# Using Bluetooth

In this section, you'll learn how to interact directly with Bluetooth devices including other phones and Bluetooth headsets. Using Bluetooth, you can pair with other devices within range, initiate an RFCOMMSocket, and transmit and receive streams of data from or for your applications.

**The Bluetooth libraries have been removed for the Android version 1.0 release. The following sections are based on earlier SDK releases and are included as a guide to functionality that is expected to be made available in subsequent releases.**

## *Introducing the Bluetooth Service*

The Android Bluetooth service is represented by the BluetoothDevice class.

Bluetooth is a system service accessed using the getSystemService method. Get a reference to the BluetoothDevice by passing in the Context.BLUETOOTH constant as the service name parameter, as shown in the following code snippet:

```
String context = Context.BLUETOOTH_SERVICE;

final BluetoothDevice bluetooth = (BluetoothDevice)getSystemService(context);
```

To use the Bluetooth Service, your application needs to have the BLUETOOTH permission as shown here:
<uses-permission android:name="android.permission.BLUETOOTH"/>

## *Controlling the Local Bluetooth Device*

The Bluetooth Device offers several methods that let you control the Bluetooth hardware. The enable and disable methods let you enable or disable the Bluetooth adapter. The getName and setName methods let you modify the local device name, and getAddress can be used to determine the local device address. You can fi nd and change the discovery mode and discovery time-out settings using the getMode and getDiscoverableTimeout methods and their setter equivalents.

The following code snippet enables the Bluetooth adapter and waits until it has connected before changing the device name and setting the mode to "discoverable":

```
bluetooth.enable(new IBluetoothDeviceCallback.Stub() {
public void onCreateBondingResult(String _address, int _result)
throws RemoteException
{
String friendlyName = bluetooth.getRemoteName(_address);
}
public void onEnableResult(int _result) throws RemoteException {
if (_result == BluetoothDevice.RESULT_SUCCESS) {
bluetooth.setName("BLACKFANG");
bluetooth.setMode(BluetoothDevice.MODE_DISCOVERABLE);
}
}
});
```

## *Discovering and Bonding with Bluetooth Devices*

Before you can establish a data communications socket, the local Bluetooth device must fi rst discover, connect, and bond with the remote device.

### *Discovery*

Looking for remote devices to connect to is called *discovery*. For other devices to discover your handset, you need to set the mode to "discoverable" or "connectable" using the setMode method as shown previously.

To discover other devices, initiate a discovery session using the startDiscovery or startPeriodicDiscovery methods, as shown below:

```
if (discoverPeriodically)
bluetooth.startPeriodicDiscovery();
else
bluetooth.startDiscovery(true);
```

Both of these calls are asynchronous, broadcasting a REMOTE_DEVICE_FOUND_ACTION whenever a new remote Bluetooth device is discovered.

To get a list of the remote devices that have been discovered, call listRemoteDevices on the Bluetooth device object. The returned String array contains the address of each remote device found. You can fi nd their "friendly" names by passing in the device address to getRemoteName.

## *Bonding*

*Bonding*, also known as *pairing*, lets you create an authenticated connection between two Bluetooth devices using a four-digit PIN. This ensures that your Bluetooth connections aren't hijacked. Android requires you to bond with remote devices before you can establish application-layer communication sessions such as RFCOMM.

To bond with a remote device, call the createBonding method on the Bluetooth Device after using setPin to set the unique identifi er PIN to use for this pairing request. To abort the bonding attempt, call cancelBonding and use cancelPin to re-set the PIN if required.

Once a remote device has been bonded, it will be added to the native database and will automatically bond with the local device if it is discovered in the future.

In the following code snippet, the Bluetooth Service is queried for a list of all the available remote devices. The list is then checked to see if any of these devices are not yet bonded with the local Bluetooth service, in which case, pairing is initiated.

```
String[] devices = bluetooth.listRemoteDevices();
for (String device : devices) {
if (!bluetooth.hasBonding(device)) {
// Set the pairing PIN. In real life it's probably a smart
// move to make this user enterable and dynamic.
bluetooth.setPin(device, new byte[] {1,2,1,2});
bluetooth.createBonding(device, new IBluetoothDeviceCallback.Stub() {
public void onCreateBondingResult(String _address, int _result)
throws RemoteException {
if (_result == BluetoothDevice.RESULT_SUCCESS) {
String connectText = "Connected to " + bluetooth.getRemoteName(_address);
Toast.makeText(getApplicationContext(), connectText, Toast.LENGTH_SHORT);
}

}
public void onEnableResult(int _result) throws RemoteException {}
});
}
}
```

To listen for remote-device bonding requests, implement and register a BroadcastListener that fi lters for the ACTION_PAIRING_REQUEST Intent.

# *Managing Bluetooth Connections*

Calling listRemoteDevices returns a list of the currently discovered devices, while listBondings returns the address of each remote device currently bonded to the local device. As a shortcut, hasBonding lets you specify a device address and returns true if you have bonded with it.

Further details on each device can be found using the lastSeen and lastUsed methods. These methods return the last time a device was seen (through discovery) or accessed.

Use removeBonding to sever a bond with a remote device. This will also close any application layer communications sockets you've established.

# Communication with Bluetooth

The most likely reason for bonding to a remote Bluetooth device is to communicate with it. Bluetooth data transfer is handled using the RfcommSocket class, which provides a wrapper for the Bluetooth radiofrequency communications (RFCOMM) protocol that supports RS232 serial communication over an underlying Logical Link Control and Adaptation Protocol (L2CAP) layer.

In practice, this alphabet soup provides a mechanism for opening communication sockets between two paired Bluetooth devices.

In order for an RFCOMM communication channel to be established, a listening port on one device must be connected to an outgoing port on the other. As a result, for bidirectional communication, two socket connections must be established.

## Opening a Socket Connection

Before you can transfer data between Bluetooth devices, you need to open a new RFCommSocket. Start by creating a new RFCommSocket object and calling its create method. This constructs a new socket for you to use on your Bluetooth device, returning a FileDescriptor for transferring data.

The FileDescriptor is the lowest-level representation of an I/O source. You can create any of the I/O class objects (FileStream, DataOutputStream, etc.) using a FileDescriptor as a constructor parameter. Later you'll use one of these I/O classes to transfer data with a remote device.

The following skeleton code shows the basic RFCommSocket implementation that creates a new socket connection ready to either initiate or respond to communications requests:

```
FileDescriptor localFile;
RfcommSocket localSocket = new RfcommSocket();
try {
localFile = localSocket.create();
} catch (IOException e) { }
```

Once the socket has been created, you need to either bind it to the local device to listen for connection requests or initiate a connection with a remote device.

## Listening for Data

To listen for incoming data, use the bind method to create a socket to use as a listening port for the local device. If this is successful, use the listen method to open the socket to start listening for incoming data transfer requests.

Once you've initialized your listener socket, use the accept method to check for, and respond to, any incoming socket connection requests.

The accept method takes a new RfcommSocket object that will be used to represent the remote socket connection, and a time-out for a connection request to be received. If a successful connection is made, accept returns a FileDescriptor that represents the input stream. Use this File Descriptor to process the incoming data stream from the remote device.

The following skeleton code shows how to confi gure a new socket that listens for, and accepts, an incoming socket connection:

```
FileDescriptor localFile;
FileDescriptor remoteFile;
RfcommSocket localSocket = new RfcommSocket();
try {
localFile = localSocket.create();
localSocket.bind(null);
localSocket.listen(1);
RfcommSocket remotesocket = new RfcommSocket();
remoteFile = localSocket.accept(remotesocket, 10000);
}
catch (IOException e) { }
```

If no connection request is made within the time-out period, accept returns null.

### *Transmitting Data*

To transmit data using an RfcommSocket, use the connect method to specify a bonded remote device address and port number to transmit data to. The connection request can also be made asynchronously using the connectAsync method to initiate the connection; waitForAsyncConnect can then be used to block on a separate thread until a response is received.

Once a connection has been established, you can transmit data with any of the I/O output classes using the local socket's FileDescriptor as a constructor parameter, as shown in the following code snippet:

```
FileDescriptor localFile;
String remoteAddress = bluetooth.listBondings()[0];
RfcommSocket localSocket = new RfcommSocket();
try {
localFile = localSocket.create();
// Select an unused port
if (localSocket.connect(remoteAddress, 0)) {
FileWriter output = new FileWriter(localFile);
output.write("Hello, Android");
output.close();
}
} catch (IOException e) { }
```

## *Using a Bluetooth Headset*

Wireless headsets are one of the most common uses of Bluetooth on mobile phones. The BluetoothHeadset class provides specialized support for interacting with Bluetooth headsets. In this context, a headset includes any headset or hands-free device.

To use the Bluetooth headset API, create a new BluetoothHeadset object on your application context,
as shown in the code snippet below:

```
BluetoothHeadset headset = new BluetoothHeadset(this);
```

This object will act as a proxy to the Bluetooth Headset Service that services any Bluetooth headsets bonded with the system.

Android only supports a single headset connection at a time, but you can change the connected headset using this API. Call connectHeadset, passing in the address of the headset to connect to, as shown in the code snippet below:

```
    headset.connectHeadset(address, new IBluetoothHeadsetCallback.Stub() {
    public void onConnectHeadsetResult(String _address, int _resultCode)
    throws RemoteException {
    if (_resultCode == BluetoothHeadset.RESULT_SUCCESS) {
    // Connected to a new headset device.
    }
    }
    });
```

The Headset Service is not guaranteed to be connected to a headset at all times, so it's good practice to use the getState method to confi rm a valid connection before performing any actions, as shown in the code snippet below:

```
if (headset.getState() == BluetoothHeadset.STATE_CONNECTED) {
// TODO Perform actions on headset.
}
```

When you've fi nished interacting with the headset, you should always call close on the BluetoothHeadset object to let the proxy unbind from the underlying service:

```
BluetoothHeadset headset = new BluetoothHeadset(this);
// [ … Perform headset actions … ]
headset.close();
```