IrDA communications, use the `AF_IRDA` constant to indicate that the address is an IrDA device address.

IrLmpDiscoveryMethodType Enum

Purpose Define values for the discovery method specified in the `IrLmpDeviceInfoType` structure.

Declared In `IrDA.h`

Constants

Constant	Definition
`irLmpSniffing`	The device was discovered while it was performing an IrLAP sniffing procedure, which is a special low-power discovery procedure.
`irLmpActiveDiscovery`	The device was discovered during an IrLAP discovery procedure initiated by the current device.
`irLmpPassiveDiscovery`	The device was discovered as the result of an IrLAP discovery procedure initiated by the discovered device.

IAS Constants

Purpose Constants pertaining to IAS.

Declared In `IAS.h`

Constants

Constant	Definition
`iasMaxAttribNameLen`	The maximum number of characters in an IAS attribute's name.
`iasMaxClassNameLen`	The maximum number of characters in an IAS object's class name.

Constant	Definition
`iasMaxOctetStringLen`	The maximum number of bytes in an octet-string IAS attribute.
`iasMaxUserStringLen`	The maximum number of characters in an IAS user-string attribute.

setsockopt() commands

Purpose Define IrDA-specific `setockopt()` commands.

Declared In `IrDA.h`

Constants

Constant	Definition
`SO_IRIDLE`	Invokes the `LM_Idle.Request`.
`SO_IREXCLUSIVE`	Invokes the `LM_AccessMode.Request`.
`SO_IRMTU`	Gets the MTU for the socket.
`SO_IRMRU`	Sets or gets the socket's MRU.

Special IrDA Device Addresses

Purpose Define special-purpose IrDA device addresses.

Declared In `IrDA.h`

Constants

Constant	Definition
`IRADDR_ANY`	Refers to any available device.
`IRADDR_BROADCAST`	The IrDA device broadcast address.

NOTE: Connecting to `IRADDR_BROADCAST` causes IrLMP to automatically perform a discovery procedure, then connect to one of the devices it discovers. Connecting to `IRADDR_ANY` will initiate an IrLMP connection to whatever device is currently at the other end of an established IrLAP connection; if IrLAP isn't already connected, the `connect()` will fail.

Special IrLMP SAP Values

Purpose Describe Service Access Points (SAPs) to which connection-oriented sockets can be bound.

Declared In `IrDA.h`

Constants

Constant	Definition
`irLsapIAS`	The IAS query management interface.
`irLsapUnitdata`	The Unitdata interface.
`irLsapAny`	Any SAP.

IrDA Data Types and Structures

IASAttribValueType Typedef

Purpose Describes the type and value of an IAS attribute.

Declared In `IAS.h`

Prototype

```
typedef PACKED struct IASAttribValueTag {
   uint8_t attribType;
   PACKED union {
      uint32_t integer;
      PACKED struct {
         uint16_t length;
         uint8_t bytes[0];
      } octetString;
      PACKED struct {
         uint8_t charSet;
         uint8_t length;
         uint8_t chars[1];
      } userString;
   } value;
} IASAttribValueType;
```

Fields

`attribType`
: An `IASAttribTypeType` value specifying the type of attribute the structure represents.

`value`
: A union containing the value of the attribute, depending on the attribute type:

  `integer`
  : The value of the attribute if it's of type `iasAttribInteger`.

  `octetString`
  : The value of the attribute if it's of type `iasAttribOctetString`. The `length` field indicates the number of bytes in the attribute string, and `bytes` is the string itself.

`userString`
The value of the attribute if it's of type `iasAttribUserString`. The `charSet` field indicates an [`IASCharSetType`] value specifying the character set in which the string is defined, `length` indicates the length of the string in bytes (not characters), and `chars` is a `length`-byte long array containing the string itself.

IASObjectType Struct

Purpose Describes an IAS object.

Declared In `IAS.h`

Prototype

```
typedef struct IASObjectTag {
   const char *className;
   const uint8_t *hintBits;
   uint16_t attribCount;
   const char **attribNames;
   IASAttribValueType **attribValues;
} IASObjectType;
```

Fields

`className`
The user-supplied class name.

`hintBits`
Bits to be included in IrLMP's "IrLMPSupport" attribute.

`attribCount`
The number of attributes included in the object.

`attribNames`
An array of `attribCount` strings naming each of the object's attributes.

`attribValues`
An array of `attribCount` `IASAttribValueType` structures, each describing the value of the corresponding attribute.

IASQueryType Struct

Purpose Describes an IAS query.

Declared In `IAS.h`

Prototype

```
typedef struct IASQueryTag {
   IrLapDeviceAddrType addr;
   const char *className;
   const char *attribName;
   uint8_t *resultBuf;
   uint32_t resultBufLen;
   uint16_t attribCount;
   IASAttribValueType **attribValues;
   uitn16_t *objectIDs;
   uint32_t resultSize;
} IASQueryType;
```

Fields

`addr`
: Address of the device to query. Can be `IRADDR_ANY` or `IRADDR_BROADCAST`.

`className`
: The name of the class to look for.

`attribName`
: The name of the attribute to look for.

`resultBuf`
: A pointer to a buffer to store the results into. This must point to a buffer before using the structure in a query.

`resultBufLen`
: The length of the `resultBuf` buffer in bytes. This must be set to the length of the buffer before using the structure in a query.

`atttribCount`
: The `IASGetValueByClass()` function fills in this value with the number of attributes retrieved as a result of the query.

`attribValues`
: The `IASQueryValueByClass()` function fills this out to point to an array of the retrieved attribute values.

`objectIDs`
The `IASGetValueByClass()` function fills this field out with a pointer to an array of the object IDs of all the found attributes.

`resultSize`
The `IASGetValueByClass()` function fills out this field with the size of the result in bytes.

IrLapDeviceAddrType Typedef

Purpose Specifies the address of an IrDA device.

Declared In `IrDA.h`

Prototype `typedef uint32_t IrLapDeviceAddrType;`

Comments IrDA device addresses are 32-bit integer values. A device's IrDA address is generated internally by its IrDA stack, and can be changed by the stack without warning.

Additionally, a device address' byte order is not defined; currently addresses are in little-endian format, but that could change in the future. Applications should therefore treat device addresses as an opaque identifier with short lifespans. For example, the `sockaddr_irda` structure's `sir_addr` field should only be populated with the results of a discovery procedure, and this should be done soon after the discovery procedure has completed.

There are two special device addresses that are exempt to this rule: `IRADDR_ANY` and `IRADDR_BROADCAST` are never returned by a discovery procedure, and can be used in `sockaddr_irda` at any time.

IrLmpDeviceInfoType Struct

Purpose Provides information describing an IrDA device.

Declared In `IrDA.h`

Prototype

```
typedef struct IrLmpDeviceInfoTag {
   IrLapDeviceAddrType deviceAddr;
   uint8_t method;
   uint8_t __pad0[2];
   uint8_t infoLen;
   uint8_t deviceInfo[32];
} IrLmpDeviceInfoType;
```

Fields

`deviceAddr`
: The 32-bit IrLAP device address of the device described by the structure.

`method`
: The method by which the device was discovered. This will be one of the values specified by the `IrLmpDiscoveryMethodType` enum.

`__pad0`
: Reserved for system use. Do not use this field.

`infoLen`
: The number of bytes of valid data in the *`deviceInfo`* array.

`deviceInfo`
: Information about the device. Usually this is the name of the device, with the first byte indicating the character set in which the name is encoded, followed by a null-terminated string naming the device. Only the first *`infoLen`* bytes contain valid data.

IrLmpSAPType Typedef

Purpose Selects the Service Access Point (SAP) for IrLMP and TinyTP.

Declared In `IrDA.h`

Prototype
```
typedef uint8_t IrLmpSAPType;
```

Comments The `IrLmpSAPType` is an 8-bit value that describes an IrLMP Service Access Point. Constants for special SAPs (the IAS server SAP and the unitdata SAP) are provided.

See Also `IrLmpSAPType`

sockaddr_irda Struct

Purpose Describes an IrLMP address for use with the Sockets API.

Declared In `IrDA.h`

Prototype
```
struct sockaddr_irda {
   sa_family_t sir_family;
   char sir_name[25];
   IrLmpSAPType sir_lsap;
   IrLapDeviceAddrType sir_addr;
};
```

Fields

`sir_family`
The IrDA address family; this must be set to `AF_IRDA`.

`sir_name`
Provides a simplified interface to the IrDA stack's IAS query and IAS entry management functionality. If this field is non-null when the `sockaddr_irda` structure is passed to `bind()`, the IrDA stack will add an IAS entry to the information base and associate it with the specified LSAP. The IAS entry's class will be the value of the `sir_name` field, and the entry will contain a single attribute specifying the socket's LSAP selector. The attribute name will depend on the socket's protocol; for example, an `IRPROTO_LMP` socket would be "`IrDA:IrLMP:LsapSel`".

If the `sockaddr_irda` passed to `connect()` contains a non-null `sir_name` field, the IrDA stack will perform an IAS query to resolve the specified name into an LSAP identifier. A get-value-by-class query is used to perform the resolution; the value of the `sir_name` field specifies which class to query, and the attribute name is based on the socket's protocol. If the `sir_name` field is empty (the first character is '`\0`'), the IrDA stack will simply connect to the remote LSAP specified by the `sir_lsap` field.

`sir_lsap`
Specifies the LSAP the connection should use.

`sir_addr`
Specifies the IrDA device address to which the connection is to be established.

IrDALib Functions

IASGetValueByClass Function

Purpose Queries the Information Access Service (IAS) to find devices matching the specified query parameters.

Declared In `IAS.h`

Prototype
```
status_t IASGetValueByClass(IASQueryType
    *ioQuery)
```

Parameters ↔ *`ioQuery`*
A pointer to an `IASQueryType` structure describing the query to perform. On return, the structure is filled out with information about the results of the query.

Returns `errNone` if no error occurred; otherwise returns an appropriate error code.

IASRegisterObject Function

Purpose Registers an IAS object into the handheld's IAS directory.

Declared In `IAS.h`

Prototype
```
int32_t IASRegisterObject( int iSocket,
    const IASObjectType *iObject,
    Boolean iExclusive )
```

Parameters

→ *`iSocket`*
The socket number of an IrDA socket with which to associate the new IAS service entry.

→ *`iObject`*
The object to register into the IAS directory.

→ *`iExclusive`*
Specify `true` to create an exclusive-access entry in the IAS directory.

Returns Returns the object ID of the newly-registered IAS object. If the returned value is negative, it's an error code.

> **NOTE:** An object ID is always in the range `0x0000` through `0xffff`, so it can be safely cast to a `uint16_t` and passed to `IASUnregisterObject()` as needed.

IASRegisterService Function

Purpose Registers a service name in the handheld's IAS directory.

Declared In `IAS.h`

Prototype
```
int32_t IASRegisterService( int iSocket,
    const char *iServiceClass,
    const uint8_t *iHintsBits,
    Boolean iExclusive )
```

Parameters

→ *`iSocket`*
The socket number of an IrDA socket with which to associate the new IAS service entry.

→ *`iServiceClass`*
The name of the service class to register with IAS.

→ *`iHintsBits`*
Bits to be included in IrLMP's "IrLMPSupport" attribute in the new IAS entry.

→ *`iExclusive`*
Specify `true` to create an exclusive-access entry in the IAS directory.

Returns Returns the object ID of the newly-created IAS directory entry. If the returned value is negative, it's an error code.

NOTE: An object ID is always in the range `0x0000` through `0xffff`, so it can be safely cast to a `uint16_t` and passed to `IASUnregisterObject()` as needed.

IASUnregisterObject Function

Purpose Unregisters an IAS object from the IAS directory.

Declared In `IAS.h`

Prototype
```
status_t IASUnregisterObject( int iSocket,
   uint16_t iObjectID )
```

Parameters

→ *`iSocket`*
The socket number of the IrDA service to be unregistered.

→ *`iObjectID`*
The object ID of the service to unregister, as returned by either `IASRegisterObject()` or `IASRegisterService()`.

IrDADiscoverDevices Function

Purpose Discovers available IrDA devices within range of the handheld's IrDA transceiver.

Declared In `IrDALib.h`

Prototype
```
status_t IrDADiscoverDevices(
   uint32_t *ioNumLogs,
   IrLmpDeviceInfoType *oLogs, Boolean *oCached )
```

Parameters

↔ *ioNumLogs*
On input, this should point to an integer indicating the size of the *oLogs* array. On output, this value has been changed to indicate how many devices were successfully discovered.

← *oLogs*
Points to an array of `IrLmpDeviceInfoType` structures that the function will fill out with information about the discovered devices.

← *oCached*
On return, indicates whether the returned device list is from the cache (`true`) or a fresh discovery procedure (`false`).

Returns `errNone` if no error. Otherwise an appropriate error code.

Comments Discovery is performed through the IrLMP protocol module. If discovery is requested while an IrLAP link is established, IrLMP will return the results of the last discovery procedure, and *oCached* will be set to `true`. If there is not an established link, `IrDADiscoverDevices()` will perform a new discovery procedure and will return `false` in the *oCached* field.

NOTE: IrLMP will never find more than six devices, so you don't need to look for more than that.

Although the IrDA specification doesn't require it, the deviceInfo field in a discovery log usually contains the name of the IrDA device. On a Palm OS device, this is the HotSync® ID assigned by the user the first time the device is synchronized. This name is usually presented as a null-terminated string and is prefixed with a byte indicating the string's character set.

Part III Bluetooth

Palm OS® provides extensive support for Bluetooth, which can be used for serial-style communications, BSD Sockets communication, and object exchange. The following chapters cover developing applications that use Bluetooth for communications.

10

The Palm OS Bluetooth System

The Bluetooth APIs provide developers a way to access the Palm OS® Bluetooth system and write Bluetooth-enabled applications.

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.

NOTE: Palm OS supports version 1.2 of the Bluetooth specification; however, no Bluetooth 1.2 specific features have been exposed in the API at this time. However, two Bluetooth 1.2 compliant devices communicating with each other gain some performance advantages that are transparent to both the developer and to the user.

Capabilities of the Palm OS Bluetooth System

The Palm OS Bluetooth system enables a Palm Powered™ device 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.
- act as a Bluetooth modem, thus providing a gateway to the Internet for other Bluetooth devices.
- use a Bluetooth headset.
- be controlled by a Bluetooth hands-free device.

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.

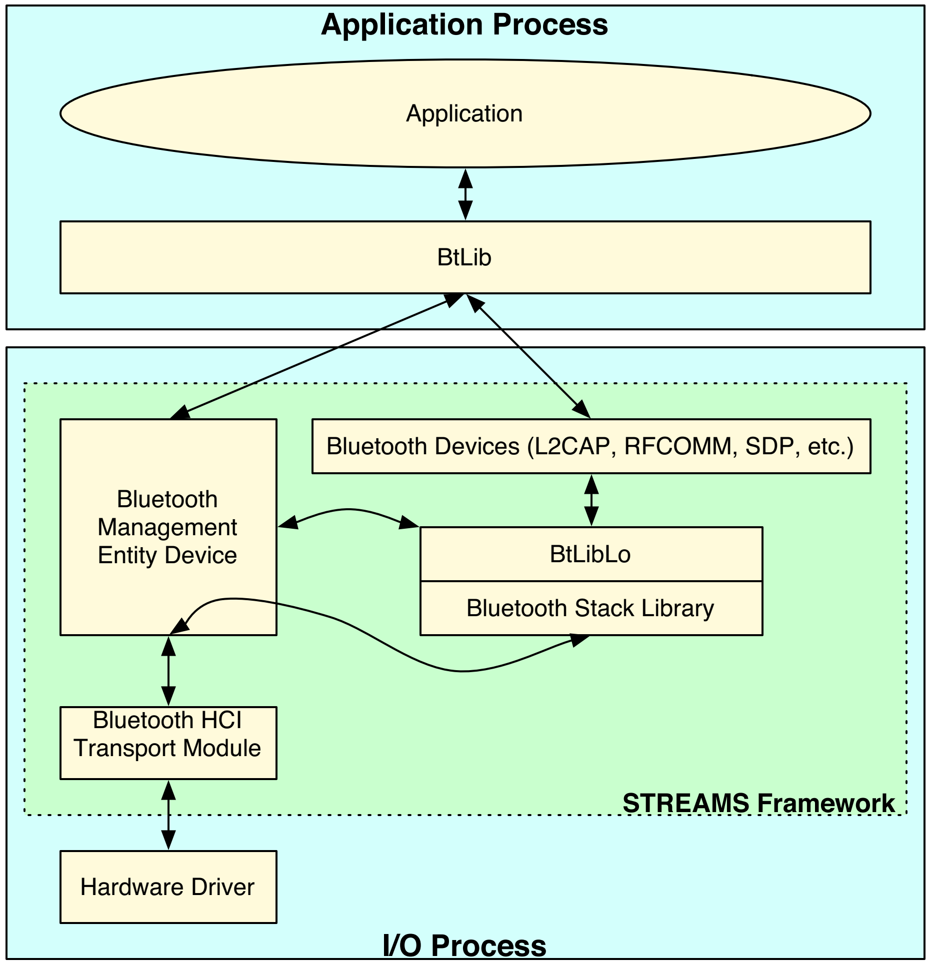
Bluetooth System Components

The Palm OS Bluetooth system contains the following components, which are built on top of the I/O Subsystem:

- Bluetooth Library
- Bluetooth Exchange Library
- Bluetooth Stack Library
- Bluetooth Devices
- Bluetooth HCI Transport Modules
- Hardware Device Drivers

This hierarchy is shown in Figure 10.1.

Figure 10.1 The hierarchy of the Palm OS Bluetooth system



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, RFCOMM, and BNEP protocols, as well as SCO links
- Advertising services and querying for remote services using SDP

- Maintaining a list of favorite devices, some or all of which may be paired with the local device
- Managing persistent service applications

The Bluetooth Library is actually split into two libraries. Applications, which run outside the I/O Process, link to BtLib, while BtLibLo executes within the I/O Process as part of the STREAMS framework.

Bluetooth Exchange Library

The Bluetooth Exchange Library allows applications to use the Palm OS Exchange Manager with Bluetooth as the link. 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 Library

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 Devices

Bluetooth Devices are STREAMS drivers for the Management Entity, and for the L2CAP, RFCOMM, BNEP, and SDP protocols, as well as for SCO links.

Bluetooth HCI Transport Modules

Bluetooth HCI Transport Modules are STREAMS modules that provide an interface between the STREAMS architecture and the lower-level, hardware device driver for the radio.

Hardware Device Drivers

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

Profiles

Table 10.1 lists the profiles supported by the Palm OS Bluetooth system.

Table 10.1 Supported Bluetooth profiles

Profile	Description
Generic Access	Defines the generic procedures related to discovery of Bluetooth devices, as well as link aspects of connecting to Bluetooth devices. Also defines procedures related to the use of different levels of security and common format requirements for parameters accessible at the user interface level.
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).

Table 10.1 Supported Bluetooth profiles *(continued)*

Profile	Description
Dial-up Networking	Defines the protocols and procedures used by devices implementing the “Internet Bridge” usage model; the usage of a cellular phone or modem to connect to a dial-up Internet access server or other dial-up service. Support is provided for both the Terminal role (the device connecting to the modem) and the Gateway role (the device acting as the phone or modem).
LAN Access	Defines LAN access using PPP over RFCOMM. This profile has been deprecated in favor of the newer Personal Area Networking profile described below.
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.

Table 10.1 Supported Bluetooth profiles *(continued)*

Profile	Description
Headset (Gateway role)	Palm OS supports the HSP in the gateway role. HSP supports the transport and control of voice-grade audio between an audio gateway such as a phone and a headset, using an SCO link to transport the audio and an RfComm channel for control functions.
Hands-Free (Gateway role)	The HFP lets the user use an audio gateway such as a phone, transporting and controlling voice-grade audio between the two. A hands-free device is typically installed in a car to allow the driver to use a phone without removing his or her hands from the steering wheel. Like HSP, HFP uses an SCO link for the audio and an RfComm channel for control functions. However, HFP is a richer profile, providing features to initiate and accept calls, among other things.

Table 10.1 Supported Bluetooth profiles *(continued)*

Profile	Description
Personal-Area Networking (PANU role)	PANP makes a Bluetooth piconet look like an Ethernet network, letting it be used under TCP/IP. The PANU role lets the device connect to another PANU, to a Group Ad-Hoc Network Service, or to a Network Access Point.
Personal-Area Networking (GN role)	The GN role lets the device servce as a bridge, to which multiple PANUs can connect.

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 devices 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:

- Palm OS device connects to the Internet via a cell phone to access email, browse the Web, or perform a remote HotSync operation.
  - Generic Access Protocol
  - Serial Port Profile
  - Dialup Networking Profile, Terminal role

- Palm OS device connects to the Internet via an access point or a desktop computer to access email, browse the Web, or perform a remote HotSync operation.
  - Generic Access Protocol
  - Personal-Area Networking Profile, PANU role
- Palm OS device connects to a desktop computer to perform a HotSync operation.
  - Generic Access Protocol
  - Serial Port Profile
- Palm OS device connects to a cell phone to dial or to manage SMS messages.
  - Generic Access Protocol
  - Serial Port Profile
- Palm OS device sends and receives addresses, appointments, or Palm OS applications to or from some other device.
  - Generic Access Protocol
  - Serial Port Profile
  - Generic Object-Exchange Profile
  - Object Push Profile, client or server role
- Another device connects to a Palm OS telephony device to access the Internet.
  - Generic Access Protocol
  - Serial Port Profile
  - Dialup Networking Profile, Gateway role
- Palm OS device uses a Bluetooth headset.
  - Generic Access Protocol
  - Serial Port Profile
  - Headset Profile, Audio Gateway role
- Palm OS telephony device controlled by a hands-free device.
  - Generic Access Protocol
  - Serial Port Profile
  - Hands-Free Profile, Audio Gateway role

- Palm OS device hosts or joins an ad-hoc personal area network, for multi-user games or other collaborative operations.
  - Generic Access Profile
  - Personal-Area Networking Profile, GN (or PANU) role

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.

Telephony and Bluetooth

The Dialup Networking Profile (DUNP), the Headset Profile (HSP), and the Hands-Free Profile (HFP) provide features that are relevant for smart phones. DUNP is integrated with the Palm OS Telephony Manager. Sample source code is provided for both HSP and HFP.

Dial-up Networking Profile

Palm OS supports both the gateway and data terminal roles for DUNP. As a data terminal, DUNP allows the Palm OS device to use a Bluetooth phone as a modem. As a gateway, DUNP lets a Palm OS smart phone be used as a modem by other devices, such as a laptop computer.

Support for the gateway role is new to Palm OS Cobalt, version 6.1. Any Palm OS device that supports the gateway role must also support telephony. When a data terminal connects to a Palm OS DUNP gateway, the Palm OS device will open a data connection to its local telephony service, then serve as a bridge between the data terminal and the network.

Headset Profile

PalmSource provides a sample application—including source code—for HSP that can be used both as a normal application and as a persistent Bluetooth service application. Users can launch it as a normal application, but because it registers itself as a Bluetooth service, it can automatically launch in a thread in the System Process when an inbound connection is detected from the headset.

The sample application provides the following features:

- Connections (the ACL link, RfComm connection, and SCO link) can be establishes either from the audio gateway or the headset.
- Connections can be released either by the audio gateway or the headset.
- Volume control can be performed remotely, by the audio gateway.

The sample HSP service application can be found in the `samples/Bluetooth/BtHeadset` directory in the SDK.

Hands-Free Profile

PalmSource also provides a sample HFP application that can be used as either a normal application and as a service application. The HFP sample application simulates all the interaction between the audio gateway and the network. For example, there is user interface to let the user simulate an inbound connection from the network to the audio gateway.

The HFP sample application includes the following features (assuming the hands-free kit being used supports them):

- Service-level connection management; establishment and release of the ACL link and RfComm connection are possible from either the audio gateway or the hands-free unit. Each side of the connection informs the other of which features they support.
- Phone status information; changes in the registration status, call status, and call setup status that the audio gateway detects are reported to the hands-free kit.
- Audio connection handling; the SCO link can be established or released by either the audio gateway or the hands-free kit.
- Accept or reject an incoming voice call.
- Terminate a call.
- Audio connection transfer during an ongoing call; the SCO link can be established and released during an ongoing call without disturbing the call, from either the audio gateway or the hands-free kit.
- Place a call with a phone number supplied by the hands-free kit. Support for this feature is optional in the hands-free kit.
- Place a call using memory dialing. Support for this feature is optional in the hands-free kit.
- Place a call to the last number dialed. Support for this feature is optional in the hands-free kit.
- Call waiting notification. Support for this feature is optional in the hands-free kit.
- Calling line identification. Support for this feature is optional in the hands-free kit.

- Ability to transmit DTMF codes. Support for this feature is optional in the hands-free kit.
- Remote audio volume control. Support for this feature is optional in the hands-free kit.

The sample application does not support the following features, which are optional in both the audio gateway and in the hands-free kit:

- Three way calling.
- Echo canceling and noise reduction.
- Voice recognition activiation.
- Attaching a phone number to a voice tag.

The sample HFP service application can be found in the `samples/Bluetooth/BtHandsfree` directory in the SDK.

Personal-Area Networking

PANP lets a Bluetooth piconet look like an ethernet, using that ethernet beneath TCP/IP. PANP uses the Basic Network Encapsulation Protocol (BNEP) below IP and above L2Cap to provide the illusion of ethernet.

PANP lets a distributed application, such as multi-player games or communication tools, work over Bluetooth without having to write any Bluetooth specific code. Any TCP/IP application can work over Bluetooth—the user simply needs to select “BluetoothPAN” as the connection type in the connection preferences.

NOTE: At this time, Palm OS only supports the PAN User (PANU) and Group Ad-Hoc Network Service (GN) roles for PANP. The Network Access Point (NAP) role is not supported at this time.

Radio Power Management

The extended battery life of Palm Powered devices is considered to be a key competitive advantage by many Palm Powered device 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.

11

Developing Bluetooth-enabled Applications

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 IOS STDIO calls. Object transfer is supported through the Exchange Manager using the Bluetooth Exchange Library, which is discussed in Chapter 12, "Bluetooth Exchange Library Support." 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 device before using any Bluetooth APIs. To do so, use the code shown in Listing 11.1:

Listing 11.1 Making sure the device has Bluetooth support

```
UInt32 btVersion;

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

Overview of the Bluetooth Library

From a programmer's perspective, the functions of the Bluetooth library fall into six areas: management entity, sockets, service discovery, security, persistent services, and utility.

- The management entity functions deal with the radio, baseband, and link manager parts of the Bluetooth specification. You use them to find nearby devices and to establish ACL links.
- The socket functions enable communication with L2CAP, RFCOMM, and SDP protocols, as well as with SCO links.
- The service discovery functions manage the local service database and query remote devices' service databases.
- The security functions manage a set of trusted devices—devices that do not have to authenticate when they create a secure connection with the Palm OS device.
- The persistent service functions provide a means of installing applications that run in the background and respond to inbound connections from remote devices.
- The utility functions perform useful data conversions.

Compatibility

The entire communications architecture has changed with Palm OS Cobalt. While existing applications will continue to run, using a compatibility library, applications written to the Palm OS Cobalt and later Bluetooth need to conform to a few modest changes to the API.

Deprecated Functions

The functions `BtLibRegisterManagementNotification()` and `BtLibUnregisterManagementNotification()` no longer exist; instead, applications read events from the Management Entity device directly by polling its file descriptor.

The `BtLibServicesOpen()`, `BtLibServicesClose()`, and `BtLibServicesIndicateSessionStart()` functions have been removed as well. Services are no longer a special case.

Additionally, the `BtLibDiscoverSingleDevice()`, `BtLibDiscoverMultipleDevices()`, and `BtLibGetSelectedDevices()` functions have been replaced by one function: `BtLibDiscoverDevices()`.

Parameter Changes

Functions that used to take a Bluetooth library reference number as an input parameter now require a file descriptor to one of the Management Entity, L2CAP, RFCOMM, or SDP device, depending on the specific function.

The `BtLibOpen()` function now returns an IOS file descriptor to the Management Entity device, and `BtLibClose()` closes that file descriptor. `BtLibOpen()` also no longer necessarily causes a radio state event; applications should not wait for a radio state event after calling `BtLibOpen()`. If the hardware is not available, the call to `BtLibOpen()` will simply fail. Likewise, `BtLibOpen()` will no longer necessarily cause an accessibility event; if the application needs to know the accessibility state, it should call `BtLibGetGeneralPreference()`.

`BtLibSocketCreate()` no longer has callback procedure and callback context parameters. The function now returns a file descriptor opened to an L2CAP, RFCOMM, or SDP device. `BtLibSocketClose()` closes the file descriptor.

The `BtLibSocketRef` type is now a 32-bit value. It is a file descriptor to the underlying STREAMS device.

New Functions

There are several new functions:

- `BtLibGetRemoteDeviceNameSynchronous()`
- `BtLibMEEventName()`
- `BtLibSocketEventName()`
- `BtLibRegisterService()`

Events

Applications now obtain events by polling IOS file descriptors, instead of through a callback function. See "Polling for Management Entity Events".

The Management Entity

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 poll for events on the STREAMS devices for the relevant protocols.

Opening the Library

To open the Bluetooth library, use the `BtLibOpen()` function. If this returns without error, the Bluetooth Management Entity device is open and ready to go.

`BtLibOpen()` returns a file descriptor to the Management Entity. Every Management Entity file descriptor sees the same Management Entity; every program monitoring ME events receives the same events.

Polling for Management Entity Events

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 events posted on the Management Entity's event queue.

In some cases, a management function fails before starting the asynchronous operation. In this case, an event does not get sent. You can tell whether or not to expect to receive an event as a result of the call by looking at the management function's return code:

`btLibErrNoError`
: The operation has completed and no event will be sent.

`btLibErrPending`
: The operation has started successfully and an appropriate event will be sent,

`any other error code`
: The operation failed and no event will be sent.

You can poll for these events either by calling `IOSPoll()` directly, or by using a PollBox. See Chapter 18, "Polling STREAMS File Descriptors," on page 375 for an introduction to event polling.

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 code in Listing 11.2 is a skeleton of the PollBox callback you need.

Listing 11.2 Polling for Management Entity events using a PollBox

```
void HandlePbxMEEvent( struct PollBox *pbx, struct pollfd *pollFd, void * ) {
  status_t error;
  int32_t flags;

  static BtLibManagementEventTypemEvent;
  static char mData[sizeof(BtLibFriendlyNameType)];
  static struct strbuf ctlBuf = { sizeof(mEvent), 0, (char*)&mEvent };
  static struct strbuf datBuf = { sizeof(mData),  0, (char*)&mData[0] };

  // We must be here for a reason...

  ErrFatalErrorIf(!(pollFd->revents & (POLLIN|POLLERR|POLLHUP|POLLNVAL)),
          "no event flag" );

  // We must have the Management Entity file descriptor.
  ErrFatalErrorIf( pollFd->fd != gFdME, "not the ME fd" );
  ErrFatalErrorIf( pollFd->fd < 0, "ME fd closed" );

  // Check for error/eof from poll, read the event message.
  flags = 0;
  if ((pollFd->revents & (POLLERR|POLLHUP|POLLNVAL)) ||
    IOSGetmsg(pollFd->fd, &ctlBuf, &datBuf, &flags, &error) != 0) {
    PbxRemoveFd(pbx, pollFd->fd);
    BtLibClose(pollFd->fd);
    gFdME = -1;
    return;
  }

  // We must have an event struct in the control part.
  ErrFatalErrorIf(ctlBuf.len != sizeof(BtLibManagementEventType),
          "no event struct");
```

```
   // Decode the event.

   switch (mEvent.event) {
      case btLibManagementEventRadioState:
         // The radio state has changed.
         break;

      case btLibManagementEventInquiryResult:
         // A device has been found.  Save it in a list.
         break;

      case btLibManagementEventInquiryComplete:
         // The inquiry is finished.
         break;

      case btLibManagementEventInquiryCanceled:
         // The inquiry has been canceled.
         break;
   }
}
```

This example includes some simple error condition checks for the PollBox callback being called with events that aren't Management Entity events.

To install this PollBox event handler, you would use code similar to that shown in Listing 11.3.

Listing 11.3 Installing the Management Event handler PollBox callback

```
int32_t gFdME;

error = BtLibOpen(&gFdME);
if (error) {
   // Unable to open the Bluetooth Library.
} else {
   PbxAddFd(gPollBox, gFdME, POLLIN, HandlePbxMEEvent, 0);
}
```

For a list of management events, see "BtLibManagementEventEnum" in Chapter 13, "Bluetooth Reference."

Finding Nearby Devices

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

- Use the `BtLibDiscoverDevices()` function 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 the discovery function, 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; the status code will indicate why the connection was terminated.

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.

Closing the Management Entity

When you're finished using the Bluetooth Library, you should call `BtLibClose()`, passing the Management Entity's file descriptor. When you do this, and there are no longer any open ME file descriptors or open and connected L2CAP or RFCOMM file descriptors, any remaining ACL links will be disconnected.

Bluetooth 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.

Five 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.

SCO links are seen as a new socket type in BtLib. You can use `BtLibSocketCreate()` and `BtLibSocketClose()` to establish and break SCO links; however, once they're established, all data transfer is managed in hardware, so there is nothing further for software to do with them.

BNEP sockets are only used within the Bluetooth system and are generally not useful to developers. Sending data over a BNEP socket using `BtLibSocketSend()` must involve sending valid ethernet frames containing a 14-byte ethernet header followed by data.

NOTE: Versions of Palm OS prior to 6.0 required that the data buffer remain unchanged until the `btLibSocketEventSendComplete` event arrives. This is no longer the case; you can immediately release or reuse the buffer after `BtLibSocketSend()` returns.

When incoming data arrives, the `IOSGetmsg()` function returns a message with no control part and a data part containing the received data.

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 when using the Serial Manager API to perform Bluetooth communications, as well as by 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. When incoming data arrives, the `IOSGetmsg()` function returns a message with no control part and a data part containing the received data.

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.

Using Serial-on-L2CAP and Serial-on-RFCOMM

The Serial-on-L2CAP and Serial-on-RFCOMM modules, whose names are `btModSerL2cName` and `btModSerRfcName` in `BtLibTypes.h`, are STREAMS modules that can be pushed onto an L2CAP or RFCOMM file descriptor.

These modules can be pushed onto the file descriptor either before or after connecting the socket. If pushed before connecting, `BtLibSocketXXX()` functions and events will be transparent to the module until the connection is established. In particular, the connection event will be visible to the application.

Once the socket is connected, or if the module is pushed after establishing the connection, then only pure data can be read or written by the application; the module handles things like flow control for you.

A disconnect event appears as an error condition from `IOSRead()` or `IOSWrite()`. Closing the file descriptor will disconnect the socket if it's connected.

SCO

SCO sockets are used to transmit audio between a Palm OS smart phone and a hands-free kit or headset. The only operations that can be performed on SCO sockets are to create, connect, and close them. Everything else is done in hardware.

BSD Sockets

You can use the standard BSD Sockets API to perform Bluetooth communications using the RFCOMM protocol. Use the `sockaddr_bth` structure to define a Bluetooth device address when using the BSD Sockets API.

Creating a Socket

You obtain a Bluetooth RFCOMM sotcket by specifying the address family `AF_BTH`, the socket type `SOCK_STREAM`, and the protocol `BTHPROTO_RFCOMM` when calling `socket()`:

```
myBtSocket = socket( AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM );
```

Restrictions

A listening socket can have no backlog, and can only accept a single incoming connection, after which it becomes dead (meaning that it doesn't listen for incoming connections anymore). In addition, once `accept()` has been called and has returned a newly connected socket, the listening socket must not be closed until the accepted connection is closed first.

Listing 11.4 Listening for an incoming connection

```
listenerSocket = socket();
bind(listenerSocket);
listen(listenerSocket);
select(listenerSocket);
dataSocket = accept(listenerSocket);
...
close(dataSocket);
close(listenerSocket);
```

If the application wishes to listen for further connections, it needs to explicitly start listening again by calling `listen()`.

For more information about Palm OS support for the BSD Sockets API, see Part IV, "Networking and Sockets."

Service Discovery

The service discovery functions are used to create and advertise service records to remote devices, and to discover services available on remote devices.

NOTE: While Palm OS Cobalt, version 6.1 supports service discovery, it does not support the full Service Discovery Application Profile, since there is no service browser provided.

Service Records

A service record is a sequence of service attributes. A service attribute consists of two components: an attribute ID and an attribute value.

Universal attributes are those service attributes whose definitions are common to all service records. Among them is the ServiceClassIDList, which is a list of **service class** identifiers. Every service record must have a ServiceClassIDList.

An attribute ID is a 16-bit unsigned integer that distinguishes each service attribute from other service attributes within a service record. The attribute ID also identifies the semantics of the associated attribute value.

An attribute value is a data element whose meaning is determined by the corresponding attribute ID and the service class of the service record in which the attribute is contained.

A data element is a typed data representation consisting of two fields: a header field and a data field. The header field, in turn, is composed of two parts: a type descriptor and a size descriptor. The data is a sequence of bytes whose length is specified by the size descriptor and whose meaning is partially specified by the type descriptor.

To fully understand SDP service records, how they are encoded, interpreted, and so forth, see the Service Discovery Protocol section in Volume 1 of the *Bluetooth Specification*, version 1.2.

NOTE: Only one outstanding query at a time is allowed per SDP socket.

Creating Persistent Services

Applications that support Bluetooth can register themselves as persistent Bluetooth services, that are automatically started in a thread in the System Process when other Bluetooth enabled devices connect to them.

NOTE: For examples of Bluetooth persistent service applications, see the `samples/Bluetooth/BtHeadset` and `samples/Bluetooth/BtHandsfree` directories in the SDK. The code snippets shown here are adapted from these examples, but the complete source code is a valuable resource you should review.

Registering a Persistent Service

To register as a persistent service, an application must register the service at boot and system reset time, as shown in Listing 11.5.

Listing 11.5 Registering a persistent service

```
uint32_t PilotMain(uint16_t cmd, MemPtr cmdPBP, uint16_t launchFlags) {
  BtLibServiceRegistrationParamsType params;
  status_t error = errNone;
  ...
  switch(cmd) {
    case sysLaunchCmdBoot:
    case sysLaunchCmdSystemReset:
      params.appCodeRscId = sysResIDDefault;
      params.appType = myAppRscType;
      params.appCreator = myAppCreator;
      params.stackSize = 5000;
      params.protocol = btLibRfCommProtocol;
      params.pushSerialModule = false;
      params.execAsNormalApp = false;
      error = BtLibRegisterService(&params);
      break;
    ...
```

```
  }
  return error;
}
```

Describing a Persistent Service

The Bluetooth system may shut down or reinitialize itself at any time for a variety of reasons: the on/auto/off preference may be set to “off,”, the device may be powered off, the radio hardware may be physically detached, or some application may close its last Bluetooth file descriptor that had been used to communicate with remote devices. To support this, the service application needs to implement the `sysBtLaunchCmdDescribeService` launch code to fill out a `BtLibServiceDescriptionType` structure. This is demonstrated in Listing 11.6.

Listing 11.6 Describing a Bluetooth persistent service

```
    case sysBtLaunchCmdDescribeService:
      int size;
      MemHandle theResHdl;
      MemPtr theResPtr;

      // Describe our service for the Bluetooth panel services view.
      ((BtLibServiceDescriptionType*)cmdPBP)->flags = 0;

      // Get the profile service name str
      theResHdl = DmGetResource(myDmOpenRef, (DmResourceType) strRsc,
            infoNameStrRscId);
      theResPtr = MemHandleLock(theResHdl);
      size = strlen(theResPtr) + 1;
      if ((((BtLibServiceDescriptionType*)cmdPBP)->nameP = MemPtrNew(size))
            == NULL ) {
        return btLibErrOutOfMemory;
      }
      MemMove(((BtLibServiceDescriptionType*)cmdPBP)->nameP, theResPtr,
            size );
      MemHandleUnlock(theResHdl);
      DmReleaseResource(theResHdl);

      // Get the profile service description str
      theResHdl = DmGetResource(myDmOpenRef, (DmResourceType) strRsc,
            infoDescStrRscId);
      theResPtr = MemHandleLock(theResHdl);
      size = strlen(theResPtr) + 1;
```

```
if ((((BtLibServiceDescriptionType*)cmdPBP)->descriptionP =
        MemPtrNew( size )) == NULL ) {
   return btLibErrOutOfMemory;
}
MemMove(((BtLibServiceDescriptionType*)cmdPBP)->descriptionP,
        theResPtr, size);
MemHandleUnlock(theResHdl);
DmReleaseResource(theResHdl);
break;
```

The example loads the strings from resources, copies them into the `BtLibServiceDescriptionType` structure, and releases the resources.

Providing Advanced Configuration Options

If the persistent service needs to provide the ability for the user to configure it, it should implement the `sysBtLaunchCmdDoServiceUI` launch code, which is sent when the user clicks the "Advanced" button in the Bluetooth services view. In response to this launch code, the application can present user interface to configure the service.

Preparing the Service to Listen for Incoming Connections

When the system is ready for the service application to begin listening for incoming connections, it sends the application the `sysBtLaunchCmdPrepareService` launch code with a `BtLibServicePreparationParamsType` structure as input.

This structure includes a handle to an empty SDP service record, which the service application needs to fill out to describe the offered serice, as well as the file descriptor for the socket on which the service application should listen for incoming connections.

Listing 11.7 Preparing the service

```
BtLibServicePreparationParamsType *params =
        (BtLibServicePreparationParamsType *) cmdPBP;
BtLibSdpUuidType gGWSdpUUIDList[3];
status_t error;
...
case sysBtLaunchCmdPrepareService:
  BtLibSdpUuidInitialize(gGWSdpUUIDList[0],
        btLibSdpUUID_SC_HEADSET_AUDIO_GATEWAY, btLibUuidSize16);
```

```
    BtLibSdpUuidInitialize(gGWSdpUUIDList[1],
         btLibSdpUUID_SC_GENERIC_AUDIO, btLibUuidSize16);

    error = BtLibSdpServiceRecordSetAttributesForSocket(
           params->fdListener,
           gGWSdpUUIDList,
           2,
           HEADSET_GW_SERVICE_NAME,
           strlen(HEADSET_GW_SERVICE_NAME);
           params->serviceRecH
           );
    break;
```

The code snippet in Listing 11.7 comes from the BtHeadset sample program. It supports the audio gateway and generic audio services, for which it builds UUIDs using the `BtLibSdpUuidInitialize()` macro and inserts them into an array. It then sets the attributes on the listener socket to watch for attempts to connect to those particular services, and sets the name of the service, by calling `BtLibSdpServiceRecordSetAttributesForSocket()`.

Executing the Service

When a connection attempt arrives at the listener socket, Palm OS sends the `sysBtLaunchCmdExecuteService` launch code to the service application. This sublaunch occurs in a thread within the System Process, and the application should not exit until the service finishes the transaction.

Listing 11.8 Processing the service transaction

```
BtLibServiceExecutionParamsType *params;
...
  case sysBtLaunchCmdExecuteService:
    params = (BtLibServiceExecutionParamsType *) cmdPBP;
    if (AppStart() == errNone) {
      ...
      /* perform the transaction */
      ...
    }
    AppStop();
    break;
...
```

When the service begins running, it needs to initialize itself by opening its database, creating a PollBox to process events, open the Bluetooth Management Entity file descriptor, and so forth. In this example, this is all done by the `AppStart()` function.

Then the application can run an event loop to process incoming data and respond to that data, as well as to provide progress user interface and so forth.

Once the transaction is finished, the service application must close the data socket, which is specified by the `fdData` field in the `BtLibServiceExecutionParamsType` structure, before exiting.

Dealing with Bluetooth Shutdown

The Bluetooth system shuts down when the user changes the on/off preference or the radio hardware is physically detached. When this happens, all opened Bluetooth file descriptors start to produce errors.

When an application detects `M_ERROR` on any Bluetooth file descriptor, it must immediately close all its Bluetooth file descriptors.

12

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 Chapter 4, "Object Exchange," on page 105 of the book *Exploring Palm OS: High-Level Communications*.

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 `ExgLibGet()` and `ExgLibRequest()` 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 the `ExgControl()` function code `exgLibCtlGetUrl` 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);
```

ExgLibGet() and ExgLibRequest()

The Bluetooth Exchange Library does not support the pull functions provided by `ExgLibGet()` and `ExgLibRequest()`. If you want to perform these functions, you must use the general Bluetooth Library APIs. See “Developing Bluetooth-enabled Applications”.

13
Bluetooth Reference

This chapter provides complete reference material to the Palm OS® Bluetooth Library, BtLib.

The header file `BtLibTypes.h` declares the types and constants that this chapter describes, while the file `BtLib.h` declares the functions and macros.

Bluetooth Structures and Types

BtLibClassOfDeviceType Typedef

Purpose A bit pattern representing the class of device and the services it supports.

Declared In `BtLibTypes.h`

Prototype `typedef uint32_t BtLibClassOfDeviceType`

Comments A device may support multiple services but only belongs to a single class. The class is specified in two parts: the major class, which broadly classifies the type of device, and the minor class, which together with the major class specifies the type of device in more detail.

An example is a simple cellular telephone. It provides Telephony and Object Exchange services. Its major device class is Phone, and its minor device class is Cellular.

The *Bluetooth Assigned Numbers* specification defines a “Class of Device/Service” (CoD) value as having three bit fields. One field specifies the major service classes supported by the device. Another field specifies the major device class. The third field specifies the minor device class.

The constants provided here allow you to construct a CoD that conforms to the Bluetooth specification. You simply perform a logical OR of the constants representing the service classes the device supports, the constant representing the device's major class, and the constant representing the device's minor class.

For example, device class of the simple cellular telephone can be computed as follows:

```
cellPhoneCOD = btLibCOD_Telephony |
   btLibCOD_ObjectTransfer |
   btLibCOD_Major_Phone |
   BtLibCOD_Minor_Phone_Cellular;
```

Constants are also provided to mask the individual bit fields in a device class.

Major Service Classes

These constants define the Bluetooth major service classes. The service classes are described in the *Specification of the Bluetooth System*.

Table 13.1 Major service classes

Constant
btLibCOD_Audio
btLibCOD_Capturing
btLibCOD_Information
btLibCOD_LimitedDiscoverableMode
btLibCOD_Networking
btLibCOD_ObjectTransfer
btLibCOD_Positioning

Table 13.1 Major service classes

Constant
`btLibCOD_Rendering`
`btLibCOD_Telephony`

Major Device Classes

These constants define the Bluetooth major device classes. The major device classes are described in the *Specification of the Bluetooth System*.

Table 13.2 Major device classes

Constant
`btLibCOD_Major_Audio`
`btLibCOD_Major_Computer`
`btLibCOD_Major_Imaging`
`btLibCOD_Major_Lan_Access_Point`
`btLibCOD_Major_Misc`
`btLibCOD_Major_Peripheral`
`btLibCOD_Major_Phone`
`btLibCOD_Major_Unclassified`

Computer Minor Device Classes

These constants define the minor device classes associated with the computer major class. They are described in the *Bluetooth Assigned Numbers* specification.

Table 13.3 Computer minor device classes

Constant
`btLibCOD_Minor_Comp_Desktop`
`btLibCOD_Minor_Comp_Handheld`
`btLibCOD_Minor_Comp_Laptop`
`btLibCOD_Minor_Comp_Palm`
`btLibCOD_Minor_Comp_Server`
`btLibCOD_Minor_Comp_Unclassified`

Phone Minor Device Classes

These constants define the minor device classes that are associated with the phone major class. They are described in the *Bluetooth Assigned Numbers* specification.

Table 13.4 Phone minor device classes

Constant
`btLibCOD_Minor_Phone_Unclassified`
`btLibCOD_Minor_Phone_Cellular`
`btLibCOD_Minor_Phone_Cordless`
`btLibCOD_Minor_Phone_ISDN`
`btLibCOD_Minor_Phone_Smart`
`btLibCOD_Minor_Phone_Modem`

LAN Access Point Minor Device Classes

These constants define load factors for the LAN access point major device class. LAN access point load factors are described in more detail in the *Bluetooth Assigned Numbers* specification.

Table 13.5 LAN access point minor device classes

Constant	Meaning
`btLibCOD_Minor_Lan_0`	Fully available
`btLibCOD_Minor_Lan_17`	1-17% utilized
`btLibCOD_Minor_Lan_33`	17-33% utilized
`btLibCOD_Minor_Lan_50`	33-50% utilized
`btLibCOD_Minor_Lan_67`	50-67% utilized
`btLibCOD_Minor_Lan_83`	67-83% utilized
`btLibCOD_Minor_Lan_99`	83-99% utilized
`btLibCOD_Minor_Lan_NoService`	Fully utilized

Audio Minor Device Classes

These constants define the minor classes associated with the audio major class. They are described in more detail in the *Bluetooth Assigned Numbers* specification.

Table 13.6 Audio minor device classes

Constant
`btLibCOD_Minor_Audio_Unclassified`
`btLibCOD_Minor_Audio_Headset`
`btLibCOD_Minor_Audio_CamCorder`
`btLibCOD_Minor_Audio_CarAudio`
`btLibCOD_Minor_Audio_GameToy`
`btLibCOD_Minor_Audio_HandFree`
`btLibCOD_Minor_Audio_HeadPhone`
`btLibCOD_Minor_Audio_HifiAudio`

Table 13.6 Audio minor device classes

Constant
`btLibCOD_Minor_Audio_LoudSpeaker`
`btLibCOD_Minor_Audio_MicroPhone`
`btLibCOD_Minor_Audio_PortableAudio`
`btLibCOD_Minor_Audio_SetTopBox`
`btLibCOD_Minor_Audio_VCR`
`btLibCOD_Minor_Audio_VideoCamera`
`btLibCOD_Minor_Audio_VideoConf`
`btLibCOD_Minor_Audio_VideoDisplayAndLoudSpeaker`
`btLibCOD_Minor_Audio_VideoMonitor`

Peripheral Minor Device Classes

These constants define the minor classes associated with the peripheral major class. They are described in more detail in the *Bluetooth Assigned Numbers* specification.

Table 13.7 Peripheral minor device classes

Constant
`btLibCOD_Minor_Peripheral_CardReader`
`btLibCOD_Minor_Peripheral_Combo`
`btLibCOD_Minor_Peripheral_DigitizerTablet`
`btLibCOD_Minor_Peripheral_GamePad`
`btLibCOD_Minor_Peripheral_Joystick`
`btLibCOD_Minor_Peripheral_Keyboard`
`btLibCOD_Minor_Peripheral_Pointing`
`btLibCOD_Minor_Peripheral_RemoteControl`

Table 13.7 Peripheral minor device classes

Constant
btLibCOD_Minor_Peripheral_Sensing
btLibCOD_Minor_Peripheral_Unclassified

Imaging Minor Device Classes

These constants define the minor classes associated with the imaging major class. They are described in more detail in the *Bluetooth Assigned Numbers* specification.

Table 13.8 Imaging minor device classes

Constant
btLibCOD_Minor_Imaging_Camera
btLibCOD_Minor_Imaging_Display
btLibCOD_Minor_Imaging_Printer
btLibCOD_Minor_Imaging_Scanner
btLibCOD_Minor_Imaging_Unclassified

Masks

These constants define bit masks to isolate certain fields of the device class.

Table 13.9 Masks

Constant	Meaning
btLibCOD_Service_Mask	A mask to isolate the major service class field from the other fields of the device class.
btLibCOD_Major_Mask	A mask to isolate the major device class field from the other fields of the device class.

Table 13.9 Masks

Constant	Meaning
`btLibCOD_Minor_Mask`	A mask to isolate the minor device class field from the other fields of the device class.
`btLibCOD_ServiceAny`	Used as a device filter for the `BtLibDiscoverDevices()` function. With this filter, devices providing any service appear in the device list. Same as `btLibCOD_Service_Mask`.
`btLibCOD_Major_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. With this filter, devices in any major device class appear in the device list. Same as `btLibCOD_Major_Mask`.
`btLibCOD_Minor_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. With this filter, devices in any minor device class appear in the device list. Same as `btLibCOD_Minor_Mask`.
`btLibCOD_Minor_Comp_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. When this filter is used in conjunction with `btLibCOD_Major_Computer`, all devices broadcasting themselves as computers appear in the device list. Same as `btLibCOD_Minor_Any`.

Table 13.9 Masks

Constant	Meaning
`btLibCOD_Minor_Phone_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. When this filter is used in conjunction with `btLibCOD_Major_Phone`, all devices broadcasting themselves as phones appear in the device list. Same as `btLibCOD_Minor_Any`.
`btLibCOD_Minor_LAN_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. When this filter is used in conjunction with `btLibCOD_Major_Lan_Access_Point`, all devices broadcasting themselves as LAN access points appear in the device list. Same as `btLibCOD_Minor_Any`.
`btLibCOD_Minor_Audio_Any`	Used as a device filter for the `BtLibDiscoverDevices()` function. When this filter is used in conjunction with `btLibCOD_Major_Audio`, all devices broadcasting themselves as audio devices appear in the device list. Same as `btLibCOD_Minor_Any`.

BtLibDeviceAddressType Struct

Purpose Defines the address of a Bluetooth device.

Declared In `BtLibTypes.h`

Prototype
```
typedef struct BtLibDeviceAddressType {
   uint8_t address[btLibDeviceAddressSize];
} BtLibDeviceAddressType
```

Fields

`address`
: `btLibDeviceAddressSize` byte long Bluetooth device address.

BtLibDeviceAddressTypePtr Typedef

Purpose A pointer to a Bluetooth address.

Declared In `BtLibTypes.h`

Prototype
```
typedef BtLibDeviceAddressType
      *BtLibDeviceAddressTypePtr
```

BtLibFriendlyNameType Struct

Purpose Contains the user-friendly name of a device.

Declared In `BtLibTypes.h`

Prototype
```
typedef struct BtLibFriendlyNameType {
   uint8_t nameLength;
   uint8_t name[btLibMaxDeviceNameLength];
} BtLibFriendlyNameType
```

Fields

`nameLength`
: The length of the name, including the null terminator.

`name`
: Buffer for the null-terminated device name.

Comments The `BtLibFriendlyNameType` structure is used to get and set a device's friendly name.

NOTE: The `nameLength` field includes the name string's null terminator.

BtLibFriendlyNameTypePtr Typedef

Purpose Defines a pointer to a friendly Bluetooth device name.

Declared In `BtLibTypes.h`

Prototype
```
typedef BtLibFriendlyNameType
      *BtLibFriendlyNameTypePtr
```

BtLibL2CapChannelIdType Typedef

Purpose Specifies an L2CAP channel ID.

Declared In `BtLibTypes.h`

Prototype
```
typedef uint16_t BtLibL2CapChannelIdType
```

Comments An L2CAP channel ID uniquely identifies the local endpoint of an L2CAP connection on a given device. L2CAP channel IDs are assigned by the system when an L2CAP connectoin is established.

BtLibL2CapPsmType Typedef

Purpose The `BtLibL2CapPsmType` type represents a Protocol and Server Multiplexer (PSM) value. See the "Logical Link and Adaptation Protocol Specification" chapter of the *Specification of the Bluetooth System* for more information about PSM values. The Bluetooth library only supports two-byte PSM values.

Declared In `BtLibTypes.h`

Prototype
```
typedef uint16_t BtLibL2CapPsmType
```

BtLibLanguageBaseTripletType Struct

Purpose The `BtLibLanguageBaseTripletType` structure represents a language base attribute identifier list attribute. See the "Service

Discovery Protocol" chapter of the *Specification of the Bluetooth System* for more information.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibLanguageBaseTripletType {
   uint16_t naturalLanguageIdentifier;
   uint16_t characterEncoding;
   uint16_t baseAttributeID;
} BtLibLanguageBaseTripletType
```

Fields

`naturalLanguageIdentifier`
: A `uint16_t` representing a natural language. See Language ID Constants for a set of constants that can be used in this field.

`characterEncoding`
: A `uint16_t` representing a character set encoding. See Character Encoding Constants for a set of constants that can be used in this field.

`baseAttributeID`
: Base attribute identifiers for attributes represented in this language. See Attribute Identifier Constants for offsets that are added to this value to get the attribute identifiers for specific attributes represented in this language.

BtLibManagementEventType Struct

Purpose The `BtLibManagementEventType` structure contains detailed information regarding a management event. All management events have some common data. Most management events have data specific to those events. The specific data uses a union that is part of the `BtLibManagementEvent` data structure.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct _BtLibManagementEventType {
   BtLibManagementEventEnum event;
   uint8_t padding1;
   uint16_t padding2;
   status_t status;
   union {
      BtLibDeviceAddressType bdAddr;
      BtLibAccessibleModeEnum accessible;
```

```
      struct {
         BtLibDeviceAddressType bdAddr;
      } nameResult;
      struct {
         BtLibDeviceAddressType bdAddr;
         uint16_t padding;
         BtLibClassOfDeviceType classOfDevice;
      } inquiryResult;
      struct {
         BtLibDeviceAddressType bdAddr;
         BtLibLinkModeEnum curMode;
         uint8_t padding;
         uint16_t interval;
      } modeChange;
      struct {
         BtLibDeviceAddressType bdAddr;
         Boolean enabled;
      } encryptionChange;
      struct {
         BtLibDeviceAddressType bdAddr;
         BtLibConnectionRoleEnum newRole;
      } roleChange;
      struct {
         BtLibDeviceAddressType bdAddr;
         int8_t rssi;
      } rssi;
   } eventData;
} BtLibManagementEventType
```

Fields

`event`
The event opcode.

`padding1`
Reserved for system use.

`padding2`
Reserved for system use.

`status`
The event's error code.

`bdAddr`
The Bluetooth device address; used by events `btLibManagementEventACLConnectInbound`, `btLibManagementEventACLConnectOutbound`,

btLibManagementEventACLDisconnect, and btLibManagementEventAuthenticationComplete.

`accessible`
: Indicates the state of the Bluetooth radio's accessibility. Used by the btLibManagementEventAccessibilityChange event.

`nameResult`
: `bdAddr` contains the Bluetooth device's address. The data part of the message contains a BtLibFriendlyNameType structure. Used by the btLibManagementEventNameResult and btLibManagementEventLocalNameChange events.

`inquiryResult`
: Information about a single device found during an inquiry procedure. `bdAddr` contains the address of the device found and `classOfDevice` identifies the device class. `padding` is reserved for system use. The data part of the message contains a structure of type BtLibFriendlyNameType with the remote device's name according to the local name cache; if the name isn't in the cache, the string is null. Used by the btLibManagementEventInquiryResult

`modeChange`
: Used by the btLibManagementEventModeChange event. `bdAddr` specifies the address of an ACL link whose mode has changed. `curMode` indicates the new current mode, and `interval` indicates the length of time to remain in that mode, if applicable. `padding` is, as usual, reserved for system use.

`encryptionChange`
: Used by btLibManagementEventEncryptionChange. `bdAddr` specifies the address of the ACL link whose encryption has changed, and `enabled` indicates whether encryption is on or off.

`roleChange`
: Used by btLibManagementEventRoleChange. `bdAddr` indicates the address of the device whose role has changed, and `newRole` specifies the device's new role.

`rssi`
: The Receiver Signal Strength Indicator indicates whether the signal strength of the receiver is below (negative), within (zero), or above (positive) the "Golden Receive Power Range," in units of one decibel. *Not used in Palm OS Cobalt.0.*

Comments Applications obtain Management Entity events by calling `IOSGetmsg()` on a file descriptor opened to a Management Entity device. The control part of the message obtained this way contains a `BtLibManagementEventType` object. For some events, there's also a data part containing additional information.

The `eventData` union lets the structure only include data needed by the particular event message.

BtLibProfileDescriptorListEntryType Struct

Purpose The `BtLibProfileDescriptorListEntryType` structure represents an entry in a profile descriptor list attribute. See the "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System* for more information about profile descriptor list attributes.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct
BtLibProfileDescriptorListEntryType {
   BtLibSdpUuidType profUUID;
   uint8_t padding1;
   uint16_t version;
   uint16_t padding2;
} BtLibProfileDescriptorListEntryType
```

Fields

`profUUID`
: The profile's UUID.

`padding1`
: Reserved for system use.

`version`
: The profile version.

`padding2`
: Reserved for system use.

BtLibProtocolDescriptorListEntryType Struct

Purpose The `BtLibProtocolDescriptorListEntryType` structure represents an entry in a protocol descriptor list attribute. See the "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System* for more information.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct
BtLibProtocolDescriptorListEntryType {
   BtLibSdpUuidType protoUUID;
   uint8_t padding1;
   uint16_t padding2;
   union {
      BtLibL2CapPsmType psm;
      BtLibRfCommServerIdType channel;
   } param;
} BtLibProtocolDescriptorListEntryType
```

Fields

`protoUUID`
: The protocol's UUID.

`padding1`
: Reserved for system use.

`padding2`
: Reserved for system use.

`param`
: A union containing two members: `psm` and `channel`. `psm` is applicable for a L2Cap protocol descriptor and specifies the Protocol and Service Multiplexor. `channel` is applicable to a RfComm protocol descriptor and specifies the server channel.

BtLibProtocolEnum Typedef

Purpose Specifies the protocol being used on a Bluetooth connection.

Declared In `BtLibTypes.h`

Prototype

```
typedef enum {
   btLibL2CapProtocol,
   btLibRfCommProtocol,
   btLibSdpProtocol,
   btLibSCOProtocol,
   btLibBNEPProtocol
} BtLibProtocolEnum
```

Fields

`btLibL2CapProtocol`
L2Cap.

`btLibRfCommProtocol`
RfComm.

`btLibSdpProtocol`
SDP.

`btLibSCOProtocol`
SCO.

`btLibBNEPProtocol`
BNEP.

BtLibRfCommServerIdType Typedef

Purpose The `BtLibRfCommServerIdType` type represents a RfComm server channel. See the "RFCOMM with TS 07.10" chapter of the *Specification of the Bluetooth System* for more information about server channels.

Declared In `BtLibTypes.h`

Prototype `typedef uint8_t BtLibRfCommServerIdType`

BtLibSdpAttributeDataType Struct

Purpose The `BtLibSdpAttributeDataType` union is used to encapsulate an SDP attribute or a list entry in an SDP attribute. The `BtLibSdpServiceRecordGetAttribute()` function gets an attribute or a list entry and return its contents in a `BtLibSdpAttributeDataType`. The `BtLibSdpServiceRecordSetAttribute()` function sets an attribute or list entry according to the contents of a `BtLibSdpAttributeDataType`. This type supports the universal attributes defined in the *Specification of the Bluetooth System*.

Declared In `BtLibTypes.h`

Prototype

```
typedef union BtLibSdpAttributeDataType {
   BtLibSdpUuidType serviceClassUuid;
   uint32_t serviceRecordState;
   BtLibSdpUuidType serviceIdUuid;
   BtLibProtocolDescriptorListEntryType
protocolDescriptorListEntry;
   BtLibSdpUuidType browseGroupUuid;
   BtLibLanguageBaseTripletType
languageBaseTripletListEntry;
   uint32_t timeToLive;
   uint8_t availability;
   BtLibProfileDescriptorListEntryType
profileDescriptorListEntry;
   BtLibUrlType documentationUrl;
   BtLibUrlType clientExecutableUrl;
   BtLibUrlType iconUrl;
   BtLibStringType serviceName;
   BtLibStringType serviceDescription;
   BtLibStringType providerName;
} BtLibSdpAttributeDataType
```

Fields

`serviceClassUuid`
: The service class UUID.

`serviceRecordState`
: Used to cache service attributes. If this attribute is contained in a service record, its value is guaranteed to change each time any other attribute is added to, deleted from, or changed within the service record. This lets a client detect whether or not the record has changed by simply looking at the value of this attribute; if the value is has changed since the last time it was checked, the record has been altered.

`serviceIdUuid`
: The service's UUID.

`protocolDescriptorListEntry`
: See "BtLibProtocolDescriptorListEntryType" on page 160.

`browseGroupUuid`
: A list of UUIDs, each of which represents a browse group to which the service record belongs. The top level browser group ID, called PublicBrowseRoot, represents the root of the browsing directory. Its value is 00001002-0000-1000-8000-00805F9B34FB (UUID16 0x1002), as specified in the *Bluetooth Assigned Numbers* document.

`languageBaseTripletListEntry`
: Describes a language triplet. See "BtLibLanguageBaseTripletType" on page 155.

`timeToLive`
: The number of seconds for which the information in the service record is expected to remain valid and unchanged. This interval is measured from the time that the attribute value is retrieved from the SDP server. It doesn't guarantee that the record will be available or unchanged, but instead recommends a polling interval for monitoring the service record for changes.

`availability`
: Represents the relative ability of the service to accept additional clients. A value of 0xFF indicates that the service is not in use and is fully available to accept clients, while a value of 0x00 means the service is not accepting new clients. For services that support multiple simultaneous clients, intermediate values indicate the relative availability of the service on a linear scale.

  For example, a service that can accept up to three clients should provide service availability values of 0xFF, 0xAA, 0x55, and 0x00 when 0, 1, 2, or 3 clients are using the service.

  A non-zero value for `availability` doesn't necessarily guarantee availability; it should be considered a hint as to how likely a connection is to be accepted.

`profileDescriptorListEntry`
: Describes an entry in a profile descriptor list. See "BtLibProfileDescriptorListEntryType" on page 159.

`documentationUrl`
: A URL to documentation for the service.

`clientExecutableUrl`
An URL to the program that is executed to run the service.

`iconUrl`
An URL to an icon to use to represent the service.

`serviceName`
The name of the service.

`serviceDescription`
A human-readable description of the service.

`providerName`
A string containing the name of the person or organization providing the service. The offset 0x0002 must be added to the attribute ID base (contained in the `LangugaeBaseAttributeIDList` attribute) in order to compute the attribute ID for this attribute.

Comments Note that if you're retrieving a string or a URL using the `BtLibSdpServiceRecordGetAttribute()` function, you first need to allocate a buffer in addition to this union. This buffer must be large enough to contain the anticipated size of the string or URL. You must also initialize the string pointer and string length fields of the appropriate `BtLibAttributeDataType` union member. For example, if you're retrieving an icon URL, you need to set `iconURL.url` to point to the buffer. You also need to set `iconURL.urllen` to the length of the buffer.

See Also BtLibSdpUuidType, BtLibSocketEventType, BtLibProfileDescriptorListEntryType, BtLibLanguageBaseTripletType, BtLibUrlType, BtLibStringType

BtLibSdpAttributeIdType Typedef

Purpose The `BtLibSdpAttributeIdType` type represents a SDP attribute identifier.

Declared In `BtLibTypes.h`

Prototype `typedef uint16_t BtLibSdpAttributeIdType`

BtLibSdpRecordHandle Typedef

Purpose The `BtLibSdpRecordHandle` type, also called an **SDP memory handle**, provides a memory handle to an **SDP memory record**.

Declared In `BtLibTypes.h`

Prototype `typedef MemHandle BtLibSdpRecordHandle`

Comments A SDP memory record can have two roles: it can contain a local SDP service record or it can refer to an SDP service record on a remote device. In the latter role, the SDP memory record is said to be **mapped** to a service record on the remote device. The `BtLibSdpServiceRecordMapRemote()` function performs this mapping.

BtLibSdpRemoteServiceRecordHandle Typedef

Purpose The `BtLibSdpRemoteServiceRecordHandle` type represents a SDP service record handle on a remote device as defined in the "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System*. The documentation refers to this type as a **remote service record handle**.

Declared In `BtLibTypes.h`

Prototype

```
typedef uint32_t
      BtLibSdpRemoteServiceRecordHandle
```

Comments Note that this type is different from the `BtLibSdpRecordHandle` type, which refers to a memory chunk containing an SDP service record.

BtLibSdpUuidSizeEnum Typedef

Purpose The `BtLibSdpUuidSizeEnum` enum specifies the sizes that a UUID can have. See `BtLibSdpUuidType` for more information.

Declared In `BtLibTypes.h`

Prototype `typedef Enum8 BtLibSdpUuidSizeEnum`

BtLibSdpUuidType Struct

Purpose The `BtLibSdpUuidType` structure represents a Universally Unique Identifier (UUID). A UUID is a 128-bit value that is generated in a manner that guarantees (with very high probability) that it is different from every other UUID.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibSdpUuidType {
   BtLibSdpUuidSizeEnum size;
   uint8_t UUID[16];
} BtLibSdpUuidType
```

Fields

`size`
: The number of bits used to specify the UUID. See `BtLibSdpUuidSizeEnum`.

`UUID`
: The value of the UUID. If you're setting the value of this field, use the `BtLibSdpUuidInitialize()` macro.

Comments The "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System* reserves a set of UUIDs for common Bluetooth services and protocols. You can specify these with 32 bits—the remaining 96 bits have a fixed value. A subset of these can be specified with 16 bits zero-extended to 32 bits. Therefore you can specify a UUID using 16, 32, or 128 bits.

You generally don't set this type directly. Instead, you use the `BtLibSdpVerifyRawDataElement()` macro.

BtLibServiceDescriptionType Struct

Purpose Parameters returned from a service application's `sysBtLaunchCmdDescribeService` launch code handler.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct {
   uint32_t flags;
   char *nameP;
   char *descriptionP;
} BtLibServiceDescriptionType
```

Fields

`flags`
: A bit mask of service description flags. See "Service Description Flags" on page 205 for possible values.

`nameP`
: A pointer to a brief name of the service, to be displayed in the Bluetooth panel.

`descriptionP`
: A pointer to a verbose description of what the service offers, which is also displayed by the Bluetooth panel.

Comments The Bluetooth panel sends this launch code to obtain the information it needs to display in its services view.

The `nameP` and `descriptionP` must be set to localized strings in buffers allocated using `MemPtrNew()` or `malloc()`. `nameP` should be a short name for display in a menu, while `descriptionP` should be a longer description that is displayed when the service is selected in the services view.

For example, nameP might be "Personal Area Networking" while `descriptionP` might be "Allow other devices to connect and form an ad-hoc local network."

BtLibServiceExecutionParamsType Struct

Purpose Specifies parameters passed to a service application when the `sysBtLaunchCmdExecuteService` launch code is sent.

Declared In `BtLibTypes.h`

Prototype
```
typedef struct {
   int32_t fdData;
} BtLibServiceExecutionParamsType
```

Fields `fdData`
The connected L2Cap or RfComm socket.

Comments The *`fdData`* parameter is a file descriptor opened to a connected L2Cap or RfComm device instance, with a serial interface module optionally pushed onto that depending on the `pushSerialModule` registration flag.

On entry, the file descriptor is connected to its remote peer and is ready for data transfer. On exit, the file descriptor must be closed.

BtLibServicePreparationParamsType Struct

Purpose Parameters passed to a service application's `sysBtLaunchCmdPrepareService` launch code handler.

Declared In `BtLibTypes.h`

Prototype
```
typedef struct {
   int32_t fdListener;
   BtLibSdpRecordHandle serviceRecH;
} BtLibServicePreparationParamsType
```

Fields `fdListener`
The L2Cap or RfComm listener file descriptor.

`serviceRecH`
Empty service record to be filled out.

Comments The *`fdListener`* parameter is a file descriptor opened to an L2Cap or RfComm device instance. On entry, it's already been marked as a listener. On return it must be left unchanged; the Bluetooth system will take care of calling `BtLibSdpServiceRecordStartAdvertising()` to advertise the service, and `BtLibSocketClose()` after an inbound connection is made.

The *serviceRecH* parameter is a handle on a local SDP service record. On entry, it's empty. On exit, it must be set up to describe the service the application has to offer.

In most cases, the application can respond to this launch code by simply calling `BtLibSdpServiceRecordSetAttributesForSocket()`, passing the *fdListener* and *serviceRecH* parameters along with a class UUID and a service name.

In more complicated cases, the application may need to use other SDP functions to make needed changes to the service record. In these cases, it's the application's responsibility to open a Management Entity device instance to pass to those functions and to close that instance before returning.

BtLibServiceRegistrationParamsType Struct

Purpose Service parameters passed to the `BtLibRegisterService()` function.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct {
   uint32_t stackSize;
   uint32_t appType;
   uint32_t appCreator;
   uint16_t appCodeRscId;
   BtLibProtocolEnum protocol;
   uint8_t execAsNormalApp:1, pushSerialModule:1;
} BtLibServiceRegistrationParamsType
```

Fields

`stackSize`
: The service thread's stack size in bytes.

`appType`
: The service application's resource database type.

`appCreator`
: The service application's resource database creator.

`appCodeRscId`
: The resource ID of the application's code resource.

`protocol`
: Which protocol the service uses (L2Cap or RfComm).

`execAsNormalApp`
: A bit flag indicating whether the application should run in the Application Process (1) or the System Process (0).

`pushSerialModule`
: A bit flag indicating whether a serial interface module should be pushed onto the protocol device instance (1) or not (0).

Comments The thread or threads that execute the service's entry points will be created with a stack of at least *`stackSize`* bytes.

The service's preparation entry point is always invoked in the System Process, regardless of the setting of the *`execAsNormalApp`* flag; this flag only controlls where the execution entry point is invoked.

NOTE: In the current version of Palm OS Cobalt, only execution in the System Process is supported, so `execAsNormalApp` should always be 0.

BtLibSocketConnectInfoType Struct

Purpose The `BtLibSocketConnectInfoType` structure allows you to specify the address of the remote device and data specific to the protocol of the socket. The protocol-specific data is stored as a union; the member of the union that is valid depends on the protocol.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibSocketConnectInfoType {
   BtLibDeviceAddressTypePtr remoteDeviceP;
   union {
      struct {
         BtLibL2CapPsmType remotePsm;
         uint16_t minRemoteMtu;
         uint16_t localMtu;
```

```
        } L2Cap;
        struct {
           BtLibRfCommServerIdType remoteService;
           uint8_t advancedCredit;
           uint16_t maxFrameSize;
        } RfComm;
        struct {
           uint16_t localService;
           uint16_t remoteService;
        } bnep;
     } data;
     uint16_t padding;
  } BtLibSocketConnectInfoType
```

Fields

`remoteDeviceP`
: A pointer to a `BtLibDeviceAddressType` that contains the address of the remote device.

`data`
: A union containing protocol-specific information. This union has three members: L2Cap, RfComm, and bnep.

L2Cap

For L2Cap, there are three fields:

`remotePsm`
: A `BtLibL2CapPsmType` representing the protocol and service multiplexer (PSM) identifier of the protocol to which this socket should connect. This identifier is obtained using the Service Discovery Protocol (SDP).

`minRemoteMtu`
: The minimum MTU, or packet size, that your application can support.

`localMtu`
: The MTU, or packet size, of the local device.

RfComm

For RfComm, there are three fields as well:

`remoteService`
: A `BtLibRfCommServerIdType` representing the RfComm service channel on the remote device to

which this socket should connect. This identifier is obtained using the Service Discovery Protocol (SDP).

`advancedCredit`
An amount of credit the socket advances to the remote device when it successfully connects. Additional credit can be advanced using the `BtLibSocketCreate` function once the connection has been established.

`maxFrameSize`
The maximum frame size your application can handle. This value must be between `BT_RF_MINFRAMESIZE` and `BT_RF_MAXFRAMESIZE`. If your application can handle any frame size, set this value to `BT_RF_DEFAULT_FRAMESIZE`.

bnep
There are two fields for BNEP, which indicate which role the local and remove devices should each play. The roles must be one of three 16-bit UUIDs: 0x1115 for PANU, 0x1116 for NAP, and 0x1117 for GN.

`localService`
The UUID of the local role.

`remoteService`
The UUID of the remote role.

`padding`
Reserved for system use.

See Also `BtLibSocketSend()`, `BtLibSocketClose()`

BtLibSocketEventType Struct

Purpose The `BtLibSocketEventType` structure contains detailed information regarding a socket event. All socket events have some common data. Most socket events have additional data specific to those events. The specific data is stored in a union that is part of the `BtLibSocketEvent` data structure.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct _BtLibSocketEventType {
   BtLibSocketEventEnum event;
   uint8_t padding1;
   uint16_t padding2;
   status_t status;
   union {
      BtLibSocketRef newSocket;
      BtLibDeviceAddressType requestingDevice;
      struct {
         BtLibSdpRemoteServiceRecordHandle
remoteHandle;
         union {
            BtLibL2CapPsmType psm;
            BtLibRfCommServerIdType channel;
         } param;
         uint16_t padding;
      } sdpByUuid;
      struct {
         uint16_t numSrvRec;
      } sdpServiceRecordHandles;
      struct {
         BtLibSdpAttributeIdType attributeID;
         uint16_t padding;
         BtLibSdpRecordHandle recordH;
         union {
            struct {
               BtLibSdpAttributeDataType
attributeValues;
               uint16_t listNumber;
               uint16_t listEntry;
            } data;
            struct {
               uint16_t valSize;
            } rawData;
            uint16_t valSize;
            uint16_t strLength;
```

```
            uint16_t numItems;
         } info;
      } sdpAttribute;
   } eventData;
} BtLibSocketEventType
```

Fields

`event`
: `BtLibSocketEventEnum` enum member that indicates which socket event has occurred.

`padding1`
: Reserved for system use.

`padding2`
: Reserved for system use.

`status`
: Status of the event. See "BtLibSocketEventEnum" on page 211 for more details about how to interpret this field for specific events.

`eventData`
: `fieldData` associated with the event. The member of this union that is valid depends on the event. See BtLibSocketEventEnum for more information.

BtLibSocketListenInfoType Struct

Purpose The `BtLibSocketListenInfoType` structure allows you to specify data specific to the protocol of the listening socket. This data is stored in the `data` field, which is a union consisting of two members: L2Cap, and RfComm. The member of the union that is valid depends on the protocol of the listening socket.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibSocketListenInfoType {
   union {
      struct {
         BtLibL2CapPsmType localPsm;
         uint16_t localMtu;
         uint16_t minRemoteMtu;
      } L2Cap;
      struct {
         BtLibRfCommServerIdType serviceID;
```

```
            uint8_t advancedCredit;
            uint16_t maxFrameSize;
         } RfComm;
         struct {
            Boolean listenNAP;
            Boolean listenGN;
            Boolean listenPANU;
         } BNEP;
      } data;
      uint16_t padding;
   } BtLibSocketListenInfoType
```

Fields `data`

A union which can represent L2Cap, RfComm, or BNEP.

L2Cap

L2Cap has the following fields:

`localPsm`

A `BtLibL2CapPsmType` representing the protocol and service multiplexer (PSM) identifier of the protocol to be used with this socket. You can identify your own protocol provided that its PSM value is odd, is within the range of 0x1001 to 0xFFFF, and has the 9th bit (0x0100) set to zero. These limitations are specified by the *Specification of the Bluetooth System*. If you set this field to `BT_L2CAP_RANDOM_PSM`, the `BtLibSocketListen` function automatically creates a suitable PSM for the channel and returns it in this structure.

`localMtu`

The maximum transmission unit (MTU), or packet size, of the local device.

`minRemoteMtu`

The minimum packet size that your application can support.

RfComm

RfComm has the following fields:

`serviceID`
: A `BtLibRfCommServerIdType` representing the socket's RfComm service channel. It is assigned by RfComm and returned in this field when you call `BtLibSocketListen`.

`advancedCredit`
: An amount of credit the socket advances to the remote device when it successfully connects. Additional credit can be advanced using the `BtLibSocketCreate` function once the connection has been established.

`maxFrameSize`
: The maximum frame size your application can handle. This value must be between `BT_RF_MINFRAMESIZE` and `BT_RF_MAXFRAMESIZE`. If your application can handle any frame size, set this value to `BT_RF_DEFAULT_FRAMESIZE`.

BNEP

BNEP has the following fields, which specify which of the three PAN profile services it is willing to support:

`listenNAP`
: `true` if the NAP service is supported.

`listenGN`
: `true` if the GN service is supported.

`listenPANU`
: `true` if the PANU service is supported.

`padding`
: Reserved for system use.

See Also `BtLibSocketClose()`

BtLibSocketRef Typedef

Purpose The `BtLibSocketRef` type identifies a socket.

Declared In `BtLibTypes.h`

Prototype `typedef int32_t BtLibSocketRef`

Comments Note that in versions of Palm OS prior to 6.0, the `BtLibSocketRef` was a 16-bit value.

BtLibStringType Struct

Purpose The `BtLibStringType` structure represents a string in an SDP attribute.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibStringType {
   char *str;
   uint16_t strLen;
   uint16_t padding;
} BtLibStringType
```

Fields

`str`
An array of characters representing the string. This array is not null-terminated.

`strLen`
The length of the string, in bytes.

`padding`
Reserved for system use.

BtLibUrlType Struct

Purpose The `BtLibUrlType` structure represents a uniform resource locator (URL) in an SDP attribute.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct BtLibUrlType {
   char *url;
   uint16_t urlLen;
   uint16_t padding;
} BtLibUrlType
```

Fields

`url`
An array of characters representing the URL. This array is not null-terminated.

`urlLen`
The length of the string, in bytes.

`padding`
Reserved for system use.

sockaddr_bth Struct

Purpose A variant of the BSD Sockets `sockaddr` structure for use with Bluetooth.

Declared In `BtLibTypes.h`

Prototype

```
typedef struct sockaddr_bth {
   sa_family_t sa_family;
   BtLibDeviceAddressType btAddr;
   BtLibSdpUuidType serviceClassId;
   uint8_t padding1;
   uint16_t padding2;
} sockaddr_bth
```

Fields

`sa_family`
The socket address family; for Bluetooth, this should be `AF_BTH`.

`btAddr`
A `BtLibDeviceAddressType` indicating the address of the Bluetooth device. This address is used on the client side to specify the remote Bluetooth device to which to connect. A value of all zeros implies that a discovery operation must be performed to allow the user to select the remote device.

`serviceClassId`
The UUID of the SDP service. On the client side, it specifies the service class to which to connect; on the server side, it specifies the service class to advertise.

`padding1`
Reserved for system use.

`padding2`
Reserved for system use.

Bluetooth Constants

Bluetooth Data Element Sizes

Purpose Define the possible sizes of Bluetooth Data Elements.

Declared In `BtLibTypes.h`

Constants

Table 13.10Bluetooth Data Element sizes

Constant	Meaning
`btLibDESD_1BYTE`	Specifies a 1-byte element. However, if the element type is `btLibDETD_NIL`, then the size is actually 0.
`btLibDESD_2BYTES`	Specifies a 2-byte element.
`btLibDESD_4BYTES`	Specifies a 4-byte element.
`btLibDESD_8BYTES`	Specifies an 8-byte element.
`btLibDESD_16BYTES`	Specifies a 16-byte element.
`btLibDESD_ADD_8BITS`	The element's actual data size, in bytes, is contained in the next eight bits.
`btLibDESD_ADD_16BITS`	The element's actual data size, in bytes, is contained in the next 16 bits.
`btLibDESD_ADD_32BITS`	The element's actual data size, in bytes, is contained in the next 32 bits.
`btLibDESD_MASK`	AND this value with the first byte of a Data Element to obtain the element's size.

See Also "Bluetooth Data Element Types"

Bluetooth Data Element Types

Purpose Define the types of Data Elements supported by the Bluetooth system.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`btLibDETD_ALT`	Specifies a Data Element alternative. The data contains a sequence of Data Elements. This type is sometimes used to distinguish between two possible sequences. Must use size `btLibDESD_ADD_8BITS`, `btLibDESD_ADD_16BITS`, or `btLibDESD_ADD_32BITS`.
`btLibDETD_BOOL`	Specifies a Boolean value. Must use size `btLibDESD_1BYTE`.
`btLibDETD_NIL`	Specifies nil, the null type. Requires a size of `btLibDESD_1BYTE`, which for this type actually means 0 bytes.
`btLibDETD_SEQ`	Specifies a Data Element sequence. The data contains a sequence of Data Elements. Must use size `btLibDESD_ADD_8BITS`, `btLibDESD_ADD_16BITS`, or `btLibDESD_ADD_32BITS`.
`btLibDETD_SINT`	Specifies a signed integer. Must use size `btLibDESD_1BYTE`, `btLibDESD_2BYTES`, `btLibDESD_4BYTES`, `btLibDESD_8BYTES`, or `btLibDESD_16BYTES`

Constant	Meaning
`btLibDETD_TEXT`	Specifies a text string. Must use size `btLibDESD_ADD_8BITS`, `btLibDESD_ADD_16BITS`, or `btLibDESD_ADD_32BITS`.
`btLibDETD_UINT`	Specifies an unsigned integer. Must use size `btLibDESD_1BYTE`, `btLibDESD_2BYTES`, `btLibDESD_4BYTES`, `btLibDESD_8BYTES`, or `btLibDESD_16BYTES`.
`btLibDETD_URL`	Specifies a Uniform Resource Locator (URL). Must use size `btLibDESD_ADD_8BITS`, `btLibDESD_ADD_16BITS`, or `btLibDESD_ADD_32BITS`.
`btLibDETD_UUID`	Specifies a Universally Unique Identifier (UUID). Must use size `btLibDESD_2BYTES`, `btLibDESD_4BYTES`, or `btLibDESD_16BYTES`.
`btLibDETD_MASK`	AND this value with the first byte of a Data Element to obtain the element’s type.

See Also “Bluetooth Data Element Sizes”

Bluetooth Device Names

Purpose Define the names of Bluetooth STREAMS devices.

Declared In `BtLibTypes.h`

Constants

Table 13.11 Bluetooth device names

Constant	Meaning
`btDevMeName`	Management Entity device.
`btDevL2cName`	L2Cap device.
`btDevRfcName`	RfComm device.
`btDevSdpName`	SDP device.
`btDevSCOName`	SCO device.
`btDevBNEPName`	BNEP device.

Bluetooth Disconnection Codes

Purpose Values for the status field of `btLibSocketEventDisconnected` events, which explain why the disconnect occurred.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`btLibL2DiscReasonUnknown`	Unknown reason.
`btLibL2DiscUserRequest`	Either the local or remote user requested disconnection.
`btLibL2DiscRequestTimeout`	An L2Cap request timed out.
`btLibL2DiscLinkDisc`	The underlying ACL link disconnected.
`btLibL2DiscQosViolation`	Quality of Service violation.
`btLibL2DiscSecurityBlock`	Local Security Manager refused the connection.

Constant	Meaning
`btLibL2DiscConnPsmUnsupported`	The remote device does not support the requested PSM.
`btLibL2DiscConnSecurityBlock`	The remote Security Manager refused the connection.
`btLibL2DiscConnNoResources`	Remote device is out of resources.
`btLibL2DiscConfigUnacceptable`	Configuration failed due to invalid parameters.
`btLibL2DiscConfigReject`	Configuration rejected for unknown reasons.
`btLibL2DiscConfigOptions`	Configuration failed due to unrecognized configuration options.

Bluetooth Error Codes

Purpose Error codes that can occur when issuing Bluetooth calls.

Declared In `BtLibTypes.h`

Constants

Error	Description
`btLibErrNoError`	Success.
`btLibErrAlreadyConnected`	A connection is already in place.
`btLibErrAlreadyOpen`	The Bluetooth Library is already open (this isn't an error, just a friendly notification).
`btLibErrBatteryTooLow`	The battery power is too low to perform the requested operation.
`btLibErrBluetoothOff`	The user has turned off Bluetooth.
`btLibErrBusy`	A needed resource is busy.
`btLibErrCanceled`	The operation was canceled.

Error	Description
`btLibErrError`	Generic error.
`btLibErrFailed`	Remote operation completed but failed.
`btLibErrInProgress`	An operation is already in progress.
`btLibErrInUseByService`	The resource is in use by a service.
`btLibErrNoAclLink`	No ACL link to the remote device.
`btLibErrNoAdminDaemon`	The daemon has not opened the admin device.
`btLibErrNoConnection`	No connection on socket.
`btLibErrNoPiconet`	A piconet is required for this operation.
`btLibErrNoPrefs`	The preferences are missing.
`btLibErrNotFound`	The requested value was not found.
`btLibErrNotInProgress`	Operation is not in progress.
`btLibErrOutOfMemory`	Memory allocation failed.
`btLibErrParamError`	Invalid parameter to function.
`btLibErrPending`	Operation will complete later; status and results will arrive in an event.
`btLibErrRadioFatal`	The Bluetooth hardware has failed while in use.
`btLibErrRadioInitFailed`	Initialization of the Bluetooth radio failed.
`btLibErrRadioInitialized`	The Bluetooth hardware was initialized successfully. This isn't an error, just a notification.
`btLibErrRadioSleepWake`	The Bluetooth hareware failed because the device went to sleep.
`btLibErrRoleChange`	Could not perform master/slave role switch.

Error	Description
`btLibErrError`	Generic error.
`btLibErrFailed`	Remote operation completed but failed.
`btLibErrInProgress`	An operation is already in progress.
`btLibErrInUseByService`	The resource is in use by a service.
`btLibErrNoAclLink`	No ACL link to the remote device.
`btLibErrNoAdminDaemon`	The daemon has not opened the admin device.
`btLibErrNoConnection`	No connection on socket.
`btLibErrNoPiconet`	A piconet is required for this operation.
`btLibErrNoPrefs`	The preferences are missing.
`btLibErrNotFound`	The requested value was not found.
`btLibErrNotInProgress`	Operation is not in progress.
`btLibErrOutOfMemory`	Memory allocation failed.
`btLibErrParamError`	Invalid parameter to function.
`btLibErrPending`	Operation will complete later; status and results will arrive in an event.
`btLibErrRadioFatal`	The Bluetooth hardware has failed while in use.
`btLibErrRadioInitFailed`	Initialization of the Bluetooth radio failed.
`btLibErrRadioInitialized`	The Bluetooth hardware was initialized successfully. This isn't an error, just a notification.
`btLibErrRadioSleepWake`	The Bluetooth hareware failed because the device went to sleep.
`btLibErrRoleChange`	Could not perform master/slave role switch.

Error	Description
`btLibErrSdpAdvertised`	Invalid operation on an advertised record.
`btLibErrSdpAttributeNotSet`	Attribute is not set for record.
`btLibErrSdpFormat`	Service record is improperly formatted.
`btLibErrSdpInvalidResponse`	Invalid data in SDP response.
`btLibErrSdpMapped`	Invalid operation on mapped record.
`btLibErrSdpNotAdvertised`	Invalid operation on an unadvertised record.
`btLibErrSdpNotMapped`	Invalid operation on an unmapped record.
`btLibErrSdpQueryContinuation`	Invalid continuation data.
`btLibErrSdpQueryDisconnect`	SDP disconnected.
`btLibErrSdpQueryHandle`	Invalid service record handle.
`btLibErrSdpQueryPduSize`	Invalid Protocol Data Unit (PDU) size.
`btLibErrSdpQueryResources`	Insufficient resources for request.
`btLibErrSdpQuerySyntax`	Invalid request syntax.
`btLibErrSdpQueryVersion`	Invalid or unsupported SDP version.
`btLibErrSdpRemoteRecord`	Invalid operation on the remote SDP record.
`btLibErrSocket`	Invalid socket reference.
`btLibErrSocketChannelUnavailable`	Channel unavailable on remote device.
`btLibErrSocketProtocol`	Invalid protocol for operation.
`btLibErrSocketPsmUnavailable`	PSM is already in use.
`btLibErrSocketRole`	Invalid role (connecor/listener).
`btLibErrSocketUserDisconnect`	The user terminated the connection.

Error	Description
`btLibErrTooMany`	Capacity reached (specific meaning varies depending on the function called).
`btLibNotYetSupported`	Unsupported feature.

Bluetooth Module Names

Purpose Names of the Bluetooth STREAMS modules.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`btModSerL2cName`	Serial-on-L2Cap module.
`btModSerRfcName`	Serial-on-RfComm module.
`btModTPISerRfcName`	TPI-on-serial-on-RfComm module.

BSD Sockets Constants

Purpose Constants used when utilizing the BSD Sockets API.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`BTADDR_ANY`	Represents any Bluetooth device address for BSD Sockets API calls.
`BTHPROTO_RFCOMM`	The protocol to use when creating an RfComm socket using the BSD Sockets API.

Character Encoding Constants

Purpose Define character encodings for Bluetooth.

Declared In `BtLibTypes.h`

Constants

btLibCharSet_Adobe_Standard_Encoding	btLibCharSet_Adobe_Symbol_Encoding
btLibCharSet_ANSI_X3_110_1983	btLibCharSet_ASMO_449
btLibCharSet_Big5	btLibCharSet_Big5_HKSCS
btLibCharSet_BS_4730	btLibCharSet_BS_viewdata
btLibCharSet_CSA_Z243_4_1985_1	btLibCharSet_CSA_Z243_4_1985_2
btLibCharSet_CSA_Z243_4_1985_gr	btLibCharSet_CSN_369103
btLibCharSet_DEC_MCS	btLibCharSet_DIN_66003
btLibCharSet_dk_us	btLibCharSet_DS_2089
btLibCharSet_EBCDIC_AT_DE	btLibCharSet_EBCDIC_AT_DE_A
btLibCharSet_EBCDIC_CA_FR	btLibCharSet_EBCDIC_DK_NO
btLibCharSet_EBCDIC_DK_NO_A	btLibCharSet_EBCDIC_ES
btLibCharSet_EBCDIC_ES_A	btLibCharSet_EBCDIC_ES_S
btLibCharSet_EBCDIC_FI_SE	btLibCharSet_EBCDIC_FI_SE_A
btLibCharSet_EBCDIC_FR	btLibCharSet_EBCDIC_IT
btLibCharSet_EBCDIC_PT	btLibCharSet_EBCDIC_UK
btLibCharSet_EBCDIC_US	btLibCharSet_ECMA_cyrillic
btLibCharSet_ES	btLibCharSet_ES2
btLibCharSet_EUC_JP	btLibCharSet_EUC_KR
btLibCharSet_Extended_UNIX_Code_Fixed_Width_for_Japanese	btLibCharSet_GB2312
btLibCharSet_GB_1988_80	btLibCharSet_GB_2312_80
btLibCharSet_GOST_19768_74	btLibCharSet_greek7

btLibCharSet_greek7_old	btLibCharSet_greek_ccitt
btLibCharSet_HP_DeskTop	btLibCharSet_HP_Legal
btLibCharSet_HP_Math8	btLibCharSet_HP_Pi_font
btLibCharSet_hp_roman8	btLibCharSet_HZ_GB_2312
btLibCharSet_IBM00858	btLibCharSet_IBM00924
btLibCharSet_IBM01140	btLibCharSet_IBM01141
btLibCharSet_IBM01142	btLibCharSet_IBM01143
btLibCharSet_IBM01144	btLibCharSet_IBM01145
btLibCharSet_IBM01146	btLibCharSet_IBM01147
btLibCharSet_IBM01148	btLibCharSet_IBM01149
btLibCharSet_IBM037	btLibCharSet_IBM038
btLibCharSet_IBM1026	btLibCharSet_IBM273
btLibCharSet_IBM274	btLibCharSet_IBM275
btLibCharSet_IBM277	btLibCharSet_IBM278
btLibCharSet_IBM280	btLibCharSet_IBM281
btLibCharSet_IBM284	btLibCharSet_IBM285
btLibCharSet_IBM290	btLibCharSet_IBM297
btLibCharSet_IBM420	btLibCharSet_IBM423
btLibCharSet_IBM424	btLibCharSet_IBM437
btLibCharSet_IBM500	btLibCharSet_IBM775
btLibCharSet_IBM850	btLibCharSet_IBM851
btLibCharSet_IBM852	btLibCharSet_IBM855
btLibCharSet_IBM857	btLibCharSet_IBM860
btLibCharSet_IBM861	btLibCharSet_IBM862
btLibCharSet_IBM863	btLibCharSet_IBM864

btLibCharSet_IBM865	btLibCharSet_IBM866
btLibCharSet_IBM868	btLibCharSet_IBM869
btLibCharSet_IBM870	btLibCharSet_IBM871
btLibCharSet_IBM880	btLibCharSet_IBM891
btLibCharSet_IBM903	btLibCharSet_IBM904
btLibCharSet_IBM905	btLibCharSet_IBM918
btLibCharSet_IBM_Symbols	btLibCharSet_IBM_Thai
btLibCharSet_IEC_P27_1	btLibCharSet_INIS
btLibCharSet_INIS_8	btLibCharSet_INIS_cyrillic
btLibCharSet_INVARIANT	btLibCharSet_ISO_10367_box
btLibCharSet_ISO_10646_UCS_2	btLibCharSet_ISO_10646_UCS_4
btLibCharSet_ISO_10646_UCS_Basic	btLibCharSet_ISO_10646_Unicode_Latin1
btLibCharSet_ISO_10646_UTF_1	btLibCharSet_ISO_2022_CN
btLibCharSet_ISO_2022_CN_EXT	btLibCharSet_ISO_2022_JP
btLibCharSet_ISO_2022_JP_2	btLibCharSet_ISO_2022_KR
btLibCharSet_ISO_2033_1983	btLibCharSet_ISO_5427
btLibCharSet_ISO_5427_1981	btLibCharSet_ISO_5428_1980
btLibCharSet_ISO_646_basic_1983	btLibCharSet_ISO_646_irv_1983
btLibCharSet_ISO_6937_2_25	btLibCharSet_ISO_6937_2_add
btLibCharSet_ISO_8859_1	btLibCharSet_ISO_8859_10
btLibCharSet_iso_8859_13	btLibCharSet_iso_8859_14
btLibCharSet_ISO_8859_15	btLibCharSet_ISO_8859_1_Windows_3_0_Latin_1
btLibCharSet_ISO_8859_1_Windows_3_1_Latin_1	btLibCharSet_ISO_8859_2

btLibCharSet_ISO_8859_2_Windows_Latin_2	btLibCharSet_ISO_8859_3
btLibCharSet_ISO_8859_4	btLibCharSet_ISO_8859_5
btLibCharSet_ISO_8859_6	btLibCharSet_ISO_8859_6_E
btLibCharSet_ISO_8859_6_I	btLibCharSet_ISO_8859_7
btLibCharSet_ISO_8859_8	btLibCharSet_ISO_8859_8_E
btLibCharSet_ISO_8859_8_I	btLibCharSet_ISO_8859_9
btLibCharSet_ISO_8859_9_Windows_Latin_5	btLibCharSet_ISO_8859_supp
btLibCharSet_iso_ir_90	btLibCharSet_ISO_Unicode_IBM_1261
btLibCharSet_ISO_Unicode_IBM_1264	btLibCharSet_ISO_Unicode_IBM_1265
btLibCharSet_ISO_Unicode_IBM_1268	btLibCharSet_ISO_Unicode_IBM_1276
btLibCharSet_IT	btLibCharSet_JIS_C6220_1969_jp
btLibCharSet_JIS_C6220_1969_ro	btLibCharSet_JIS_C6226_1978
btLibCharSet_JIS_C6226_1983	btLibCharSet_JIS_C6229_1984_a
btLibCharSet_JIS_C6229_1984_b	btLibCharSet_JIS_C6229_1984_b_add
btLibCharSet_JIS_C6229_1984_hand	btLibCharSet_JIS_C6229_1984_hand_add
btLibCharSet_JIS_C6229_1984_kana	btLibCharSet_JIS_Encoding
btLibCharSet_JIS_X0201	btLibCharSet_JIS_X0212_1990
btLibCharSet_JUS_I_B1_002	btLibCharSet_JUS_I_B1_003_mac
btLibCharSet_JUS_I_B1_003_serb	btLibCharSet_KOI8_R
btLibCharSet_KOI8_U	btLibCharSet_KSC5636
btLibCharSet_KS_C_5601_1987	btLibCharSet_latin_greek
btLibCharSet_Latin_greek_1	btLibCharSet_latin_lap
btLibCharSet_macintosh	btLibCharSet_Microsoft_Publishing
btLibCharSet_MNEM	btLibCharSet_MNEMONIC

btLibCharSet_MSZ_7795_3	btLibCharSet_NATS_DANO
btLibCharSet_NATS_DANO_ADD	btLibCharSet_NATS_SEFI
btLibCharSet_NATS_SEFI_ADD	btLibCharSet_NC_NC00_10_81
btLibCharSet_NF_Z_62_010	btLibCharSet_NF_Z_62_010__1973_
btLibCharSet_NS_4551_1	btLibCharSet_NS_4551_2
btLibCharSet_PC8_Danish_Norwegian	btLibCharSet_PC8_Turkish
btLibCharSet_PT	btLibCharSet_PT2
btLibCharSet_SCSU	btLibCharSet_SEN_850200_B
btLibCharSet_SEN_850200_C	btLibCharSet_Shift_JIS
btLibCharSet_TIS_620	btLibCharSet_T_101_G2
btLibCharSet_T_61_7bit	btLibCharSet_T_61_8bit
btLibCharSet_UNICODE_1_1	btLibCharSet_UNICODE_1_1_UTF_7
btLibCharSet_UNKNOWN_8BIT	btLibCharSet_US_ASCII
btLibCharSet_us_dk	btLibCharSet_UTF_16
btLibCharSet_UTF_16BE	btLibCharSet_UTF_16LE
btLibCharSet_UTF_7	btLibCharSet_UTF_8
btLibCharSet_Ventura_International	btLibCharSet_Ventura_Math
btLibCharSet_Ventura_US	btLibCharSet_videotex_suppl
btLibCharSet_VIQR	btLibCharSet_VISCII
btLibCharSet_windows_1250	btLibCharSet_windows_1251
btLibCharSet_windows_1252	btLibCharSet_windows_1253
btLibCharSet_windows_1254	btLibCharSet_windows_1255
btLibCharSet_windows_1256	btLibCharSet_windows_1257
btLibCharSet_windows_1258	btLibCharSet_Windows_31J

L2Cap Constants

Purpose Constants for the L2Cap protocol.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`BT_L2CAP_MTU`	The maximum size for L2Cap frames.
`BT_L2CAP_RANDOM_PSM`	Used when creating a listener socket; instructs the system to select a random, unused PSM.

Comments The `BT_L2CAP_RANDOM_PSM` constant lets you ask the system to select an available Protocol Service Multiplexor (PSM) for you when creating an L2Cap listener socket, as seen in Listing 13.1.

Listing 13.1 Creating an L2Cap listener socket

```
listenInfo.data.L2Cap.localPsm = BT_L2CAP_RANDOM_PSM;
listenInfo.data.L2Cap.localMtu = MAX_FRAME_SIZE_L2CAP;
listenInfo.data.L2Cap.minRemoteMtu = MAX_FRAME_SIZE_L2CAP;
err = BtLibSocketListen(socket, &listenInfo);
```

Language ID Constants

Purpose Define languages supported by the Bluetooth system.

Declared In `BtLibTypes.h`

Constants

`btLibLangAbkihazian`	`btLibLangAfar`
`btLibLangAfrikaans`	`btLibLangAlbanian`
`btLibLangAmharic`	`btLibLangArabic`
`btLibLangArmenian`	`btLibLangAssamese`

`btLibLangAymara`	`btLibLangAzerbaijani`
`btLibLangBashkir`	`btLibLangBasque`
`btLibLangBengali`	`btLibLangBhutani`
`btLibLangBihari`	`btLibLangBislama`
`btLibLangBreton`	`btLibLangBulgarian`
`btLibLangBurmese`	`btLibLangByelorussian`
`btLibLangCambodian`	`btLibLangCatalan`
`btLibLangChinese`	`btLibLangCorsican`
`btLibLangCroation`	`btLibLangCzech`
`btLibLangDanish`	`btLibLangDutch`
`btLibLangEnglish`	`btLibLangEsperanto`
`btLibLangEstonian`	`btLibLangFaroese`
`btLibLangFiji`	`btLibLangFinnish`
`btLibLangFrench`	`btLibLangFrisian`
`btLibLangGalician`	`btLibLangGeorgian`
`btLibLangGerman`	`btLibLangGreek`
`btLibLangGreenlandic`	`btLibLangGuarani`
`btLibLangGujarati`	`btLibLangHausa`
`btLibLangHebrew`	`btLibLangHindi`
`btLibLangHungarian`	`btLibLangIcelandic`
`btLibLangIndonesian`	`btLibLangInterlingua`
`btLibLangInterlingue`	`btLibLangInupiak`
`btLibLangIrish`	`btLibLangItalian`
`btLibLangJapanese`	`btLibLangJavanese`
`btLibLangKannada`	`btLibLangKashmiri`

`btLibLangKazakh`	`btLibLangKinyarwanda`
`btLibLangKirghiz`	`btLibLangKirundi`
`btLibLangKorean`	`btLibLangKurdish`
`btLibLangLaothian`	`btLibLangLatin`
`btLibLangLatvian`	`btLibLangLingala`
`btLibLangLithuanian`	`btLibLangMacedonian`
`btLibLangMalagasy`	`btLibLangMalay`
`btLibLangMalayalam`	`btLibLangMaltese`
`btLibLangMaori`	`btLibLangMarathi`
`btLibLangMoldavian`	`btLibLangMongolian`
`btLibLangNaura`	`btLibLangNepali`
`btLibLangNorwegian`	`btLibLangOccitan`
`btLibLangOriya`	`btLibLangOromo`
`btLibLangPashto`	`btLibLangPersian`
`btLibLangPolish`	`btLibLangPortuguese`
`btLibLangPunjabi`	`btLibLangQuechua`
`btLibLangRhaeto_Romance`	`btLibLangRomanian`
`btLibLangRussian`	`btLibLangSamoan`
`btLibLangSangho`	`btLibLangSanskrit`
`btLibLangScotsGaelic`	`btLibLangSerbian`
`btLibLangSerbo_Croation`	`btLibLangSesotho`
`btLibLangSetswanna`	`btLibLangShona`
`btLibLangSindhi`	`btLibLangSinghalese`
`btLibLangSiswati`	`btLibLangSlovak`
`btLibLangSlovenian`	`btLibLangSomali`

btLibLangSpanish	btLibLangSundanese
btLibLangSwahili	btLibLangSwedish
btLibLangTagalog	btLibLangTajik
btLibLangTamil	btLibLangTatar
btLibLangTelugu	btLibLangThai
btLibLangTibetan	btLibLangTigrinya
btLibLangTonga	btLibLangTsonga
btLibLangTurkish	btLibLangTurkmen
btLibLangTwi	btLibLangUkranian
btLibLangUrdu	btLibLangUzbek
btLibLangVietnamese	btLibLangVolapuk
btLibLangWelsh	btLibLangWolof
btLibLangXhosa	btLibLangYiddish
btLibLangYoruba	btLibLangZulu

Management Event Status Codes

Purpose When a management event is generated, the `status` field of the associated `BtLibStringType` provides information about why the event occurred. The following status codes can occur with a management event.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
btLibErrNoError	Success.
btLibMeStatusAuthenticateFailure	Authentication failure.
btLibMeStatusCommandDisallowed	Command disallowed.

Constant	Meaning
`btLibMeStatusConnnectionTimeout`	Connection timed out.
`btLibMeStatusHardwareFailure`	Hardware failure.
`btLibMeStatusHostTimeout`	Host timeout.
`btLibMeStatusInvalidHciParam`	Invalid HCI command parameters.
`btLibMeStatusInvalidLmpParam`	Invalid LMP parameters.
`btLibMeStatusLimitedResources`	Host rejected due to limited resources.
`btLibMeStatusLmpPduNotAllowed`	LMP PDU not allowed.
`btLibMeStatusLmpResponseTimeout`	Timeout waiting for LMP response.
`btLibMeStatusLmpTransdCollision`	LMP error transaction collission.
`btLibMeStatusLocalTerminated`	Connection terminated by local host.
`btLibMeStatusLowResources`	Connection terminated by remote device due to low resources.
`btLibMeStatusMaxAclConnections`	Reached maximum number of ACL connections.
`btLibMeStatusMaxConnections`	Reached maximum number of connections.
`btLibMeStatusMaxScoConnections`	Reached maximum number of SCO connections.
`btLibMeStatusMemoryFull`	Not enough memory.
`btLibMeStatusMissingKey`	Missing key.
`btLibMeStatusNoConnection`	No connection.
`btLibMeStatusPageTimeout`	Page timeout.
`btLibMeStatusPairingNotAllowed`	Pairing not allowed.
`btLibMeStatusPersonalDevice`	Host rejected; remote is a personal device.

Constant	Meaning
`btLibMeStatusPowerOff`	Connection terminated due to remote device powering off.
`btLibMeStatusRepeatedAttempts`	Repeated attempts.
`btLibMeStatusRoleChangeNotAllowed`	Can't perform master/slave role switch.
`btLibMeStatusScoAirModeRejected`	SCO air mode rejected.
`btLibMeStatusScoIntervalRejected`	SCO interval rejected.
`btLibMeStatusScoOffsetRejected`	SCO offset rejected.
`btLibMeStatusSecurityError`	Host rejected for security reasons.
`btLibMeStatusUnknownHciCommand`	Unknown HCI command.Unknown HCI command.
`btLibMeStatusUnknownLmpPDU`	Unknown LMP PDU.
`btLibMeStatusUnspecifiedError`	Unspecified error.
`btLibMeStatusUnsupportedFeature`	Unsupported feature or parameter value.
`btLibMeStatusUnsupportedLmpParam`	Unsupported LMP parameter value.
`btLibMeStatusUnsupportedRemote`	Unsupported remote feature.
`btLibMeStatusUserTerminated`	Remote user terminated the connection.

Miscellaneous Bluetooth Constants

Purpose These constants don't fit into other categories, but are important nonetheless.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`btLibDeviceAddressSize`	The size, in bytes, of a Bluetooth address.
`btLibFeatureCreator`	The Bluetooth Library's creator ID, for use when calling the Feature Manager.
`btLibFeatureVersion`	The feature ID of the Bluetooth Library's version number.
`btLibMaxDeviceNameLength`	The maximum length of a Bluetooth device's user-friendly name.
`btLibMaxSrvRecListLen`	The maximum number of entries in a service record list.

Attribute Identifier Constants

Purpose Define offsets for human-readable attributes that can be provided in multiple languages.

Declared In `BtLibTypes.h`

Constants

Constant	Meaning
`btLibServiceNameOffset`	Offset to the human-readable service name attribute.
`btLibServiceDescriptionOffset`	Offset to the human-readable service description attribute
`btLibProviderNameOffset`	Offset to the human-readable provider name attribute.

Comments In order to support multiple natural languages for human-readable attributes, a service record can contain a `btLibLanguageBaseAttributeIdList` attribute. This attribute is a list of triplets indicating the language ID, character encoding ID, and base attribute ID for each language for which a language is available.

Then these language support offsets are used in tandem with the language base attribute ID list to locate the actual string for the attribute in the desired language. For example, to locate the French version of the service's name, you would search the service record's `btLibLanguageBaseAttributeIdList` attribute for a triplet whose language ID is `btLibLangFrench`, get the base attribute ID from that triplet, and add `btLibServiceNameOffset` to that.

The resulting value is the ID of the attribute containing the service name in French. Your application can then display the string using the character encoding from the triplet.

Protocol UUIDs

Purpose Raw values for protocol UUIDs that are predefined by the Bluetooth specification.

Declared In `BtLibTypes.h`

Constants

Constant
`btLibSdpUUID_PROT_AVCTP`
`btLibSdpUUID_PROT_AVDTP`
`btLibSdpUUID_PROT_BNEP`
`btLibSdpUUID_PROT_CMTP`
`btLibSdpUUID_PROT_FTP`
`btLibSdpUUID_PROT_HARDCOPY_CONTROL_CHANNEL`
`btLibSdpUUID_PROT_HARDCOPY_DATA_CHANNEL`
`btLibSdpUUID_PROT_HARDCOPY_NOTIFICATION`
`btLibSdpUUID_PROT_HIDP`
`btLibSdpUUID_PROT_HTTP`
`btLibSdpUUID_PROT_IP`
`btLibSdpUUID_PROT_L2CAP`
`btLibSdpUUID_PROT_OBEX`
`btLibSdpUUID_PROT_RFCOMM`
`btLibSdpUUID_PROT_SDP`
`btLibSdpUUID_PROT_TCP`
`btLibSdpUUID_PROT_TCS_AT`
`btLibSdpUUID_PROT_TCS_BIN`
`btLibSdpUUID_PROT_UDI_C_PLANE`
`btLibSdpUUID_PROT_UDP`

Constant
btLibSdpUUID_PROT_UPNP
btLibSdpUUID_PROT_WSP

RfComm Constants

Purpose Constants for the RfCommprotocol.

Declared In BtLibTypes.h

Constants

Constant	Meaning
BT_RF_DEFAULT_FRAMESIZE	The default size of an RFCOMM frame.
BT_RF_MAX_FRAMESIZE	The maximum size of an RFCOMM frame.
BT_RF_MIN_FRAMESIZE	The minimum size of an RFCOMM frame.

Service Class UUIDs

Purpose Raw values for service class UUIDs predefined by the Bluetooth specification.

Declared In BtLibTypes.h

Constants

Constant
btLibSdpUUID_SC_ADVANCED_AUDIO_DISTRIBUTION
btLibSdpUUID_SC_AUDIO_SINK
btLibSdpUUID_SC_AUDIO_SOURCE
btLibSdpUUID_SC_AUDIO_VIDEO

Constant
btLibSdpUUID_SC_AV_REMOTE_CONTROL
btLibSdpUUID_SC_AV_REMOTE_CONTROL_TARGET
btLibSdpUUID_SC_BASIC_PRINTING
btLibSdpUUID_SC_BROWSE_GROUP_DESC
btLibSdpUUID_SC_COMMON_ISDN_ACCESS
btLibSdpUUID_SC_CORDLESS_TELEPHONY
btLibSdpUUID_SC_DIALUP_NETWORKING
btLibSdpUUID_SC_DIRECT_PRINTING
btLibSdpUUID_SC_DIRECT_PRINTING_REF_OBJ
btLibSdpUUID_SC_ESDP_UPNP_IP_LAP
btLibSdpUUID_SC_ESDP_UPNP_IP_PAN
btLibSdpUUID_SC_ESDP_UPNP_L2CAP
btLibSdpUUID_SC_FAX
btLibSdpUUID_SC_GENERIC_AUDIO
btLibSdpUUID_SC_GENERIC_FILE_TRANSFER
btLibSdpUUID_SC_GENERIC_NETWORKING
btLibSdpUUID_SC_GENERIC_TELEPHONY
btLibSdpUUID_SC_GN
btLibSdpUUID_SC_HANDSFREE
btLibSdpUUID_SC_HANDSFREE_AUDIO_GATEWAY
btLibSdpUUID_SC_HARDCOPY_CABLE_REPLACEMENT
btLibSdpUUID_SC_HCR_PRINT
btLibSdpUUID_SC_HCR_SCAN
btLibSdpUUID_SC_HEADSET

Constant
btLibSdpUUID_SC_HEADSET_AUDIO_GATEWAY
btLibSdpUUID_SC_HUMAN_INTERFACE_DEVICE
btLibSdpUUID_SC_IMAGING
btLibSdpUUID_SC_IMAGING_AUTOMATIC_ARCHIVE
btLibSdpUUID_SC_IMAGING_REFERENCED_OBJECTS
btLibSdpUUID_SC_IMAGING_RESPONDER
btLibSdpUUID_SC_INTERCOM
btLibSdpUUID_SC_IRMC_SYNC
btLibSdpUUID_SC_IRMC_SYNC_COMMAND
btLibSdpUUID_SC_IRMC_SYNC_COMMAND
btLibSdpUUID_SC_LAN_ACCESS_PPP
btLibSdpUUID_SC_NAP
btLibSdpUUID_SC_OBEX_FILE_TRANSFER
btLibSdpUUID_SC_OBEX_OBJECT_PUSH
btLibSdpUUID_SC_PANU
btLibSdpUUID_SC_PNP_INFORMATION
btLibSdpUUID_SC_PRINTING_STATUS
btLibSdpUUID_SC_PUBLIC_BROWSE_GROUP
btLibSdpUUID_SC_REFERENCE_PRINTING
btLibSdpUUID_SC_REFLECTED_UI
btLibSdpUUID_SC_SERIAL_PORT
btLibSdpUUID_SC_SERVICE_DISCOVERY_SERVER
btLibSdpUUID_SC_SIM_ACCESS
btLibSdpUUID_SC_UDI_MT

Constant
`btLibSdpUUID_SC_UDI_TA`
`btLibSdpUUID_SC_UPNP_IP_SERVICE`
`btLibSdpUUID_SC_UPNP_SERVICE`
`btLibSdpUUID_SC_VIDEO_CONFERENCING`
`btLibSdpUUID_SC_VIDEO_CONFERENCING_GW`
`btLibSdpUUID_SC_WAP`
`btLibSdpUUID_SC_WAP_CLIENT`

Service Description Flags

Purpose Flags used by the sysBtLaunchCmdDescribeLaunchService launch code.

Declared In `BtLibTypes.h`

Constants `btLibServDescFlag_CAN_DO_UI`

If set, indicates that the service application is capable of responding to the `sysBtLaunchCmdDoServiceUI` launch code. When the service is selected in the services view of the Bluetooth panel, an “Advanced” button will appear, and tapping that button will cause the launch code to be sent to the service, which should do some sort of user interface specific to the service.

BtLibAccessibleModeEnum Enum

Purpose The `BtLibAccessibleModeEnum` enum specifies a device’s accessibility modes. See the “Generic Access Profile” chapter of the *Specification of the Bluetooth System* for more information about accessibility.

Declared In `BtLibTypes.h`

Constants `btLibNotAccessible = 0x00`

The device does not respond to a page or an inquiry.

`btLibConnectableOnly = 0x02`
The device responds to a page but not an inquiry.

`btLibDiscoverableAndConnectable = 0x03`
The device responds to both a page and an inquiry.

BtLibConnectionRoleEnum Enum

Purpose The `BtLibConnectionRoleEnum` enum specifies all the connection roles a device can have. A device can either be a master or a slave.

Declared In `BtLibTypes.h`

Constants

`btLibMasterRole`
The device is a master.

`btLibSlaveRole`
The device is a slave.

BtLibGeneralPrefEnum Enum

Purpose The `BtLibGeneralPreferenceEnum` enum specifies the general preferences that can be accessed using the `BtLibSetGeneralPreference()` and `BtLibGetGeneralPreference()` functions.

Declared In `BtLibTypes.h`

Constants

`btLibPref_Name`
This preference is a `BtLibFriendlyNameType` containing the user-friendly name of the local device.

`btLibPref_UnconnectedAccessible`
preference is a `BtLibAccessibleModeEnum` indicating the accessibility mode of the local device when it is unconnected.

`btLibPref_CurrentAccessible`
This preference is a `BtLibAccessibleModeEnum` indicating the current accessibility mode of the local device. You cannot set this preference.

`btLibPref_LocalClassOfDevice`
This preference is a `BtLibClassOfDeviceType` indicating the class of the local device.

`btLibPref_LocalDeviceAddress`
This preference is a `BtLibDeviceAddressType` indicating the address of the local device. You cannot set this preference.

See Also `BtLibGetGeneralPreference()`, `BtLibSetGeneralPreference()`

BtLibGetNameEnum Enum

Purpose The `BtLibGetNameEnum` enum specifies whether to retrieve a device name from the cache, the remote device, or both.

Declared In `BtLibTypes.h`

Constants

`btLibCachedThenRemote`
Look for a name in the cache. If the name is not in the cache, ask the remote device.

`btLibCachedOnly`
Look for a name in the cache. If the name is not in the cache, fail.

`btLibRemoteOnly`
Ignore any cached names and ask the remote device for its name.

See Also `BtLibGetRemoteDeviceName()`, `BtLibGetRemoteDeviceNameSynchronous()`

BtLibLinkModeEnum Enum

Purpose The `BtLibLinkModeEnum` enum specifies the modes a slave can have. According to the *Specification of the Bluetooth System*, a slave can be in active, sniff, hold, or park mode. However, the Bluetooth library only supports the hold and active modes.

Declared In `BtLibTypes.h`

Constants

`btLibSniffMode`
The slave is in sniff mode. This mode is not currently supported.

`btLibHoldMode`
The slave is in hold mode.

`btLibParkMode`
The slave is in park mode. This mode is not currently supported.

`btLibActiveMode`
The slave is active.

Comments `btLibManagementEventModeChange`

BtLibLinkPrefsEnum Enum

Purpose The `BtLibLinkPrefsEnum` enum specifies the link state preferences that can be accessed with the `BtLibLinkGetState()` and `BtLibLinkSetState()` functions.

Declared In `BtLibTypes.h`

Constants `btLibLinkPref_Authenticated`
This preference is a `Boolean` and indicates whether the link has been authenticated or not.

`btLibLinkPref_Encrypted`
This preference is a `Boolean` and indicates whether the link is encrypted or not.

`btLibLinkPref_LinkRole`
This preference is a `BtLibConnectionRoleEnum` and indicates whether the remote device is a master or a slave. You cannot set this preference but you can get its value.

See Also `BtLibLinkGetState()`, `BtLibLinkSetState()`

BtLibManagementEventEnum Enum

Purpose These event codes are posted on the Management Entity's file descriptor. Your application can poll the file descriptor to receive notification that they have occurred.

Declared In `BtLibTypes.h`

Constants `btLibManagementEventRadioState`
This event is generated when the Bluetooth radio changes state. The radio changes state when the radio is disconnected, the power is turned on or off, the radio resets, or the radio

fails to initialize. The status code for this event explains why the event gets generated.

`btLibManagementEventInquiryResult`
: A remote device has responded to an inquiry that was started with the `BtLibStartInquiry()` function.

`btLibManagementEventInquiryComplete`
: The device inquiry started with the `BtLibStartInquiry()` function has completed.

`btLibManagementEventInquiryCanceled`
: The device inquiry has been canceled because the application called `BtLibCancelInquiry()`.

`btLibManagementEventACLDisconnect`
: An ACL link has been disconnected. The `status` field indicates the reason the link was disconnected.

`btLibManagementEventACLConnectInbound`
: A remote device has established an ACL link to the local device.

`btLibManagementEventACLConnectOutbound`
: An attempt to establish an ACL link to a remote device has completed; the status field indicates whether or not the attempt was successful.

`btLibManagementEventPiconetCreated`
: The piconet has been created. This event can result from calling `BtLibPiconetCreate()`.

`btLibManagementEventPiconetDestroyed`
: The piconet has been destroyed. This event can result from calling `BtLibPiconetDestroy()`.

`btLibManagementEventModeChange`
: A slave has changed its mode. A slave can be in active, sniff, hold, or park mode.

`btLibManagementEventAccessibilityChange`
: The accessibility mode of the local device has changed.

`btLibManagementEventEncryptionChange`
: Encryption for a link has been enabled or disabled.

`btLibManagementEventRoleChange`
: The master and slave devices for a link have switched roles.

`btLibManagementEventNameResult`
A remote device name request has completed.

`btLibManagementEventLocalNameChange`
The user-friendly name of the local device has changed.

`btLibManagementEventAuthenticationComplete`
The authentication of a remote device has completed.

`btLibManagementEventPasskeyRequest`
A remote device has requested a passkey. Your application does not have to respond to this request—the Bluetooth library automatically handles it.

Because a passkey can be requested during or after a link is established, consider disabling any failure timers while the passkey dialog is up. The btLibManagementEventPasskeyRequestComplete event signals the completion of the passkey entry.

`btLibManagementEventPasskeyRequestComplete`
A passkey request has been processed. The status code for this event is set to `btLibErrNoError` if the passkey was entered or `btLibErrCanceled` if passkey entry was cancelled. Note that this event does *not* tell you that the authentication completed.

`btLibManagementEventPairingComplete`
Pairing has successfully completed and the link is authenticated.

`btLibManagementEventRSSI`
A radio strength indication event has occurred.

BtLibProtocolEnum Enum

Purpose Define protocols supported by the Bluetooth system.

Declared In `BtLibTypes.h`

Constants

`btLibL2CapProtocol`
L2CAP.

`btLibRfCommProtocol`
RFCOMM.

`btLibSdpProtocol`
SDP.

`btLibBNEPProtocol`
BNEP.

`btLibSCOProtocol`
SCO.

BtLibSdpUuidSizeEnum Enum

Purpose The `BtLibSdpUuidSizeEnum` enum specifies the sizes that a UUID can have. See `BtLibSdpUuidType` for more information.

Declared In `BtLibTypes.h`

Constants

`btLibUuidSize16 = 2`
16-bit UUID.

`btLibUuidSize32 = 4`
32-bit UUID.

`btLibUuidSize128 = 16`
Full-size 128-bit UUID.

BtLibSocketEventEnum Enum

Purpose Specify events that can occur in response to socket operations; these are used by the event field in the `BtLibSocketEventType` structure; see that structure's description for details on the data specific to each event.

Declared In `BtLibTypes.h`

Constants

`btLibSocketEventConnectRequest`
A remote device has requested a connection.

You must respond to this event with a call to `BtLibSocketRespondToConnection()`.

If the remote device requests a L2Cap connection, this event is sent to the L2Cap listener socket with a PSM that matches the PSM of the request.

If the remote device requests an RfComm connection, this event is sent to the RfComm listener socket with a server channel that matches the server channel of the request.

To convert a socket into a listener socket use the `BtLibSocketListen()` function.

`btLibSocketEventConnectedOutbound`
An outbound connection initiated by a call to `BtLibSocketConnect()` has completed. The `status` field is `btLibErrNoError` if the connection has completed successfully. Otherwise, the `status` field indicates why the connection failed.

`btLibSocketEventConnectedInbound`
A remote connection has been accepted because the application has called `BtLibSocketRespondToConnection()`.

If the remote device requests a L2Cap connection, this event is sent to the L2Cap listener socket with a PSM that matches the PSM of the requested connection. The Bluetooth library creates a new socket that exchanges data with the remote device.

If the remote device requests an RfComm connection, this event is sent to the RfComm listener socket with a server channel that matches the server channel of the requested connection. The Bluetooth library creates a new socket that exchanges data with the remote device.

`btLibSocketEventDisconnected`

If this event arrives on a data socket, then it means that the data socket has been disconnected, and the `status` field indicates the reson for hte disconnection.

If this event arrives on a listener socket, then it means that an inbound connection couldn't be established following a call to `BtLibSocketRespondToConnection()`, and the `status` field indicates the reason why the inbound connection failed.

IMPORTANT: In the case of failure of an inbound connection attempt, a new data socket is still returned in the `eventData.newSocket` field of the event. You *must* call `BtLibSocketClose()` to close it.

`btLibSocketEventSendComplete`

A previous send operation has completed. The application initiated this request by calling `BtLibSocketSend()`.

NOTE: This event is only provided to maintain compatibility with previous versions of Palm OS. Applications do not have to wait for this event before reusing the data buffer passed to `BtLibSocketSend()`, which they had to do in versions of Palm OS prior to Palm OS Cobalt, version 6.0.

`btLibSocketEventSdpServiceRecordHandle`

A request for remote service records matching a list of service classes has completed. The application initiated this request by calling the `BtLibSdpServiceRecordsGetByServiceClass()` function.

`btLibSocketEventSdpGetAttribute`

An attribute request has completed. The application initiated this request by calling the `BtLibSdpServiceRecordGetAttribute()` function.

`btLibSocketEventSdpGetStringLen`

A string or URL length request has completed. The application initiated this request by calling `BtLibSdpServiceRecordGetStringOrUrlLength()`.

`btLibSocketEventSdpGetNumListEntries`
A number of list entries request has completed. The application initiated this request by calling `BtLibSdpServiceRecordGetNumListEntries()`.

`btLibSocketEventSdpGetNumLists`
A number of lists request has completed. The application initiated this request by calling `BtLibSdpServiceRecordGetNumLists()`.

`btLibSocketEventSdpGetRawAttribute`
A get raw attribute request has completed. The application initiated the request by calling `BtLibSdpServiceRecordGetRawAttribute()`.

`btLibSocketEventSdpGetRawAttributeSize`
A get raw attribute size request has completed. The application initiated this request by calling `BtLibSdpServiceRecordGetSizeOfRawAttribute()`.

`btLibSocketEventSdpGetServerChannelByUuid`
A get server channel request has completed. The application initiated this request by calling `BtLibSdpGetServerChannelByUuid()`.

`btLibSocketEventSdpGetPsmByUuid`
A get PSM request has completed. The application initiated this request by calling `BtLibSdpGetPsmByUuid()`.

BtLibSocketInfoEnum Enum

Purpose The `BtLibSocketInfoEnum` enum allows you to specify which information you want to retrieve using the `BtLibSocketGetInfo` function.

Declared In `BtLibTypes.h`

Constants `btLibSocketInfo_Protocol = 0`
`BtLibSocketGetInfo()` returns a `BtLibProtocolEnum` representing the socket's protocol.

`btLibSocketInfo_RemoteDeviceAddress`
`BtLibSocketGetInfo()` returns a `BtLibDeviceAddressType` representing the address of the device at the other end of this socket.

`btLibSocketInfo_SendPending = 100`
: BtLibSocketGetInfo() returns a `Boolean` indicating whether a send is currently in progress.

`btLibSocketInfo_MaxTxSize`
: BtLibSocketGetInfo() returns a `uint32_t` representing the maximum packet size the local device can transmit.

`btLibSocketInfo_MaxRxSize`
: BtLibSocketGetInfo() returns a `uint32_t` representing the maximum packet size the local device can receive.

`btLibSocketInfo_L2CapPsm = 200`
: BtLibSocketGetInfo() returns a BtLibL2CapPsmType that represents the Protocol and Service Multiplexer (PSM) this socket is using to route packets. This information is only valid for L2Cap sockets.

`btLibSocketInfo_L2CapChannel`
: BtLibSocketGetInfo() returns a BtLibL2CapChannelIdType that represents the channel identifier for this socket. This information is valid for L2Cap sockets only. See the "Logical Link Control and Adaptation Protocol Specification" chapter of the *Specification of the Bluetooth System* for more information about channel identifiers.

`btLibSocketInfo_RfCommServerId = 300`
: BtLibSocketGetInfo() returns a BtLibRfCommServerIdType that represents the socket's RfComm server channel. This information is valid for RfComm sockets only.

`btLibSocketInfo_RfCommOutstandingCredits`
: BtLibSocketGetInfo() returns a `uint16_t` containing the number of remaining credits on this socket. This information is valid for RfComm sockets only.

`btLibSocketInfo_SdpServiceRecordHandle = 400`
: BtLibSocketGetInfo() returns the BtLibSdpRemoteServiceRecordHandle for the service record associated with this socket. This information is valid for SDP sockets only.

`btLibSocketInfo_DeviceNum = 1000`
: Used to get the minor device number of the STREAMS L2Cap or RfComm device instance.

Universal Service Attribute IDs

Purpose Service attributes whose definitions are common to all service records.

Declared In `BtLibTypes.h`

Constants

Constant	Definition
`btLibServiceRecordHandle`	An SDP service record handle.
`btLibServiceClassIdList`	A list of class IDs.
`btLibServiceRecordState`	A service record state.
`btLibServiceId`	A service ID.
`btLibProtocolDescriptorList`	A protocol descriptor list.
`btLibBrowseGroupList`	A browse group list.
`btLibLanguageBaseAttributeIdList`	A language attribute ID list. See "Attribute Identifier Constants" on page 200.
`btLibTimeToLive`	A time-to-live value.
`btLibAvailability`	Availability information.
`btLibProfileDescriptorList`	A profile descriptor list.
`btLibDocumentationUrl`	An URL to documentation.
`btLibClientExecutableUrl`	The URL to a client executable.
`btLibIconUrl`	The URL to an icon.

Comments Universal attributes aren't necessarily all used in every service record; they're simply standard attributes that may be used. If a service record has an attribute with an attribute ID assigned to a universal attribute, the attribute value must conform to the universal attribute's definition.

Only two attributes are required to exist in every service record instance: `btLibServiceRecordHandle` and `btLibServiceClassIdList`.

Bluetooth Application Launch Codes

sysBtLaunchCmdDoServiceUI

Purpose Sent to Bluetooth service applications when the user taps the "Advanced" button in the services view of the Bluetooth panel. This gives the service the opportunity to display and manage custom UI to let the user configure the service.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysBtLaunchCmdDoServiceUI 89`

Parameters None.

Comments **NOTE:** This launch code is only set if the `btLibServDescFlag_CAN_DO_UI` flag is set in the response when `sysBtLaunchCmdDescribeService` is called.

sysBtLaunchCmdDescribeService

Purpose Sent to Bluetooth service applications to obtain information it needs in order to display its services view.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysBtLaunchCmdDescribeService 86`

Parameters The launch code's parameter block pointer references a `BtLibServiceDescriptionType` structure, in which the service application should return information about the service offered by the application.

sysBtLaunchCmdExecuteService

Purpose Sent to Bluetooth service applications to let them know that there is an inbound-connected data socket.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysBtLaunchCmdExecuteService 77`

Parameters The launch code's parameter block pointer references a `BtLibServiceExecutionParamsType` structure. This structure identifies the connected L2Cap or RFComm socket.

Comments Applications register themselves as Bluetooth services by calling `BtLibRegisterService()`. The service application receives this launch code each time a remote client connects. It receives the launch code in the context of the System process or the Application process, according to the `execAsNormalApp` registration flag. Bluetooth service applications must respond to this launch code.

The `BtLibServiceExecutionParamsType` structure contains a file descriptor opened to a connected L2Cap or RFComm device instance, with a serial interface module optionally pushed onto that (depending upon the `pushSerialModule` registration flag). Upon entry, it is connected to its remote peer and ready for data transfer. Upon exit, it must be closed.

See Also `sysBtLaunchCmdPrepareService`

sysBtLaunchCmdPrepareService

Purpose Sent to Bluetooth service applications to let them know that a listener socket has been created and to request an SDP service record.

`CmnLaunchCodes.h`

Prototype `#define sysBtLaunchCmdPrepareService 76`

Parameters The launch code's parameter block pointer references a `BtLibServicePreparationParamsType` structure. This structure identifies both a L2Cap or RFComm listener socket and an SDP service record that the Bluetooth service application fills in to describe the service that it is offering.

Comments Applications register themselves as Bluetooth services by calling `BtLibRegisterService()`. The service application receives this launch code once after it registers itself, and then after each service execution session, in the context of the System Process. All Bluetooth service applications must respond to this launch code.

The `BtLibServicePreparationParamsType` structure contains a file descriptor opened to an L2Cap or RFComm device instance. Upon entry it has already been marked as a listener. Upon return it must be left unchanged; the Bluetooth system will take care of calling `BtLibSdpServiceRecordStartAdvertising()` to advertise the service, and `BtLibSocketClose()` after an inbound connection has been made.

The `BtLibServicePreparationParamsType` structure also contains a handle on a local SDP service record that, upon entry, is empty. Upon exit, it must be set up to describe the service that the application has to offer.

In most cases, the application can respond to this launch code by simply calling `BtLibSdpServiceRecordSetAttributesForSocket()`, passing the `BtLibServicePreparationParamsType` structure's fields along with a service class UUID and a service name. In more complex cases, the application will need to use other `BtLibSdp`*`xxx`*`()` functions to construct the service record. In such cases it is the application's responsibility to open a Management Entity device instance to pass to those functions, and to close it before returning.

See Also `sysBtLaunchCmdExecuteService`

Bluetooth Functions and Macros

BtLibAddrAToBtd Function

Purpose Convert an ASCII string a Bluetooth device address in colon-separated form to a 48-bit `BtLibDeviceAddressType`.

Declared In `BtLib.h`

Prototype
```
status_t BtLibAddrAToBtd (const char *strBuf,
   BtLibDeviceAddressType *devAddrP)
```

Parameters

→ *strBuf*
String containing ASCII colon-separated Bluetooth device address.

← *devAddrP*

Returns Pointer to a `BtLibDeviceAddressType` to store the converted device address.

Returns Returns `btLibErrNoError` to indicate that the conversion was successful.

See Also `BtLibAddrBtdToA()`

BtLibAddrBtdToA Function

Purpose Convert 48-bit `BtLibDeviceAddressType` to an ASCII string in colon-separated form.

Declared In `BtLib.h`

Prototype
```
status_t BtLibAddrBtdToA
   (BtLibDeviceAddressType *devAddrP,
   char *strBuf, uint16_t strBufSize)
```

Parameters

→ *devAddrP*
Address of a Bluetooth device. This parameter must not be `NULL`.

← *strBuf*
Pointer to a buffer to store the ASCII formatted Bluetooth devices address upon return. This parameter must not be `NULL`.

→ *strBufSize*
Size of the *strBuf* buffer, in bytes. Must be at least 18.

Returns Returns `btLibErrNoError` if successful. Returns `btLibErrParamErr` if

- *devAddrP* is `NULL`
- *strBuf* is `NULL`
- *strBufSize* is less than 18, the number of bytes required to store the ASCII formatted address

BtLibCancelInquiry Function

Purpose Cancel a Bluetooth inquiry in process.

Declared In `BtLib.h`

Prototype `status_t BtLibCancelInquiry (int32_t fdME)`

Parameters → *fdME*
The ME's file descriptor.

Returns Returns one of the following values:

`btLibErrNoError`
The inquiry process was canceled before it started.

`btLibErrPending`
The cancellation is pending. When it succeeds, notification will be provided through a management event.

`btLibErrInProgress`
The inquiry is already being canceled.

`btLibErrNotInProgress`
No inquiry is in progress to be canceled.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments The function cancels inquiries initiated by `BtLibStartInquiry()`. The `btLibManagementEventInquiryCanceled` event indicates that the cancellation has completed.

A Bluetooth discovery initiated using `BtLibDiscoverDevices()` cannot be canceled with this function. Only the user can cancel these inquiries by tapping the Cancel button.

See Also `BtLibStartInquiry()`

BtLibClose Function

Purpose Close the Bluetooth Management Entity.

Declared In `BtLib.h`

Prototype `status_t BtLibClose (int32_t fdME)`

Parameters

→ *fdME*
The Management Entity's file descriptor.

Returns

`btLibErrNoError`
Success.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Applications must call this function when they're done using the Management Entity file descriptor they obtained by calling `BtLibOpen()`.

If this function closes the last Management Entity file descriptor, and there are no connected L2CAP, RFCOMM, SCO, or BNEP file descriptors open, then the following steps are taken:

- If there are any remaining ACL links, they are destroyed.
- If the radio hardware has been used since the last reinitialization, the stack and radio are shut down and reinitialized.

See Also `BtLibOpen()`

BtLibDiscoverDevices Function

Purpose Perform remote device discovery, presenting a user interface to let the user select remote devices or cancel the operation.

Declared In `BtLib.h`

Prototype

```
status_t BtLibDiscoverDevices (int32_t fdME,
   char *instructionTxt, char *buttonTxt,
   Boolean addressAsName,
   BtLibClassOfDeviceType *filterTable,
   uint8_t filterTableLen, Boolean hideFavorites,
   BtLibDeviceAddressType *deviceTable,
   uint8_t deviceTableLen,
   uint8_t *numSelectedPtr)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *instructionTxt*
Text to appear at the top of the selection box. Specify `NULL` to use the default text, which is "Select a device:" or "Select one or more devices:" depending on whether the *deviceTableLen* parameter is 1 or greater than one.

→ *buttonTxt*
Text to appear in the "done" button. Specify `NULL` to use the default text.

→ *addressAsName*
If `true`, devices' addresses will be displayed instead of their names.

→ *filterTable*
Pointer to a list of devices classes that should appear in the list. Specify `NULL` to list all devices.

→ *filterTableLen*
The number of entries in the *filterTable* list.

→ *hideFavorites*
If `true`, devices that are in the user's favorite devices list are not shown.

← *deviceTable*
Pointer to a table to receive the addresses of the devices the user selects. Must not be `NULL`.

→ *`deviceTableLen`*
The number of slots in the *`deviceTable`* array.

← *`numSelectedPtr`*
Receives the number of devices returned in the *`deviceTable`* list.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrCanceled`
The user canceled the discovery process.

`iosErrBadFD`
The specified file descriptor is invalid.

`iosErrNotOpened`
The file descriptor specified is not for an opened Management Entity.

Comments If the *`addressAsName`* parameter is `true`, the user will be presented with the discovered devices' Bluetooth addresses. If it's `false`, the Bluetooth system will attempt to obtain each device's user-friendly name, either from the cache or by connecting to the remote device and requesting it. If this is successful, the name will be displayed.

The *`filterTable`* can be used to restrict the devices that are presented to the user based on class of device; for example, if the application needs to locate a Bluetooth headset, it can specify `btLibCOD_Minor_Audio_Headset` in the *`filterTable`*.

The user will be prevented from selecting more than *`deviceTableLen`* devices.

BtLibGetGeneralPreference Function

Purpose Get one of the general management preferences.

Declared In `BtLib.h`

Prototype
```
status_t BtLibGetGeneralPreference (int32_t fdME,
    BtLibGeneralPrefEnum pref, void *prefValueP,
    uint16_t prefValueSize)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *pref*
The general preference to get.

← *prefValueP*
Pointer to a buffer to receive the preference's value. You must allocate this buffer, and this pointer must not be `NULL`.

→ *prefValueSize*
The size, in bytes, of the *prefValueP* buffer. You must set this size to match the size of the requested preference.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrParamError`
One or more parameters is invalid. Be sure that the *prefValueSize* parameter matches the size of the preference value.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Specify the preference with a member of the `BtLibGeneralPrefEnum`.

IMPORTANT: The 68K compatibility version of the Bluetooth library does not include the null terminator in the length of string preferences when you call this function; this is to maintain compatiblity with a bug in previous versions of Palm OS. The ARM-native version of this function, however, correctly includes the null terminator in the length of strings.

See Also `BtLibSetGeneralPreference()`

BtLibGetRemoteDeviceName Function

Purpose Get the name of the remote device with the specified address.

Declared In `BtLib.h`

Prototype
```
status_t BtLibGetRemoteDeviceName (int32_t fdME,
   BtLibDeviceAddressType *remoteDeviceP,
   BtLibGetNameEnum retrievalMethod)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *remoteDeviceP*
Pointer to a `BtLibDeviceAddressType` containing the address of the device whose name you wish to retrieve.

→ *retrievalMethod*
Method used to retrieve the user-friendly remote device name. See `BtLibGetNameEnum`.

Returns Returns one of the following values:

`btLibErrBusy`
There is already a name request pending.

`btLibErrPending`
The results will be returned through a notification.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function returns `btLibErrPending` and generates a `btLibManagementEventNameResult` event when the name is available.

The Bluetooth library maintains a cache of 50 device names. If the *`retrievalMethod`* parameter is `btLibCachedThenRemote`, this function first checks the cache for a name. If the name is not in the cache, the function queries the remote device for its name, forming a temporary ACL connection if one is not already in place. In this case,

Other values of the *`retrievalMethod`* parameter can instruct this function to look for the name only in the cache or only on the remote device. See `BtLibGetNameEnum` for more information.

BtLibGetRemoteDeviceNameSynchronous Function

Purpose Return the user-friendly name of the given remote device, blocking until the name is determined.

Declared In `BtLib.h`

Prototype
```
status_t BtLibGetRemoteDeviceNameSynchronous
    (int32_t fdME,
    BtLibDeviceAddressType *remoteDeviceP,
    BtLibGetNameEnum retrievalMethod,
    char *buffer, size_t bufferLen)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`remoteDeviceP`*
The address of the remote Bluetooth device.

→ *`retrievalMethod`*
A `BtLibGetNameEnum` indicating the method to use when obtaining the name.

← *`buffer`*
A buffer to receive the name of the remote device. This buffer must be at least `btLibMaxDeviceNameLength` bytes long.

→ *`bufferLen`*
Size, in bytes, of the *`buffer`*.

Returns Returns one of the following values:

`btLibErrNoError`
: The name structure was successfully retrieved from the cache. No event will be generated.

`btLibErrBusy`
: There is already a name request pending.

`iosErrBadFD`
: The file descriptor is not valid.

`iosErrNotOpened`
: The specified file descriptor isn't open.

Comments This function blocks until the name retrieval attempt is completed. The resulting name is a null-terminated string. If the name is not found, an empty string is returned.

BtLibL2CapHToNL Macro

Purpose Macro that converts a 32-bit value from host to L2Cap byte order. L2Cap byte order is little endian.

Declared In `BtLib.h`

Prototype `#define BtLibL2CapHToNL (x)`

Parameters → *x*
: 32-bit value to convert.

Returns Returns *x* in L2Cap byte order.

See Also `BtLibL2CapHToNS()`, `BtLibL2CapNToHL()`, `BtLibL2CapNToHS()`

BtLibL2CapHToNS Macro

Purpose Macro that converts a 16-bit value from host to L2Cap byte order. L2Cap byte order is little endian.

Declared In `BtLib.h`

Prototype `#define BtLibL2CapHToNS (x)`

Parameters → *x*
: 16-bit value to convert.

Returns Returns *x* in L2Cap byte order.

See Also `BtLibL2CapHToNL()`, `BtLibL2CapNToHS()`, `BtLibL2CapNToHL()`

BtLibL2CapNToHL Macro

Purpose Macro that converts a 32-bit value from L2Cap to host byte order. L2Cap byte order is little endian.

Declared In `BtLib.h`

Prototype `#define BtLibL2CapNToHL (x)`

Parameters → *x*
32-bit value to convert.

Returns Returns *x* in host byte order.

See Also `BtLibL2CapNToHS()`, `BtLibL2CapHToNL()`, `BtLibL2CapHToNS()`

BtLibL2CapNToHS Macro

Purpose Macro that converts a 16-bit value from L2Cap to host byte order. L2Cap byte order is little endian.

Declared In `BtLib.h`

Prototype `#define BtLibL2CapNToHS (x)`

Parameters → *x*
16-bit value to convert.

Returns Returns *x* in host byte order.

See Also `BtLibL2CapNToHL()`, `BtLibL2CapHToNS()`, `BtLibL2CapNToHL()`

BtLibLinkConnect Function

Purpose Create a Bluetooth Asynchronous Connectionless (ACL) link.

Declared In `BtLib.h`

Prototype
```
status_t BtLibLinkConnect (int32_t fdME,
   BtLibDeviceAddressType *remoteDeviceP)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *remoteDeviceP*
Pointer to a `BtLibDeviceAddressType` containing the address of the remote device.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through an event.

`btLibErrAlreadyConnected`
An ACL link already exists between the local device and the specified remote device.

`btLibErrBluetoothOff`
The Bluetooth radio is off. The user can turn the radio on and off with a setting in the preferences panel.

`btLibErrBusy`
A piconet is currently being created or destroyed.

`btLibErrTooMany`
Cannot create another ACL link because the maximum allowed number has already been reached.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments An ACL link is a packet-switched physical level connection between two devices that is needed before the devices can form a RfComm or L2Cap connection.

When the connection is established or if it fails to be established, the `btLibManagementEventACLConnectOutbound` event is generated.

See Also `BtLibLinkDisconnect()`

BtLibLinkDisconnect Function

Purpose Disconnect an existing ACL Link.

Declared In `BtLib.h`

Prototype

```
status_t BtLibLinkDisconnect (int32_t fdME,
    BtLibDeviceAddressType *remoteDeviceP)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *remoteDeviceP*
Pointer to a `BtLibDeviceAddressType` containing the address of the remote device.

Returns Returns one of the following values:

`btLibErrNoError`
The connection attempt was canceled before it started. No event is generated.

`btLibErrPending`
When the link actually disconnects, a `btLibManagementEventACLDisconnect` event is generated.

`btLibErrBusy`
Can't disconnect the link because the piconet is being destroyed.

`btLibErrNoConnection`
No link to the specified device exists.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments When the link disconnects, a `btLibManagementEventACLDisconnect` event is generated.

See Also `BtLibLinkDisconnect()`

BtLibLinkGetState Function

Purpose Get the state of an ACL link.

Declared In `BtLib.h`

Prototype
```
status_t BtLibLinkGetState (int32_t fdME,
   BtLibDeviceAddressType *remoteDeviceP,
   BtLibLinkPrefsEnum pref, void *linkStateP,
   uint16_t linkStateSize)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`remoteDeviceP`*
Pointer to a `BtLibDeviceAddressType` containing the address of the remote device.

→ *`pref`*
The link preference to retrieve. See `BtLibLinkPrefsEnum`.

← *`linkStateP`*
Pointer to a buffer to receive the value of the preference. You must allocate this buffer, and this pointer must not be `NULL`. See `BtLibLinkPrefsEnum` for more information.

→ *`linkStateSize`*
The size, in bytes, of the *`linkStateP`* buffer.

Returns Returns one of the following values:

`btLibErrNoError`
Success. The `linkState` variable has been filled in.

`btLibErrNoAclLink`
No link to the specified remote device exists.

`btLibErrParamError`
The *`linkStateSize`* parameter is not same as the size of the preference value.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibLinkPrefsEnum`

BtLibLinkSetState Function

Purpose Set the state of an ACL link.

Declared In `BtLib.h`

Prototype
```
status_t BtLibLinkSetState (int32_t fdME,
    BtLibDeviceAddressType *remoteDeviceP,
    BtLibLinkPrefsEnum pref, void *linkStateP,
    uint16_t linkStateSize)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *remoteDeviceP*
Pointer to a `BtLibDeviceAddressType` containing the address of the remote device.

→ *pref*
The link preference to set. See `BtLibLinkPrefsEnum`.

→ *linkStateP*
Pointer to the preference's new value. If this is `NULL`, the call is ignored and no error occurs. See `BtLibLinkPrefsEnum`.

→ *linkStateSize*
Size, in bytes, of the *linkStateP* buffer.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through an event.

`btLibErrFailed`
An attempt was made to encrypt a link before authenticating it.

`btLibErrNoAclLink`
No link to the specified remote device exists.

`btLibErrParamError`
The preference cannot be set or *linkStateSize* is invalid.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Applications use this function to set the state of an ACL link. This function may generate events depending on the preference you change. The `btLibManagementEventAuthenticationComplete` event indicates the link authentication has completed. The `btLibManagementEventEncryptionChange` event indicates that the encryption has been enabled or disabled.

See Also `BtLibLinkGetState()`

BtLibMEEventName Function

Purpose Return the name of the specified Management Entity event code.

Declared In `BtLib.h`

Prototype
```
const char *BtLibMEEventName
    (BtLibManagementEventEnum event)
```

Parameters → *event*
The event whose name should be returned.

Returns Returns a pointer to a null-terminated string indicating the name of the ME event code.

Comments This function is provided primarily for debugging purposes.

BtLibOpen Function

Purpose Open a file descriptor to the Management Entity device, and wait for reinitialization of the stack and radio hardware if necessary.

Declared In `BtLib.h`

Prototype
```
status_t BtLibOpen (int32_t *fdME)
```

Parameters ← *fdME*
Receives the ME's file descriptor.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrOutOfMemory`
Not enough memory available to open the library.

`btLibErrRadioInitFailed`
The Bluetooth stack or radio could not be initialized.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Applications must call this function before using the Bluetooth library. If this function is called just after a call to `BtLibClose()` or `BtLibSocketClose()` that caused stack and radio reinitialization, then this function will block until reinitialization is complete. If it does block, and it is being executed on the main UI thread, then a progress dialog is displayed.

> **NOTE:** Previous versions of Palm OS would return from this function before actually initializing the radio hardware, and would inform you of success or failure through a series of events including `btLibManagementEventRadioState`, `btLibManagementEventLocalNameChange`, and `btLibManagementEventAccessibilityChange`. Under Palm OS Cobalt, this call simply fails with an error if the hardware cannot be initialized.

See Also `BtLibClose()`

BtLibPiconetCreate Function

Purpose Set up the local device to be the master of a piconet.

Declared In `BtLib.h`

Prototype

```
status_t BtLibPiconetCreate (int32_t fdME,
    Boolean unlockInbound, Boolean discoverable)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *unlockInbound*
: If `true`, the piconet accepts inbound connections. Otherwise, the piconet only allows outbound connections.

→ *discoverable*
: If `true`, configures the radio to be discoverable. In other words, the radio responds to inquiries. If `false`, configures the radio to be only connectable. In other words, only devices that know the radio's Bluetooth device address can connect to it. This parameter is ignored if *unlockInbound* is `false`.

Returns Returns one of the following values:

`btLibErrNoError`
: Successfully created the piconet with the local device as the master. No event is generated.

`btLibErrPending`
: An ACL link exists, and a role change and/or accessibility change is necessary. The result will be returned in a `btLibManagementEventPiconetCreated` event.

`btLibErrInProgress`
: A previous call to this function returned `btLibErrPending`, and the result is still pending.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments Despite its name, this function doesn't really create a piconet; it simply sets the local device's link management policy such that the local device can be the master of a piconet. It's still necessary to create ACL links with other devices to actually form the piconet.

This function may be called when there are no ACL links, or when there is already one ACL link. In the latter case, if the local device isn't already master, a master-slave switch will be initiated. Once the local device has been set up to be piconet master, more ACL links may be established.

If this function returns `btLibErrPending`, then a `btLibManagementEventInquiryResult` event is generated, and the status field of that event indicates whether the local device can be set up to be piconet master.

If the accessibility of the radio changes due to this operation, a `btLibManagementEventAccessibilityChange` event is generated.

See Also `BtLibPiconetDestroy()`, `BtLibPiconetLockInbound()`, `BtLibPiconetUnlockInbound()`

BtLibPiconetDestroy Function

Purpose Destroy the piconet by disconnecting links to all devices and removing all restrictions on whether the local device is a master or a slave.

Declared In `BtLib.h`

Prototype `status_t BtLibPiconetDestroy (int32_t` *`fdME`*`)`

Parameters → *`fdME`*

The ME's file descriptor.

Returns Returns one of the following values:

`btLibErrNoError`
: Successfully destroyed the piconet. A `btLibManagementEventPiconetDestroyed` event is not generated.

`btLibErrPending`
: The piconet is being destroyed, and a `btLibManagementEventPiconetDestroyed` event will be generated when the operation succeeds or fails.

`btLibErrBusy`
: The piconet is already in the process of being destroyed.

`btLibErrNoPiconet`
: No piconet exists to be destroyed.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments A `btLibManagementEventACLDisconnect` event is generated for each ACL link that is disconnected. When the piconet is successfully destroyed or fails to be destroyed, a `btLibManagementEventPiconetDestroyed` is generated. The `status` field of the `BtLibStringType` structure accompanying the event indicates whether the piconet was destroyed or not.

See Also `BtLibPiconetCreate()`

BtLibPiconetLockInbound Function

Purpose Prevent remote devices from creating ACL links into the piconet.

Declared In `BtLib.h`

Prototype `status_t BtLibPiconetLockInbound (int32_t fdME)`

Parameters

→ *`fdME`*
The ME's file descriptor.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrBusy`
The piconet is in the process of being destroyed.

`btLibErrNoPiconet`
No piconet exists.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments After locking inbound connections, outbound connections are still allowed. Locking inbound connections maximizes the bandwidth for members of the piconet to transmit data to each other.

See Also `BtLibPiconetUnlockInbound()`

BtLibPiconetUnlockInbound Function

Purpose Allow remote devices to create ACL links into the piconet.

Declared In `BtLib.h`

Prototype
```
status_t BtLibPiconetUnlockInbound (int32_t fdME,
    Boolean discoverable)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`discoverable`*
If `true`, configures the radio to be discoverable. In other words, the radio responds to inquiries. If `false`, configures the radio to be only connectable. In other words, only devices that know the radio's Bluetooth device address can connect to it.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrBusy`
The piconet is in the process of being destroyed.

`btLibErrNoPiconet`
No piconet exists.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Allowing inbound connections lowers the bandwidth available to transmit data between members of the piconet because the radio must periodically scan for incoming links.

See Also `BtLibPiconetLockInbound()`

BtLibRegisterService Function

Purpose Register a persistent Bluetooth service application.

Declared In `BtLib.h`

Prototype

```
status_t BtLibRegisterService
    (BtLibServiceRegistrationParamsType *params)
```

Parameters

→ *params*
: A `BtLibServiceRegistrationParamsType` structure describing the service the application wishes to register.

Returns Returns one of the following:

`btLibErrNoError`
: Success.

`btLibErrTooMany`
: The maximum number of services is already registered.

Comments An application only needs to register a service once after a system boot; subsequent registrations are ignored.

Service applications must respond to the `sysBtLaunchCmdPrepareService` and `sysBtLaunchCmdExecuteService` launch codes.

BtLibRfCommHToNL Macro

Purpose Macro that converts a 32-bit value from host to RfComm byte order. RfComm byte order is big endian.

Declared In `BtLib.h`

Prototype

```
#define BtLibRfCommHToNL (x)
```

Parameters

→ *x*
: 32-bit integer to convert.

Returns Returns *x* in RfComm byte order.

See Also `BtLibRfCommHToNS()`, `BtLibRfCommNToHL()`, `BtLibRfCommNToHS()`

BtLibRfCommHToNS Macro

Purpose Macro that converts a 16-bit value from host to RfComm byte order. RfComm byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibRfCommHToNS (x)`

Parameters → *x*

16-bit integer to convert.

Returns Returns *x* in RfComm byte order.

See Also `BtLibRfCommHToNL()`, `BtLibRfCommNToHL()`, `BtLibRfCommNToHS()`

BtLibRfCommNToHL Macro

Purpose Macro that converts a 32-bit value from RfComm to host byte order. RfComm byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibRfCommNToHL (x)`

Parameters → *x*

32-bit integer to convert.

Returns Returns *x* in host byte order.

See Also `BtLibRfCommNToHS()`, `BtLibRfCommHToNL()`, `BtLibRfCommHToNS()`

BtLibRfCommNToHS Macro

Purpose Macro that converts a 16-bit value from RfComm to host byte order. RfComm byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibRfCommNToHS (x)`

Parameters → *x*

16-bit integer to convert.

Returns Returns *x* in host byte order.

See Also `BtLibRfCommNToHL()`, `BtLibRfCommHToNL()`, `BtLibRfCommHToNS()`

BtLibSdpCompareUuids Function

Purpose Compare two UUIDs.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpCompareUuids (int32_t fdME,
   BtLibSdpUuidType *uuid1,
   BtLibSdpUuidType *uuid2)
```

Parameters

→ *fdME*
: The ME's file descriptor.

→ *uuid1*
: The first UUID to compare.

→ *uuid2*
: The second UUID to compare.

Returns Returns one of the following values:

`btLibErrNoError`
: UUIDs are the same

`btLibErrError`
: UUIDs are different.

`btLibErrParamError`
: One or both UUIDs are invalid.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

BtLibSdpGetPsmByUuid Function

Purpose Get an available L2Cap PSM using SDP.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpGetPsmByUuid
    (BtLibSocketRef socketRef,
    BtLibDeviceAddressType *rDev,
    BtLibSdpUuidType *serviceUUIDList,
    uint8_t uuidListLen)
```

Parameters

→ *`socketRef`*
An SDP socket.

→ *`rDev`*
Device address of a remote device to query. This parameter must not be `NULL`.

→ *`serviceUUIDList`*
Array of UUIDs that must match those of the service record. This parameter must not be `NULL`.

→ *`uuidListLen`*
Length of *`serviceUuidList`*. A maximum of 12 entries is allowed.

Returns Returns one of the following values:

`btLibErrPending`
The PSM value will be returned through an event.

`btLibErrOutOfMemory`
Not enough memory to complete request.

`btLibErrParamError`
One or more parameters is invalid.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketRole`
The specified socket is not connected.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function returns the L2Cap PSM of the first SDP record on the remote device that contains all the specified UUIDs.

This function generates a `btLibSocketEventSdpGetPsmByUuid` event when the query completes or fails.

See Also `BtLibSdpGetServerChannelByUuid()`

BtLibSdpGetRawDataElementSize Macro

Purpose Macro that returns a constant representing the data element's size.

Declared In `BtLib.h`

Prototype `BtLibSdpGetRawDataElementSize (header)`

Parameters → *header*
First byte of a data element.

Returns A constant representing the size of the data element.

Comments The first byte of a SDP data element contains the type and size of the data element.

See Also `BtLibSdpGetRawElementType()`, `BtLibSdpParseRawDataElement()`, `BtLibSdpVerifyRawDataElement()`, "Bluetooth Data Element Sizes"

BtLibSdpGetRawElementType Macro

Purpose Macro that returns an SDP data element's type.

Declared In `BtLib.h`

Prototype `BtLibSdpGetRawElementType (header)`

Parameters → *header*
The first byte of a data element.

Returns The type of the data element.

Comments The first byte of a SDP data element contains the type and size of the data element.

See Also `BtLibSdpGetRawDataElementSize()`, `BtLibSdpParseRawDataElement()`, `BtLibSdpVerifyRawDataElement()`, "Bluetooth Data Element Types"

BtLibSdpGetServerChannelByUuid Function

Purpose Get an available RfComm server channel using SDP.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpGetServerChannelByUuid
    (BtLibSocketRef socketRef,
    BtLibDeviceAddressType *rDev,
    BtLibSdpUuidType *serviceUUIDList,
    uint8_t uuidListLen)
```

Parameters

→ *socketRef*
An SDP socket.

→ *rDev*
Device address of a remote device to query. This parameter must not be `NULL`.

→ *serviceUUIDList*
Array of UUIDs that must match those of the service record. This parameter must not be `NULL`.

→ *uuidListLen*
Length of *serviceUuidList*. A maximum of 12 entries is allowed.

Returns Returns one of the following values:

`btLibErrPending`
The server channel will be returned through an event.

`btLibErrOutOfMemory`
Not enough memory to complete request.

`btLibErrParamError`
One or more parameters is invalid.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketRole`
The specified socket is not connected.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function returns the RfComm server channel number of the first SDP record on the remote device that contains all the specified UUIDs.

This function generates a `btLibSocketEventSdpGetServerChannelByUuid` event when the query completes or fails.

See Also `BtLibSdpGetPsmByUuid()`

BtLibSdpHToNL Macro

Purpose Macro that converts a 32-bit value from host to Service Discovery Protocol (SDP) byte order. SDP byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibSdpHToNL (`*x*`)`

Parameters → *x*
32-bit value to convert.

Returns Returns *x* in SDP byte order.

See Also `BtLibSdpHToNS()`, `BtLibSdpNToHL()`, `BtLibSdpNToHS()`

BtLibSdpHToNS Macro

Purpose Macro that converts a 16-bit value from host to Service Discovery Protocol (SDP) byte order. SDP byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibSdpHToNS (x)`

Parameters → *x*
16-bit value to convert.

Returns Returns *x* in SDP byte order.

See Also `BtLibSdpHToNL()`, `BtLibSdpNToHL()`, `BtLibSdpNToHS()`

BtLibSdpNToHL Macro

Purpose Macro that converts a 32-bit value from Service Discovery Protocol (SDP) to host byte order. SDP byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibSdpNToHL (x)`

Parameters → *x*
32-bit value to convert.

Returns Returns *x* in host byte order.

See Also `BtLibSdpNToHS()`, `BtLibSdpHToNL()`, `BtLibSdpHToNS()`

BtLibSdpNToHS Macro

Purpose Macro that converts a 16-bit value from Service Discovery Protocol (SDP) to host byte order. SDP byte order is big endian.

Declared In `BtLib.h`

Prototype `#define BtLibSdpNToHS (x)`

Parameters → *x*
16-bit value to convert.

Returns Returns *x* in host byte order.

See Also `BtLibSdpNToHL()`, `BtLibSdpHToNL()`, `BtLibSdpHToNS()`

BtLibSdpParseRawDataElement Function

Purpose Parse a raw SDP data element to determine where the data field begins and the size of the data field.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpParseRawDataElement
    (int32_t fdME, const uint8_t *value,
    uint16_t *offset, uint32_t *length)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`value`*
Pointer to a raw SDP data element.

← *`offset`*
Offset, in bytes, between *`value`* and the start of the data field.

← *`length`*
Length, in bytes, of the data field.

Returns Returns one of the following values:

`btLibErrNoError`
Successfully parsed the attribute.

`btLibErrNotOpen`
The reference Bluetooth Management Entity is not open.

`btLibErrParamError`
`dataElementP`, *`offset`*, or *`length`* is `NULL`.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments A data element has three fields. The first field, called the **header field**, identifies the type of value stored in the data element and the size of the element. The second field, called the **size field**, contains more information about the size of the data if it's not completely specified by the header. Otherwise the size field is omitted. The third field, called the **data field**, contains the data element's actual value.

The offset this function returns is the offset between the start of the data element and the data field. The size this function returns is the the size of the data field. Note that the sum of the offset and the size is the size of the data element.

This function is especially useful for iterating through entries in a list attribute.

The *Specification of the Bluetooth System* has more information about the structure of a data element.

See Also `BtLibSdpVerifyRawDataElement()`, `BtLibSdpGetRawElementType()`, `BtLibSdpGetRawDataElementSize()`

BtLibSdpServiceRecordCreate Function

Purpose Allocate a memory chunk that represents an SDP service record.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordCreate
    (int32_t fdME, BtLibSdpRecordHandle *recordH)
```

Parameters

→ *fdME*
The ME's file descriptor.

← *recordH*
SDP memory handle for the new SDP memory record.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrOutOfMemory`
Not enough memory to allocate the memory chunk.

`btLibErrParamError`
recordH is `NULL`.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibSdpServiceRecordDestroy()`, `BtLibSdpServiceRecordStartAdvertising()`, `BtLibSdpServiceRecordStopAdvertising()`

BtLibSdpServiceRecordDestroy Function

Purpose Free the memory associated with a SDP memory record.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordDestroy
   (int32_t fdME, BtLibSdpRecordHandle recordH)
```

Parameters

→ *fdME*
The ME’s file descriptor.

→ *recordH*
SDP memory handle associated with the memory chunk to be freed.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrParamError`
recordH does not refer to an valid SDP memory record.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function stops advertising the record before it frees it.

See Also `BtLibSdpServiceRecordCreate()`, `BtLibSdpServiceRecordStartAdvertising()`, `BtLibSdpServiceRecordStopAdvertising()`

BtLibSdpServiceRecordGetAttribute Function

Purpose Retrieve the value of a specific attribute in a SDP memory record. If the attribute is a list or a protocol descriptor list (a list of lists), this function retrieves the value of a specific list entry.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordGetAttribute
    (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    BtLibSdpAttributeDataType *attributeValue,
    uint16_t listNumber, uint16_t listEntry)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *recordH*
Handle identifying the SDP memory record.

→ *attributeID*
Attribute identifier of the attribute to retrieve.

← *attributeValue*
Buffer into which this function stores the attribute's value. You must allocate this buffer. This pointer must not be `NULL`.

→ *listNumber*
List to query if the attribute is a protocol descriptor list. Otherwise this parameter is ignored.

→ *listEntry*
Item to get in the list if the attribute is a list attribute. Otherwise this parameter is ignored.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrPending`
The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
: A query is already pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
: An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
: Not enough memory to perform the query.

`btLibErrParamError`
: *`recordH`* is an invalid handle or *`attributeValues`* is `NULL`.

`btLibErrSdpAttributeNotSet`
: The specified attribute does not exist in the specified service record.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments If the specified SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetAttribute` event when the result is available or the query fails. In this case, the attribute value is contained within the control and data parts of the event when it is received by a call to `IOSGetmsg()`. The main part of the event is in the control part, and the string or URL associated with it, if there is one, is in the data part.

If you are retrieving a string or a URL, you need to allocate additional space. See the documentation for `BtLibSdpAttributeDataType` for more information.

This function supports the universal attributes defined in "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System*.

See Also `BtLibSdpServiceRecordSetAttribute()`, `BtLibSdpServiceRecordMapRemote()`, `BtLibSdpServiceRecordGetNumListEntries()`, `BtLibSdpServiceRecordGetNumLists()`, `BtLibSdpServiceRecordGetStringOrUrlLength()`

BtLibSdpServiceRecordGetNumListEntries Function

Purpose Get the number of entries in a list attribute.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordGetNumListEntries
    (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    uint16_t listNumber, uint16_t *numEntries)
```

Parameters

→ *fdME*
: The ME's file descriptor.

→ *recordH*
: Handle identifying the SDP memory record.

→ *attributeID*
: Attribute identifier of the attribute whose number of list entries is retrieved.

→ *listNumber*
: List to query if the attribute is a `ProfileDescriptorListEntry`. Otherwise this parameter is ignored.

← *numEntries*
: On return, indicates the number of entries in the list.

Returns Returns one of the following values:

`btLibErrNoError`
: Success

`btLibErrPending`
: The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
: The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
: Another query is pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
: An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
: Not enough memory to perform this query.

`btLibErrParamError`
: *`recordH`* is an invalid handle or *`numEntries`* is `NULL`.

`btLibErrSdpAttributeNotSet`
: The specified attribute does not exist in the specified service record.

`btLibErrStackNotOpen`
: The Bluetooth stack failed to open when the library was opened.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments This function supports the universal attributes defined in "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System*. Specifically, this function gives valid results for ServiceClassIdList, ProtocolDescriptorList, BrowseGroupList, LanguageBaseAttributeIDList, and ProfileDescriptorList attributes.

If the specified SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetNumListEntries` event when the result is available or the query fails.

See Also `BtLibSdpServiceRecordGetNumLists()`, `BtLibSdpServiceRecordGetAttribute()`, `BtLibSdpServiceRecordGetStringOrUrlLength()`, `BtLibSdpServiceRecordMapRemote()`

BtLibSdpServiceRecordGetNumLists Function

Purpose Get the number of lists in a protocol descriptor list SDP attribute.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordGetNumLists
   (int32_t fdME, BtLibSdpRecordHandle recordH,
   BtLibSdpAttributeIdType attributeID,
   uint16_t *numLists)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *recordH*
Handle identifying the SDP memory record.

→ *attributeID*
Attribute identifier of the attribute whose number of lists is retrieved.

← *numLists*
On return, indicates the number of lists.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrPending`
The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
Another query is pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
Not enough memory to perform this query.

`btLibErrParamError`
: `recordH` is an invalid handle or *`numLists`* is `NULL`.

`btLibErrSdpAttributeNotSet`
: The specified attribute does not exist in the specified service record.

`btLibErrStackNotOpen`
: The Bluetooth stack failed to open when the library was opened.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments If the specified SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetNumListEntries` event when the result is available or the query fails.

See Also `BtLibSdpServiceRecordGetNumListEntries()`, `BtLibSdpServiceRecordGetAttribute()`, `BtLibSdpServiceRecordGetStringOrUrlLength()`, `BtLibSdpServiceRecordMapRemote()`

BtLibSdpServiceRecordGetRawAttribute Function

Purpose Retrieve the value of an attribute of an SDP memory record. The retrieved attribute is in the format defined in the "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System*.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordGetRawAttribute
    (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    uint8_t *value, uint16_t *valSize)
```

Parameters

→ *`fdME`*
: The ME's file descriptor.

→ *`recordH`*
: Handle identifying the SDP memory record.

→ *`attributeID`*
Attribute identifier of the attribute to retrieve.

← *`value`*
Buffer into which this function stores the retrieved SDP attribute data. You must allocate this buffer. This pointer must not be `NULL`.

← *`valSize`*
Size of the *`value`* buffer upon entry. This parameter must not be zero. Upon return, contains the number of bytes retrieved.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrPending`
The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
A query is already pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
Not enough memory to perform the query.

`btLibErrParamError`
`recordH` is an invalid handle, *`value`* is `NULL`, *`valSize`* is `0`, or the size of the attribute value is larger than *`valSize`*.

`btLibErrSdpAttributeNotSet`
The specified attribute does not exist in the specified service record.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments If the specified SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetRawAttribute` event when the result is available or the query fails.

See Also `BtLibSdpServiceRecordSetRawAttribute()`, `BtLibSdpServiceRecordGetSizeOfRawAttribute()`, `BtLibSdpServiceRecordMapRemote()`

BtLibSdpServiceRecordGetSizeOfRawAttribute Function

Purpose Return the size, in bytes, of any attribute of an SDP memory record.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordGetSizeOfRawAttribute
    (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    uint16_t *size)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *recordH*
Handle identifying the SDP memory record.

→ *attributeID*
Attribute identifier of the attribute whose size is retrieved.

← *size*
Pointer to a `uint16_t` into which the size of the attribute will be stored by this function. Must not be `NULL`.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrPending`
The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
A query is already pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
Not enough memory to perform the query.

`btLibErrParamError`
`recordH` is an invalid handle or *`size`* is `NULL`.

`btLibErrSdpAttributeNotSet`
The specified attribute does not exist in the specified service record.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments If the specified SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetRawAttributeSize` event when the result is available or the query fails.

See Also `BtLibSdpServiceRecordGetRawAttribute()`, `BtLibSdpServiceRecordMapRemote()`, `BtLibSdpServiceRecordSetRawAttribute()`

BtLibSdpServiceRecordGetStringOrUrlLength Function

Purpose Get the length of a string or URL attribute in a SDP memory record.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordGetStringOrUrlLengt
    h (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    uint16_t *size)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`recordH`*
Handle identifying the SDP memory record.

→ *`attributeID`*
Attribute identifier of the attribute whose length is retrieved.

← *`size`*
Pointer to a `uint16_t` into which the length of the string or URL will be stored. This parameter cannot be `NULL`.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrPending`
The specified SDP memory record refers to a service record on a remote device. The result will be returned through an event.

`btLibErrBusy`
The connection is parked. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrInProgress`
A query is already pending on this socket. This error can occur only if the SDP memory record refers to a service record on a remote device.

`btLibErrNoAclLink`
An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
Not enough memory to perform the query.

`btLibErrParamError`
The *`recordH`* does not refer to a valid handle, *`length`* is `NULL`, or the attribute is not a string or a URL.

`btLibErrSdpAttributeNotSet`
The specified attribute does not exist in the specified SDP record.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments Bluetooth strings do not include a null terminator.

If the SDP memory record refers to a service record on a remote device, this function generates a `btLibSocketEventSdpGetStringLen` event when the result is available or the query fails.

See Also `BtLibSdpServiceRecordGetAttribute()`, `BtLibSdpServiceRecordGetNumListEntries()`, `BtLibSdpServiceRecordGetNumLists()`, `BtLibSdpServiceRecordMapRemote()`

BtLibSdpServiceRecordMapRemote Function

Purpose Configure an SDP memory record so it refers to a service record on a remote device.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordMapRemote
    (BtLibSocketRef socketRef,
    BtLibDeviceAddressType *rDev,
    BtLibSdpRemoteServiceRecordHandle remoteHandle
    , BtLibSdpRecordHandle recordH)
```

Parameters

→ *`socketRef`*
The SDP socket.

→ *`rDev`*
The device to query.

→ *remoteHandle*
Remote service record handle.

→ *recordH*
SDP memory handle of an empty SDP record.

Returns Returns one of the following values:

`btLibErrNoError`
The mapping was successful.

`btLibErrOutOfMemory`
Not enough memory to perform mapping.

`btLibErrParamError`
recordH is invalid or refers to an invalid memory chunk.

`btLibErrSdpMapped`
The SDP memory record is already mapped to a remote service record.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketProtocol`
The specified socket is not an SDP socket.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments You must create an SDP memory record using `BtLibSdpServiceRecordCreate()` before using this function.

Note that this function does not copy the contents of the remote service record to the SDP memory record in local memory.

BtLibSdpServiceRecordSetAttribute Function

Purpose Set the value of an attribute in an SDP memory record. If the attribute is a list or a protocol descriptor list (a list of lists), this function sets the value of a specific list entry. The SDP memory record must represent a local unadvertised service record.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordSetAttribute
    (int32_t fdME, BtLibSdpRecordHandle recordH,
    BtLibSdpAttributeIdType attributeID,
    BtLibSdpAttributeDataType *attributeValue,
    uint16_t listNumber, uint16_t listEntry)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *recordH*
Handle of the service record to modify.

→ *attributeID*
Attribute identifier of the attribute to set.

→ *attributeValue*
Pointer to the new value for the attribute. This pointer must not be `NULL`.

→ *listNumber*
to modify if the attribute is a protocol descriptor list. Otherwise this parameter is ignored.

→ *listEntry*
Item to set in the list if the attribute is a list attribute. Otherwise this parameter is ignored.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrAdvertised`
An advertised record was passed in *recordH*. The record must not be advertised.

`btLibErrOutOfMemory`
Not enough memory to set the attribute.

`btLibErrParamError`
recordH is invalid or *attributeValue* is `NULL`.

`btLibErrRemoteRecord`
A remote record was passed in *recordH*. The record must be local.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function only works on SDP memory records that are local and not advertised. You can advertise the record after you finish modifying it.

This function supports the universal attributes defined in the *Specification of the Bluetooth System*.

See Also `BtLibSdpServiceRecordGetAttribute()`, `BtLibSdpServiceRecordStartAdvertising()`, `BtLibSdpServiceRecordStopAdvertising()`

BtLibSdpServiceRecordSetAttributesForSocket Function

Purpose Initialize an SDP memory record so it can represent an existing L2Cap or RfComm listener socket as a service.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordSetAttributesForSocket(
   BtLibSocketRef socketRef,
   BtLibSdpUuidType *serviceUuidList,
   uint8_t uuidListLen, const char *serviceName,
   uint16_t serviceNameLen,
   BtLibSdpRecordHandle recordH)
```

Parameters

→ *`socketRef`*
Reference number for an RfComm or L2Cap socket in listening mode.

→ *`serviceUuidList`*
List of UUIDs for the service record.

→ *`uuidListLen`*
Number of entries in *`serviceUUIDList`*. A maximum of 12 entries is allowed.

→ *`serviceName`*
User-friendly name for the service in English; if you want to use another language, you should use the lower-level functions and a language base attribute ID list.

→ *`serviceNameLen`*
Size, in bytes, of *`serviceName`*.

→ *`recordH`*
Handle of the service record to be initialized.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrAdvertised`
The record specified by *`recordH`* is being advertised. You must stop advertising the record before you can change it.

`btLibErrNotOpen`
The Bluetooth library Entity is not open.

`btLibErrOutOfMemory`
Not enough memory to store the contents of the SDP record.

`btLibErrParamError`
`recordH` is not a valid record handle.

`btLibErrRemoteRecord`
A remote record was passed in *`recordH`*. Because the service is local, the record must be local.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketRole`
The specified socket is not a listener socket.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments You must first create an SDP record using `BtLibSdpServiceRecordCreate()`. However, the record must not be advertised. In other words, don't call

[`BtLibSdpServiceRecordStartAdvertising()`]() until after calling this function.

See Also `BtLibSdpServiceRecordCreate()`, `BtLibSocketListen()`

BtLibSdpServiceRecordSetRawAttribute Function

Purpose Set the value for an attribute of a SDP memory record. This function allows you to specify the attribute as an array of bytes in the format defined in the "Service Discovery Protocol" chapter of the *Specification of the Bluetooth System*. The SDP memory record must represent a local unadvertised service record.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordSetRawAttribute
   (int32_t fdME, BtLibSdpRecordHandle recordH,
   BtLibSdpAttributeIdType attributeID,
   const uint8_t *value, uint16_t valSize)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`recordH`*
Handle identifying the SDP memory record.

→ *`attributeID`*
Attribute identifier of the attribute to set.

→ *`value`*
Array of bytes containing SDP attribute data in the format defined in the SDP protocol. This parameter must not be `NULL`.

→ *`valSize`*
Size, in bytes, of *`value`*. This parameter must not be 0.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrAdvertised`
`recordH` is being advertised. The record must not be advertised.

`btLibErrOutOfMemory`
Not enough memory to set the attribute.

`btLibErrParamError`
recordH is invalid, *value* is `NULL`, or *valSize* is 0.

`btLibErrRemoteRecord`
recordH refers to a service record on a remote device. The service record must be local.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments If the service record is being advertised, you must stop advertising it before you modify it.

See Also `BtLibSdpServiceRecordGetRawAttribute()`, `BtLibSdpServiceRecordSetAttribute()`, `BtLibSdpServiceRecordStartAdvertising()`, `BtLibSdpServiceRecordStopAdvertising()`

BtLibSdpServiceRecordsGetByServiceClass Function

Purpose Get the service record handles corresponding to the service classes advertised on a remote device.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordsGetByServiceClass
    (BtLibSocketRef socketRef,
    BtLibDeviceAddressType *rDev,
    BtLibSdpUuidType *uuidList,
    uint16_t uuidListLen)
```

Parameters

→ *socketRef*
The SDP socket.

→ *rDev*
Remote device to query.

→ *uuidList*
Array of UUIDs identifying the service classes. This parameter must not be `NULL`.

→ *`uuidListLen`*
Number of elements in the *`uuidList`*. You can specify a maximum of 12 UUIDs.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through an event.

`btLibErrBusy`
The connection to the remote device is parked.

`btLibErrInProgress`
A SDP query is already in progress on this socket.

`btLibErrNoAclLink`
An ACL link to the remote device does not exist.

`btLibErrOutOfMemory`
Not enough memory to perform the query.

`btLibErrParamError`
One or more parameters are invalid.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketProtocol`
The specified socket is not an SDP socket.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function generates a `btLibSocketEventSdpServiceRecordHandle` event when the matching service records are available or the query fails.

BtLibSdpServiceRecordStartAdvertising Function

Purpose Make visible an SDP memory record representing a local SDP service record. Remote devices can access visible service records through SDP.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpServiceRecordStartAdvertising
    (int32_t fdME, BtLibSdpRecordHandle recordH)
```

Parameters

→ *fdME*
The ME's file descriptor.

→ *recordH*
Handle of the service record to make available to remote devices.

Returns Returns one of the following values:

`btLibErrNoError`
Success

`btLibErrParamError`
recordH is not a valid record handle.

`btLibErrRemoteRecord`
recordH refers to a remote record. The record must be local.

`btLibErrSdpAdvertised`
The service record is already accessible by remote devices.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments You cannot modify an SDP memory record while it is available to remote devices.

See Also `BtLibSdpServiceRecordStopAdvertising()`

BtLibSdpServiceRecordStopAdvertising Function

Purpose Hide an SDP memory record representing a local SDP service record. Remote devices cannot access hidden service records through SDP.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSdpServiceRecordStopAdvertising
   (int32_t fdME, BtLibSdpRecordHandle recordH)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`recordH`*
Handle of the service record to hide.

Returns Returns one of the following values:

`btLibErrNoError`
Success. The SDP record is no longer available to remote devices.

`btLibErrParamError`
`recordH` is not a valid record handle.

`btLibErrRemoteRecord`
`recordH` refers to a remote record. The record must be local.

`btLibErrSdpNotAdvertised`
The service record is already hidden from remote devices.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibSdpServiceRecordStartAdvertising()`

BtLibSdpUuidInitialize Macro

Purpose Macro that sets the value of a UUID.

Declared In `BtLibTypes.h`

Prototype
```
#define BtLibSdpUuidInitialize (uuidVar,
    rawValue, uuidSize)
```

Parameters

→ *uuidVar*
: `BtLibSdpUuidType` to initialize.

→ *rawValue*
: Array of bytes representing the UUID. The size of this array depends on *uuidSize*.

→ *uuidSize*
: `BtLibSdpUuidType` member specifying the size of the *rawValue* array.

Returns Nothing.

BtLibSdpVerifyRawDataElement Function

Purpose Verify that a raw SDP data element is properly formed.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSdpVerifyRawDataElement
    (int32_t fdME, const uint8_t *value,
    uint16_t valSize, uint8_t maxLevel)
```

Parameters

→ *fdME*
: The ME's file descriptor.

→ *value*
: Raw SDP attribute data.

→ *valSize*
: Size of *value*, in bytes. The size of the data element must be less than or equal to this parameter, otherwise this function fails.

→ *maxLevel*
: Maximum level of recursion over which this function verifies the data element. Must be at least one.

Returns Returns one of the following values:

`btLibErrNoError`
SDP data element is properly formatted.

`btLibErrError`
SDP data element is not properly formatted.

`btLibErrNotOpen`
The reference Bluetooth library is not open.

`btLibErrParamError`
`value` is `NULL`.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments This function checks all size descriptors in the element to ensure that the data element fits into the indicated length. In the case of data element sequences or alternates, this function calls itself recursively.

The *`maxLevel`* parameter specifies the maximum number of times this function calls itself. Limiting the recursion level prevents an infinite loop if the data is bad. *`maxLevel`* must be large enough to handle the complete data element. For example, to verify a simple data element such as an unsigned integer, *`maxLevel`* must be at least 1. To verify a data element sequence of UUIDs, *`maxLevel`* must be at least 2.

See Also `BtLibSdpParseRawDataElement()`, `BtLibSdpGetRawDataElementSize()`, `BtLibSdpGetRawElementType()`

BtLibSecurityFindTrustedDeviceRecord Function

Purpose Search the device database for the device with the specified Bluetooth address. Return the index of the corresponding device record in the database.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSecurityFindTrustedDeviceRecord
    (int32_t fdME, BtLibDeviceAddressType *addrP,
    uint16_t *indexP)
```

Parameters

→ *fdME*
File descriptor of the Management Entity.

→ *addrP*
Bluetooth address of remote device.

← *indexP*
Index of the found record.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrNotFound`
No record with the specified remote device address was found.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibSecurityGetTrustedDeviceRecordInfo()`, `BtLibSecurityRemoveTrustedDeviceRecord()`

BtLibSecurityGetTrustedDeviceRecordInfo Function

Purpose Get information from a device record in the device database.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSecurityGetTrustedDeviceRecordInfo
    (int32_t fdME, uint16_t index,
    BtLibDeviceAddressType *addrP,
    char *nameBuffer, uint8_t nameBufferSize,
    BtLibClassOfDeviceType *codP,
    uint32_t *lastConnectedP, Boolean *trustedP)
```

Parameters

→ *fdME*
File descriptor of the Management Entity.

→ *index*
Index of the record.

← *addrP*
Bluetooth address of remote device.

← *nameBuffer*
Pointer to buffer to store user-friendly name of remote device. You must allocate this buffer. Provide a `NULL` pointer if the user-friendly name is not needed.

→ *nameBufferSize*
Size of the *nameBuffer* buffer on entry. On exit, the size of the name.

↔ *codP*
Pointer to a `BtLibClassOfDeviceType` representing the class of the device. You must allocate this structure. Provide a `NULL` pointer if the device class is not needed.

← *lastConnectedP*
The date since the device last connected. This date is measured in seconds since midnight January 1, 1904. Provide a `NULL` pointer if the date of last connection is not needed.

← *trustedP*
If `true`, the device is bonded and can connect to the local device without authentication. If `false`, the device is paired but not bonded—it will need to reauthenticate if it connects again. Provide a `NULL` pointer if this information is not needed.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`dmErrIndexOutOfRange`
A record with the specified index could not be found.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibSecurityFindTrustedDeviceRecord()`

BtLibSecurityNumTrustedDeviceRecords Function

Purpose Return the number of bonded devices in the device database or return the total number of devices in the device database.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSecurityNumTrustedDeviceRecords
    (int32_t fdME, Boolean trustedOnly,
    uint16_t *numP)
```

Parameters

→ *fdME*
File descriptor of the Management Entity.

→ *trustedOnly*
true to only obtain the total number of trusted devices in the database. false will obtain the total number of devices in the devices database, including both bonded and paired but not bonded devices.

← *numP*
On return, contains the number of trusted devices.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`iosErrBadFD`
The Management Entity file descriptor is bad.

`iosErrNotOpened`
The Management Entity file descriptor isn't open.

See Also `BtLibSecurityFindTrustedDeviceRecord()`, `BtLibSecurityGetTrustedDeviceRecordInfo()`

BtLibSecurityRemoveTrustedDeviceRecord Function

Purpose Removes a device from the device database.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSecurityRemoveTrustedDeviceRecord
    (int32_t fdME, uint16_t index)
```

Parameters

→ *`fdME`*
The file descriptor of the Management Entity.

→ *`index`*
Index of the record to remove.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`dmErrIndexOutOfRange`
A record with the specified index could not be found.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

See Also `BtLibSecurityFindTrustedDeviceRecord()`

BtLibSetGeneralPreference Function

Purpose Set one of the general management preferences.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSetGeneralPreference (int32_t fdME,
   BtLibGeneralPrefEnum pref, void *prefValueP,
   uint16_t prefValueSize)
```

Parameters

→ *fdME*
: The ME's file descriptor.

→ *pref*
: General preference to set. See `BtLibGeneralPrefEnum`.

→ *prefValueP*
: Pointer to the value of the preference. This parameter must not be `NULL`. See `BtLibGeneralPrefEnum`.

→ *prefValueSize*
: The size, in bytes, of *prevValueP*.

Returns Returns one of the following values:

`btLibErrNoError`
: Success.

`btLibErrPending`
: The results will be returned through an event.

`btLibErrParamError`
: One or more parameters is invalid. Be sure that *prefValueSize* matches the size of the preference value.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments See the `BtLibGeneralPrefEnum` description for a list of the preferences.

This function may generate events depending on the preference you change. The `btLibManagementEventAccessibilityChange` event indicates that the accessibility of the local device has changed.

See Also `BtLibGetGeneralPreference()`

BtLibSocketAdvanceCredit Function

Purpose Advance credit to a given RfComm connection socket.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSocketAdvanceCredit
   (BtLibSocketRef socketRef, uint8_t credit)
```

Parameters

→ *socketRef*
: The `BtLibSocketRef` indicating the socket.

→ *credit*
: Number credits to add to the total number of credits for this socket. The total number of credits represents the number of packets the remote device can send before data flow stops.

Returns Returns one of the following values:

`btLibErrNoError`
: Success

`btLibErrFailed`
: Too many credits advanced.

`btLibErrSocket`
: The specified socket is invalid.

`btLibErrSocketProtocol`
: The specified socket is not an RfComm socket.

`btLibErrSocketRole`
: The specified socket is not connected.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments RfComm uses a credit based flow control mechanism. For each credit the connection has, one packet of data can be sent. When the credits are spent, data flow stops until you advance more credits using this function.

Multiple calls to this function have a cumulative effect.

BtLibSocketClose Function

Purpose Close a socket, free associated resources, and kill all associated socket connections.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSocketClose
    (BtLibSocketRef socketRef)
```

Parameters

→ *socketRef*
The `BtLibSocketRef` indicating the socket.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrSocket`
The specified socket is invalid.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments No events are generated when closing a socket.

If there are no Management Entity file descriptors open, and this function closes the last connected L2CAP, RFCOMM, SCO, or BNEP file descriptor, then the following steps are taken:

- If there are any remaining ACL links, they are destroyed.
- If the radio has been used since the last reinitialization, the stack and radio are shut down and reinitialized.

See Also `BtLibSocketCreate()`, `BtLibSocketListen()`, `BtLibSocketConnect()`, `BtLibSocketRespondToConnection()`

BtLibSocketConnect Function

Purpose Create an outbound L2Cap, RfComm, SCO, or BNEP connection.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSocketConnect
    (BtLibSocketRef socketRef,
    BtLibSocketConnectInfoType *connectInfo)
```

Parameters

→ *socketRef*
The socket to connect.

→ *connectInfo*
`BtLibSocketConnectInfoType` containing Bluetooth device address and protocol-specific connection information.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through an event.

`btLibErrNoAclLink`
An ACL link for the remote device does not exist

`btLibErrSocket`
The specified socket is invalid.

`btLibErrSocketProtocol`
The protocol of the specified socket is not supported. This function only supports the L2Cap and RfComm protocols.

`btLibErrSocketRole`
The specified socket is already connected or listening.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments If the connection succeeds, the `btLibSocketEventConnectedOutbound` event is generated and its `status` field is set to `btLibErrNoError`. If connection fails, the same event is generated with a non-zero `status` field, or a `btLibSocketEventDisconnected` is generated. In both cases, the `status` field indicates the reason for the failure.

If the connection succeeds, when inbound data arrives, `IOSGetmsg()` will return a message with an empty control part and a data part containing the received data. When the channel disconnects, a `btLibSocketEventDisconnected` event is generated.

See Also `BtLibSocketSend()`, `BtLibSocketClose()`

BtLibSocketCreate Function

Purpose Create a socket (by opening a file descriptor) to an L2CAP, RFCOMM, SDP, SCO, or BNEP device.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSocketCreate
    (BtLibSocketRef *socketRefP,
    BtLibProtocolEnum socketProtocol)
```

Parameters

← *socketRefP*
: Pointer to an allocated `BtLibSocketRef` that will receive the socket value on return. This pointer must not be `NULL`.

→ *socketProtocol*
: The protocol (L2Cap, RFComm, or SDP) to use on the socket.

Returns Returns one of the following values:

`btLibErrNoError`
: Success.

`btLibErrParamError`
: *socketRefP* is `NULL`.

`btLibErrTooMany`
: The maximum number of sockets allocated for the system has already been reached. The Bluetooth library supports a maximum of 16 socket connections.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments No events are generated when creating a socket.

Before terminating, applications should close all of the sockets that they have created.

NOTE: A socket reference is the same thing as an IOS file descriptor.

See Also `BtLibSocketConnect()`, `BtLibSocketListen()`, `BtLibSocketClose()`

BtLibSocketEventName Function

Purpose Return the name of the given socket event code.

Declared In `BtLib.h`

Prototype
```
const char *BtLibSocketEventName
    (BtLibSocketEventEnum event)
```

Parameters
→ *event*
The socket event code whose name should be returned.

Returns Returns a pointer to a null-terminated string with the human-readable name of the event.

Comments This function is primarily provided for debugging purposes.

BtLibSocketGetInfo Function

Purpose Retrieve information for a currently open socket.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSocketGetInfo
    (BtLibSocketRef socketRef,
    BtLibSocketInfoEnum infoType, void *valueP,
    uint32_t valueSize)
```

Parameters
→ *socketRef*
The socket to query.

→ *infoType*
Type of information to retrieve. See `BtLibSocketInfoEnum`.

← *valueP*
Buffer into which this function stores the result. You must allocate the buffer.

→ *`valueSize`*
Size, in bytes, of the *`valueP`* buffer. This size must match that of the requested information.

Returns Returns one of the following values:

`btLibErrNoError`
Success.

`btLibErrParamError`
One or more parameters is invalid. Be sure that the *`valueSize`* parameter matches the size of the information you're retrieving.

`btLibErrSdpNotMapped`
The SDP socket has not been mapped to a remote SDP service record. This error occurs when you try to obtain the SDP service record handle before you map socket to a remote service record using `BtLibSdpServiceRecordMapRemote`.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketRole`
The specified socket is not connected or has the wrong role for the request.

`btlibErrSocketProtocol`
The specified socket has the wrong protocol for the request.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

BtLibSocketListen Function

Purpose Set up an L2Cap, RFComm, SCO, or BNEP socket as a listener.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSocketListen
    (BtLibSocketRef socketRef,
    BtLibSocketListenInfoType *listenInfoP)
```

Parameters

→ *`socketRef`*
: The socket to listen on.

↔ *`listenInfoP`*
: Protocol-specific listening information. For more information see `BtLibSocketListenInfoType`. This parameter must be `NULL` if the socket is an SCO socket, otherwise it must not be `NULL`.

Returns Returns one of the following values:

`btLibErrNoError`
: Success. The socket is listening for incoming connections.

`btLibErrBusy`
: The given PSM is in use (L2Cap only)

`btLibErrParamError`
: *`listenInfoP`* is `NULL`.

`btLibErrSocket`
: The specified socket is invalid.

`btLibErrSocketProtocol`
: The protocol of the specified socket is not supported. This function only supports the L2Cap and RfComm protocols.

`btLibErrSocketRole`
: The specified socket is already listening or connected.

`btLibErrTooMany`
: There are no resources to create a listener socket of this type.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments A listener socket waits for a remote device to initiate a connection to the local device and then generates a `btLibSocketEventConnectRequest` event to notify the application that it needs to handle the connection attempt.

You need to respond to this event with a call to `BtLibSocketConnect()` on the listener socket to accept or reject the connection.

Under certain circumstances, the *`listenInfo`* parameter acts as an output as well as an input. See the discussion of `BtLibSocketListenInfoType`.

See Also `BtLibSocketClose()`

BtLibSocketRespondToConnection Function

Purpose Accept or reject an in-bound connection on a given listener socket.

Declared In `BtLib.h`

Prototype

```
status_t BtLibSocketRespondToConnection
    (BtLibSocketRef socketRef, Boolean accept)
```

Parameters

→ *`socketRef`*
The listener socket that needs to respond to a connection attempt.

→ *`accept`*
`true` to accept the connection; `false` to reject the connection.

Returns Returns one of the following values:

`btLibErrNoError`
Success. This status is returned when *`accept`* is `false`.

`btLibErrFailed`
One or more parameters is invalid.

`btLibErrPending`
The results will be returned through an event.

`btLibErrSocket`
The specified socket is invalid or not in use.

`btLibErrSocketProtocol`
The protocol of the specified socket is not supported. This function only supports the L2Cap and RfComm protocols.

`btLibErrSocketRole`
The specified socket is not a listener socket.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments You should call this function when you respond to a `btLibSocketEventConnectRequest` event delivered to a listener socket.

If the connection succeeds, the `btLibSocketEventConnectedInbound` event is generated and its `status` field is set to `btLibErrNoError`. If connection fails, the same event is generated with a non-zero `status` field, or a `btLibSocketEventDisconnected` is generated. In both cases, the `status` field indicates the reason for the failure.

Once the connection succeeds, future calls to `IOSPoll()` will receive data messages whenever data is received from the remote device. If the channel disconnects, a `btLibSocketEventDisconnected` event is generated.

RfComm listener sockets and L2Cap listener sockets behave differently when you call this function. When you respond to an inbound L2Cap connection, a new L2Cap socket is created to exchange data with the remote device, and the L2Cap listener socket continues to listen for more connections. In other words, a single L2Cap listener socket can "spawn" several L2Cap sockets. This mechanism allows you to create a piconet.

When you respond to an RfComm connection, a new data socket is created through which you can exchange data with the remote device. However, unlike L2Cap, the listener socket remains intact but may not be reused; it also cannot be closed until the data socket is closed. You may, however, create a new listener socket to detect more inbound connections.

See Also `BtLibSocketListen()`, `BtLibSocketSend()`, `BtLibSocketClose()`

BtLibSocketSend Function

Purpose Send data over a connected L2Cap, RfComm, BNEP socket.

Declared In `BtLib.h`

Prototype
```
status_t BtLibSocketSend
    (BtLibSocketRef socketRef, uint8_t *data,
    uint32_t dataLen)
```

Parameters

→ *socketRef*
The transmitting socket.

→ *data*
Pointer to data to send.

→ *dataLen*
Length of data to send. This value must be less than the Maximum Transmission Unit (MTU) for the socket. The MTU indicates the size of the largest packet that the remote device can receive and is determined when the socket is connected.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through an event.

`btLibErrBusy`
A send is already in process.

`btLibErrNoAclLink`
An ACL link for the remote device does not exist

`btLibErrSocket`
The specified socket is invalid.

`btLibErrSocketProtocol`
The protocol of the specified socket is not supported by this function. You can only send using the L2Cap and RfCommprotocols.

`btLibErrSocketRole`
The specified socket is not connected.

`iosErrBadFd`
The specified file descriptor is invalid.

`iosErrNotOpened`
The specified file descriptor is not open.

Comments If the socket is not L2Cap, RfComm, or BNEP, or the socket is not connected, the data is silently discarded.

When the data has been sent successfully, a `btLibSocketEventSendComplete` event is generated and its `status` field is set to `btLibErrNoError`. If the data is not sent successfully, the same event is generated with a non-zero `status` field.

NOTE: Unlike previous versions of Palm OS, you can immediately reuse or dispose of the memory containing the *`data`*.

See Also `BtLibSocketClose()`

BtLibStartInquiry Function

Purpose Start a Bluetooth inquiry.

Declared In `BtLib.h`

Prototype
```
status_t BtLibStartInquiry (int32_t fdME,
   uint8_t timeOut, uint8_t maxResp)
```

Parameters

→ *`fdME`*
The ME's file descriptor.

→ *`timeOut`*
Time, in seconds, this inquiry is allowed to take. If the inquiry does not complete within this time, it is canceled. The actual time is rounded to the nearest multiple of 1.28 seconds. If you specify a timeout period larger than 60 seconds, this function acts as if you specified a timeout period of 60 seconds. If this parameter is 0, the timeout period defaults to 10.24 seconds as specified in the Generic Access Profile.

→ *`maxResp`*
Maximum number of responses the inquiry accepts. Responses are not guaranteed to be unique.

Returns Returns one of the following values:

`btLibErrPending`
The results will be returned through events.

`btLibErrBluetoothOff`
: The Bluetooth radio is off. The user can turn the radio on and off with a setting in the preferences panel.

`btLibErrInProgress`
: Another inquiry is already in progress.

`iosErrBadFd`
: The specified file descriptor is invalid.

`iosErrNotOpened`
: The specified file descriptor is not open.

Comments The function performs a low-level Bluetooth inquiry, as opposed to a full device discovery. Specifically, inquiries started with this function only return the Bluetooth address and the class of the discovered device. This function does not have a user interface.

Every time a device is discovered, a `btLibManagementEventInquiryResult` event is generated. When the inquiry is complete, a `btLibManagementEventInquiryComplete` event is generated. If the application calls `BtLibCancelInquiry()`, a `btLibManagementEventInquiryCanceled` event is generated.

See Also `BtLibCancelInquiry()`

Part IV
Networking and Sockets

Palm OS® supports the standard 4.3BSD Sockets API for networking. This Part of *Exploring Palm OS: Low-Level Communications* serves as an overview of the Sockets API; for more detailed coverage, you should refer to any of the numerous books that cover this common API. The Sockets API deprecates the Netlib API provided by previous versions of Palm OS.

14

Introduction to Sockets on Palm OS

Overview

Whereas previous versions of Palm OS® used a proprietary API for handling network connectivity, Palm OS Cobalt introduces the standard 4.3BSD Sockets API. Applications that use the old Netlib API will continue to work, but it is not possible to use the Palm OS Cobalt SDK to compile software to use the Netlib API. Applications designed to run on Palm OS Cobalt must be modified to use the Sockets API.

This API is provided as two elements:

- A socket shared library that exports BSD socket functions for system services and applications.
- A STREAMS module that performs TPI messaging adaptation and contains socket states.

NOTE: If you need to develop a 68K application using the Netlib API, your application should be built against the Palm OS Garnet SDK.

Unsupported Sockets Features

While the Palm OS Sockets API is almost totally compatible with the 4.3BSD Sockets API, there are three key differences:

AF_UNIX and PF_UNIX Unsupported

Palm OS does not support the `AF_UNIX` address family and the `PF_UNIX` protocol family, since it's not UNIX.

No socketpair() Function

The `socketpair()` function is not provided by Palm OS.

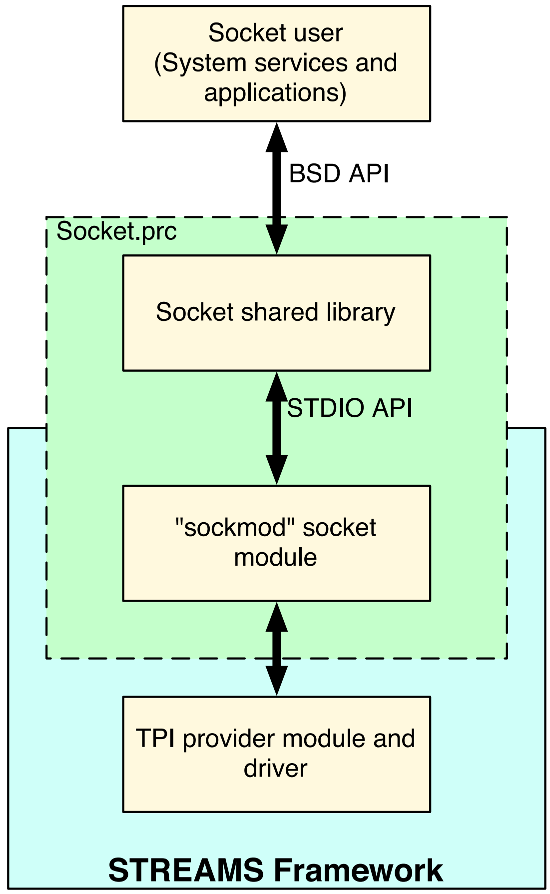
No UNIX-Style Asynchronous Features

Palm OS does not support UNIX-style asynchronous signals, options, or flags.

Architecture of the Sockets Support System

The Sockets API for Palm OS is implemented as an I/O Subsystem (IOS) module. It's implemented in a shared library that translates 4.3 BSD Sockets API calls into IOS STDIO function calls. This architecture is shown in easily-digestible diagram form in Figure 14.1.

Figure 14.1 Architecture of sockets on Palm OS



Protocol Mapping

The Sockets API determines the name of the device to open when request to open a particular socket is processed. The device that gets opened must build a TPI-compliant stream. Its name is derived from the family/domain, type, and protocol given in the `socket()` call. The device name is constructed using the following C code:

```
sprintf(device_name, "SX%02x%02x%02x", domain,
type, protocol);
```

15

Sockets & Network Support Reference

Overview

The structures, types, and functions described in this chapter are provided in several header files, including:

`sys/socket.h`
: Defines functions and types for sending and receiving data using sockets, as well as for listening for and opening connections to remote devices.

`posix/arpa/inet.h`
: Defines functions used for manipulating Internet addresses.

`posix/netinet/in.h`
: Defines structures and functions used for converting between host and network addresses.

`posix/netdb.h`
: Defines structures and functions used for performing network database operations, particularly Domain Name Resolution operations.

`posix/sys/select.h`
: Provides the `select()` function, which provides a means of detecting the readiness state of file desriptors.

Each structure, type, or function indicates the header file in which it is defined.

Structures and Types

addrinfo Struct

Purpose This structure contains the information obtained from the address.

Declared In `posix/netdb.h`

Prototype

```
struct addrinfo {
   int ai_flags;
   int ai_family;
   int ai_socktype;
   int ai_protocol;
   size_t ai_addrlen;
   char *ai_canonname;
   struct sockaddr *ai_addr;
   struct addrinfo *ai_next;
}
```

Fields

`ai_flags`
: `AI_PASSIVE`, `AI_CANONNAME`, `AI_NUMERICHOST`.

`ai_family`
: `PF_xxx`.

`ai_socktype`
: `SOCK_xxx`.

`ai_protocol`
: 0 or `IPPROTO_xxx` for IPv4 and IPv6.

`ai_addrlen`
: The length of `ai_addr`.

`ai_canonname`
: Canonical name for hostname.

`ai_addr`
: Binary address.

`ai_next`
: Next structure in linked list.

Comments All addresses are supplied in host order and returned in network order (suitable for use in system calls).

hostent Struct

Purpose This structure contains either the information obtained from the name server or database entries supplied by the system.

Declared In `posix/netdb.h`

Prototype

```
struct hostent {
   char *h_name;
   char **h_aliases;
   int h_addrtype;
   int h_length;
   char **h_addr_list;
}
```

Fields

`h_name`
: Official name of the host.

`h_aliases`
: A list of alternative names for the host.

`h_addrtype`
: Host address type.

`h_length`
: The length, in bytes, of the address.

`h_addr_list`
: List of addresses from name server.

Comments All addresses are supplied in host order and returned in network order (suitable for use in system calls).

netent Struct

Purpose This structure contains the information obtained from the network.

Declared In `posix/netdb.h`

Prototype

```
struct netent {
   char *n_name;
   char **n_aliases;
   int n_addrtype;
   unsigned long n_net;
}
```

Fields `n_name`
Official name of the network.

`n_aliases`
A list of alternative names for the network.

`n_addrtype`
Network address type.

`n_net`
The network number.

Comments All addresses are supplied in host order and returned in network order (suitable for use in system calls).

protoent Struct

Purpose This structure contains the information obtained from the protocol.

Declared In `posix/netdb.h`

Prototype
```
struct protoent {
   char *p_name;
   char **p_aliases;
   int p_proto;
}
```

Fields `p_name`
Official name of the protocol.

`p_aliases`
A list of alternative names for the protocol.

`p_proto`
The protocol number.

Comments All addresses are supplied in host order and returned in network order (suitable for use in system calls).

servent Struct

Purpose This structure contains the information obtained from the service.

Declared In `posix/netdb.h`

Prototype
```
struct servent {
   char *s_name;
   char **s_aliases;
   int s_port;
   char *s_proto;
}
```

Fields

`s_name`
Official name of the service.

`s_aliases`
A list of alternative names for the service.

`s_port`
The port number.

`s_proto`
The protocol to use.

Comments All addresses are supplied in host order and returned in network order (suitable for use in system calls).

sockaddr Struct

Purpose Defines a structure used by the kernel to store most addresses.

Declared In `posix/sys/socket.h`

Prototype
```
struct sockaddr {
   sa_family_t sa_family;
   char sa_data[14];
}
```

Fields

`sa_family`
The address family.

`sa_data`
The address value.

sockaddr_in Struct

Purpose Defines a structure used to store Internet addresses.

Declared In `posix/netinet/in.h`

Prototype
```
struct sockaddr_in {
```

```
   sa_family_t sin_family;
   in_port_t sin_port;
   struct in_addr sin_addr;
   uint8_t sin_zero[8];
}
```

Fields

`sin_family`
: `AF_INET`.

`sin_port`
: The port number.

`sin_addr`
: The IP address.

`sin_zero`
: The address value; must be initialized to zero.

socklen_t Typedef

Purpose A data type used to represent the size in bytes of socket related data.

Declared In `posix/sys/socket.h`

Prototype `typedef unsigned int socklen_t`

Functions and Macros

accept Function

Purpose Accepts a connection on a socket by extracting the first connection request on the queue of pending connections, creating a new socket with the same properties of *sock* and allocating a new file descriptor for the socket.

Declared In `posix/sys/socket.h`

Prototype
```
int accept (int sock, struct sockaddr *addr,
   socklen_t *addrlen)
```

Parameters

→ *sock*
: A socket that has been created with `socket()`, bound to an address with `bind()`, and listening for connections after a `listen()`.

← *addr*
A result parameter that is filled in with the source address of the connecting entity, as known to the communications layer.

↔ *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address returned.

Returns Returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

Comments This function is used to accept a connection when a remote system attempts to connect to a socket on which you have previously called `listen()`.

See Also `bind()`, `connect()`, `listen()`, `select()`, `socket()`

bind Function

Purpose Assigns a name to an unnamed socket.

Declared In `posix/sys/socket.h`

Prototype
```
int bind (int sock, const struct sockaddr *addr,
    socklen_t addrlen)
```

Parameters

→ *sock*
A socket that has been created with `socket()` that exists in a namespace but has no name defined.

← *addr*
A result parameter that is filled in with the source address of the connecting entity, as known to the communications layer.

↔ *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address returned.

Returns Returns zero (0) if the bind is successful. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `connect()`, `getsockname()`, `listen()`, `socket()`

connect Function

Purpose Initiates a connection on a socket.

Declared In `posix/sys/socket.h`

Prototype
```
int connect (int sock,
   const struct sockaddr *addr,
   socklen_t addrlen)
```

Parameters

→ *sock*
A socket.

← *addr*
A result parameter that is filled in with the source address of the connecting entity, as known to the communications layer.

↔ *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address returned.

Returns Returns zero (0) if the connection or binding is successful. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `accept()`, `getsockname()`, `getsockopt()`, `select()`, `socket()`

endhostent Function

Purpose Closes the TCP connection.

Declared In `posix/netdb.h`

Prototype `void endhostent (void)`

Parameters None.

Returns Nothing.

endnetent Function

Purpose Closes the connection to the database, releasing any open file descriptor.

Declared In `posix/netdb.h`

Prototype `void endnetent (void)`

Parameters None.

Returns Nothing.

endprotoent Function

Purpose Closes the connection to the database, releasing any open file descriptor.

Declared In `posix/netdb.h`

Prototype `void endprotoent (void)`

Parameters None.

Returns Nothing.

endservent Function

Purpose Closes the connection to the database, releasing any open file descriptor.

Declared In `posix/netdb.h`

Prototype `void endservent (void)`

Parameters None.

Returns Nothing.

freeaddrinfo Function

Purpose Returns the socket address structures and canonical node name strings pointed to by the `addrinfo` structures.

Declared In `posix/netdb.h`

Prototype `void freeaddrinfo (struct addrinfo *ai)`

Parameters

→ *ai*

The `addrinfo` structure pointed to by the *ai* argument is freed, along with any dynamic storage pointed to by the structure. This operation is repeated until a `NULL` `ai_next` pointer is encountered.

Returns Nothing.

freehostent Function

Purpose Releases the dynamically allocated memory of the `hostent` structure.

Declared In `posix/netdb.h`

Prototype `void freehostent (struct hostent *ip)`

Parameters → *ip*
A pointer to an object of the `hostent` structure.

Returns Returns a pointer to an object of the `hostent` structure.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

gai_strerror Function

Purpose Aids applications in printing error messages based on the `EAI_xxx` codes.

Declared In `posix/netdb.h`

Prototype `const char *gai_strerror (int ecode)`

Parameters → *ecode*
An `EAI_xxx` code, such as `EAI_ADDRFAMILY`.

Returns Returns a pointer to a string whose contents indicate an unknown error.

getaddrinfo Function

Purpose Protocol-independent nodename-to-address translation.

Declared In `posix/netdb.h`

Prototype
```
int getaddrinfo (const char *nodename,
   const char *servname,
   const struct addrinfo *hints,
   struct addrinfo **res)
```

Parameters

→ *nodename*
A pointer to null-terminated strings or `NULL`.

→ *servname*
A pointer to null-terminated strings or `NULL`.

→ *hints*
Hints concerning the type of socket that the caller supports.

← *res*
A pointer to a linked list of one or more `addrinfo` structures.

Returns Returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

Comments One or both of the *nodename* and *servname* parameters must be a non-`NULL` pointer.

If *nodename* is not `NULL`, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller. If *servname* is `NULL`, the call returns network-level addresses for the specified *nodename*. If *servname* is not `NULL`, it is a null-terminated character string identifying the requested service.

See Also `gethostbyname()`, `getservbyname()`

gethostbyaddr Function

Purpose Searches for the specified host in the current domain and its parents unless the name ends in a dot.

Declared In `posix/netdb.h`

Prototype

```
struct hostent *gethostbyaddr (const char *addr,
    int len, int type)
```

Parameters

→ *addr*
Host address type.

→ *len*
The length, in bytes, of the address.

→ *type*
A named constant that indicates the naming scheme under which the lookup is performed. Must be specified as `AF_INET`.

Returns Returns a pointer to an object of the `hostent` structure, describing an Internet host referenced by address.

gethostbyname Function

Purpose Searches for the specified host in the current domain and its parents unless the name ends in a dot.

Declared In `posix/netdb.h`

Prototype

```
struct hostent *gethostbyname (const char *name)
```

Parameters

→ *name*
Official name of the host.

Returns Returns a pointer to an object of the `hostent` structure, describing an Internet host referenced by name.

gethostbyname2 Function

Purpose An evolution of `gethostbyname()` that allows lookups in address families other than `AF_INET`.

Declared In `posix/netdb.h`

Prototype

```
struct hostent *gethostbyname2 (const char *name,
    int af)
```

Parameters

→ *name*
Official name of the host.

→ *af*
Must be specified as `AF_INET` or `AF_INET6`.

Returns Returns a pointer to an object of the `hostent` structure, describing an Internet host referenced by name.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

gethostent Function

Purpose Reads the next entry in the database, opening and closing a connection to the database as necessary.

Declared In `posix/netdb.h`

Prototype
```
struct hostent *gethostent (void)
```

Parameters None.

Returns Returns a pointer to an object of the `hostent` structure.

getipnodebyaddr Function

Purpose Returns the address of a network host.

Declared In `posix/netdb.h`

Prototype
```
struct hostent *getipnodebyaddr (const void *src,
    size_t len, int af, int *error_num)
```

Parameters

→ *src*
The name of the host whose network address to look up.

→ *len*
The length, in bytes, of the address.

→ *af*
Must be specified as `AF_INET` or `AF_INET6`.

← *error_num*
A `NULL` pointer is returned if an error occurred, and *error_num* contains an error code from the following list: `HOST_NOT_FOUND`, `NO_ADDRESS`, `NO_RECOVERY`, or `TRY_AGAIN`.

Returns Returns a pointer to an object of the `hostent` structure, describing an Internet host referenced by address.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

getipnodebyname Function

Purpose Returns the name of a network host.

Declared In `posix/netdb.h`

Prototype
```
struct hostent *getipnodebyname
    (const char *name, int af, int flags,
    int *error_num)
```

Parameters

→ *name*
Official name of the host.

→ *af*
Must be specified as `AF_INET` or `AF_INET6`.

→ *flags*
Specifies additional options: `AI_V4MAPPED`, `AI_ALL`, or `AI_ADDRCONFIG`. More than one option can be specified by logically ORing them together. *flags* should be set to zero (0) if no options are desired.

← *error_num*
A `NULL` pointer is returned if an error occurred, and *error_num* contains an error code from the following list: `HOST_NOT_FOUND`, `NO_ADDRESS`, `NO_RECOVERY`, or `TRY_AGAIN`.

Returns Returns a pointer to an object of the `hostent` structure, describing an Internet host referenced by name.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

getnameinfo Function

Purpose Translates address-to-nodename in a protocol-independent manner.

Declared In `posix/netdb.h`

Prototype
```
int getnameinfo (const struct sockaddr *sa,
    size_t salen, char *host, size_t hostlen,
    char *serv, size_t servlen, int flags)
```

Parameters

→ *sa*
A `sockaddr` structure.

→ *salen*
The length, in bytes, of the `sockaddr` structure.

→ *host*
The buffer that holds the IP address.

→ *hostlen*
The length, in bytes, of the IP address buffer.

→ *serv*
The buffer that holds the port number.

→ *servlen*
The length, in bytes, of the port number buffer.

→ *flags*
Changes the default actions of this function.

Returns Returns text strings for the IP address and port number in user-provided buffers.

getnetbyaddr Function

Purpose Searches from the beginning of the file until a matching network address is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct netent *getnetbyaddr (unsigned long net,
    int type)
```

Parameters
→ *net*
The network number.

→ *type*
Network address type.

Returns Returns a pointer to an object of the `netent` structure, describing the network database.

getnetbyname Function

Purpose Searches from the beginning of the file until a matching network name is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct netent *getnetbyname (const char *name)
```

Parameters
→ *name*
Official name of the network.

Returns Returns a pointer to an object of the `netent` structure, describing the network database.

getnetent Function

Purpose Reads the next line of the file, opening the file if necessary.

Declared In `posix/netdb.h`

Prototype
```
struct netent *getnetent (void)
```

Parameters None.

Returns Returns a pointer to an object of the `netent` structure, describing the network database.

getpeername Function

Purpose Gets the name of the connected peer.

Declared In `posix/sys/socket.h`

Prototype
```
int getpeername (int sock, struct sockaddr *addr,
    socklen_t addrlen)
```

Parameters

→ *sock*
A socket.

← *addr*
A result parameter that is filled in with the source address of the connecting entity, as known to the communications layer.

↔ *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address returned.

Returns Returns the name of the peer connected to the specified socket.

See Also `accept()`, `bind()`, `getsockname()`, `socket()`

getsockname Function

Purpose Gets the socket name.

Declared In `posix/sys/socket.h`

Prototype
```
int getsockname (int sock, struct sockaddr *addr,
    socklen_t addrlen)
```

Parameters → *sock*
A socket.

← *addr*
A result parameter that is filled in with the source address of the connecting entity, as known to the communications layer.

↔ *addrlen*
Initially contains the amount of space pointed to by `addr`; on return, it contains the actual length (in bytes) of the address returned.

Returns Returns the current name for the specified socket.

See Also `bind()`, `socket()`

getprotobyname Function

Purpose Sequentially searches from the beginning of the file until a matching protocol name is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct protoent *getprotobyname
    (const char *name)
```

Parameters → *name*
Official name of the protocol.

Returns Returns a pointer to an object of the `protoent` structure, describing the network database.

getprotobynumber Function

Purpose Sequentially searches from the beginning of the file until a matching protocol number is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct protoent *getprotobynumber (int proto)
```

Parameters → *proto*
Official name of the protocol.

Returns Returns a pointer to an object of the `protoent` structure, describing the network database.

getprotoent Function

Purpose Reads the next line of the file, opening the file if necessary.

Declared In `posix/netdb.h`

Prototype
```
struct protoent *getprotoent (void)
```

Parameters None.

Returns Returns a pointer to an object of the `protoent` structure, describing the network database.

getservbyname Function

Purpose Searches from the beginning of the file until a matching protocol name is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct servent *getservbyname (const char *name,
   const char *proto)
```

Parameters

→ *name*
Official name of the network.

→ *proto*
The protocol.

Returns Returns a pointer to an object of the `servent` structure, describing the network services database.

getservbyport Function

Purpose Searches from the beginning of the file until a matching port number is found, or until EOF is encountered.

Declared In `posix/netdb.h`

Prototype
```
struct servent *getservbyport (int port,
   const char *proto)
```

Parameters

→ *port*
The port number.

→ *proto*
The protocol to use

Returns Returns a pointer to an object of the `servent` structure, describing the network services database.

getservent Function

Purpose Reads the next line of the file, opening the file if necessary.

Declared In `posix/netdb.h`

Prototype `struct servent *getservent (void)`

Parameters None.

Returns Returns a pointer to an object of the `servent` structure, describing the network services database.

getsockopt Function

Purpose Gets the options on sockets.

Declared In `posix/sys/socket.h`

Prototype

```
int getsockopt (int sock, int level, int option,
    void *optval, socklen_t *optlen)
```

Parameters

→ *sock*
: A socket.

→ *level*
: To manipulate options at the socket level, *level* is specified as `SOL_SOCKET`.

→ *option*
: *option* and any specified options are passed uninterpreted to the appropriate protocol module for interpretation.

Returns Returns zero (0) if the connection or binding is successful. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `getprotoent()`, `select()`, `socket()`, `setsockopt()`

hstrerror Function

Purpose Returns a string that is the message text corresponding to the value of the *err* parameter.

Declared In `posix/netdb.h`

Prototype `const char *hstrerror (int err)`

Parameters → *err*
The error.

Returns Returns a string that is the message text corresponding to the value of the *err* parameter.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

htonl Function

Purpose Converts 32-bit values between host byte order and network byte order.

Declared In `posix/netinet/in.h`

Prototype `uint32_t htonl (uint32_t host32)`

Parameters → *host32*
The value being converted.

Returns Returns an unsigned integer.

See Also `gethostbyname()`, `getservent()`

htons Function

Purpose Converts 16-bit values between host byte order and network byte order.

Declared In `posix/netinet/in.h`

Prototype `uint16_t htons (uint16_t host16)`

Parameters → *host16*
The value being converted.

Returns Returns an unsigned short integer.

See Also `gethostbyname()`, `getservent()`

inet_addr Function

Purpose Interprets the specified character string (the name of a computer on the Internet) and returns a number suitable for use as an Internet address.

Declared In `posix/arpa/inet.h`

Prototype `in_addr_t inet_addr (const char *cp)`

Parameters → *cp*
A character string indicating the name of a computer on the Internet.

Returns Returns a number suitable for use as an Internet address.

Comments The string *cp* should be a name such as "palmsource.com" or "foo.bar.com".

See Also `inet_network()`

inet_aton Function

Purpose Interprets the specified character string as an Internet address, placing the address into the structure provided.

Declared In `posix/arpa/inet.h`

Prototype
```
int inet_aton (const char *cp,
    struct in_addr *addr)
```

Parameters → *cp*
A character string. In order for this function to work successfully, the string must be a standard dotted-quad format IP address, such as "127.0.0.1".

→ *addr*
An Internet address.

Returns Returns 1 if the string was successfully interpreted, or zero (0) if the string is invalid.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

inet_lnaof Function

Purpose Breaks apart the specified Internet host address and returns the local network address part (in host order).

Declared In `posix/arpa/inet.h`

Prototype `in_addr_t inet_lnaof (struct in_addr in)`

Parameters → *in*
An Internet address.

Returns Returns the local network address (in host order).

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `inet_netof()`

inet_makeaddr Function

Purpose Takes an Internet network number and a local network address (both in host order) and constructs an Internet address from it.

Declared In `posix/arpa/inet.h`

Prototype `struct in_addr inet_makeaddr (int net, int lna)`

Parameters → *net*
An Internet network number.

→ *lna*
A local network address.

Returns Returns an Internet address.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

inet_netof Function

Purpose Breaks apart the specified Internet host address and returns the network number part (in host order).

Declared In `posix/arpa/inet.h`

Prototype `in_addr_t inet_netof (struct in_addr in)`

Parameters → *in*
An Internet address.

Returns Returns the network number (in host order).

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `inet_lnaof()`

inet_network Function

Purpose Interprets the specified character string and returns a number suitable for use as an Internet network number.

Declared In `posix/arpa/inet.h`

Prototype `in_addr_t inet_network (const char *cp)`

Parameters → *cp*
A character string.

Returns Returns a number suitable for use as an Internet network number.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `inet_addr()`

inet_ntoa Function

Purpose Takes an Internet address and returns an ASCII string representing the address.

Declared In `posix/arpa/inet.h`

Prototype `const char *inet_ntoa (struct in_addr in)`

Parameters → *in*
An Internet address.

Returns Returns a pointer to an ASCII string representing the address.

inet_ntop Function

Purpose Converts a network format address to presentation format.

Declared In `posix/arpa/inet.h`

Prototype
```
const char *inet_ntop (int af, const void *src,
    char *dst, size_t size)
```

Parameters

→ *af*
The address family.

→ *src*
The source buffer.

→ *dst*
The destination buffer.

→ *size*
The size of the destination buffer.

Returns Returns a pointer to the destination buffer. Otherwise, `NULL` is returned if a system error occurs and the global variable `errno` is set to indicate the error.

inet_pton Function

Purpose Converts a presentation format address to network format.

Declared In `posix/arpa/inet.h`

Prototype

```
int inet_pton (int af, const char *src,
   void *dst)
```

Parameters

→ *af*
The address family.

→ *src*
The printable form as specified in a character string.

→ *dst*
The destination string.

Returns Returns 1 if the address was valid for the specified address family, or zero (0) if the address was not parseable in the specified address family, or -1 if some system error occurred (in which case the global variable `errno` is set to indicate the error).

listen Function

Purpose Listens for connections on a socket.

Declared In `posix/sys/socket.h`

Prototype

```
int listen (int sock, int backlog)
```

Parameters

→ *sock*
The socket on which to listen for incoming connection attempts.

→ *backlog*
The maximum length the queue of pending connections may grow to.

Returns

Returns zero (0) if the connection or binding is successful. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also

`accept()`, `connect()`, `socket()`

ntohl Function

Purpose Converts 32-bit values between network byte order and host byte order.

Declared In `posix/netinet/in.h`

Prototype `uint32_t ntohl (uint32_t *net32*)`

Parameters → *net32*
The value being converted.

Returns Returns an unsigned integer.

See Also `gethostbyname()`, `getservent()`

ntohs Function

Purpose Converts 16-bit values between network byte order and host byte order.

Declared In `posix/netinet/in.h`

Prototype `uint16_t ntohs (uint16_t *net16*)`

Parameters → *net16*
The value being converted.

Returns Returns an unsigned short integer.

See Also `gethostbyname()`, `getservent()`

recv Function

Purpose Normally used only on a connected socket and is identical to `recvfrom()` with a `NULL` *addr* parameter.

Declared In `posix/sys/socket.h`

Prototype
```
ssize_t recv (int sock, void *data,
    size_t datalen, int flags)
```

Parameters → *sock*
A socket.

→ *data*
The message.

→ *datalen*
The length of the message.

→ *flags*
ORs together one or more of the values: MSG_OOB, MSG_PEEK, MSG_WAITALL.

Returns Returns the length of the message upon successful completion. Otherwise, -1 is returned and the global variable errno is set to indicate the error. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

See Also connect(), recvfrom(), recvmsg()

recvfrom Function

Purpose Receives messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

Declared In posix/sys/socket.h

Prototype
```
ssize_t recvfrom (int sock, void *data,
    size_t datalen, int flags,
    struct sockaddr *addr, socklen_t *addrlen)
```

Parameters

→ *sock*
A socket.

→ *data*
The message.

→ *datalen*
The length of the message.

→ *flags*
ORs together one or more of the values: MSG_OOB, MSG_PEEK, MSG_WAITALL.

→ *addr*
If *addr* is non-NULL, and the socket is not connection-oriented, the source address of the message is filled in.

← *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address stored there.

Returns Returns the length of the message upon successful completion. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

See Also `connect()`, `recv()`, `recvmsg()`

recvmsg Function

Purpose Receives messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

Declared In `posix/sys/socket.h`

Prototype
```
ssize_t recvmsg (int sd, struct msghdr *msg,
   int flags)
```

Parameters

→ *sd*
A socket.

→ *msg*
The message.

→ *flags*
ORs together one or more of the values: `MSG_OOB`, `MSG_PEEK`, `MSG_WAITALL`.

Returns Returns the length of the message upon successful completion. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

See Also `connect()`, `recv()`, `recvfrom()`

select Function

Purpose Examines the I/O descriptor sets whose addresses are passed in to see if some of their descriptors are ready.

Declared In `posix/sys/select.h`

Prototype
```
int select (int fd, fd_set *rfds, fd_set *wfds,
   fd_set *efds, struct timeval *timeout)
```

Parameters

→ *fd*
: The descriptors are checked in each set; that is, the descriptors from zero (0) through *fd* - 1 in the descriptor sets are examined.

→ *rfds*
: The descriptors are checked to see if some of them are ready for reading.

→ *wfds*
: The descriptors are checked to see if some of them are ready for writing.

→ *efds*
: The descriptors are checked to see if some of them have an exceptional condition pending.

→ *timeout*
: If *timeout* is a non-`NULL` pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a `NULL` pointer, then `select()` blocks indefinitely. To affect a poll, the *timeout* argument should be non-`NULL`, pointing to a zero-valued `timeval` structure.

Returns Returns the number of ready descriptors that are contained in the descriptor sets. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error. If the time limit expires, `select()` returns zero (0). If `select()` returns with an error, including one due to an interrupted call, the descriptor sets are unmodified.

See Also `accept()`, `connect()`, `recv()`, `send()`

send Function

Purpose Sends a message from a socket.

Declared In `posix/sys/socket.h`

Prototype

```
ssize_t send (int sock, const void *data,
    size_t datalen, int flags)
```

Parameters

→ *sock*
: A socket.

→ *`data`*
The message.

→ *`datalen`*
The length of the message.

→ *`flags`*
ORs together one or more of the values: `MSG_OOB`, `MSG_DONTROUTE`.

Returns Returns the number of characters sent. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

Comments May be used only when the socket is in a connected state.

See Also `select()`, `sendmsg()`, `sendto()`

sendmsg Function

Purpose Sends a message from a socket.

Declared In `posix/sys/socket.h`

Prototype
```
ssize_t sendmsg (int sd,
   const struct msghdr *msg, int flags)
```

Parameters

→ *`sd`*
A socket.

→ *`msg`*
The message.

→ *`flags`*
ORs together one or more of the values: `MSG_OOB`, `MSG_DONTROUTE`.

Returns Returns the number of characters sent. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `select()`, `send()`, `sendto()`

sendto Function

Purpose Sends a message from a socket.

Declared In `posix/sys/socket.h`

Prototype
```
ssize_t sendto (int sock, const void *data,
    size_t datalen, int flags,
    const struct sockaddr *addr,
    socklen_t addrlen)
```

Parameters

→ *sock*
A socket.

→ *data*
The message.

→ *datalen*
The length of the message.

→ *flags*
ORs together one or more of the values: `MSG_OOB`, `MSG_DONTROUTE`.

→ *addr*
If *addr* is non-`NULL`, and the socket is not connection-oriented, the source address of the message is filled in.

← *addrlen*
Initially contains the amount of space pointed to by *addr*; on return, it contains the actual length (in bytes) of the address stored there.

Returns Returns the number of characters sent. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `select()`, `send()`, `sendmsg()`

sethostent Function

Purpose Requests the use of a connected TCP socket for queries.

Declared In `posix/netdb.h`

Prototype `void sethostent (int stayopen)`

Parameters

→ *stayopen*
If the *stayopen* flag is non-zero, sets the option to send all queries to the name server using TCP and to retain the connection after each call to `gethostbyname()`, `gethostbyname2()`, or `gethostbyaddr()`. Otherwise, queries are performed using UDP datagrams.

See Also `gethostbyaddr()`, `gethostbyname()`, `gethostbyname2()`

setnetent Function

Purpose Opens and rewinds a file.

Declared In `posix/netdb.h`

Prototype `void setnetent (int stayopen)`

Parameters → *stayopen*
If non-zero, the network database is not closed after each call to `getnetbyname()` or `getnetbyaddr()`.

See Also `getnetbyaddr()`, `getnetbyname()`

setprotoent Function

Purpose Opens and rewinds a file.

Declared In `posix/netdb.h`

Prototype `void setprotoent (int stayopen)`

Parameters → *stayopen*
If non-zero, the network database is not closed after each call to `getprotobyname()` or `getprotobynumber()`.

See Also `getprotobyname()`, `getprotobynumber()`

setservent Function

Purpose Opens and rewinds a file.

Declared In `posix/netdb.h`

Prototype `void setservent (int stayopen)`

Parameters → *stayopen*
If non-zero, the network database is not closed after each call to `getservbyname()` or `getservbyport()`.

See Also `getservbyname()`, `getservbyport()`

setsockopt Function

Purpose Sets options on sockets.

Declared In `posix/sys/socket.h`

Prototype
```
int setsockopt (int sock, int level, int option,
    const void *optval, socklen_t optlen)
```

Parameters

→ *sock*
A socket.

→ *level*
To manipulate options at the socket level, *level* is specified as `SOL_SOCKET`.

→ *option*
Any specified option(s) passed uninterpreted to the appropriate protocol module for interpretation.

→ *optval*
Used to access option values. Identifies a buffer in which the value for the requested option is returned.

→ *optlen*
Used to access option values. Identifies a buffer in which the length for the requested option is returned.

Returns Returns zero (0) if the connection or binding is successful. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `getprotoent()`, `getsockopt()`, `select()`, `socket()`

shutdown Function

Purpose Disables subsequent send and/or receive operations on a socket.

Declared In `posix/sys/socket.h`

Prototype
```
int shutdown (int sock, int direction)
```

Parameters

→ *sock*
A socket.

→ *direction*
Specifies the type of shutdown. The values are as follows:

`SHUT_RD`
Disables further receive operations.

`SHUT_WR`
Disables further send operations.

`SHUT_RDWR`
Disables further send and receive operations.

Returns Returns zero (0) upon successful completion. Otherwise, 1 is returned and the global variable `errno` is set to indicate the error.

socket Function

Purpose Creates an endpoint for communication.

Declared In `posix/sys/socket.h`

Prototype `int socket (int` *`family`*`, int` *`type`*`, int` *`proto`*`)`

Parameters
→ *family*
A communications domain within which communication takes place; this selects the protocol family that should be used.

→ *type*
The semantics of communication.

→ *proto*
A particular protocol to be used with the socket.

Returns Returns a descriptor referencing the socket. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `getsockopt()`

Part V
WiFi

Palm OS® Cobalt, version 6.1 introduced support for 802.11 (WiFi) wireless networking. These chapters cover the APIs that allow your applications to manage WiFi connectivity.

16

Introduction to Wireless Networking

Overview

Palm OS® provides direct support for WiFi wireless networking through a set of `ioctl` commands that allow an application to find, connect to, and disconnect from wireless networks. Other commands allow applications to monitor the status of a wireless network, configure wireless security, and perform other standard management tasks necessary when using WiFi networking.

While the operating system includes the necessary user interface to manage WiFi connectivity, these `ioctl` commands are available for developers to provide custom solutions.

WiFi Concepts

A WiFi network is identified by an **SSID**. The SSID is an ASCII string of up to 32 characters.

There are two types of WiFi networks. The typical network, operating in **infrastructure mode**, is formed by devices connecting wirelessly to an access **access point**, which is a dedicated device. This device is also sometimes referred to as a base station. Each access point is a **Basic Service Set** (BSS). An **Extended Service Set** (ESS) is a entwork of one or more access points that is referred to by a single SSID.

The other type of network, called an **ad-hoc network**, is created when one or more devices are connected together wirelessly without a dedicated access point.

Locating and Opening a WiFi Interface

Before you can manage WiFi, you need to find and open the WiFi interface. This is done using `IOSGetDriverNameByIndex()` and `IOSOpen()`. See Listing 16.1 for an example.

Listing 16.1 Finding and opening the WiFi interface

```
char name[64];
uint16_t nameLen sizeof(name);
status_t err;

IOSGetDriverNameByIndex(iosDriverClassWifi, 0, (char *) name,
       &nameLen);
int32 wifiDataFD = IOSOpen(name, 0, &err);

strcat(name, "_mgmt");
int32 wifiMgmtFD = IOSOpen(name, 0, &err);
```

To send data over WiFi, you simply open the interface using the name returned by `IOSGetDriverNameByIndex()`. If, however, you want to send ioctl commands to the WiFi interface, you need to append the string "_mgmt" to the returned name to access the management interface.

Getting Information About the WiFi Interface

Before using the WiFi interface, your application may need to obtain information about the device's capabilities or status. For example, you may need to determine what forms of encryption it supports, whether or not it's already connected to a network, or what channels and transmission rates it supports.

Determining Supported Encryption Modes

To determine which encryption modes the interface supports, use the `WIOCGETSECCAPS` command, as demonstrated in Listing 16.2.

Listing 16.2 Getting supported encryption modes

```
WifiGetSecCapType modes;
status_t err;
```

```
IOSIoctl(wifiFD, WIOCGETSECCAPS, (int32_t) &modes, &err);

if (modes.capabilities & WifiSecOpen) {
   /* Open System is supported */
}

if (modes.capabilities & WifiSecWEP) {
   / * WEP is supported */
}
```

Getting the Interface Status

You can obtain information about the current status of the WiFi interface using the `WIOCGETSTATUS` `ioctl`. This is seen in Listing 16.3.

Listing 16.3 Getting the current status of the WiFi interface

```
uint32_t status;
status_t err;

IOSIoctl(wifiFD, WIOCGETSTATUS, (int32_t) &status, &err);
```

After `IOSIoctl()` returns, `status` contains a value indicating the current state of the WiFi interface:

- `WifiStatusDisconnected` indicates that the interface is not currently connected to a network.
- `WifiStatusConnectedAccessPoint` indicates that the interface is connected to an access point.
- `WifiStatusConnectedAdHoc` indicates that the interface is connected to an ad-hoc network.
- `WifiStatusOutOfRange` indicates that the interface is currently connected, but that the network is not currently in range. The state will automatically return to `WifiStatusConnectedAccessPoint` or `WifiStatusConnectedAdHoc` when the network is in range again.
- `WifiStatusConnecting` indicates that the interface is in the process of attempting to establish a connection.

- `WifiStatusConnectionFailed` indicates that the most recent connection attempt failed.
- `WifiStatusUndefined` indicates that for whatever reason, the interface's status could not be determined.

Identifying the Currently Connected Network

If you wish to determine the SSID or BSSID of the network to which the interface is connected, use the `WIOCGETSSID` or `WIOCGETBSSID` command.

Listing 16.4 Getting the name and BSSID of the access point or ad-hoc network

```
WifiSSIDType ssid;
WifiBSSIDType bssid;
status_t err;

IOSIoctl(wifiFD, WIOCGETSSID, (int32_t) &ssid, &err);
IOSIoctl(wifiFD, WIOCGETBSSID, (int32_t) &bssid, &err);
```

Determining Supported Channels and Transmission Rates

To determine which channels the WiFi interface supports, use the `WIOCGETCHANNEL` command. This also reports the channel the interface is currently using.

Listing 16.5 Determining supported channels

```
status_t err;
WifiChannelType channels;

channels.current = 0;
channels.supportedMask = 0;

IOSIoctl(wifiFD, WIOCGETCHANNEL, (int32_t) &channels, &err);
```

After this code executes, `channels.current` is set to the channel on which the interface is currently communicating, and `channels.supportedMask` is a bit mask of all the channels the interface supports. See "Channel Constants" on page 347 for a list of the channel number flags.

The code in Listing 16.6 determines the rates supported by the interface, which rates are preferred, and what rate is currently in use.

Listing 16.6 Determining supported transmission rates

```
WifiGetRatesType rates;
status_t err;

rates.preferred_rates = 0;
rates.supported_rates = 0;
rates.current_rate = 0;

IOSIoctl(wifiFD, WIOCGETRATES, (int32_t) &rates, &err);
```

On return, `rates.current_rate` indicates the transmission rate currently in effect, `rates.supported_rates` is a bit mask of all the transmission rates the interface supports, and `rates.preferred_rates` is a bit mask of the rates the interface is best suited for. See "Transmission Rate Flags" on page 351 for the possible values.

NOTE: The preferred rates always default to the complete set of supported rates. You may change them if you wish, using the `WIOCSETRATES` ioctl.

Getting the Signal Strength

There are two ways to keep apprised of the current signal strength. You can manually poll the signal strength using the `WIOCGETCURRENTRSSI` command, or you can enable automatic signal strength update notification.

Listing 16.7 Getting the current signal strength

```
WifiGetRSSIType current;

IOSIoctl(wifiFD, WIOCGETCURRENTRSSI, (int32_t) &current,
            &err);
```

After the code in Listing 16.7 executes, `current.signal` contains the current signal strength, as a percentage between 0 and 100.

To receive periodic notification of changes to the signal strength, use the `WIOCSETRSSIUPDATE` command. With this command, you can choose to receive notification events whenever any change to signal strength occurs, whenever the signal strength changes by a given amount, or at a specific interval.

Listing 16.8 Enabling automatic signal strength notifications

```
WifiRSSIUpdateType update;
status_t err;

/* notify me when signal strength changes by +/- 2% */

update.updateMode = WifiRSSIUpdateOnDelta;
update.updateValue = 2;

/* notify me every 1000 milliseconds */

update.updateMode = WifiRSSIUpdatePeriodic;
update.updateValue = 1000;

/* notify me every time the signal strength changes */

update.updateMode = WifiRSSIUpdateAlways;
update.updateValue = 0;

/* never notify me of signal strength changes */

update.updateMode = WifiRSSIUpdateNever;
update.updateValue = 0;

IOSIoctl(wifiFD, WIOCSETRSSIUPDATE, (int32_t) &update, &err);
```

The example in Listing 16.8 shows how to set up the `WifiRSSIUpdateType` structure for each of the four notification modes. Once the structure is prepared, call IOSIoctl() to issue the request.

Finding an Access Point or Ad-hoc Network

To locate an access point or ad-hoc network to which you can connect, you need to use the WIOCSCAN or WIOCPASSIVESCAN command.

Active Scanning

If you want to simply perform a one-time scan of the airwaves for available ad-hoc networks and access points, use WIOCSCAN. See Listing 16.9.

Listing 16.9 Performing a one-shot scan for access points and ad-hoc networks

```
WifiScanRequestType cmd;
status_t err;

memset(&cmd, 0, sizeof(WifiScanRequestType));

cmd.channels = WifiChannel_All;
cmd.rates = WifiRate_All;
cmd.timeout = 2000;
cmd.blockTillCompletion = 0;
IOSIoctl(wifiFD, WIOCSCAN, (int32_t) &cmd, &err);
```

The scan is performed asynchronously; the IOSIoctl() will return immediately. Your application's event loop will receive WiFi events with the scan results. See "Obtaining Scan Results" on page 340 for details on how to parse the results.

The blockTillCompletion flag indicates whether or not you want the ioctl to block until the first scan result arrives. This example sets it to 0, indicating that we want to return immediately.

Passive Scanning

If you would prefer to constantly be kept informed of the available access points and ad-hoc networks, as they move in and out of range, or are turned on and off, you can enable passive scanning mode. While in passive scanning mode, your application's event loop will receive scan result events when appropriate. See Listing 16.10 for an example of how to enable passive scanning.

Listing 16.10Enabling passive scanning

```
WifiPassiveScanType scan;
status_t err;

memset(&scan, 0, sizeof(WifiPassiveScanType));

scan.enableScanning = true;
scan.channelMask = WifiChannel_All;
scan.rateMask = WifiRate_All;
scan.interval = 1000;

IOSIoctl(wifiFD, WIOCPASSIVESCAN, (int32_t) &scan, &err);
```

The example above enables scanning for access points or ad-hoc networks operating on any channel and at any transmission rate. Scan results will be delivered to the application every 1,000 milliseconds.

To disable passive scanning, issue the `WIOCPASSIVESCAN` command again, with the `enableScanning` field set to `false`.

Obtaining Scan Results

Normally, your application receives scan results as a `wifiScanResults` event in its event loop. The event's `WifiEventType` structure describes the detected access point in detail. Each time an access point or ad-hoc network is found, a `wifiScanResults` event is delivered.

You can also manually fetch the scan results from the WiFi adapter by using the `WIOCGETSCANRESULTS` command. This command can be used in a loop to fetch all the scan results available, as seen in Listing 16.11.

Listing 16.11Using WIOCGETSCANRESULTS

```
uint16_t index = 0;
uint16_t last = 0;
WifiGetScanResultsType scan;
status_t err;

do {
   memset(&scan, 0, sizeof(WifiGetScanResultsType));
   scan.last = last;
   scan.index = index;

   IOSIoctl(m_fd, WIOCGETSCANRESULTS, (int32_t)&scan, &err);

   if (err == P_OK) {
      /* results received successfully in scan */
   }
   else {
      /* error receiving scan results */
   }

   last = scan.last;
   index = scan.index;

   index++;
} while (index <= last);
```

Configuring Encryption

WiFi supports the concept of encryption to protect data security. There are two security modes currently supported by Palm OS: open system (unencrypted) and Wired Equivalent Privacy (WEP).

To use WEP encryption, an encryption key needs to be configured prior to connecting to the network. There are three steps required to accomplish this. First, it's necessary to store the key in the adapter. A WiFi adapter can store up to four encryption keys, which can then be selected among depending on which network is being accessed.

Listing 16.12Setting an encryption key

```
WifiSetWEPKeyType arg;
status_t err;
```

```
arg.key = 0;                /* key number to set */

memset(arg.data, 0, 16);
arg.data_len = Ascii2Binary(arg.data, keyString, 16);

IOSIoctl(wifiFD, WIOCSETKEY, (int32_t) &arg, &err);
```

The code in Listing 16.12 sets key 0 to the string specified by `keyString`. See Listing 16.17 for the code for the `Ascii2Binary()` function.

Once the key has been stored on the adapter, it must be selected using the `WIOCSETDEFAULTKEY` command. See Listing 16.13.

Listing 16.13Selecting the default key

```
status_t err;

IOSIoctl(wifiFD, WIOCSETDEFAULTKEY, (int32_t) 0, &err);
```

Finally, once the key has been selected, it's possible to put the interface into WEP mode by using the `WIOCSETSECMODE` command, as seen in Listing 16.14.

Listing 16.14Enabling encryption

```
status_t err;
uint32_t mode = WifiSecWEP;

IOSIoctl(wifiFD, WIOCSETSECMODE, (int32_t) &mode, &err);
```

To disable encryption, simply set the mode to `WifiSecOpen`.

Connecting To a Network

Once you have found an access point or ad-hoc network to which you wish to connect, you can connect to that network using either the `WIOCCONNECT` or the `WIOCJOIN` `ioctl`.

If you have the SSID of the network or ad-hoc network, you use the `WIOCCONNECT` command, as shown in Listing 16.15.

Listing 16.15Connecting to a wireless network using an SSID

```
WifiConnectType arg;
status_t err;

strncpy(arg.ssid, theSSID, 32);
arg.timeout = 3000;
arg.blockTillCompletion = false;
IOSIoctl(wifiFD, WIOCCONNECT, (int32_t) &arg, &err);
```

In this example, we choose to try for three seconds (3,000 milliseconds) before timing out. In addition, since the `blockTillCompletion` flag is set to false, the call will return at once. We must then check the status of the connection periodically to detect when the connection is actually opened (or if the connection fails to open).

Your event loop (using either a Pollbox or `IOSPoll()`) will receive notifications as the status of the connection changes: `wifiConnectAccessPoint` or `wifiConnectAdHoc` when the connection is established, `wifiConnecting` while connection is being attempted, `wifiOutOfRange` if the access point is out of range but was opened anyway under the assumption that it will be eventually, `wifiMediaUnavailable` if the 802.11 hardware is missing, or `wifiConnectFailed` if the connection could not be established.

If you have the BSSID (MAC address) and channel number of a network to which you wish to connect, you can use the `WIOCJOIN` command instead, as shown in Listing 16.16.

Listing 16.16Connecting to a wireless network using a BSSID and channel number

```
WifiJoinType arg;
status_t err;

Ascii2Binary(arg.bssid, "FF:FF:FF:FF:FF", 6);
arg.channel = theChannel;
IOSIoctl(wifiFD, WIOCJOIN, (int32_t) &arg, &err);
```

This code uses a function called `Ascii2Binary()` to convert the MAC ID string into the proper format. `Ascii2Binary()` is shown in Listing 16.17.

Listing 16.17Converting a hex string into packed binary format

```
int Ascii2Binary(uint8_t * buf, const char* p, size_t length)
{
   uint8_t nibble;
   size_t i = 0;
   static char map[22] = {'0', '1', '2', '3', '4', '5', '6',
'7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c',
'd', 'e', 'f'};

   while (*p != NULL && i < length) {

      // Skip over MAC octet separators
      if (*p == ':') {
         p++;
         continue;
      }
      nibble = 0;
      for(int j = 0; j < 22; j++) {
         if (p[0] == map[j]) {

            if (j >= 16)
               nibble = j - 6;
            else
               nibble = j;
            break;
         }
      }

      buf[i] |= nibble << 4;
      p++;

      if (*p == NULL)
         break;

      nibble = 0;
      for(int j = 0; j < 22; j++) {
         if (p[0] == map[j]) {
            if (j >= 16)
               nibble = j - 6;
            else
               nibble = j;
            break;
```

```
            }
        }

        buf[i] |= nibble;
        i++;
        p++;
    }

    return i;
}
```

Once a connection has been established, the wireless network can be used just like any other network connection, using the Sockets API or IOS STDIO calls.

Managing a Wireless Connection

Once the connection is established, your application's event loop will receive events on the WiFi management file descriptor, informing you of changes in the status of the connection, as well as results of specific requests you issue. Your event loop needs to either poll the file descriptor, or use a Pollbox. These concepts are covered in "Polling STREAMS File Descriptors" on page 375.

WiFi events use the `WifiEventType` structure to return data to your event handler. The possible events are listed in "Event Type Constants" on page 349.

Disconnecting From a Network

To disconnect from a WiFi network, use the `WIOCDISCONNECT` command. See Listing 16.18.

Listing 16.18Disconnecting from a WiFi network

```
status_t err;
IOSIoctl(wifiFD, WIOCDISCONNECT, NULL, &err);
```

Creating an Ad-hoc Network

Palm OS devices can create an ad-hoc network using the `WIOCCREATEIBSS` `ioctl`.

Listing 16.19Creating an ad-hoc network

```
WifiCreateIBSSType ibss;
status_t err;

memset(&ibss, 0, sizeof(WifiCreateIBSSType));
strncpy(ibss.ssid, "MyAdhocNet", 32);
ibss.channel = 8;

IOSIoctl(wifiFD, WIOCCREATEIBSS, (int32_t) &ibss, &err);
```

The example in Listing 16.19 creates a new ad-hoc network named "MyAdhocNet" operating on channel 8. If this is successful, other WiFi-enabled devices can then connect to the new ad-hoc network just like any other network.

17

WiFi Reference

Declared In `wifi.h`

Overview

This chapter describes the constants, data types, and ioctl command codes used to control WiFi connectivity.

WiFi Constants

Channel Constants

Purpose Define which channels are usable or preferable when multiple channels can be used.

Constants

`WifiChannel_1`
    Channel 1

`WifiChannel_2`
    Channel 2

`WifiChannel_3`
    Channel 3

`WifiChannel_4`
    Channel 4

`WifiChannel_5`
    Channel 5

`WifiChannel_6`
    Channel 6

`WifiChannel_7`
    Channel 7

`WifiChannel_8`
    Channel 8

`WifiChannel_9`
Channel 9

`WifiChannel_10`
Channel 10

`WifiChannel_11`
Channel 11

`WifiChannel_12`
Channel 12

`WifiChannel_13`
Channel 13

`WifiChannel_14`
Channel 14

`WifiChannel_All`
All channels are allowed.

Connection Status Constants

Purpose Values that may be used when reporting the association status of the connection.

Constants `WifiStatusUndefined`
The current connection state is undefined.

`WifiStatusDisconnected`
The interface is disconnected.

`WifiStatusConnecting`
The interface is in the process of attempting to connect.

`WifiStatusConnectedAccessPoint`
The interface is connected to an access point.

`WifiStatusConnectedAdHoc`
The interface is connected to an ad-hoc network.

`WifiStatusOutOfRange`
The interface is connected to an access point that is out of range.

`WifiStatusConnectionFailed`
The connection attempt has failed.

Event Type Constants

Purpose Define events that can be delivered to indicate changed conditions on the 802.11 network.

Constants

`wifiUndefinedEvent`
: An event that does not fall under any of the other defined event codes.

`wifiConnecting`
: The station is currently attempting to connect to an access point or ad-hoc network.

`wifiConnectAccessPoint`
: The station has successfully connected to an access point.

`wifiConnectAdHoc`
: The station has successfully connected to an ad-hoc network.

`wifiConnectFailed`
: The interface could not establish a connection. If the reason for the failure can be determined, the reason code will indicate what it is.

`wifiOutOfRange`
: The access point to which the station is connected is out of range. If the station moves back into range of the access point, the interface should return to the `wifiConnectAccessPoint` or `wifiConnectAdHoc` state.

`wifiDisconnect`
: The station is not connected.

`wifiScanResults`
: The `wifiScanResults` event contains information about a scanned access point or ad-hoc network that has been found.

`wifiSignalStrength`
: The interface is reporting updated signal strength information.

`wifiMediaUnavailable`
: The 802.11 hardware has been removed from the device, or has been disabled. The file descriptor for the connection, as well as the device name, are no longer valid.

`wifiScanFailed`
: The 802.11 interface did not find any access points or ad-hoc networks. If the reason for the failure can be determined, the reason code will specify why.

Power Mode Constants

Purpose Define the power mode for the radio hardware.

Constants

`WifiPowerOff`
: The radio hardware is off.

`WifiPowerOn`
: The radio hardware is on.

`WifiPowerOnPowerSave`
: The radio hardware is in a reduced-power, but still operational, mode.

`WifiPowerOffHardSwitch`
: The radio hardware has been switched off using an external switch.

RSSI Update Mode Constants

Purpose Define the RSSI update modes.

Constants

`WifiRSSIUpdateNever`
: Disables asynchronous RSSI updates.

`WifiRSSIUpdateOnDelta`
: Specifies that RSSI updates will be sent when the signal strength changes by a given percentage.

`WifiRSSIUpdatePeriodic`
: Specifies that RSSI updates will be generated at a regular interval, which is specified in milliseconds.

`WifiRSSIUpdateAlways`
: Specifies that RSSI update events will be generated every time new signal strength information is available.

Comments Changing the RSSI update mode allows applications to control how frequently they receive an update. For example, an application designed to test signal strength may want updates constantly, while a typical status bar signal strength monitor would prefer less frequent updates to reduce processor impact.

Scan Result Capability Constants

Purpose Define bit flags indicating the capabilities of an 802.11 station that responded to a scan. These constants correspond to values defined in section 7.3.1.4 of the IEEE 802.11 Specification.

Constants

`WifiCapAccessPoint`
The station is an access point.

`WifiCapAdhocNetwork`
The station can be connected to to form an ad-hoc network.

`WifiCapPrivacy`
The station supports some form of security protocol. The security protocol may be WEP or WPA.

Comments `WifiCapAccessPoint` and `WifiCapAdhocNetwork` are mutually exclusive.

Security Capability Constants

Purpose Define the security capabilities of the WiFi network.

Constants

`WifiSecOpen`
The network is not secure.

`WifiSecWEP`
The network supports WEP.

Transmission Rate Flags

Purpose Define the data transmission rate capabilities and preferences of the client hardware and access points.

Constants

`WifiRate_0Mbit`
0 Mbps

`WifiRate_1Mbit`
1 Mbps

`WifiRate_2Mbit`
2 Mbps

`WifiRate_5_5Mbit`
5.5 Mbps

`WifiRate_6Mbit`
6 Mbps

`WifiRate_9Mbit`
9 Mbps

`WifiRate_11Mbit`
11 Mbps

`WifiRate_12Mbit`
12 Mbps

`WifiRate_18Mbit`
18 Mbps

`WifiRate_22Mbit`
22 Mbps

`WifiRate_24Mbit`
24 Mbps

`WifiRate_33Mbit`
33 Mbps

`WifiRate_36Mbit`
36 Mbps

`WifiRate_48Mbit`
48 Mbps

`WifiRate_54Mbit`
54 Mbps

`WifiRate_All`
Shorthand indicating that all rates are supported.

WEP Flag Constants

Purpose Define flags used to configure WEP. These constants are only valid when the WiFi encryption mode is set to `WifiSecWEP`.

Constants

`WifiWEPExcludeUnencrpyted`
: If this flag is set, unencrypted frames are discarded automatically.

`WifiWEPIVReuseEvery`
: Reuse the initialization vector every frame.

`WifiWEPIVReuse10`
: Reuse the initialization vector every 10 frames.

`WifiWEPIVReuse50`
: Reuse the initialization vector every 50 frames.

`WifiWEPIVReuse100`
: Reuse the initialization vector every 100 frames.

Comments The initialization vector reuse flags are mutually exclusive.

WiFi Data Structures and Types

WifiEventType Struct

Purpose Describes a single WiFi event.

Prototype

```
typedef struct {
   uint32_t event;

   union {
      struct scan{
         uint16_t index;
         uint16_t last;
         WifiScanResultsType results;
      } scan;

      struct connectAccessPoint {
         char   ssid[33];
         uint8_t padding[3];
         uint8_t bssid[6];
      } connectAccessPoint;
```

```
        struct connectAdhoc {
           char ssid[33];
           uint8_t padding[3];
           uint8_t bssid[6];
        } connectAdhoc;

        struct connectFailed {
           status_t reasonCode;
        } connectFailed;

        struct scanFailed {
           status_t reasonCode;
        } scanFailed;

        struct signalStrength {
           uint8_t signal;
        } signalStrength;
     } data;
} WifiEventType;
```

Fields

`scan`
: Describes a scan result. This event includes the following additional data:

  `index`
  : The index within the series of scan results that are being reported.

  `last`
  : The index of the last element in the series.

  `results`
  : A description of the scanned station or access point.

`connectAccessPoint`
: Indicates that a connection to an access point has occurred.

  `ssid`
  : The ESSID of the network to which the connection has been established.

  `bssid`
  : The 6-byte MAC address of the access point to which the connection has been established.

`connectAdHoc`
: Indicates that a connection to an ad-hoc network has occurred.

  `ssid`
  : The ESSID of the ad-hoc network to which the connection has been established.

  `bssid`
  : The 6-byte MAC address of the ad-hoc network to which the connection has been established.

`connectFailed`
: Indicates that a connection attempt has failed.

  `reasonCode`
  : An integer indicating the reason for the failure.

`scanFailed`
: Indicates that a scan attempt has failed.

  `reasonCode`
  : An integer indicating the reason for the failure.

`signalStrength`
: Provides updated signal strength information.

  `signal`
  : A value from 0 to 100 indicating the current signal strength as a percentage of maximum strength.

WifiScanResultsType Struct

Purpose Describes an access point or ad-hoc network discovered during a scan.

Prototype

```
typedef struct {
   char ssid[33];
   int8_t signal;
   int8_t noise;
   uint8_t channel;
   uint8_t bssid[6];
   uint16_t ATIMInterval;
   uint32_t supportedRates;
```

```
    uint32_t responseRate;
    uint32_t capabilities;
    uint16_t beaconInterval;
    uint8_t padding[2];
} WifiScanResultsType;
```

Fields

`ssid`
The network name of the scanned access point or ad-hoc network. An SSID is a null terminated ASCII string of 1 to 32 characters in length.

`signal`
The signal level at which the probe response was received, in dbm.

`noise`
The average noise level detected while the probe response was being received, in dbm.

`channel`
The channel number on which the access point or ad-hoc network is operating.

NOTE: The channel number is an integer, not a constant from the "Channel Constants" list.

`bssid`
The MAC address that identifies the access point or station that created the ad-hoc network.

`ATIMInterval`
The ATIM time window, in units of 100 microseconds. This field is only valid for ad-hoc networks.

`supportedRates`
A bit mask of all the transmission rates supported by the scanned access point or station. See "Transmission Rate Flags" on page 351 for the possible values.

`capabilities`
A bit mask of the capabilities of the scanned access point or station. See "Scan Result Capability Constants" on page 351 for the possible flag values.

`beaconInterval`
Specifies how frequently the access point or station transmits a beacon frame. This is an integer value in units of 100 microseconds.

IOCTL Commands

WIOCCONNECT

Purpose Initiates a connection to an access point or ad-hoc network.

Prototype

```
struct {
   uint32_t timeout;
   char ssid[33];
   uint8_t blockTillCompletion;
   uint8_t pad[2];
} WifiConnectType
```

Fields

→ *`timeout`*
The length of time, in microseconds, for the 802.11 framework to wait before giving up on the connection attempt.

→ *`ssid`*
A null-terminated ASCII string containing the SSID of the network to which to connect. The string length must be between one and 32 characters.

→ *`blockTillCompletion`*
If `true`, the `WIOCONNECT` ioctl will block until the connection has been established or a timeout occurs.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

Comments Using this command while already connected to a network causes the existing connection to be terminated and a new one to be opened.

See Also `WIOCDISCONNECT`

WIOCCREATEIBSS

Purpose Creates an independent BSS (ad-hoc network).

Prototype
```
struct {
   char ssid[33];
   uint8_t channel;
   uint8_t padding[2];
} WifiCreateIBSSType
```

Fields

← *ssid*
The name to give the newly created ad-hoc network. The name must be a null terminated ASCII string of 1 to 32 characters in length

← *channel*
The channel number on which to create the ad-hoc network. This must be one of the channels supported by the radio hardware, as reported by `WIOCGETCHANNEL`. The value must be an integer, rather than one of the channel flag values.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

Comments This command will terminate any existing connection.

WIOCDISCONNECT

Purpose Disconnects from the access point or ad-hoc network to which the device is currently connected, or terminates a connection attempt in progress.

Prototype None.

Fields None.

Returns Nothing.

Comments This call should only be used on a connected interface, or one that is in the process of trying to connect.

See Also `WIOCCONNECT`

WIOCGETBSSID

Purpose Retrieves the the BSSID of the access point or ad-hoc network to which the 802.11 interface is connected.

Prototype

```
struct {
   uint8_t bssid[6];
} WifiBSSIDType
```

Fields ← *bssid*

The 48-bit MAC address representing the BSSID of the access point or ad-hoc network.

Returns `errNone`

Success.

`EINVAL`

One of the parameters is invalid.

Other errors as appropriate.

WIOCGETCHANNEL

Purpose Returns the channel on which the interface is connected, as well as a bit mask indicating which channels are supported by the radio hardware.

Prototype

```
struct {
   uint32_t current;
   uint32_t supportedMask;
} WifiChannelType
```

Fields ← *current*

The integer number of the channel on which the hardware is currently connected.

← *supportedMask*

A bit mask indicating which channels the radio hardware supports. See "Channel Constants" on page 347. for possible values.

Returns `errNone`

Success.

`EINVAL`

One of the parameters is invalid.

Other errors as appropriate.

WIOCGETCURRENTRSSI

Purpose Returns the current signal strength of the wireless connection.

Prototype

```
struct {
   uint8_t signal;
   uint8_t padding[3];
} WifiGetRSSIType
```

Fields ← *signal*

The signal strength of the connection, as a percentage from 0 to 100.

Returns `errNone` if successful, otherwise an appropriate negative error code.

WIOCGETMACADDR

Purpose Returns the MAC address of the WiFi interface.

Prototype

```
struct {
   uint8_t bssid[6];
} WiFiBSSIDType
```

Fields ← *bssid*

The BSSID (MAC address) of the WiFi interface.

Returns `errNone` if successful, otherwise an appropriate negative error code.

WIOCGETPOWERMODE

Purpose Returns the current power mode for the radio hardware.

Prototype `uint32_t mode`

Fields ← *mode*

The retrieved power mode setting. See "Power Mode Constants" on page 350 for possible values.

Returns `errNone` if successful, otherwise an appropriate negative error code.

See Also [WIOCSETPOWERMODE]

WIOCGETRATES

Purpose Retrieves information about the transmission settings and capabilities of the radio hardware.

Prototype

```
struct {
   uint32_t supported_rates;
   uint32_t preferred_rates;
   uint32_t current_rate;
} WifiGetRatesType
```

Fields

← *supported_rates*
A mask of all the transmission rates supported by the radio hardware.

← *preferred_rates*
A subset of the supported_rates, which may be used when negotiating the transmission rate with an access point or ad-hoc network.

← *current_rate*
The transmission rate currently in use, if the device is associated with an access point or ad-hoc network.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

See Also [WIOCSETRATES]

WIOCGETRSSIUPDATE

Purpose Retrieves the rules governing when the 802.11 adapter sends an RSSI update event.

Prototype

```
struct {
   uint32_t updateMode;
   uint32_t updateValue;
} WifiRSSIUpdateType
```

Fields

← *updateMode*
The RSSI update mode currently in use. See "RSSI Update Mode Constants" on page 350 for a list of possible values.

← *updateValue*
The frequency of updates. If *updateMode* is `WifiRSSIUpdatePeriodic`, this is the time interval in milliseconds between updates. If *updateMode* is `WifiRSSIUpdateOnDelta`, then this value is the amount of change in signal strength, in percentage points, that must be exceeded before an update is sent. This value is 0 for other modes.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

See Also `WIOCSETRSSIUPDATE`

WIOCGETSCANRESULTS

Purpose Retrieves scan results from the results cache stored in the 802.11 adapter.

Prototype

```
struct {
   uint16_t index;
   uint16_t last;
   WifiScanResultsType results;
} WifiGetScanResultsType
```

Fields

→ *index*
The index number of the scan result to retrieve. On the first invocation of this command, the value should be zero. For subsequent invocations, the value should be set to a value in the range 0 to the value returned in *last*.

← *last*
The index of the last scan result. On the first invocation of this command, set this value to zero. When the command returns, it is set to the index of the last scan result available.

Returns

errNone
Success.

EINVAL
One of the parameters is invalid.

Other errors as appropriate.

Comments Use this command to fetch the results of either a passive or active scan.

See Also WIOCPASSIVESCAN, WIOCSCAN

WIOCGETSECCAPS

Purpose Retrieves information about the current security settings and capabilities for the 802.11 interface.

Prototype

```
struct {
   uint32_t current;
   uint32_t capabilities;
} WifiGetSecCapType
```

Fields

← *current*
The current security mode in effect on the 802.11 interface. See "Security Capability Constants" on page 351. Only one security mode can be in effect.

← *capabilities*
A bit mask of all the security modes supported by the 802.11 interface.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

See Also `WIOCSETSECMODE`

WIOCGETSSID

Purpose Retrieves the SSID of the access point or ad-hoc network to which the 802.11 interface is currently connected.

Prototype

```
struct {
   char ssid[33];
   uint8_t padding[3];
} WifiSSIDType
```

Fields

← *ssid*
A null terminated ASCII string with a length between 1 and 32 characters, indicating the SSID of the access point or ad-hoc network to which the interface is connected.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

WIOCGETSTATUS

Purpose Gets the current connection status for the 802.11 interface.

Prototype `uint32_t status`

Fields ← *`status`*
The current connection status of the 802.11 interface. See "Connection Status Constants" on page 348.

Returns `errNone`
Success.

`EINVAL`
The *`status`* parameter is invalid.

Other errors as appropriate.

WIOCGETWEPFLAGS

Purpose Retrieves the current WEP configuration flags.

Prototype `uint32_t flags`

Fields ← *`flags`*
A mask of all the flags currently set. See "WEP Flag Constants" on page 353.

Returns `errNone`
Success.

`EINVAL`
The *`flags`* parameter is invalid.

Other errors as appropriate.

See Also `WIOCSETWEPFLAGS`

WIOCJOIN

Purpose Initiates a connection to an access point or ad-hoc network using a BSSID instead of an SSID to specify the network.

Prototype

```
struct {
   uint8_t bssid[6];
   uint16_t channel;
} WifiJoinType
```

Fields

→ *bssid*
: The 48-bit MAC address of the BSS the 802.11 interface should join.

→ *channel*
: The channel number on which to create the ad-hoc network. This must be one of the channels supported by the radio hardware, as reported by `WIOCGETCHANNEL`. The value must be an integer, rather than one of the channel flag values.

Returns

`errNone`
: Success.

`EINVAL`
: One of the parameters is invalid.

Other errors as appropriate.

Comments Using this command while already connected to a network causes the existing connection to be terminated and a new one to be opened.

See Also `WIOCDISCONNECT`

WIOCPASSIVESCAN

Purpose Instructs the 802.11 interface to passively scan for available access points and ad-hoc networks at regular intervals. This command is also used to terminate passive scanning.

Prototype

```
struct {
   uint32_t interval;
   uint32_t channelMask;
   uint32_t rateMask;
   uint8_t ssid[33];
   uint8_t enableScanning;
   uint8_t padding[2]
} WifiPassiveScanType;
```

Fields

→ *interval*
The time interval between scans, in milliseconds.

→ *channelMask*
A bit mask of all the channels that should be checked during the scan. See "Channel Constants" on page 347 for possible values.

→ *rateMask*
A bit mask of all the transmission rates to check for while performing the scan. See "Transmission Rate Flags" on page 351 for possible values.

→ *ssid*
This parameter, which is optional, lets you specify the SSID of a specific network to search for access points within. If you do not wish to use this restriction, this parameter should be set to an empty string.

→ *enableScanning*
If `true`, passive scanning is started; if `false`, passive scanning is canceled and all other parameters are ignored.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

Comments After this ioctl is issued, use the `WIOCGETSCANRESULTS` ioctl to obtain the results.

See Also `WIOCSCAN`, `WIOCGETSCANRESULTS`

WIOCSCAN

Purpose Instructs the 802.11 interface to perform an active scan for available access points and ad-hoc networks.

Prototype

```
struct {
   uint32_t channels;
   uint32_t rates;
   uint32_t timeout;
   uint8_t ssid[33];
   uint8_t blockTillCompletion;
   uint8_t padding[2]
} WifiScanRequestType;
```

Fields

→ *`channels`*
A bit mask of all the channels that should be checked during the scan. See "Channel Constants" on page 347 for possible values.

→ *`rates`*
A bit mask of all the transmission rates to check for while performing the scan. See "Transmission Rate Flags" on page 351 for possible values.

→ *`timeout`*
The length of time, in milliseconds, that the 802.11 framework will wait before giving up on a scan. A timeout of zero may be specified if the scan should not time out.

→ *`ssid`*
This parameter, which is optional, lets you specify the SSID of a specific network to search for access points within. If you do not wish to use this restriction, this parameter should put to an empty string.

→ *`blockTillCompletion`*
If `true`, the `WIOCSCAN` ioctl will block until the first scan result comes in.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

Comments

After this ioctl is issued, use the `WIOCGETSCANRESULTS` ioctl to obtain the results.

See Also

`WIOCPASSIVESCAN`, `WIOCGETSCANRESULTS`

WIOCSETDEFAULTKEY

Purpose

Sets the default WEP key for the radio hardware.

Prototype

```
uint32_t key
```

Fields

→ *key*
The key number to use as the default key. Must be a value in the range 0 to 3.

Returns

`errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

Comments

If WEP encryption is disabled, the default key will be set but not used.

WIOCSETKEY

Purpose

Sets one of the four WEP keys for the radio hardware.

Prototype

```
struct {
   uint16_t key;
   uint16_t len;
   uint8_t buffer[MAX_KEY_VALUE_LEN];
} WifiSetWEPKeyType
```

Fields

→ *key*
The key number to set. Must be a value in the range 0 to 3.

→ *len*
The length of the key, in bytes. A 40-bit key requires a 5-byte buffer, while a 104-bit key requires a 13-byte buffer.

→ *buffer*
The key itself.

Returns `errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

WIOCSETPOWERMODE

Purpose Sets the current power mode for the radio hardware.

Prototype `uint32_t mode`

Fields ↔ *mode*
The power mode to use. See "Power Mode Constants" on page 350 for possible values.

Returns `errNone` if successful, otherwise an appropriate negative error code.

Comments The caller should check `mode` on output to verify that the requested mode was in fact set. For example, if there is a hardware switch locking WiFi power off, an attempt to turn WiFi on would fail.

See Also `WIOCGETPOWERMODE`

WIOCSETRATES

Purpose Sets the preferred transmission rates for the radio hardware. The actual transmission rate is selected by negotiation turing the process of associating with an access point or ad-hoc network.

Prototype `uint32_t rate_mask`

Fields → *`rate_mask`*
A bit mask containing all the transmission rates that the station should use. See "Transmission Rate Flags" on page 351 for possible values.

Returns `errNone`
Success.

`EINVAL`
The *`rate_mask`* specified is invalid.

Other errors as appropriate.

See Also `WIOCGETRATES`

WIOCSETRSSIUPDATE

Purpose Configures the rules governing when the 802.11 adapter sends an RSSI update event.

Prototype

```
struct {
   uint32_t updateMode;
   uint32_t updateValue;
} WifiRSSIUpdateType
```

Fields → *`updateMode`*
The RSSI update mode to use. See "RSSI Update Mode Constants" on page 350 for a list of possible settings.

→ *`updateValue`*
The frequency of updates. If *`updateMode`* is `WifiRSSIUpdatePeriodic`, this is the time interval in milliseconds between updates. If *`updateMode`* is `WifiRSSIUpdateOnDelta`, then this value is the amount of change in signal strength, in percentage points, that must be exceeded before an update is sent.

Returns `errNone`
Success.

`EINVAL`
One of the parameters is invalid.

Other errors as appropriate.

See Also WIOCGETRSSIUPDATE

WIOCSETSECMODE

Purpose Selects the security scheme for the 802.11 interface.

Prototype `uint32_t mode`

Fields → *mode*
The security mode to use on the 802.11 interface. See "Security Capability Constants" on page 351. You may only specify one security mode.

Returns `errNone`
Success.

`EINVAL`
The *mode* is invalid.

Other errors as appropriate.

See Also WIOCGETSECCAPS

WIOCSETWEPFLAGS

Purpose Sets options related to WEP.

Prototype `uint32_t flags`

Fields → *flags*
A mask of all the flags to set. See "WEP Flag Constants" on page 353.

Returns `errNone`
Success.

`EINVAL`
The *flags* are invalid.

Other errors as appropriate.

See Also WIOCGETWEPFLAGS

Part VI
IOS STDIO

The Standard I/O (STDIO) interface provided by the I/O Subsystem (IOS) in Palm OS® lets programmers use familiar Posix-like functions to access Palm OS device drivers. This part covers this API, as well as other IOS APIs for managing drivers.

18

Using IOS STDIO

Introducing IOS STDIO

The IOS STDIO shared library provides a set of functions that are mostly compatible with the Posix STDIO interface. These functions forward I/O requests to the I/O Subsystem (IOS) for processing. The STDIO function calls include calls to open and close devices, read and write data, and perform control operations on devices.

In general, the functions in the Palm OS IOS STDIO library are used similarly to those in the standard Posix standard I/O library; the key difference is that the Palm OS versions have the prefix “IOS” added to the function names.

You can learn much more about these functions by reading any good Unix programming book.

Synchronization Issues

The I/O Process, in order to optimize performance, directly accesses the calling process’ memory space to read from and write into buffers. Although the calling thread is blocked, other threads in the calling process might access that memory at the same time as the I/O Process, which would cause synchronization problems.

Therefore, if this may be an issue for your application, be sure to use a semaphore or other mutual exclusion device.

Polling STREAMS File Descriptors

Your application’s main event loop can reduce its overall impact on the performance of the device by blocking until a user interface event occurs. This can be done using the `IOSPoll()` function, similar to the example in Listing 18.1.

Listing 18.1 An example main event loop that blocks until an event occurs

```
status_t error;
EventType event;
int32_t eventFd;
int32_t fdCount;
struct pollfd fdSet[1];

eventFd = EvtGetEventDescriptor();
fdSet[0].fd = (int) eventFd;
fdSet[0].events = (short)(POLLIN | POLLHUP);
fdCount = 1;

do {
  if ((error = IOSPoll(fdSet, fdCount, 10000, &fdCount)) !=
        errNone) {
    printf("IOSPoll() failed with error: 0x%08lx\n", error);
  }

  EvtGetEvent(&event, 0);

  if (!SysHandleEvent(&event)) {
    if (!MenuHandleEvent(0, &event, &error)) {
      if (!ApplicationHandleEvent(&event)) {
        FrmDispatchEvent(&event);
      }
    }
  }
} while (event.eType != appStopEvent);
```

This code calls `EvtGetEventDescriptor()` to determine the file descriptor for the event queue.

Once that's done, it calls `IOSPoll()` to block for 10,000 milliseconds or until message occurs on the event queue's file descriptor. Once an event occurs, it is fetched using `EvtGetEvent()` and the event is processed normally.

By blocking on `IOSPoll()`, this event loop avoids busy-waiting—the practice of looping constantly, non-stop, executing code that repeatedly checks for pending events. This saves processor time for other tasks.

Using a PollBox to Monitor Multiple File Descriptors

Applications that use multiple file descriptors can simplify their event loops by using a PollBox. A PollBox is a mechanism that automatically handles polling, calling a specified routine each time an event occurs that affects a file descriptor you're monitoring.

Once an application creates a PollBox, it can add and remove file descriptors from the set of file descriptors to monitor right from within its event loop. Each file descriptor has a callback routine associated with it, which is called whenever an event affects the file descriptor.

Creating a PollBox

Creating a PollBox is a simple matter of calling the `PbxCreate()` function, as shown in Listing 18.2.

Listing 18.2 Creating a PollBox

```
#include <PollBox.h>

...

PollBox *pbx = PbxCreate();
```

Destroying a PollBox

When your application is done using the PollBox, it must return resources to the system by calling `PbxDestroy()`. This will close all the file descriptors currently in the PollBox and free all memory used by the box. This is demonstrated in Listing 18.3.

Listing 18.3 Destroying a PollBox

```
PbxDestroy(pbx);
```

Adding File Descriptors to Monitor

To add a file descriptor to the set of file descriptors being monitored by a PollBox, your application should call the `PbxAddFd()` function, as shown in Listing 18.4.

Listing 18.4 Adding a file descriptor to a PollBox

```
status_t err = PbxAddFd(pbx, fd, eventMask, MyCallbackProc, myContextPtr);
```

The *eventMask* passed into the `PbxAddFd()` function is a bitwise OR of one or more of the following values:

POLLIN: Set this if your application should be informed when a non-priority message is available for the file descriptor.

POLLPRI: Set this if your application should be informed when a high-priority message is available for the file descriptor.

POLLOUT: Set this bit if your application should be informed when a message is sent on the file descriptor.

The *MyCallbackProc* parameter is a pointer to a callback routine, which will be called whenever any of the desired events occur. It will be called with the *myContextPtr* pointer as one of its parameters.

NOTE: If you want to poll without receiving callbacks, you can specify `NULL` for the callback procedure pointer.

You can poll for the existence of user interface events by using the file descriptor returned by `EvtGetEventDescriptor()`, although you can't use IOS to read the events. So to handle user interface events, your application can set up a special callback just for handling those, which calls the appropriate Event Manager and other functions to fetch and handle the events:

Listing 18.5 Adding the UI file descriptor to a PollBox

```
status_t err = PbxAddFd(pbx, EvtGetEventDescriptor(), POLLIN,
    MyUICallbackProc, NULL);
```

Removing a File Descriptor from the PollBox

The `PbxRemoveFd()` function removes a file descriptor from a PollBox, as shown in Listing 18.6.

Listing 18.6 Removing a file descriptor from a PollBox

```
PbxRemoveFd(pbx, fd);
```

Polling for Events using a PollBox

Once you've added all the file descriptors you wish to monitor to the PollBox, you can simply call the `PbxPoll()` function in a loop to watch for events, as shown in Listing 18.7. The `PbxPoll()` function automatically dispatches events to the appropriate callback handlers, so all you have to do is watch for error conditions, and possibly perform some idle activities.

Listing 18.7 Polling for events

```
for (;;) {
        err = PbxPoll( pbx, timeout, &nReady );
        if ( err ) {
            // Some unexpected error occurred.
        } else if ( nReady == 0 ) {
            if ( pbx->count == 0 ) {
                // There are no more file descriptors in the pollbox.
            } else {
                // The timer expired before any events occurred.
            }
        } else {
            // Normal case. There are pbx->count > 0 file descriptors in
            // the pollbox, and nReady of them have events. The callbacks
            // associated with the ready file descriptors have been called.
            // If you are working without callbacks, then do something here
            // using the contents of the pollbox.
        }
    }
```

The call to `PbxPoll()` blocks until at least one event is available on at least one file descriptor, or the specified timeout period elapses. The timeout is specified in milliseconds.

NOTE: If you wish the `PbxPoll()` function to return immediately if no events are pending, specify 0 as the timeout. If you don't want it to time out at all, specify -1 instead.

When `PbxPoll()` returns, any callbacks that apply have already been called; `nReady` contains the number of file descriptors that have events waiting and `pbx->count` indicates the total number of file descriptors in the PollBox. If your application isn't using callbacks, you can look at the contents of the `pbx` PollBox and perform whatever actions your application needs to.

Polling the Easy Way

As you can probably see, in the typical case, all you need to do is call `PbxPoll()` in a loop until your application is ready to quit. For this case, you can use the convenient `PbxRun()` function, as demonstrated in Listing 18.8.

Listing 18.8 The easy way to write an event loop

```
status_t err = PbxRun(pbx);
if (err) {
   // Some unexpected error occurred
} else {
   // There are no fds left in the PollBox; this is a normal exit
   PbxDestroy(pbx);
}
```

This can literally be your entire event loop. If your application uses a UI event callback on file descriptor 0, that callback can cause the application to exit by simply removing all the file descriptors from the PollBox, which will cause `PbxRun()` to exit.

Implementing a PollBox Callback

Your callback procedures must be of type `PbxCallback`:

```
void PbxCallback(PollBox *pbx, struct pollfd *pollFd, void *context);
```

The first parameter is a pointer to the PollBox itself. The second parameter is a pointer to the `pollfd` structure associated with the file descriptor on which the event has occurred. The fields your callback can access within this structure are shown in Table 18.1.

Table 18.1 pollFd fields

Field Name	Description
`fd`	The file descriptor (read-only).
`events`	The current event mask for `IOSPoll()` (read/write).
`revents`	The events returned from `IOSPoll()` (read-only).

The final parameter is a pointer to the context variable specified when your application called `PbxAddFd()` to add the file descriptor to the PollBox.

In Listing 18.9, we see an example of a callback procedure.

Listing 18.9 Sample callback procedure

```
void MyCallback( PollBox* pbx, struct pollfd *pollFd, void* context );
    {
        MyContext*  ctx = (MyContext*)context;
        status_t         err = 0;

        if ( pollFd->revents & POLLIN ) {
            IOSGetMsg( pollFd->fd, ctx->ctlBuf, ctx->dataBuf, 0, &err );
        }

        if ( err || (pollFd->revents & (POLLERR|POLLHUP)) ) {
            PbxRemoveFd( pbx, pollFd->fd );
            IOSClose( pollFd->fd );
            return;
        }

        // Handle the event that has been read into the ctl and data buffers.
        ...
    }
```

If the message received is a `POLLIN` event, the callback calls the `IOSGetmsg()` function to fetch the message. In this case, the context variable is a structure into which the data gets copied.

If an an error occurred on the file descriptor, or it's been hung up, we remove the file descriptor from the PollBox and close it, then return to the caller.

Other processing can be handled here as needed. For example, if your application is using IOS STDIO calls to communicate with a Bluetooth device, you may receive events from the Bluetooth Management Entity which need to be handled.

19

IOS STDIO Reference

Overview

This chapter covers the IOS STDIO API. IOS STDIO provides an architecture for communicating directly at a low level with any kind of communications device for which there are drivers available, through a standard, unified API. Using IOS STDIO calls, it's possible to write communications applications that can use any network or serial connection, without having to write custom code for each type of interface.

IOS STDIO also provides the pollbox—an automated event polling mechanism that can ease development of event-driven applications such as communications software.

IOS STDIO Data Structures and Types

cc_t Typedef

Purpose Specifies a control character in a `termios` structure.

Declared In `SDK/posix/termios.h`

Prototype

```
typedef unsigned char cc_t;
```

iovec Struct

Purpose The `IOSReadv()` and `IOSWritev()` functions pass an array of `iovec` data structures that represent the scattered data array to read from or write to. Each `iovec` structure contains a pointer and a byte length.

Declared In IOS.h

Prototype

```
struct iovec {
   MemPtr iAddrP;
   int32_t iLen;
};
```

Fields

`iAddrP`
: Base address of the buffer.

`iLen;`
: Length of the buffer.

Comments All of the pointers in the array of `iovec` data structures must be in the same memory segment.

PollBox Struct

Purpose Describes a PollBox. Most of the fields in this structure are private.

Declared In PollBox.h

Prototype

```
typedef struct PollBox {
   struct pollfd *pollTab;
   uint16_t count;
   uint16_t capacity;
   uint16_t flags;
   uint16_t nCallbacks;
   PbxInfo *infoTab;
} PollBox;
```

Fields

`pollTab`
: A table of `pollfd` structures, one per file descriptor that's in the PollBox.

`count`
: The number of file descriptors in the PollBox.

`capacity`
: The current maximum number of file descriptors the PollBox can contain; reserved for system use.

`flags`
: Internal flags; reserved for system use.

`nCallbacks`
The number of file descriptors that have callbacks assigned; reserved for system use.

`infoTab`
A table of `PbxInfo` structures, one per file descriptor. Reserved for system use.

pollfd Struct

Purpose The `IOSPoll()` function passes an array of `pollfd` data structures that represent the file descriptors and events to poll. Each `pollfd` structure contains a file descriptor, a mask of events to check, and a mask of events that are selected.

Declared In `IOS.h`

Prototype

```
struct pollfd {
   int32_t fd;
   int16_t events;
   int16_t revents;
};
```

Fields

`fd`
The file descriptor to poll.

`events`
The events in which you're interested.

`revents`
On return, contains the events which have occurred.

speed_t Typedef

Purpose Specifies the baud rate for a connection in a `termios` structure.

Declared In `SDK/posix/termios.h`

Prototype

```
typedef unsigned long speed_t;
```

strbuf Struct

Purpose The `IOSPutmsg()`, `IOSPutpmsg()`, `IOSGetmsg()`, and `IOSGetpmsg()` functions pass two `strbuf` structures which represent the data and control buffers. Each `strbuf` contains the maximum length of the buffer, the length of the data currently in the buffer, and a pointer to the data buffer.

Declared In `IOS.h`

Prototype

```
struct strbuf {
   int32_t iMaxLen;
   int32_t iLen;
   MemPtr iBufP;
};
```

Fields

`iMaxLen`
: The maximum number of bytes that the `iBufP` buffer can hold. Only used by `IOSGetmsg()` and `IOSGetpmsg()`.

`iLen`
: The length of the data currently in the `iBufP` buffer.

`iBufP`
: A pointer to the buffer.

tcflag_t Typedef

Purpose Specifies control modes for a connection in a `termios` structure.

Declared In `SDK/posix/termios.h`

Prototype

```
typedef unsigned long tcflag_t;
```

termios Struct

Purpose Contains all the settings for a communications channel.

Declared In `SDK/poxis/termios.h`

Prototype

```
struct termios {
   tcflag_t c_iflag;
   tcflag_t c_oflag;
   tcflag_t c_cflag;
   tcflag_t c_lflag;
   char c_line;
   cc_t c_cc[NCC];
   speed_t c_ispeed;
   speed_t c_ospeed;
};
```

Fields

`c_iflag`
Input modes.

`c_oflag`
Output modes.

`c_cflag`
Control modes.

`c_lflag`
Local modes.

`c_line`
Line discipline.

`c_cc[NCC]`
Control characters.

`c_ispeed`
Input speed.

`c_ospeed`
Output speed.

IOS STDIO Constants

Character Control Mode Constants

Purpose Define constants that control character and flow modes for a connection. Used in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants

Constant	Definition
`CSIZE`	Character sizes.
`CS7`	OR with `CSIZE` to specify 7-bit characters: `CSIZE|CS7`.
`CS8`	OR with `CSIZE` to specify 8-bit characters: `CSIZE|CS8`.
`CSTOPB`	Send two stop bits instead of the normal one.
`CREAD`	Enable the receiver.
`PARENB`	Enable transmit parity.
`PARODD`	Select odd parity. If this flag isn't set, and parity is enabled, parity is even.
`HUPCL`	If this flag is set, the line will be hung up after the last file descriptor accessing it is closed.
`CLOCAL`	If set, indicates a local line.
`XLOBLK`	Block layer output.
`CTSFLOW`	Enables CTS flow control.
`RTSFLOW`	Enables RTS flow control.

Constant	Definition
CRTSCTS (RTSFLOW | CTSFLOW)	Enables both CTS and RTS flow control.
IRDAENB	Enables IrDA encoding.

Input Control Mode Constants

Purpose Define constants that specify input mode settings for connections. Used in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants

Constant	Definition
IGNBRK	Ignore breaks.
BRKINT	Break sends an interrupt.
IGNPAR	Ignore characters with parity errors.
PARMRK	Mark parity errors.
INPCK	Enable input parity checking.
ISTRIP	Strip the high bit from received characters.
INLCR	Map newline to CR on input.
IGNCR	Ignore carriage returns.
ICRNL	Map CR to newline on input.
IUCLC	Map all upper-case characters to lower-case.
IXON	Enable software flow control on input.
IXANY	Any character received will restart input after flow control has disabled input.
IXOFF	Enables output software flow control.

Ioctl Command Constants

Purpose Define commands that can be sent to `IOSIoctl()`. This list is not exhaustive; that is, some drivers will implement additional commands, and some may not implement all of these. This is simply a list of common constants.

Declared In `SDK/posix/sys/ttycom.h`
`SDK/posix/termios.h`

Constants

Constant	Definition
`TIOCMGET`	Returns all of the modem's status flags. The `IOSIoctl()` function's *`iParam`* output pointer should point to an integer variable which will be filled with the status on return.
`TIOCGETA`	Returns a `termios` structure describing the device's settings. The *`iParam`* parameter should point to a `termios` struct, which will be filled in with the current device settings. This is the same as `TCGETA`.
`TCGETA`	Returns a `termios` structure describing the device's settings. The *`iParam`* parameter should point to a `termios` struct, which will be filled in with the current device settings. This is the same as `TIOCGETA`.
`TIOCSETA`	Sets the device's communication settings to match those in the `termios` structure passed in *`iParam`*. This is the same as `TCSETA`.
`TCSETA`	Sets the device's communication settings to match those in the `termios` structure passed in the *`iParam`* parameter. This is the same as `TIOCSETA`.
`TCSBRK`	Sets a break condition on the line. *`iParam`* is unused. This is the same as `TIOCSBRK`.
`TIOCSBRK`	Sets a break condition on the line. *`iParam`* is unused. This is the same as `TCSBRK`.

Constant	Definition
`TIOCCBRK`	Clears the break condition on the line. *`iParam`* is unused.
`TIOCSDTR`	Sets the DTR condition on the line. *`iParam`* is unused.
`TIOCCDTR`	Clears the DTR condition on the line. *`iParam`* is unused.
`TIOCDRAIN`	Blocks the calling thread until the transmit queue is empty. *`iParam`* is unused.
`TCSETIRDAMODE`	Selects read or write mode on IrDA devices. *`iParam`* should point to an integer 0 value to select write mode, or to a non-zero value to select read mode.

Local Mode Constants

Purpose Define local modes that can be specified in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants

Constant	Definition
`ISIG`	Enable signals.
`ICANON`	Canonical input.
`XCASE`	Canonical upper/lower case.
`ECHO`	Enable echo.
`ECHOE`	Echo erase as backspace/space/backspace.
`ECHOK`	Echo newline after kill.
`ECHONL`	Echo newlines.
`NOFLSH`	Disable flush after interrupt or quit.

Constant	Definition
`TOSTOP`	Stop background processes that write to the connection.
`IEXTEN`	Implementation-defined extensions begin here.

Modulation Speed Constants

Purpose Define the supported baud rates. Used in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants

Constant	Definition
`B0`	0 bps.
`B50`	50 bps.
`B75`	75 bps.
`B110`	110 bps.
`B134`	134 bps.
`B150`	150 bps.
`B200`	200 bps.
`B300`	300 bps.
`B600`	600 bps.
`B1200`	1200 bps.
`B1800`	1800 bps.
`B2400`	2400 bps.
`B4800`	4800 bps.
`B7200`	7200 bps.
`B9600`	9600 bps.
`B14400`	14,400 bps.

Constant	Definition
B19200	19,200 bps.
B28800	28,800 bps.
B38400	38,400 bps.
B56000	56,000 bps.
B57600	57,600 bps.
B76800	76,800 bps.
B115200	115,200 bps.
B128000	128,000 bps.
B230400	230,400 bps.
B256000	256,000 bps.
B31250	31,250 bps. Used by MIDI.

NCC Constant

Purpose Defines the number of control characters that can be specified in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants `NCC`

This constant specifies the size of the array of control characters in the `termios` structure.

Output Control Mode Constants

Purpose Define output control modes. These constants are used in the `termios` structure.

Declared In `SDK/posix/termios.h`

Constants

Constant	Definition
`OPOST`	Enable post-processing of output.
`OLCUC`	Maps lower case to upper case on output.
`ONLCR`	Maps newlines to CR+NL on output.
`OCRNL`	Maps CR to newline on output.
`ONOCR`	No CR output when in column 0.
`ONLRET`	Newline performs a CR.
`OFILL`	Uses fill characters for delays.
`OFDEL`	Fills are DEL, otherwise NUL.
`NLDLY`	Newline delays.
`NL0`	Add this to `NLDLY` to choose the `NL0` delay: `NLDLY+NL0`.
`NL1`	Add this to `NLDLY` to choose the `NL1` delay: `NLDLY+NL1`.
`CRDLY`	Carriage return delays.
`CR0`	Add this to `CRDLY` to choose the `CR0` delay: `CRDLY+CR0`.
`CR1`	Add this to `CRDLY` to choose the `CR1` delay: `CRDLY+CR1`.
`CR2`	Add this to `CRDLY` to choose the `CR2` delay: `CRDLY+CR2`.
`CR3`	Add this to `CRDLY` to choose the `CR3` delay: `CRDLY+CR3`.

Constant	Definition
TABDLY	Tab delays.
TAB0	Add this to TABDLY to choose the TAB0 delay.
TAB1	Add this to TABDLY to choose the TAB1 delay.
TAB2	Add this to TABDLY to choose the TAB2 delay.
TAB3	Add this to TABDLY to choose the TAB3 delay.
BSDLY	Backspace delays.
BS0	Add this to BSDLY to choose the BS0 delay.
BS1	Add this to BSDLY to choose the BS1 delay.
VTDLY	Vertical tab delays.
VT0	Add this to VTDLY to choose the VT0 delay.
VT1	Add this to VTDLY to choose the VT1 delay.
FFDLY	Form feed delays.
FF0	Add this to FFDLY to choose the FF0 delay.
FF1	Add this to FFDLY to choose the FF1 delay.

Poll Mask Constants

Purpose Define the events that can be polled for by `IOSPoll()`.

Declared In `IOS.h`

Constants

Constant	Definition
POLLIN	A non-priority message is available.
POLLPRI	A high-priority message is available.
POLLOUT	The stream is writable for non-priority messages.
POLLERR	An error message has arrived.

Constant	Definition
POLLHUP	A hangup has occurred.
POLLNVAL	The specified file descriptor isn't valid.
POLLRDNORM	A non-priority message is available.
POLLRDBAND	A priority (band > 0) message is available.
POLLWRNORM	Same as POLLOUT.
POLLWRBAND	A priority band exists and is writable.
POLLMSG	A signal message has reached the front of the queue.

Functions

IOSClose Function

Purpose Closes the specified device.

Declared In `IOS.h`

Prototype `status_t IOSClose( int32_t iFD )`

Parameters → *iFD*
The file descriptor to close.

Returns Returns `errNone` if the operation was successful; otherwise the operation failed and an appropriate error code is returned. Possible errors are:

`errNone`
No Error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
The specified file descriptor is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not reference an open device.

Comments Any thread in a process may close a file descriptor opened by that process.

When a process terminates, all associated file descriptors are closed automatically. However, since there is a limit on how many file descriptors can be opened at once, it is a good idea to close them as you're finished using them.

IOSFastIoctl Function

Purpose Performs one of a variety of control functions on a device.

Declared In `IOS.h`

Prototype

```
int32_t IOSIoctl( int32_t iFD, int32_t iRequest,
    int32_t iSendLen, MemPtr iSendP,
    int32_t iRecvLen, MemPtr iRecvP,
    status_t *oErrno )
```

Parameters

→ *iFD*
The file descriptor of the device.

→ *iRequest*
The command to be executed on the device.

→ *iSendLen*
The length of the send buffer.

→ *iSendP*
A pointer to the send buffer.

→ *iRecvLen*
The length of the receive buffer.

→ *iRecvP*
A pointer to the receive buffer.

← *oErrno*
The error code.

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrIOError`
: An I/O error occurred.

`iosErrNotOpened`
: The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
: The device does not support this operation.

Returns Returns the number of bytes actually received. If an error occurred, returns -1, and the actual error code in `oErrno`.

Comments `FastIoctl()` calls are only supported by certain Palm OS internal devices.

The maximum send and receive buffer lengths are determined by the driver.

IOSFattach Function

Purpose Attaches a STREAMS-based file descriptor to a given pathname.

Declared In `IOS.h`

Prototype

```
status_t IOSFattach( int32_t iFD,
   const Char *iPath )
```

Parameters

→ *`iFD`*
: The file descriptor of the device.

→ *`iPath`*
: The null-terminated pathname of the device.

Returns Returns `errNone` if the operation succeeded; otherwise returns one of the following error codes:

`errNone`
: No error.

`iosErrCanceled`
: The operation was canceled.

`iosErrDeviceInUse`
: The device is already opened and cannot be shared.

`iosErrDeviceNotFound`
The device pathname is not a valid IOS device.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support this operation.

Comments *`iFD`* must reference an open STREAMS-based file descriptor. All subsequent operations on *`iPath`* will operate on the STREAMS file until the STREAMS file is detached by calling `IOSFdetach()`. *`iFD`* can be attached to more than one path.

IOSFdetach Function

Purpose Detaches a STREAMS-based file descriptor from the specified pathname.

Declared In `IOS.h`

Prototype `status_t IOSDetach( const Char *iPath )`

Parameters → *`iPath`*
The null-terminated pathname of the device.

Returns Returns `errNone` if the operation succeeded; otherwise returns one of the following error codes:

`errNone`
No error.

`iosErrAccess`
The caller does not have the required permissions for this operation.

`iosErrCanceled`
The operation was canceled.

`iosErrDeviceNotFound`
The device pathname is not a valid IOS device.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrNotSupported`
: The device does not support this operation.

Comments This function detaches a STREAMS-based file descriptor from the path to which it was associated by a prior call to `IOSFattach()`. The *`iPath`* parameter points to the pathname to the attached STREAMS file.

IOSFnctl Function

Purpose Performs one of a variety of operations on an open file descriptor.

Declared In `IOS.h`

Prototype

```
int32_t IOSFnctl( int32_t iFD, int32_t iRequest,
    int32_t iArg, status_t *oErrno )
```

Parameters

→ *`iFD`*
: The file descriptor of the device.

→ *`iRequest`*
: The operation to perform on the file descriptor.

→ *`iArg`*
: Any additional information required by the command.

← *`oErrno`*
: The error code.

`errNone`
: No error.

`iosErrCanceled`
: The operation was canceled.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrNotOpened`
: The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
: The device does not support this operation.

Returns Returns a non-negative request-specific value if successful. Returns -1 if an error occurred; the specific error code is returned in *oErrno*.

Comments The allowed commands are defined in the `fcntl.h` header file. Only `F_GETFL` and `F_SETFL` are supported in the current release of Palm OS.

IOSGetmsg Function

Purpose Receives a STREAMS message.

Declared In `IOS.h`

Prototype

```
int32_t IOSGetmsg( int32_t iFD,
   struct strbuf *oCtlPtrP,
   struct strbuf *oDataPtrP, int32_t oFlags,
   status_t *oErrno )
```

Parameters

→ *iFD*
The file descriptor of the device from which a STREAMS message is to be received.

← *oCtlPtrP*
A pointer to a `strbuf` into which the control part of the message is to be stored.

← *oDataPtrP*
A pointer to a `strbuf` into which the data part of the message is to be stored.

← *oFlags*
The message's priority:

`RS_HIPRI`
High priority

`0`
Normal, non-priority

← *oErrno*
The error code indicating the result of the operation.

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support this operation.

Returns A return value of 0 indicates that a full message was successfully received. A return value of `MORECTL` indicates that more control information is waiting to be read. Similarly, a return value of `MOREDATA` indicates that more data is waiting to be read. A return value of `MORECTL | MOREDATA` indicates that more of both control information and data are waiting to be read.

Comments **NOTE:** The file descriptor must reference a STREAMS device.

IOSGetpmsg Function

Purpose Receives a STREAMS message.

Declared In `IOS.h`

Prototype

```
int32_t IOSGetpmsg( int32_t iFD,
   struct strbuf *oCtlPtrP,
   struct strbuf *oDataPtrP, int32_t *oBand,
   int32_t oFlags, status_t *oErrno )
```

Parameters

→ *iFD*
The file descriptor of the device from which a STREAMS message is to be received.

← *oCtlPtrP*
A pointer to a `strbuf` into which the control part of the message is to be stored.

← *oDataPtrP*
A pointer to a `strbuf` into which the data part of the message is to be stored.

↔ *ioBand*
The message's priority band.

↔ *ioFlags*
The message's priority:

`MSG_HIPRI`
High priority

`MSG_BAND`
Band priority

`MSG_ANY`
Any priority

← *oErrno*
The error code indicating the result of the operation.

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support this operation.

Returns A return value of 0 indicates that a full message was successfully received. A return value of `MORECTL` indicates that more control information is waiting to be read. Similarly, a return value of `MOREDATA` indicates that more data is waiting to be read. A return value of `MORECTL | MOREDATA` indicates that more of both control information and data are waiting to be read.

Comments You may choose to retrieve only high-priority messages by setting the integer pointed to by *ioFlags* to `MSG_HIPRI` and the integer pointed to by *ioBand* to 0. In this case, `IOSGetpmsg()` will only process the next message if it is a high-priority message.

Similarly, you can opt to only process a message from a given priority band by setting the integer pointed to by *ioFlags* to `MSG_BAND` and the integer pointed to by *ioBand* to the priority band of interest. In this case, `IOSGetpmsg()` will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *ioBand*, or if it is a high priority message.

If you just want to fetch the next message off the queue, set the integer pointed to by *ioFlags* to `MSG_ANY` and the integer pointed to by *ioBand* to 0.

On return, *ioBand* and *ioFlags* are set to indicate the priority band and priority of the message returned.

NOTE: The file descriptor must reference a STREAMS device.

IOSIoctl Function

Purpose Performs one of a variety of control functions on a device.

Declared In `IOS.h`

Prototype

```
int32_t IOSIoctl( int32_t iFD, int32_t iRequest,
    int32_t iParam, status_t *oErrno )
```

Parameters

→ *iFD*
The file descriptor of the device.

→ *iRequest*
The command to be executed on the device.

→ *iParam*
Any additional information required by the command.

← *oErrno*
The error code.

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support this operation.

Returns Returns a non-negative request-specific value if successful. Returns -1 if an error occurred; the specific error code is returned in *`oErrno`*.

IOSOpen Function

Purpose Opens a device for reading, writing, and control.

Declared In `IOS.h`

Prototype

```
int32_t IOSOpen( const Char *iPath,
   int32_t iFlags, status_t *oErrno )
```

Parameters

→ *`iPath`*
The null-terminated pathname of the device to open.

→ *`iFlags`*
A bitwise OR of flags specifying the access privileges requested:

`O_RDONLY`
Open for read only.

`O_WRONLY`
Open for write only.

`O_RDWR`
Open for both reading and writing.

← *`oErrno`*
On return, contains the error code indicating success or failure. Possible errors are:

`errNone`
No error.

`iosErrAuthFailed`
: The caller is not authorized to use the requested device.

`iosErrCanceled`
: The operation was canceled.

`iosErrDevice`
: The device is already opened and cannot be shared.

`iosErrDeviceNotFound`
: The requested device could not be found.

`iosErrInvalidArg`
: Invalid argument.

`iosErrIOError`
: An I/O error occurred.

`iosErrNoFileDescriptors`
: The system is out of free file descriptors.

`iosErrNoSessionEntry`
: The system is out of free file descriptors.

`iosErrNotSupported`
: This operation is not supported by the specified device.

Returns Returns a file descriptor for the opened device. Returns -1 if an error occurred; the specific error code is stored in *`oErrno`*.

Comments The pathname must not span multiple memory segments.

IOSPipe Function

Purpose Creates an interprocess communication channel (called a "pipe").

Declared In `IOS.h`

Prototype `status_t IOSPipe( int32_t oFD[2] )`

Parameters ← *`oFD`*
: Two file descriptors.

Returns Returns `errNone` if the operation succeeded. Otherwise returns an appropriate error code:

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrIOError`
An I/O error occurred.

`iosErrNoFileDescriptors`
The caller has no free file descriptors left.

`iosErrNoSessionEntry`
The system is out of free file descriptors.

`iosErrNotSupported`
The device does not support this operation.

Comments The `IOSPipe()` function creates an I/O interprocess communication channel called a pipe, returning two file descriptors in *`oFD`*[0] and *`oFD`*[1]. These file descriptors are STREAMS-based and are opened for both reading and writing.

Reading from `oFD`[0] returns data written to `oFD`[1] and vice versa.

IMPORTANT: Pipes are not supported in Palm OS Cobalt, but will be available in a future release.

IOSPoll Function

Purpose Examines a set of file descriptors to see if any of them are ready for I/O.

Declared In `IOS.h`

Prototype
```
status_t IOSPoll( struct pollfd iFDs[],
   int32_t iNfds, int32_t iTimeout,
   int32_t *oNfds )
```

Parameters

→ *iFDs*
An array of `pollfd` structures, each containing a file descriptor to be polled, the events to poll for, and the events that actually occurred.

If the value of the file descriptor field in a `pollfd` structure is less than zero, then the `iEvents` member is ignored and the `oRevents` member is set to 0 on return.

→ *iNfds*
The number of `pollfd` structures in the array.

→ *iTimeout*
The number of milliseconds to wait before timing out. If -1, `IOSPoll()` blocks indefinitely. If the timeout is 0, `IOSPoll()` does not block.

← *oNfds*
The number of file descriptors selected.

Returns Returns `errNone` if the operation is successful; otherwise returns an error code:

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotSupported`
The device does not support this operation.

Comments `IOSPoll()` is only supported for STREAMS devices. File descriptor 0 is used to poll for pending user interface events.

IOSPutmsg Function

Purpose Sends a STREAMS message.

Declared In `IOS.h`

Prototype
```
status_t IOSPutmsg( int32_t iFD,
   const struct strbuf *iCtlPtrP,
   const struct strbuf *iDataPtrP,
   int32_t iFlags )
```

Parameters

→ *iFD*
The file descriptor of the device to which the STREAMS message is to be sent.

→ *iCtlPtrP*
A pointer to a `strbuf` containing the control portion of the message.

→ *iDataPtrP*
A pointer to a `strbuf` containing the data portion of the message.

→ *iFlags*
The message's priority:

`RS_HIPRI`
High priority

`0`
Normal, non-priority

Returns Returns `errNone` if the operation was successful. Otherwise returns an error code:

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support reading data.

Comments The `IOSPutmsg()` function creates a message containing either the control portion from *`iCtlPtrP`* or the data portion from *`iDataPtrP`* (or both) and sends it to the STREAMS device specified by the file descriptor *`iFD`*, using the priority specified by *`iFlags`*.

NOTE: The file descriptor must reference a STREAMS device.

IOSPutpmsg Function

Purpose Sends a STREAMS message.

Declared In `IOS.h`

Prototype

```
status_t IOSPutpmsg( int32_t iFD,
   const struct strbuf *iCtlPtrP,
   const struct strbuf *iDataPtrP, int32_t iBand,
   int32_t iFlags )
```

Parameters

→ *`iFD`*
The file descriptor of the device to which the STREAMS message is to be sent.

→ *`iCtlPtrP`*
A pointer to a `strbuf` containing the control portion of the message.

→ *`iDataPtrP`*
A pointer to a `strbuf` containing the data portion of the message.

`iBand`
The priority band.

→ *`iFlags`*
The message's priority:

`MSG_HIPRI`
High priority

`MSG_BAND`
Band priority

Returns Returns `errNone` if the operation was successful. Otherwise returns an error code:

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support reading data.

Comments The `IOSPutmsg()` function creates a message containing either the control portion from *`iCtlPtrP`* or the data portion from *`iDataPtrP`* (or both) and sends it to the STREAMS device specified by the file descriptor *`iFD`*, using the priority specified by *`iFlags`*.

The *`iFlags`* argument is a bit mask which must be either `MSG_HIPRI` or `MSG_BAND`. If *`iFlags`* is 0, `IOSPutpmsg()` fails and returns `iosErrInvalidArg`. If a control part is specified and *`iFlags`* is set to `MSG_HIPRI` and *`iBand`* is 0, a high-priority message is sent.

If *`iFlags`* is set to `MSG_HIPRI` and either no control part is specified or *`iBand`* is non-zero, `IOSPutpmsg()` fails and returns `iosErrInvalidArg`.

If *`iFlags`* is set to `MSG_BAND`, then a message is sent in the priority band specified by *`iBand`*. If a control part and data part are not specified and *`iFlags`* is set to `MSG_BAND`, no message is sent and `errNone` is returned.

NOTE: The file descriptor must reference a STREAMS device.

IOSRead Function

Purpose Reads data from a device.

Declared In `IOS.h`

Prototype
```
int32_t IOSRead( int32_t iFD, MemPtr iBufP,
   int32_t iNbytes, status_t *oErrno )
```

Parameters

→ *iFD*
: The file descriptor of the device from which data should be read.

→ *iBufP*
: A pointer to the memory buffer into which data should be read.

→ *iNbytes*
: The number of bytes to read from the device.

← *oErrno*
: On output, contains the result code indicating the error which occurred, or `errNone` if the data was read successfully.

Returns Returns the number of bytes actually read, which may be lower than the number of bytes requested (if, for example, the end of the available data is reached). If an error occurred during the read operation, -1 is returned, and the error code is returned in the *oErrno* parameter.

`errNone`
: No error.

`iosErrCanceled`
: The operation was canceled.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrIOError`
: An I/O error occurred.

`iosErrNotOpened`
: The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
: The device does not support reading data.

`iosErrBadFD`
: The file descriptor is invalid or is not opened for reading.

IOSReadv Function

Purpose Reads data from a device into a scattered data buffer.

Declared In `IOS.h`

Prototype

```
int32_t IOSReadv( int32_t iFD,
    const struct iovec *iIovP, int32_t iIovCnt,
    status_t *oErrno )
```

Parameters

→ *`iFD`*
: The file descriptor of the device from which data should be read.

→ *`iIovP`*
: A pointer to an array of *`iIovCnt`* `iovec` structures indicating where the portions of the scattered data buffer are located.

→ *`iIovCnt`*
: The number of entries in the *`iIovP`*.

← *`oErrno`*
: On output, contains the result code indicating the error which occurred, or `errNone` if the data was read successfully.

Returns Returns the number of bytes actually read, which may be lower than the number of bytes requested (if, for example, the end of the available data is reached). If an error occurred during the read operation, -1 is returned, and the error code is returned in the *`oErrno`* parameter.

`errNone`
: No error.

`iosErrCanceled`
: The operation was canceled.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrIOError`
: An I/O error occurred.

`iosErrNotOpened`
: The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
: The device does not support reading data.

`iosErrBadFD`
: The file descriptor is invalid or is not opened for reading.

Comments

> **NOTE:** The scattered data buffer portions must all be in the same memory segment.

This function attempts to read data from the device specified by the *`iFD`* file descriptor into the scattered data buffer described by the `iovec` structures passed in the *`iIovP`* array. Each `iovec` entry specifies the base address and length of each portion of the scattered buffer. Each buffer will be filled before moving on to the next one.

IOSWrite Function

Purpose Writes data to a device.

Declared In `IOS.h`

Prototype

```
int32_t IOSWrite( int32_t iFD, MemPtr iBufP,
   int32_t iNbytes, status_t *oErrno )
```

Parameters

→ *`iFD`*
: The file descriptor of the device to which data should be written.

→ *`iBufP`*
: A pointer to the memory buffer from which data should be written.

→ *`iNbytes`*
: The number of bytes to write to the device.

← *`oErrno`*
: On output, contains the result code indicating the error which occurred, or `errNone` if the data was written successfully.

Returns Returns the number of bytes actually written. If an error occurred during the write operation, -1 is returned, and the error code is returned in the *`oErrno`* parameter.

`errNone`
No error.

`iosErrCanceled`
The operation was canceled.

`iosErrInvalidArg`
One of the parameters is invalid.

`iosErrIOError`
An I/O error occurred.

`iosErrNotOpened`
The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
The device does not support writing data.

`iosErrBadFD`
The file descriptor is invalid or is not opened for writing.

IOSWritev Function

Purpose Writes data to a device from a scattered data buffer.

Declared In `IOS.h`

Prototype

```
int32_t IOSWritev( int32_t iFD,
    const struct iovec *iIovP, int32_t iIovCnt,
    status_t *oErrno )
```

Parameters

→ *`iFD`*
The file descriptor of the device to which data should be written.

→ *`iIovP`*
A pointer to an array of *`iIovCnt`* `iovec` structures indicating where the portions of the scattered data buffer are located.

→ *`iIovCnt`*
The number of entries in the *`iIovP`*.

← *`oErrno`*
On output, contains the result code indicating the error which occurred, or `errNone` if the data was written successfully.

Returns Returns the number of bytes actually written. If an error occurred during the write operation, -1 is returned, and the error code is returned in the *`oErrno`* parameter.

`errNone`
: No error.

`iosErrCanceled`
: The operation was canceled.

`iosErrInvalidArg`
: One of the parameters is invalid.

`iosErrIOError`
: An I/O error occurred.

`iosErrNotOpened`
: The file descriptor does not correspond to an opened device.

`iosErrNotSupported`
: The device does not support writing data.

`iosErrBadFD`
: The file descriptor is invalid or is not opened for writing.

Comments **NOTE:** The scattered data buffer must all be in the same memory segment.

This function attempts to write data to the device specified by the *`iFD`* file descriptor from the scattered data buffer described by the **`iovec`** structures passed in the *`iIovP`* array. Each **`iovec`** entry specifies the base address and length of each portion of the scattered buffer. Each buffer will be completely written before moving on to the next one.

PbxAddFd Function

Purpose Adds a file descriptor to a PollBox.

Declared In `PollBox.h`

Prototype

```
status_t PbxAddFd( PollBox *pbx, int32_t fd,
   int16_t eventMask, PbxCallback *callback,
   void *context )
```

Parameters

→ *pbx*
: A pointer to the PollBox to which the file descriptor should be added.

→ *fd*
: The file descriptor to add to the PollBox.

→ *eventMask*
: A bitwise OR mask of the events to poll for.

→ *callback*
: A pointer to a callback handler that the PollBox will call when an event occurs on the file descriptor. You can specify `NULL` if your application does not require a callback.

→ *context*
: A value that will be passed into the callback handler.

Returns

`noErr`
: The file descriptor was added successfully.

`memErrNotEnoughSpace`
: Not enough memory to add the file descriptor to the PollBox.

See Also `PbxRemoveFd()`, "Adding File Descriptors to Monitor" on page 377

PbxCreate Function

Purpose Creates a new PollBox.

Declared In `PollBox.h`

Prototype `PollBox *PbxCreate( void )`

Returns Returns a pointer to the newly-created PollBox object. If an error occurred creating the PollBox, this function returns `NULL`.

See Also `PbxDestroy()`, "Creating a PollBox" on page 377

PbxDestroy Function

Purpose Destroys an existing PollBox.

Declared In `PollBox.h`

Prototype `void PbxDestroy( PollBox *pbx )`

Parameters → *pbx*
A pointer to the PollBox to destroy.

Comments `PbxDestroy()` closes all file descriptors in the PollBox and frees any memory allocated by the PollBox.

See Also `PbxCreate()`, "Destroying a PollBox" on page 377

PbxPoll Function

Purpose Polls all file descriptors in a PollBox for events that need processing.

Declared In `PollBox.h`

Prototype
```
status_t PbxPoll( PollBox *pbx, int32_t timeout,
   int32_t *nReady )
```

Parameters → *pbx*
The PollBox to poll.

→ *timeout*
The number of milliseconds to wait for an event to occur. Specify 0 to return immediately if there are no events pending, or -1 to wait indefinitely.

← *nReady*
The number of file descriptors with events pending, or `-1` if an error occurred.

Returns On return, *nReady* indicates the number of file descriptors that have events pending. Any of these events that have callbacks established have already had the callbacks run. If an error occurred during the poll operation, *nReady* is set to -1, and a non-zero result is returned.

Comments If there are no file descriptors in the PollBox, `PbxPoll()` sets *nReady* to 0 and returns zero.

Otherwise, this function blocks until one or more file descriptors in the box have events that were flagged as being of interest when they were added to the box, or until the timeout period expires. For each file descriptor that has events, the corresponding callback procedure (if one was specified when `PbxAddFd()` was called) is called, and the number of file descriptors with events is returned in *`nReady`*.

NOTE: If this function returns zero and sets *`nReady`* to zero, then there are either no file descriptors left in the PollBox or the timeout period expired. You can check which of these scenarios is the case by looking at the value of `pbx->count;` if this is nonzero, then the timeout expired.

See Also `PbxCreate()`, `PbxAddFd()`, "Polling for Events using a PollBox" on page 379

PbxRemoveFd Function

Purpose Removes a file descriptor from a PollBox.

Declared In `PollBox.h`

Prototype `void PbxRemoveFd( PollBox *pbx, int32_t fd )`

Parameters

→ *`pbx`*
The PollBox from which to remove the file descriptor

→ *`fd`*
The file descriptor to remove from the PollBox.

See Also `PbxAddFd()`, "Removing a File Descriptor from the PollBox" on page 378

PbxRun Function

Purpose Runs an event loop using a PollBox; this automatically calls `PbxPoll()` for you repeatedly until there are no more file descriptors in the box, or until an unexpected error occurs.

Declared In `PollBox.h`

Prototype `status_t PbxRun( PollBox *pbx )`

Parameters

→ *`pbx`*
The PollBox to use for the event loop.

Returns Returns 0 if the event loop terminated because there are no file descriptors left in the PollBox. If this value is non-zero, it is an error code indicating that something unexpected happened while processing the event loop.

Comments The event loop is run by calling `PbxPoll()` repeatedly with an infinite timeout. Your application can watch for user interface events by adding file descriptor 0 to the PollBox.

See Also `PbxPoll()`, `PbxAddFd()`, `PbxCreate()`, "Polling the Easy Way" on page 380

Application-Defined Functions

PbxCallback Function

Purpose Called by a PollBox when an event occurs on a file descriptor.

Declared In `PollBox.h`

Prototype
```
typedef void PbxCallback( struct PollBox *pbx,
   struct pollfd *pollFd, void *context )
```

Parameters

→ *pbx*
The PollBox that is calling into your callback routine.

→ *pollFd*
A `pollfd` structure indicating which file descriptor experienced an event, and which event or events occurred.

→ *context*
The context variable specified when your application called `PbxAddFd()`.

Comments Implement this function to handle the event or events that have occurred, as described by the *pollFd* structure.

20

Driver Attributes API

Applications can use the functions described in this section to query IOS about drivers. These C-language functions are are not meant for use by STREAMS drivers, but by the clients that depend upon them.

This chapter discusses the following topics:

Driver Attribute Constants

Driver Class Constants

Purpose Driver classes are used to classify groups of drivers. Not all drivers need to be classified. Certain classes of drivers, such as the drivers that need to work with the Serial Manager, should be classified. All driver classes are reserved creator codes.

Declared In `SDK/headers/IOSAttributes.h`

Table 20.1 Driver Classes

Class Name	Creator ID	Description
`iosDriverClassGeneric`	`'cgen'`	A generic class used for drivers that have an attributes block or a description but do not belong to a defined class.
`iosDriverClassSerial`	`'cser'`	Drivers that are designed to work with Serial Manager should use this class.
`iosDriverClassEthernet`	`'ceth'`	Drivers that are designed to support the Ethernet interface.
`iosDriverClassSlot`	`'cslt'`	Drivers that support expansion card slots.

Table 20.1 Driver Classes

Class Name	Creator ID	Description
`iosDriverClassVolume`	`'cvol'`	Volumes that support a specific file system implementation.
`iosDriverClassAdmin`	`'cadm'`	Drivers that are responsible for configuration or administration within IOS.
`iosDriverClassWifi`	`'wifi'`	Drivers that are associated with wireless (“Wi-Fi”) communication.
`iosDriverClassAll`	`'call'`	Drivers can not use `iosDriverClassAll` as their class ID. This class ID is used in the Driver Attributes API to get attributes or a description for all installed drivers.

Comments At this time, you cannot add additional classes. A driver may only belong to one class.

Driver Attribute Functions

Declared In `IOSAttributes.h`

IOSGetNumDrivers Function

Purpose Returns the number of drivers registered in IOS for the specified class. The class value `CLASS_ALL` is reserved to indicate all drivers.

Prototype
```
status_t IOSGetNumDrivers(uint32_t iClassID,
   uint16_t *oCount)
```

Parameters

→ *`iClassID`*
The class ID for the group of drivers you want to count. For the list of available class IDs, see “Driver Attribute Constants” on page 421.

← *`oCount`*
The number of drivers in the specified class.

Returns If the call is unsuccessful, this function returns an error code. Otherwise, it returns `errNone`.

IOSGetDriverAttributesByIndex Function

Purpose Returns the class-specific attributes block for a driver in the specified class at the given index. The class value `CLASS_ALL` is reserved to indicate all drivers.

Prototype

```
status_t IOSGetDriverAttributesByIndex(uint32_t
    iClassID, int16_t iIndex, MemPtr ioBuf,
    uint16_t* ioBufLen)
```

Parameters

→ *`iClassID`*
The driver's class ID. For the list of available class IDs, see "Driver Attribute Constants" on page 421.

→ *`iIndex`*
The index of the driver in the set of drivers contained in the class. Applications can get the number of drivers in the class (using `IOSGetNumDrivers()`) and loop over this function to get the attributes for each of the drivers in the class.

→ *`ioBuf`*
A buffer in the user's memory space. The attributes block will be copied into this space.

↔ *`ioBuflen`*
When calling this function, set this parameter to the number of bytes in the user buffer. Upon return, this parameter will be set to the length of the attributes block.

Returns

`errNone`
The operation completed successfully.

`iosErrNotEnoughSpace`
The buffer was too small to contain the attributes block. The length of the attribute block is in the *`ioBuflen`* parameter.

`iosErrDriverNotFound`
A matching driver could not be found at that class and index.

IOSGetDriverAttributesByName Function

Purpose Returns the class-specific attributes block for a driver, given its driver name.

Prototype
```
status_t IOSGetDriverAttributesByName(Char const
    * iIOSName, MemPtr ioBuf, uint16_t* ioBufLen)
```

Parameters

→ *iIOSName*
The name of a registered driver in IOS. A driver can be a device driver, a STREAMS module, or STREAMS driver.

WARNING! This function does not use partial name matching. The function will only return the attributes if *iIOSName* contains a complete match.

→ *ioBuf*
A buffer in the user's memory space. The attributes block will be copied into this space.

↔ *ioBuflen*
When calling this function, set this parameter to the number of bytes in the user buffer. Upon return, this parameter will be set to the length of the attributes block.

Returns

`errNone`
The operation completed successfully.

`iosErrNotEnoughSpace`
The buffer was too small to contain the attributes block. The length of the attribute block is in the *ioBuflen* parameter.

`iosErrDriverNotFound`
A matching driver could not be found for that driver name.

IOSGetDriverDescriptionByIndex Function

Purpose Returns the descriptive name for a driver in the specified class at the given index. The class value `CLASS_ALL` is reserved to indicate all drivers.

Prototype

```
status_t IOSGetDriverDescriptionByIndex(uint32_t
    iClassID, int16_t iIndex, Char* ioBuf,
    uint16_t* ioBufLen)
```

Parameters

→ *iClassID*
The driver's class ID. For the list of available class IDs, see "Driver Attribute Constants" on page 421.

→ *iIndex*
The index of the driver in the set of drivers contained in the class. Applications can get the number of drivers in the class (using `IOSGetNumDrivers()`) and loop over this function to get the description for each of the drivers in the class.

→ *ioBuf*
A buffer in the user's memory space. The descriptive name string will be copied into this space.

↔ *ioBuflen*
When calling this function, set this parameter to the number of bytes in the user buffer. Upon return, this parameter will be set to the length of the descriptive name string plus the `null` character.

Returns

`errNone`
The operation completed successfully.

`iosErrNotEnoughSpace`
The buffer was too small to contain the descriptive name string. The length of the descriptive name string + 1 is in the `ioBuflen` parameter.

`iosErrDriverNotFound`
A matching driver could not be found at that class and index.

Comments An installed driver may not have a descriptive name. In that case, the buffer will contain a zero-length string.

IOSGetDriverDescriptionByName Function

Purpose Returns the descriptive name string for a driver, given the driver's name in IOS.

Prototype

```
status_t IOSGetDriverDescriptionByName(Char const
    * iIOSName, Char* ioBuf, uint16_t* ioBufLen)
```

Parameters

→ *iIOSName*
The name of a registered driver in IOS. A driver can be a device driver, a STREAMS module, or STREAMS driver.

WARNING! This function does not use partial name matching. The function will only return the attributes if *iIOSName* contains a complete match.

→ *ioBuf*
A buffer in the user's memory space. The descriptive name string will be copied into this space.

↔ *ioBuflen*
When calling this function, set this parameter to the number of bytes in the user buffer. Upon return, this parameter will be set to the length of the descriptive name string plus the `null` character.

Returns

`errNone`
The operation completed successfully.

`iosErrNotEnoughSpace`
The buffer was too small to contain the descriptive name string. The length of the descriptive name string + 1 is in the `ioBuflen` parameter.

`iosErrDriverNotFound`
A matching driver could not be found at that driver name.

IOSGetDriverNameByIndex Function

Purpose Returns the driver name in IOS for a driver in the specified class, given the index. The class value `CLASS_ALL` is reserved to indicate all drivers.

Prototype

```
status_t IOSGetDriverDescriptionByIndex(uint32_t
    iClassID, int16_t iIndex, Char* ioBuf,
    uint16_t* ioBufLen)
```

→ *iClassID*
: The driver's class ID. For the list of available class IDs, see "Driver Attribute Constants" on page 421.

→ *iIndex*
: The index of the driver in the set of drivers contained in the class. Applications can get the number of drivers in the class (using `IOSGetNumDrivers()`) and loop over this function to get the description for each of the drivers in the class.

→ *ioBuf*
: A buffer in the user's memory space. The driver name string will be copied into this space.

↔ *ioBuflen*
: When calling this function, set this parameter to the number of bytes in the user buffer. Upon return, this parameter will be set to the length of the driver name string plus the `NULL`.

Returns

`errNone`
: The operation completed successfully.

`iosErrNotEnoughSpace`
: The buffer was too small to contain the descriptive name string. The length of the descriptive name string + 1 is in the `ioBuflen` parameter.

`iosErrDriverNotFound`
: A matching driver could not be found at that class and index.

21

Driver Installation API

All drivers are installed into the I/O Process using the I/O Subsystem's Driver Installation API. A driver's installation software must make use of the installation functions provided by this API.

This chapter discusses the following topics:

IOS Installation Functions

IOSInstallDriver Function

Purpose Installs a driver into the I/O Process.

Declared In `SDK/headers/IOSInstall.h`

Prototype

```
status_t IOSInstallDriver(uint32_t typeId,
    uint32_t creatorId, uint32_t resourceID)
```

Parameters

→ *typeId*
The type for the driver PRC. For example, `'mydr'`.

NOTE: The Device Loader will automatically install drivers of type `'drvr'`. Drivers of this type will not call `IOSInstallDriver()` explicitly. For drivers with a different database type, the program that installs the driver should call this function.

→ *creatorId*
The creator ID for the driver PRC. For example, `'MYDR'`.

→ *resourceID*
The resource ID of the driver to be installed. This is the ID that was assigned to the driver when the PRC was created. This ID should be the same as that declared in the PRC's SLD file.

Returns `iosErrDriverNotFound`
The PRC was not found (or no driver with those attributes was found in the PRC).

`iosErrInvalidArg`
The name of the driver is missing (or is not a string).

`iosErrAuthFailed`
The driver was not signed in the manner required by Palm OS.

Comments If the device's security policy requires a signature, Palm OS will verify the signature of the driver when this function is called. For more information about signing your drivers, see *Exploring Palm OS: Security and Cryptography*.

IOSRemoveDriver Function

Purpose Requests the removal of a driver from the I/O Process. Note that a driver with an active session cannot be removed.

Declared In `SDK/headers/IOSInstall.h`

Prototype
```
Status_t IOSRemoveDriver(uint32_t type, uint32_t
    creator, uint32_t resourceID)
```

Parameters → *typeId*
The type for the driver PRC. For example, `'drvr'`.

→ *creatorId*
The creator ID for the driver PRC. For example, `'MYDR'`.

→ *resourceID*
The resource ID of the driver to be removed. This is the ID that was assigned to the driver when the PRC was created. This ID should be the same as that declared in the PRC's SLD file.

Returns `iosErrDriverNotFound`
The PRC was not found (or no driver with those attributes was found in the PRC).

`iosErrInvalidArg`
The name of the driver is missing (or is not a string).

`iosErrDeviceInUse`
The driver has an active session and cannot be removed at this time.

Comments Calling `IOSRemoveDriver()` only removes the driver from the I/O Process—it does not delete the driver's PRC. After the driver has been removed, the uninstallation software may delete the PRC using `DmDeleteDatabase()`.

WARNING! A driver that has an active session cannot be removed. All active sessions with the driver must be closed before you can request removal of the driver.

Glossary

access point A WiFi station that provides access to other networks.

ad-hoc network A WiFi network consisting of multiple devices without a dedicated access point.

ATIM Announcement Traffic Indication Message. These messages are used to coordinate transmission times between the various stations that comprise an ad-hoc WiFi network.

BSS Basic Service Set. The service area of a single access point in a WiFi network.

BSSID The BSS ID is the network identifier for a single access point or ad-hoc network.

ESS Extended Service Set. The service area of a network of access points.

ESSID The ESS ID is the text string identifying a network of access points.

IEEE 802.3 The IEEE standard for wired Ethernet

IEEE 802.11 The IEEE standard for wireless Ethernet. There are several substandards that cover different transmission media, differences in RF spectrum allocation, and security.

MAC address A 48-bit identifier that uniquely identifies a device on a network.

PDU See Protocol Data Unit.

PSM See Protocol Service Multiplexer.

Protocol Data Unit The PDU is a unit of data exchanged between two protocol peers.

Protocol Service Multiplexer The PSM is the L2CAP equivalent of a TCP port number; it identifies an individual L2CAP channel.

SSID A 1-32 character string identifying an 802.11 network.

station Any node in an 802.11 network. A station can be an access point or a client of an access point.

WEP Wired Equivalent Privacy. The most common authentication and encryption algorithm for 802.11.

WPA WiFi Protected Access. A new authentication and encryption framework for 802.11 that corrects the deficiencies found in WEP.

Index

L

M

N

O

P

R

S

T

U

V

W

X