Article

Part II: The Java APIs for Bluetooth Wireless Technology

by Qusay H. Mahmoud
April 2003

The Java 2 Platform, Micro Edition (J2ME) and Bluetooth technology are two of the most exciting offerings in the wireless industry today. J2ME, most compact of the three Java platforms, is inherently portable because it shares the Java "write once run anywhere" philosophy and thus enhances developer productivity. Bluetooth is a short-range universal wireless connectivity standard for electronic appliances and mobile devices.

Imagine being able to use your Bluetooth-enabled mobile phone to lock and unlock your car, operate your garage door, and control your TV, VCR, DVD player, and other consumer appliances. If you want to make that kind of control available to your users, you'll need to be able to write Bluetooth applications that customize these appliances, and deploy them in a way that lets users download them, to a cell phone for example. Bluetooth and J2ME can work together to achieve this unified vision. Bluetooth allows devices to communicate wirelessly and J2ME allows you to write custom applications and deploy them on mobile devices.

Part I of this series, "Getting Started with Bluetooth," was a tutorial that covered the basics of Bluetooth, the Bluetooth protocol stack, and procedures for establishing Bluetooth connections. This article focuses on programming wireless applications using Bluetooth and J2ME. It provides a brief overview of the Java APIs for Bluetooth Wireless Technology (JSR 82), and shows you how to use these APIs.

Java APIs for Bluetooth Wireless Technology

While Bluetooth hardware has advanced, there has been no standardized way to develop Bluetooth applications - until JSR 82 came into play. It is the first open, non-proprietary standard for developing Bluetooth applications using the Java programming language. It hides the complexity of the Bluetooth protocol stack behind a set of Java APIs that allow you to focus on application development rather than the low-level details of Bluetooth. JSR 82 is based on version 1.1 of the Bluetooth Specification.

Like all JSRs, the Java APIs for Bluetooth are being developed through the Java Community Process. Its expert group has members representing 20 companies. The final specification is available for download.

JSR 82 consists of two optional packages: the core Bluetooth API and the Object Exchange (OBEX) API. The latter is transport-independent and can be used without the former.

Note: The Java APIs for Bluetooth do not implement the Bluetooth specification, but rather provide a set of APIs to access and control a Bluetooth-enabled device. JSR 82 concerns itself primarily with providing Bluetooth capabilities to J2ME-enabled devices.

The Java APIs for Bluetooth target devices with the following characteristics:

- 512K minimum of total memory available (ROM and RAM) (application memory requirements are additional)
- Bluetooth wireless network connection
- Compliant implementation of the J2ME Connected Limited Device Configuration (CLDC)

Bluetooth System Requirements

The underlying Bluetooth system upon which the Java APIs will be built must also meet certain requirements:

- The underlying system must be "qualified," in accordance with the Bluetooth Qualification Program, for at least the Generic Access Profile, Service Discovery Application Profile, and Serial Port Profile.
- The system must support three communication layers or protocols as defined in the 1.1 Bluetooth Specification, and the implementation of this API must have access to them: Service Discovery Protocol (SDP), Radio Frequency Communications Protocol (RFCOMM), and Logical Link Control and Adaptation Protocol (L2CAP).
- The system must provide a Bluetooth Control Center (BCC), a control panel much like the application that allows a user or OEM to define specific values for certain configuration parameters in a stack.

OBEX support can be provided in the underlying Bluetooth system or by the implementation of the API. The OBEX protocol provides support for object exchanges, and forms the basis for Bluetooth profiles such as the Synchronization Profile and the File Transfer Profile.

What Is the BCC?

Bluetooth devices that implement this API may allow multiple applications to execute concurrently. The BCC prevents any application from harming another. The BCC is a set of capabilities that allow a user or OEM to resolve conflicting application requests by defining specific values for certain configuration parameters in a Bluetooth stack. It is the central authority for local Bluetooth device settings. The BCC might be a native application, an application with a separate API, or simply a group of settings that are specified by the manufacturer and cannot be changed by the user. Note that the BCC is not a class or an interface defined in this specification but an important part of its security architecture.

Capabilities of JSR 82

The API is intended to provide the following capabilities:

- Register services
- Discover devices and services
- Establish RFCOMM, L2CAP, and OBEX connections between devices
- Using those connections, send and receive data (voice communication not supported)
- Manage and control the communication connections
- Provide security for these activities

The API Architecture

The goal of the specification was to define an open, non-proprietary standard API that can be used by all J2ME-enabled devices. Therefore, it was designed using standard J2ME APIs and CLDC/MIDP's Generic Connection Framework. Some important features:

- The specification provides basic support for Bluetooth protocols and profiles. It doesn't include specific APIs for all Bluetooth profiles simply because the number of profiles is growing.
- The specification incorporates the OBEX, L2CAP, and RFCOMM communication protocols in the JSR 82 APIs, primarily because all current Bluetooth profiles are designed to use these communication protocols.
- The JSR 82 specification addresses the Generic Access Profile, Service Discovery Application Profile, Serial Port Profile, and Generic Object Exchange Profile.
- The Service Discovery protocol is also supported. JSR 82 defines service registration in detail in order to standardize the registration process for the application programmer.

JSR 82 requires that the Bluetooth stack underlying a JSR 82 implementation be qualified for the Generic Access Profile, the Service Discovery Application Profile, and the Serial Port Profile. The stack must also provide access to its Service Discovery Protocol, and to the RFCOMM and L2CAP layers.

The APIs are designed in such a way that developers can use the Java programming language to build new Bluetooth profiles on top of this API as long as the core layer specification does not change. To promote this flexibility and extensibility, the specification is not restricted to APIs that implement Bluetooth profiles. JSR 82 includes APIs for OBEX and L2CAP so that future Bluetooth profiles can be implemented in Java, and these are already being used for that purpose. Figure 1 shows where the APIs defined in this specification fit in a CLDC/MIDP architecture.

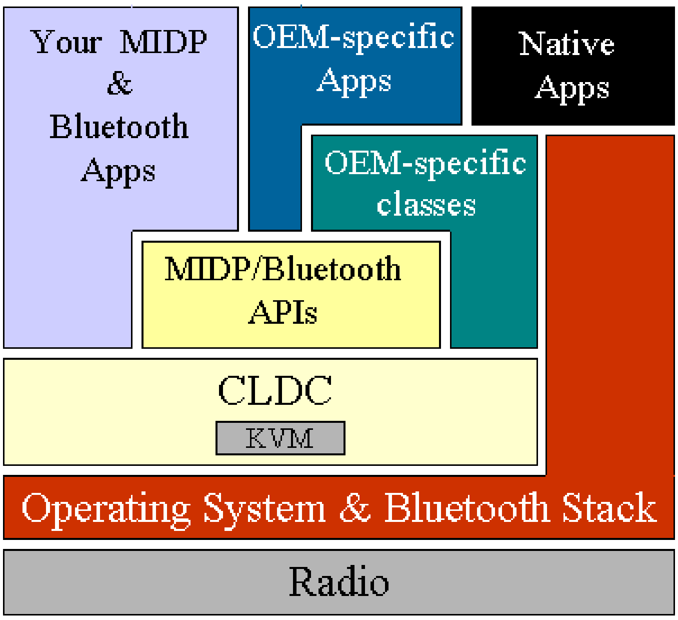

Figure 1: High-level Architecture of J2ME CDLC/MIDP and Bluetooth

JSR 82 is Flexible

The JSR 82 APIs can work with both native Bluetooth stacks and Java Bluetooth stacks. In the latter case the APIs call the stack directly; in the former case the APIs go through the virtual machine, which will interface with the native stack. As I already noted, Java developers can expand their options by creating new profiles.

JSR 82 standardizes the programming interface. Its two optional packages can be used with any of the J2ME profiles. The minimum configuration is CLDC. Because CLDC is a subset of CDC, applications using these APIs should work on CDC devices. If the Generic Connection Framework Optional Package for J2SE (JSR 197) is implemented, the JSR 82 APIs should work smoothly with J2SE.

Packages

The Java APIs for Bluetooth define two packages that depend on the CLDC `javax.microedition.io` package:

- `javax.bluetooth`: core Bluetooth API
- `javax.obex`: APIs for the Object Exchange (OBEX) protocol

Again, the OBEX APIs are defined independently of the Bluetooth transport layer and packaged separately. Each of the above packages represents a separate optional package, which means that a CLDC implementation can include either package or both. MIDP-enabled devices are expected to be the kind of devices to incorporate this specification.

Application Programming

The anatomy of a Blueooth application has five parts: stack initialization, device management, device discovery, service discovery, and communication.

Stack Initialization

The Bluetooth stack is responsible for controlling the Bluetooth device, so you need to initialize the Bluetooth stack before you can do anything else. The initialization process comprises a number of steps whose purpose is to get the device ready for wireless communication. Unfortunately, the Bluetooth specification leaves implementation of the BCC to vendors, and different vendors handle stack initialization differently. On one device, it may be an application with a GUI interface, and on another it may be a series of settings that cannot be changed by the user. As an example, Atinav's Java Bluetooth solution requires the developer to initialize the stack with a series of settings like the ones in the following code snippet - note well that the APIs invoked are not part of JSR 82:

```
...
// set the port number
BCC.setPortNumber("COM1");
// set the baud rate
BCC.setBaudRate(50000);
// set the connectable mode
BCC.setConnectable(true);
// set the discovery mode to Limited Inquiry Access Code
BCC.setDiscoverable(DiscoveryAgent.LIAC);
...
```

Device Management

The Java Bluetooth APIs contain the classes `LocalDevice` and `RemoteDevice`, which provide the device-management capabilities defined in the Generic Access Profile. `LocalDevice` depends on the `javax.bluetooth.DeviceClass` class to retrieve the device's type and the kinds of services it offers. The `RemoteDevice` class represents a remote device (a device within a range of reach) and provides methods to retrieve information about the device, including its Bluetooth address and name. The following code snippet retrieves that information for the local device:

```
...
// retrieve the local Bluetooth device object
LocalDevice local = LocalDevice.getLocalDevice();
// retrieve the Bluetooth address of the local device
String address = local.getBluetoothAddress();
// retrieve the name of the local Bluetooth device
String name = local.getFriendlyName();
...
```

You can get the same information about a remote device:

```
...
// retrieve the device that is at the other end of
// the Bluetooth Serial Port Profile connection,
// L2CAP connection, or OBEX over RFCOMM connection
RemoteDevice remote =
    RemoteDevice.getRemoteDevice(
        javax.microedition.io.Connection c);
// retrieve the Bluetooth address of the remote device
String remoteAddress = remote.getBluetoothAddress();
// retrieve the name of the remote Bluetooth device
String remoteName = local.getFriendlyName(true);
...
```

The `RemoteDevice` class also provides methods to authenticate, authorize, or encrypt data transferred between local and remote devices.

Device Discovery

Because wireless devices are mobile they need a mechanism that allows them to find other devices and gain access to their capabilities. The core Bluetooth API's `DiscoveryAgent` class and `DiscoveryListener` interface provide the necessary discovery services.

A Bluetooth device can use a `DiscoveryAgent` object to obtain a list of accessible devices, in any of three ways:

The `DiscoveryAgent.startInquiry` method places the device into an inquiry mode. To take advantage of this mode, the application must specify an event listener that will respond to inquiry-related events. `DiscoveryListener.deviceDiscovered` is called each time an inquiry finds a device. When the inquiry is completed or canceled, `DiscoveryListener.inquiryCompleted` is invoked.

If the device doesn't wish to wait for devices to be discovered, it can use the `DiscoveryAgent.retrieveDevices` method to retrieve an existing list. Depending on the parameter passed, this method will return either a list of devices that were found in a previous inquiry, or a list of *pre-known devices* that the local device has told the Bluetooth Control Center it will contact often.

These three code snippets demonstrate the various approaches:

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// place the device in inquiry mode
boolean complete = agent.startInquiry();
...
```

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// return an array of pre-known devices
RemoteDevice[] devices =
   agent.retrieveDevices(DiscoveryAgent.PREKNOWN);
...
```

```
...
// retrieve the discovery agent
DiscoveryAgent agent = local.getDiscoveryAgent();
// return an array of devices found in a previous inquiry
RemoteDevice[] devices =
    agent.retrieveDevices(DiscoveryAgent.CACHED);
...
```

Service Discovery

Once the local device has discovered at least one remote device, it can begin to search for available *services* - Bluetooth applications it can use to accomplish useful tasks. Because service discovery is much like device discovery, `DiscoveryAgent` also provides methods to discover services on a Bluetooth server device, and to initiate service-discovery transactions. Note that the API provides mechanisms to search for services on remote devices, but not for services on the local device.

Service Registration

Before a service can be discovered, it must first be *registered* - advertised on a *Bluetooth server device*. The server is responsible for:

- Creating a service record that describes the service offered
- Adding the service record to the server's Service Discovery DataBase (SDDB), so it's visible and available to potential clients
- Registering the Bluetooth security measures associated with the service (enforced for connections with clients)
- Accepting connections from clients
- Updating the service record in the SDDB whenever the service's attributes change
- Removing or disabling the service record in the SDDB when the service is no longer available

The following annotated code fragment gives you a flavor of the effort involved in registering a service using the Java APIs for Bluetooth:

1. To create a new service record that represents the service, invoke `Connector.open` with a server connection URL argument, and cast the result to a `StreamConnectionNotifier` that represents the service:

```
...
StreamConnectionNotifier service =
    (StreamConnectionNotifier) Connector.open("someURL");
```

2. Obtain the service record created by the server device:

```
ServiceRecord sr = local.getRecord(service);
```

3. Indicate that the service is ready to accept a client connection:

```
StreamConnection connection =
    (StreamConnection) service.acceptAndOpen();
```

   Note that `acceptAndOpen` blocks until a client connects.

4. When the server is ready to exit, close the connection and remove the service record:

```
service.close();
...
```

Communication

For a local device to use a service on a remote device, the two devices must share a common communications protocol. So that applications can access a wide variety of Bluetooth services, the Java APIs for Bluetooth provide mechanisms that allow connections to any service that uses RFCOMM, L2CAP, or OBEX as its protocol. If a service uses another protocol (such as TCP/IP) layered above one of these protocols, the application *can* access the service, but only if it implements the additional protocol in the application, using the CLDC Generic Connection Framework.

Because the OBEX protocol can be used over several different transmission media - wired, infrared, Bluetooth radio, and others - JSR 82 implements the OBEX API (`javax.obex`) independently of the core Bluetooth API (`javax.bluetooth`). The OBEX API is a separate optional package you can use either with the core Bluetooth package or independently.

Serial Port Profile

The RFCOMM protocol, which is layered over the L2CAP protocol, emulates an RS-232 serial connection. The Serial Port Profile (SPP) eases communication between Bluetooth devices by providing a stream-based interface to the RFCOMM protocol. Some capabilities and limitations to note:

- Two devices can share only one RFCOMM session at a time.
- Up to 60 logical serial connections can be multiplexed over this session.
- A single Bluetooth device can have at most 30 active RFCOMM services.
- A device can support only one client connection to any given service at a time.

For a server and client to communicate using the Serial Port Profile, each must perform a few simple steps.

As the following code snippet demonstrates, the server must:

1. Construct a URL that indicates how to connect to the service, and store it in the service record
2. Make the service record available to the client
3. Accept a connection from the client
4. Send and receive data to and from the client

The URL placed in the service record may look something like:

`btspp://102030405060740A1B1C1D1E100:5`

This says that a client should use the Bluetooth Serial Port Profile to establish a connection to this service, which is identified with server channel 5 on a device whose address is 102030405060740A1B1C1D1E100.

```
...
// assuming the service UID has been retrieved
String serviceURL =
    "btspp://localhost:"+serviceUID.toString());
// more explicitly:
String ServiceURL =
    "btspp://localhost:10203040607040A1B1C1DE100;name=SPP
        Server1";
try {
    // create a server connection
    StreamConnectionNotifier notifier =
       (StreamConnectionNotifier) Connector.open(serviceURL);
    // accept client connections
    StreamConnection connection = notifier.acceptAndOpen();
    // prepare to send/receive data
    byte buffer[] = new byte[100];
    String msg = "hello there, client";
    InputStream is = connection.openInputStream();
```

```
    OutputStream os = connection.openOutputStream();
    // send data to the client
    os.write(msg.getBytes());
    // read data from client
    is.read(buffer);
    connection.close();
} catch(IOException e) {
  e.printStackTrace();
}
...
```

At the other end, as the next code snippet shows, to set up an RFCOMM connection to a server the client must:

1. Initiate a service discovery to retrieve the service record
2. Construct a connection URL using the service record
3. Open a connection to the server
4. Send and receive data to and from the server

```
...
// (assuming we have the service record)
// use record to retrieve a connection URL
String url =
    record.getConnectionURL(
        record.NOAUTHENTICATE_NOENCRYPT, false);
// open a connection to the server
StreamConnection connection =
    (StreamConnection) Connector.open(url);
// Send/receive data
try {
    byte buffer[] = new byte[100];
    String msg = "hello there, server";
    InputStream is = connection.openInputStream();
    OutputStream os = connection.openOutputStream();
    // send data to the server
    os.write(msg.getBytes);
    // read data from the server
    is.read(buffer);
    connection.close();
} catch(IOException e) {
  e.printStackTrace();
}
...
```

If you'd like to learn how to use communication protocols other than RFCOMM, see the JSR 82 specification.

J2ME/Bluetooth Development Kits

Several integrated development environments provide APIs for Bluetooth. When selecting an IDE for J2ME/Bluetooth development, make sure it complies with JSR 82. If you have access to Bluetooth devices, I'd recommend the Java Bluetooth solution from Atinav. It's JSR 82-compliant and based on an all-Java stack, it includes a KVM, and it supports RS-232, USB, PCMCIA, and other Bluetooth devices.

Esmertec, Smart Network Devices, and several other companies offer unique JSR 82-compliant solutions.

Ireland-based Rococo Software has a number of J2ME/Bluetooth products including the Impronto Simulator, which allows you to run Java applications in a simulated Bluetooth environment, where you can test and configure applications before deploying them on Bluetooth-enabled devices. All this, without the need to purchase hardware and a Bluetooth stack! Here's how it works: When an application runs on Impronto Simulator, the virtual Bluetooth stack processes all Bluetooth calls, routing them to the virtual stack of the appropriate device. A discovery server allows devices to locate one another; a GUI helps developers monitor the behavior of simulated devices and networks; an event logger logs API calls and Bluetooth events; and a device editor is used to configure virtual devices. The simulator supports the standard Java APIs for Bluetooth. One limitation is that it doesn't allow you to specify which device is a master and which is a slave. Figure 2 shows a screenshot:

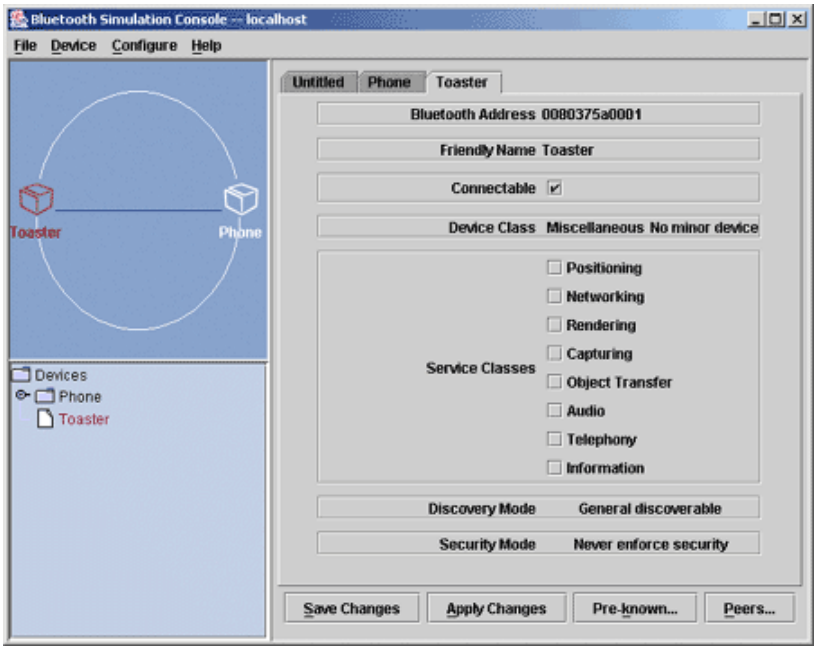

Figure 2: Impronto Simulator

Conclusion

This article presented a tutorial on the Java APIs for Bluetooth wireless technology. The sample code demonstrated how easy it is to develop wireless applications for Bluetooth-enabled devices. The APIs enable you to exploit fully the power of the Java programming language to develop wireless applications in a standard way. This set of APIs is a key enabler that will help software vendors and developers tap the potentially huge market for Bluetooth wireless technology.

For more information

- JSR 82: Java APIs for Bluetooth Wireless Technology
- Bluetooth Specification
- Bluetooth SIG, Inc. (Member Website)
- Bluetooth Qualification Process
- Bluetooth Qualified Products
- Subscribe to the JABWT Yahoo! Group
- Atinav
- Esmertec
- Smart Network Devices
- Rococo Software

Acknowledgments

Special thanks to Teck Yang Lee of Sun Microsystems whose feedback helped me improve the article.

About the Author: Qusay H. Mahmoud provides Java consulting and training services. He has published dozens of articles on Java, and is the author of *Distributed Programming with Java* (Manning Publications, 1999) and *Learning Wireless Java* (O'Reilly & Associates, 2002).

Back To Top