

## ***Using Bluetooth on Motorola Handsets***

June 1, 2006



TECHNICAL ARTICLE

---

## *Using Bluetooth on Motorola Handsets*

By  
MOTODEV Staff

---

What technology got its name from a king who introduced Christianity to Denmark and is used to communicate wirelessly between devices? The answer to that question is Bluetooth! What makes this technology so significant and exciting is that it provides several key advantages over older technology such as infrared. For example, with infrared devices, such as remote controls for televisions, it is necessary to line them up in order for communication to take place. Bluetooth eliminates this restriction and can communicate like a radio in a multi-directional fashion. Infrared devices also have very poor range and can only communicate within a few feet. Bluetooth corrects this issue and can communicate up to 30 feet (10 meters).

Due to the great advantages of Bluetooth, developers nowadays wish to implement this feature to create more engaging applications. The goal of this article is to provide a “quick-start” overview of how to use the most important features of JSR-82 (the Bluetooth API) with Motorola phones so it is not too difficult to get started. This article describes the basics of device and service discovery, client/server connections, some tips and concludes with sample code of a very simple Bluetooth client/server application.

---

### ***Device Discovery***

When it comes to Bluetooth, the first thing a developer is likely to want to know is how to discover other devices. Fortunately for developers, this is not too difficult. The classes that are required to accomplish this are:

- `javax.bluetooth.LocalDevice`
- `javax.bluetooth.DiscoveryAgent`
- `javax.bluetooth.DiscoveryListener`

The `LocalDevice` class provides access to the bluetooth device and can be used to create an instance of a `DiscoveryAgent` which performs the inquiry. The `DiscoveryListener` then listens for the event that a device has been discovered.

The following code illustrates how to perform device discovery:

```
LocalDevice localDevice = LocalDevice.getLocalDevice();  
discoveryAgent = localDevice.getDiscoveryAgent();  
discoveryAgent.startInquiry(accessCode, aDiscoveryListener)
```

The `discoveryAgent.startInquiry(int accessCode, DiscoveryListener listener)` method initiates the inquiry. It is a non-blocking method and will search for all the devices based on the inquiry access mode specified by the `accessCode` argument.

The types of access modes are as follows:

- General/Unlimited Inquiry Access (GIAC): allows the device to be discoverable for an indefinite amount of time.
- Limited Dedicated Inquiry Access (LIAC): allows the device to be discoverable for only a limited amount of time.
- NOT\_DISCOVERABLE: disables discoverable mode.

The following illustrates how each of the inquiries work:

- The `DiscoveryAgent.GIAC` inquiry finds all devices that are in either the GIAC or LIAC mode.
- The `DiscoveryAgent.LIAC` inquiry finds all devices that are just in the LIAC mode.
- The `DiscoveryAgent.NOT_DISCOVERABLE` access mode does not allow the device to be discovered.

After initiating the inquiry, the `DiscoveryListener` used as an argument in the `discoveryAgent.startInquiry()` method will “listen” for device discovery events. When a device is found, the `deviceDiscovered()` method is triggered:

```
public void deviceDiscovered(RemoteDevice remoteBTDevice, DeviceClass devClass)
```

With this method a reference to the remote device is provided which can be used to find information about it such as its services.

When the inquiry is completed the `inquiryCompleted()` method is triggered:

```
public void inquiryCompleted(int discType)
```

As illustrated above, device discovery is quite easy and requires only a few lines of code. The next topic that will be discussed will be about service discovery.

---

## ***Service Discovery***

After discovering devices, it is logical to want to know what services are available on them. Like device discovery, service discovery requires the same `javax.bluetooth.DiscoveryAgent` class.

There are 2 methods used to find services using the `DiscoveryAgent`, they are `searchServices()` and `selectService()`. The main difference is that `searchServices()` searches for services in one device, whereas `selectService()` searches for services in all the devices in the area.

The `searchServices()` method is typically used to search for services after going through the process of discovering the desired remote device (please refer to the previous section on Device Discovery). The full signature of this method is as follows:

```
public int searchServices(int[] attrSet, UUID[] uuidSet,
RemoteDevice btDev, DiscoveryListener discListener)
```

A description of the parameters of the method is as follows:

`attrSet` – specifies an array of integers that correspond to the desired attributes of the service records found

`uuidSet` – specifies an array of UUID which correspond to the desired services that are being searched. UUID stands for Universal Unique Identifier and uniquely identifies services in the bluetooth protocol. For example the UUID for the RFCOMM (virtual serial port) service is 0x0003.

`RemoteDevice` – specifies the device in which to search services on

`discListener` – the listener that will listen for events when services are discovered

Similarly to device discovery a method is called when a service discovery is made.

This method is known as `servicesDiscovered()`, and its signature is as follows:

```
public void servicesDiscovered(int transID,
ServiceRecord[] servRecord)
```

An array of `ServiceRecord` objects are provided in the method which represents the services found. After finding the desired service in the array, it is possible to connect to it by retrieving the connection URL using the `ServiceRecord.getConnectionURL()` method. Connecting to services will be described in more detail in the next section on client/server connections.

The `selectService()` method does not require discovering the desired remote device. It automatically searches all devices in the area for a particular service. Once it has found the suitable service, a connection string is returned which can be used to connect to the service. The signature of the method is as follows:

```
public java.lang.String selectService(UUID uuid, int security,
boolean master)
```

The next section describes the client/server connection. The use of the connection string returned from `ServiceRecord.getConnectionURL()` and `selectService()` will be discussed in more detail.

---

## Client/Server Connections

Many developers are interested in creating a client/server type application. An example would be a chat application or a chess game in which a client requests services from a server. Bluetooth communication works under this client/server model. The following section will describe how to create a client/server connection and transmit data through RFCOMM (the serial port profile).

Implementing this model requires 2 parts, first setting up the server to listen for client connections and second having clients connect to the server. The approach used to create Bluetooth connections are very similar to other connections that fall under the Generic Connection Framework (GCF).

Setting up a Bluetooth server so that it can listen for client connections requires only a few steps. The code is as follows:

```
StreamConnectionNotifier notifier =  
(StreamConnectionNotifier)Connector.open("btspp://localhost:" + customUUID);  
StreamConnection con = StreamConnection(notifier.acceptAndOpen());
```

The `Connector.open()` call creates a service record for the service and accepts a connection URL as an argument. The connection URL contains "Btspp://localhost:" in the beginning to indicate that the Bluetooth Serial Port Profile (RFCOMM) is to be used. Also part of the URL is a custom made UUID which is composed of any string that is 128 bits long. An example is "3c8b24e0eac144a5ad92800555c9a66". UUID generator software tools are available on the internet to create unique UUIDs for this purpose. The `acceptAndOpen()` method registers the service and allows Bluetooth clients to connect to it. This method is a blocking call and will not return until a client connection occurs.

On the client side, all that is needed to connect to the server is 1 line of code:

```
StreamConnection conn = (StreamConnection)Connector.open("btspp://" +  
serverAddress + ":" + channelNumber);
```

As with the server, `Connector.open()` accepts a connection URL as an argument. The connection URL needs to specify the protocol with "btspp:" in the beginning of the connection string, however the difference is that it will not have "localhost" in it. The connection URL can be generated through the `ServiceRecord.getConnectionURL()` or the `DiscoveryAgent.selectService()` call as mentioned in the previous section.

For example with `DiscoveryAgent.selectService()` the connection URL can be extracted and used by the client to connect to the server's service. The following code illustrates this:

```
String connectionURL = discoveryAgent.selectService(new UUID(myUUID,
false), ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);

StreamConnection conn =
StreamConnectionConnector.open(connectionURL);
```

It is evident that the code for the client and server side are very similar – both use the `Connector.open()` method. The main difference being that the client uses a `StreamConnection` and the server uses a `StreamConnectionNotifier`. After a connection is established, input and output streams may be used to transmit data.

---

## ***Important Tips***

The following is a list of issues that might be helpful to be aware of when developing with JSR82 on Motorola phones:

- Depending on how Bluetooth is implemented it is sometimes necessary to include code to initialize the Bluetooth stack for preparation for communication. With Motorola phones this step is not required
- A Bluetooth client/server connection disconnects 8 seconds after the MIDlet is suspended, which happens in the case of an interruption by a phone call or if the END key is pressed
- There can only be 1 Bluetooth client/server connection at one time
- If the `RemoteDevice.getFriendlyName()` method is called in the `DiscoveryListener.deviceDiscovered()` method not all the devices may be found. `RemoteDevice.getFriendlyName()` is a time consuming method and could cause the device inquiry to timeout before finding all the devices. Therefore it is necessary to create a separate thread that calls `RemoteDevice.getFriendlyName()` in this case
- Depending on the carrier, the MIDlet may need to be signed in order to use JSR-82

---

## ***Sample Application***

Sometimes the easiest way to illustrate a concept is by example. This section provides sample code for a very simple client/server application. The application allows a client to connect to a server and send the text "Hello from client" to it.

Instructions on how to use it are as follows:

- 1) Install the MIDlet on two Motorola phones (the MIDlet contains code for both the server and client)
- 2) On the phone which will act as the server, select “Server” by pressing the left softkey to start the server (you may receive a prompt asking for permission to make a local connection, proceed and grant permission)
- 3) On the phone which will act as the client, select “Client” by pressing the right softkey to try to connect to the server (this might take a while) and send the text “Hello from the client”

The application also provides the ability to search for devices and also make the phone discoverable. These features are available by pressing the MENU key.

The following is the code for the application:

```
import javax.microedition.midlet.*;
import javax.bluetooth.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;

public class BluetoothTest extends MIDlet implements DiscoveryListener, CommandListener {

    private Command searchDevicesCommand;
    private Command discoverableCommand;
    private Command clientConnectToServerCommand;
    private Command serverCommand;

    private boolean isDiscoverable;

    // a custom UUID that is made up
    private String myUUID = "3c8b24e00eac144a5ad92800555c9a66";

    private LocalDevice localDevice;
    private DiscoveryAgent discoveryAgent;
    private int numDevicesFound;
    private Displayable theDisplay;
    private TextBox tb;

    public BluetoothTest() {
        super();
    }
}
```

```

protected void startApp() throws MIDletStateChangeException {
    try {

        tb = new TextBox("Bluetooth Test", "", 5000, TextField.ANY);
        theDisplay = tb;

        clientConnectToServerCommand = new Command("Client", Command.OK, 0);
        serverCommand = new Command("Server", Command.OK, 0);
        searchDevicesCommand = new Command("Discover Devices", Command.OK, 0);
        discoverableCommand = new Command("Make Discoverable", Command.OK, 0);

        theDisplay.addCommand(clientConnectToServerCommand);
        theDisplay.addCommand(serverCommand);
        theDisplay.addCommand(searchDevicesCommand);
        theDisplay.addCommand(discoverableCommand);

        theDisplay.setCommandListener(this);

        Display.getDisplay(this).setCurrent(theDisplay);

        localDevice = LocalDevice.getLocalDevice();
        discoveryAgent = localDevice.getDiscoveryAgent();

    }
    catch(Throwable t) {
        t.printStackTrace();
    }
}

protected void pauseApp() {}

protected void destroyApp(boolean arg0) throws MIDletStateChangeException {}

/*****
 * deviceDiscovered is called whenever a device is found after
 * an inquiry by a DiscoveryAgent
 *****/
public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass arg1) {
    try {
        ++numDevicesFound;
    }
}

```



```

        printToScreen("* # of devices found: " +
String.valueOf(numDevicesFound));

        // retrieve the friendly name of the device:
        // this is done in a separate thread because the
        // RemoteDevice.getFriendlyName method can be
        // time consuming and the DiscoveryAgent inquiry could
        // time out before the friendly names are retrieved
        new Thread(new getFriendlyNameThread(remoteDevice, this)).start();

    }

    catch(Throwable t) {
        t.printStackTrace();
    }

}

public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {}

public void serviceSearchCompleted(int transID, int respCode) {}

/*****
* inquiryCompleted is called when the inquiry is completed i.e.
* when it is done searching for devices
*****/
public void inquiryCompleted(int arg0) {
    printToScreen("***Inquiry Complete***");
}

public void commandAction(Command arg0, Displayable arg1) {

    /*****
    * demonstrates how to discover devices
    *****/

    if (arg0 == searchDevicesCommand) {
        try {
            numDevicesFound = 0;
            if ((discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this) ==
true)) {

                printToScreen("* StartInquiry");
            }
            else {
                printToScreen("* Failed Starting Inquiry");
            }
        }
    }
}

```

```

    }

    catch (Throwable t) {
        t.printStackTrace();
    }
}

/*****
 * demonstrates how to set the device to discoverable mode
 *****/
if (arg0 == discoverableCommand) {
    printToScreen("Set Device: GIAC");
    try {
        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
        retrieveDiscoveryMode(localDevice);
    } catch (BluetoothStateException e) {
        e.printStackTrace();
    }
}

/*****
 * demonstrates how to set up an RFCOMM client connection to a server
 * and sends text to this server
 *****/
if (arg0 == clientConnectToServerCommand) {
    DiscoveryAgent discoveryAgent = localDevice.getDiscoveryAgent();

    try {
        String connectionURL = discoveryAgent.selectService(new
UUID(myUUID, false), ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);

        if(connectionURL != null) {
            try {
                StreamConnection conn =
(StreamConnection)Connector.open(connectionURL);
                if (conn == null){
                    printToScreen("conn == null");
                }

                DataOutputStream out = conn.openDataOutputStream();
                String clientMessage = "Hello from client";
                out.writeUTF(clientMessage);
                out.flush();
            }

```

```

        + "\"");

        printToScreen("* Send to server: \"" + clientMessage

        } catch (IOException e) {
            e.printStackTrace();
            printToScreen(e.getMessage());
        }
    } else {
        printToScreen("unable to connect to service");
    }

    } catch (BluetoothStateException e) {
        e.printStackTrace();
    }
}

/*****
 * demonstrates how to set up an RFCOMM server connection
 * and receive text from a client
 *****/
if (arg0 == serverCommand) {
    try {

        // make sure the server is discoverable so clients can connect
        isDiscoverable = localDevice.setDiscoverable(DiscoveryAgent.GIAC);
        if (isDiscoverable == false) {
            printToScreen("Current Device Not Discoverable");
        }

        // create server
        StreamConnectionNotifier notifier =
(StreamConnectionNotifier)Connector.open("btspp://localhost:" + myUUID);

        // open client connection
        printToScreen("* Open Client Connection");
        StreamConnection con = (StreamConnection)notifier.acceptAndOpen();

        // read in the data
        DataInputStream in = con.openDataInputStream();

        printToScreen("* Received from Client: \"" + in.readUTF() + "\"");

```

```

        printToScreen("* Close Connections");
        in.close();
        con.close();
        notifier.close();

    } catch (BluetoothStateException e) {
        e.printStackTrace();
        printToScreen(e.getMessage());
    } catch (IOException e) {
        e.printStackTrace();
        printToScreen(e.getMessage());
    }
}

/*****
 * Retrieves the discovery mode the device is currently in
 *****/
public void retrieveDiscoveryMode(LocalDevice localDevice) {
    int discoveryMode;

    printToScreen("* Discovery Mode:");

    discoveryMode = localDevice.getDiscoverable();

    switch (discoveryMode) {
        case DiscoveryAgent.GIAC:
            printToScreen("GIAC");
            break;
        case DiscoveryAgent.LIAC:
            printToScreen("LIAC");
            break;
        case DiscoveryAgent.NOT_DISCOVERABLE:
            printToScreen("NOT_DISCOVERABLE");
            break;
        default:
            printToScreen(Integer.toString(discoveryMode));
    }
}

/*****
 * prints text to the cell phone screen and also to System.out.println
 *****/

```

```

    * which shows up in the Motorola Midway Debug Log
    *****/

public void printToScreen(String message) {
    tb.insert(message + "\n", tb.size());
    // System.out.println outputs to Motorola Midway Tool debug log
    System.out.println(message);
}

}

/*****
 * a separate thread is created for retrieving the friendlyName of a
 * remote device since this is a time consuming task
 *****/

class getFriendlyNameThread implements Runnable {

    RemoteDevice remoteDevice;
    BluetoothTest b;
    String friendlyName;

    getFriendlyNameThread(RemoteDevice rDevice, BluetoothTest blueToothTest) {
        remoteDevice = rDevice;
        b = blueToothTest;
    }

    public void run() {
        b.printToScreen("* Begin to get friendly name of remote device");
        try
        {
            friendlyName = remoteDevice.getFriendlyName(true);
            b.printToScreen("* Remote device friendly name: " + "\"" + friendlyName +
"\");
        }
        catch(Exception ioe)
        {
            System.out.println("* Ex: when getFriendlyName: "+ioe);
        }
    }

}

```

---

## **Conclusion**

Bluetooth can be an overwhelming topic and this article aims to simplify the topic by providing the basics so it is easier to get started. The article began with a description of device and service discovery. It was shown that discovering devices and services are done in a similar way both using the `DiscoveryAgent` and `DiscoveryListener` classes. Then it went over how to make Bluetooth connections, which are created similarly to other connections that fall under the Generic Connection Framework (GCF). The article proceeded with explaining some tips and concluded with a sample Bluetooth application.

---

## **References**

- [1] Hopkins, B., & Antony, R. (2003). *Bluetooth for Java*. Berkeley, CA: Apress
- [2] Kumar, C., & Kline, P., & Thompson, T. (2004). *Bluetooth Application Programming With The Java APIs*. San Francisco, CA: Morgan Kaufmann Publishers
- [3] Ortiz, E. (2005, February). *Using the Java APIs for Bluetooth, Part 2 – Putting the Core APIs to Work*. Retrieved May 1, 2006, from <http://developers.sun.com/techtopics/mobility/apis/articles/bluetoothcore/index.html>
- [4] Sony Ericsson. (2004, January). *Developing Applications with the Java APIs for Bluetooth (JSR-82)*. Retrieved May 17, 2006 from <http://www.microjava.com/articles/Bluetooth-jsr-82-training.pdf>
- [5] CLDC1.1 (JSR-139)
- [6] Bluetooth API (JSR-82)