January 2004

Developing Applications with the Java APIs for Bluetooth™ (JSR-82)

Sony Ericsson

1. Purpose of this document

This course introduces the JSR 82 Java APIs for Bluetooth, an optional J2ME package defined by the Java Community Process that provides a standard API for Bluetooth connectivity.

Table of Contents

The Java APIs for Bluetooth™ (JSR-82) 1

1. Purpose of this document 2

2. What You Need To Get Started 4

3. Introduction 4

4. The Bluetooth Protocol Stack 4

4.1. Background 4

4.2. The Protocol Stack 4

4.3. Profiles 5

4.4. Bluetooth Devices and Network 6

5. The Java APIs for Bluetooth 7

5.1. Bluetooth System Requirements 7

5.2. Java APIs For Bluetooth - Organization and Packages 7

5.3. Anatomy of a JSR 82-enabled MIDlet 8

5.4. Using the Java APIs for Bluetooth 8

5.5. The Bluetooth Control Center 9

5.6. Overview of the Connection APIs 10

5.6.1. Bluetooth Connection Types 11
5.6.2. Creating a Connection 11
5.6.3. Waiting For a Connection 13
5.6.4. Sending and Receiving Data 14

5.7. Overview of the Device Management APIs 16

5.7.1. Local Device 16
5.7.2. Remote Device 17
5.7.3. Device Class 18
5.7.4. Gathering Bluetooth Properties 19

5.8. Registering a Service 20

5.9. Overview of the Device and Service Discovery APIs 20

5.9.1. The MIDlet 20
5.9.2. The DiscoveryAgent 21
5.9.3. The DiscoveryListener 21
5.9.4. The ServiceRecord 22

5.10. Security 24
6. The P900 J2ME SDK 25
7. Resources 25
7.1. Abbreviations 25
7.2. Further Information and Links 26
7.3. The Bluetooth Car Application 27
7.4. Bluepad - a Bluetooth shared sketch board for P900 phones 28

2. What You Need To Get Started

To develop Bluetooth applications for the Sony Ericsson P900, this course requires that you have the following SDKs installed on your Windows PC:

- Java 2 Standard Edition (J2SE) SDK 1.4.1 or later
- Java 2 Runtime Environment (JRE) Standard Edition 1.4
- Sun Wireless Toolkit version 2.0 or later
- Sony Ericsson's P900 J2ME SDK

3. Introduction

This paper covers the Java API for Bluetooth (JSR-82) with respect to Sony Ericsson devices. It starts by introducing the Bluetooth technology, followed by the Java APIs for Bluetooth, and how to use them.

Currently, these APIs are currently available in the Sony Ericsson P900/P908 handsets.

Note: The Java Bluetooth API's are not available in the first software release in P900 (Organizer SW version R1*). For more information about this software release, refer to "Sony Ericsson P900 MIDP 2.0 Java Developers' Guidelines".

4. The Bluetooth Protocol Stack

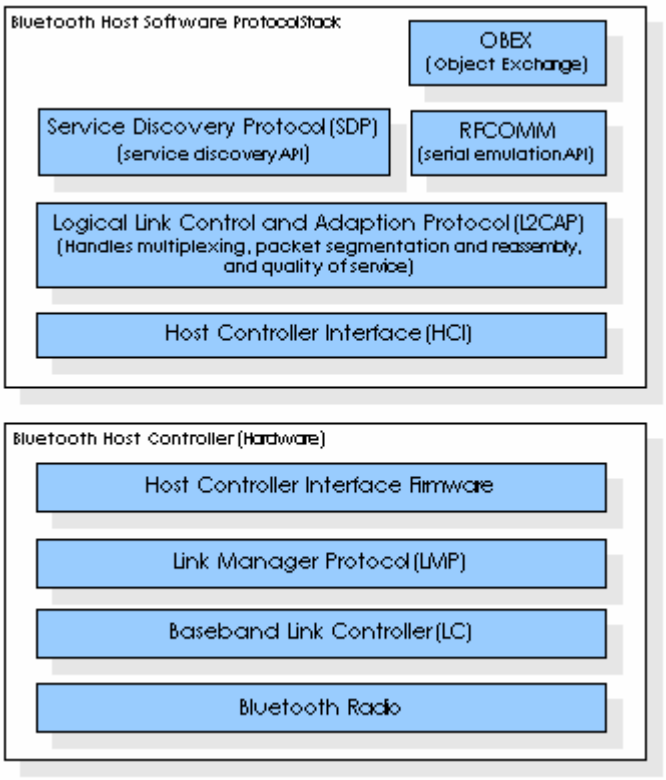
4.1. Background

Developed by the Bluetooth Special Interest Group (SIG), Bluetooth is a low-cost, short-range radio technology intended to replace data cables connecting electronic devices. Ideal for wireless personal networks it uses the unlicensed ISM frequency band of 2.45 GHz that is globally available. The Bluetooth technology, currently in version 1.1, consists of the radio technology, the software stack, and profiles.

4.2. The Protocol Stack

The Bluetooth protocol stack is divided into two parts: the controller that is typically implemented in hardware, and the on-host stack with which applications and services interact. Figure 1 represents the Bluetooth protocol stack.

Figure 1. The Bluetooth Stack



In this paper we are mainly concerned with the top layers of the implementation, which is the Bluetooth Host Software Protocol Stack. This stack consists of the following layers:

- **Host Controller Interface (HCI)** – this is the lowest layer of the Bluetooth Host stack. It interfaces directly with the host controller hardware.
- **Logical Link Control and Adaptation Layer (L2CAP)** – this layer handles packet segmentation and reassembly (SAR), protocol multiplexing, and provides quality of service information.
- **Service Discovery Protocols (SDP)** – as the name implies, applications use this layer to discover Bluetooth services that are available.
- **RFCOMM** – this layer provides serial behavior over Bluetooth, similar to using a standard serial (COM) port.
- **Object Exchange Protocol** – originally defined by the Infrared Data Association (IrDA), this protocol enables the exchange of objects such as vCard and vCalendar synchronization data.

4.3. Profiles

To ensure interoperability and consistency between devices, Bluetooth profiles define vendor-neutral device capabilities. A *profile* describes specific functions and features that use Bluetooth as its transport mechanism. Profiles supply information that makes certain that Bluetooth devices that claim these capabilities can exchange data with devices from another vendor. The Bluetooth SIG has defined a number of standard profiles:

- **Generic Access Profile (GAP)** – defines the use of the low-layers of the Bluetooth protocol stack, including device management functionality. All Bluetooth implementations implement the GAP.
- **Service Discover Application Profile (SDAP)** – describes a specific application and usage of SDP, the availability and user interface aspects of service discovery, and the use of the L2CAP and low-layer for service discovery.
- **Serial Port Profile (SPP)** – defines the RFCOMM, L2CAP, SDP, and low-layer layer interoperability requirements and capabilities, for serial cable emulation.
- **Dial-up Networking Profile (DUNP)** – defines the interoperability requirements for GAP and SPP, and dialing and control capabilities that allows a device to serve as a dial-up device.
- **Generic Object Exchange Profile (GOEP)** – defines the OBEX, SPP, and GAP interoperability requirements, and OBEX capabilities for file transfers, object push, and synchronization.

- **Object Push Profile (OPP)** – defines the user interface requirements, use of OBEX and SDP, and the object push feature to push vCard, vCalendar, vNote, and vMessage content formats.
- **File Transfer Profile (FTP)** – defines the user interface requirements, and the interoperability and use of GOEP, OBEX, and SDP.
- **Synchronization Profile (SP)** – defines the user interface requirements, and the interoperability and use of GOEP, OBEX and SDP, as well as IrMC synchronization requirements.

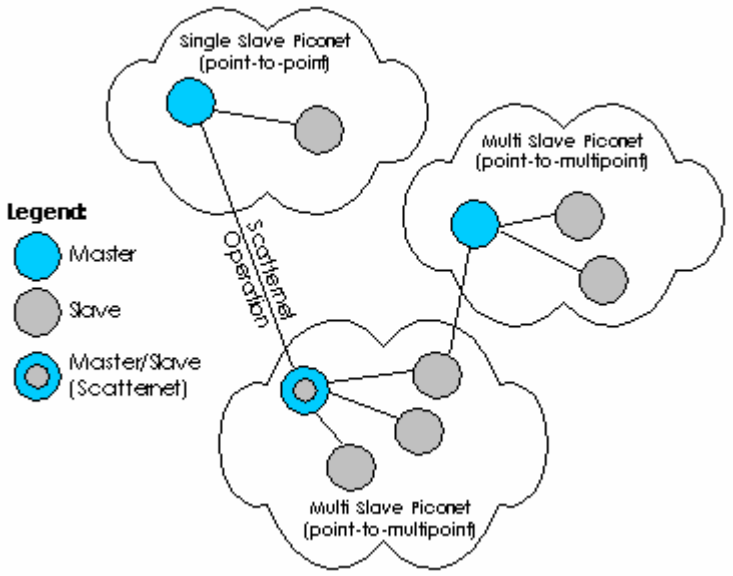
The Sony Ericsson P900 supports the following Bluetooth profiles:

- Generic Access Profile
- Serial Port Profile
- Dialup Networking Profile
- Generic Object Exchange Profile
- Object Push Profile
- Headset Profile
- File Transfer Profile

4.4. Bluetooth Devices and Network

Bluetooth devices use a unique IEEE 802 *48-bit address*. When two or more Bluetooth devices connect, they form what is referred to as a *Piconet*. This is a dynamic (ad hoc) network, where one device acts as a master while all others (up to seven) are slaves. Slaves can participate in different piconets, and two connecting masters form an overlapping piconet that is referred to as a *Scatternet*; in Figure 2 a piconet master node becomes a slave node for the other piconet.

Figure 2. The Bluetooth Piconet and Scatternet Networks.



Bluetooth supports one data channel and a maximum of three voice channels. Data can be exchanged at a rate of approximately 720 kilobits per second using point-to-point or multipoint encrypted connections. The theoretical range of Bluetooth is 100 meters. The range for Sony Ericsson P900/P908 is approximately 10 meters.

For more information about each profile, please refer to the *Profiles Bluetooth Specification* found at the Bluetooth web site, http://www.bluetooth.org/.

5. The Java APIs for Bluetooth

5.1. Bluetooth System Requirements

All Bluetooth low-level implementations must satisfy a set of requirements with respect to the supported Bluetooth profiles and protocols, as follows:

Table 1. Bluetooth System Requirements.

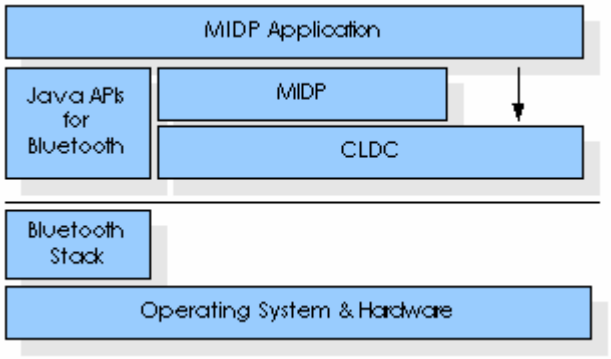
Bluetooth version 1.1 layers	Profiles
• RFCOMM • Service Discovery Protocol • L2CAP	• Generic Access Profile • Service Discovery Application Profile • Serial Port Profile

In addition, Bluetooth devices must also support what is called the Bluetooth Control Center (BCC), which is the central authority for local Bluetooth device settings. The BCC is not part of the Bluetooth Java API itself, but the Bluetooth API depends on the BCC for things such as device level configuration and security settings.

5.2. Java APIs For Bluetooth - Organization and Packages

The Java APIs for Bluetooth is a J2ME *optional package* for defined by the Java Community Process (JSR-82). This optional package provides a common API for Bluetooth development. Figure 3 illustrates the relationship between the Java APIs for Bluetooth and the J2ME platform, using the Mobile Information Device Profile (MIDP) and Connected Limited Device Configuration (CLDC) stack:

Figure 3. Bluetooth and J2ME MIDP.



At the bottom of the stack are the hardware, operating system, and Bluetooth stack, followed by the configuration (in this example, CLDC), profile (in this example, MIDP), and optional packages (in this example, the Java APIs for Bluetooth). On the top we have the MIDP application (MIDlet).

Table 2 – Java APIs for Bluetooth.

Package Name	Description
`javax.microedition.io`	The core CLDC Generic Connection Framework.
`javax.bluetooth`	Core Bluetooth API, such as Discovery, L2CAP, and device and data interfaces and classes.
`javax.obex`	Core Object Exchange (OBEX) APIs. Support is optional.

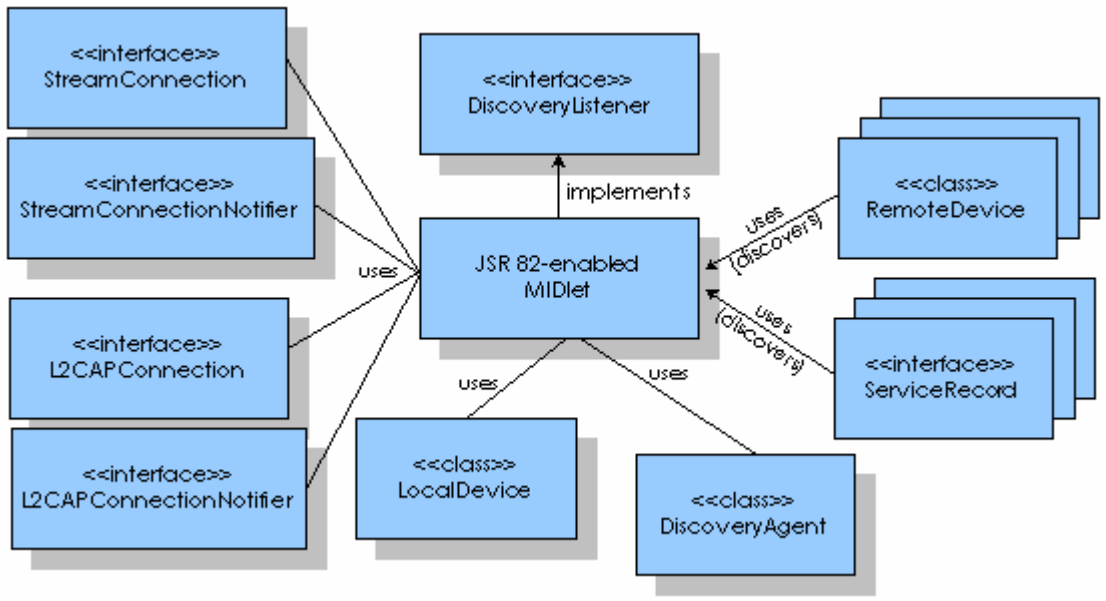
The Java APIs for Bluetooth optional package defines a new connection protocol for the GCF and an Object Exchange (OBEX) API based on the specifications of the IrDA Data Association.

Note: The `javax.obex.*` package is not supported in P900/P908. The `javax.bluetooth.*` package is a restricted API on the P900/P908. For more information about this software release, refer to “Sony Ericsson P900 MIDP 2.0 Java Developers’ Guidelines”.

5.3. Anatomy of a JSR 82-enabled MIDlet

Figure 4 shows all of the interfaces and classes available to a JSR-82-enabled MIDlet. These will be discussed in further detail in the sections that follow.

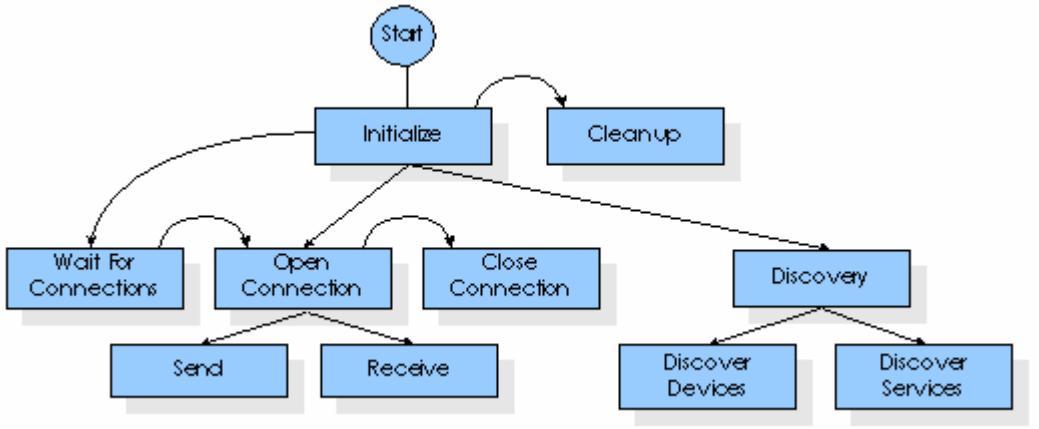
Figure 4. Anatomy of a JSR 82-enabled MIDlet



5.4. Using the Java APIs for Bluetooth

Using the Java APIs for Bluetooth consists of initializing the Bluetooth stack, discovering devices or services that are in proximity, open and close and waiting for or initiate connections, and perform I/O. Figure 5 illustrates the different Bluetooth operations your application can perform.

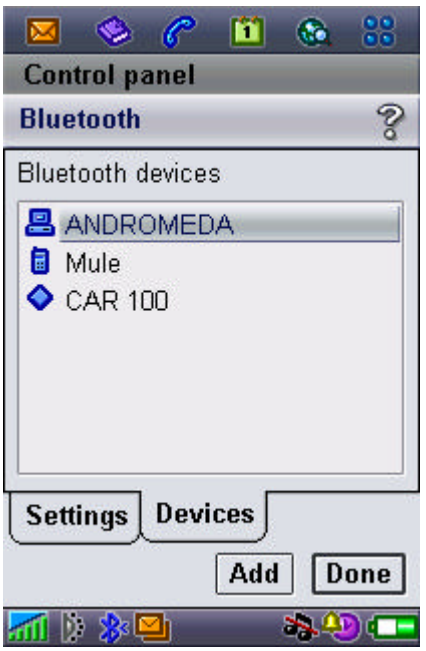
Figure 5. Using the Bluetooth API.

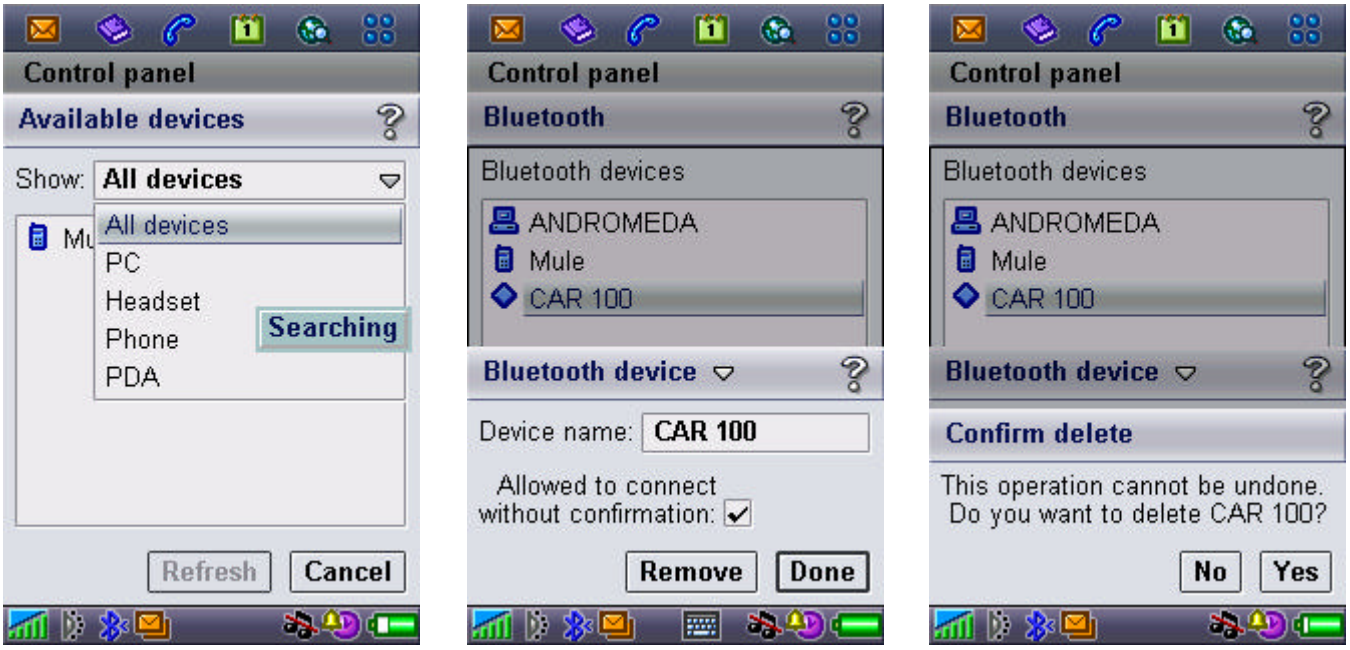


5.5. The Bluetooth Control Center

Bluetooth initialization typically entails setting the device's name, security settings, and/or turning the Bluetooth radio on/off. These aforementioned steps are done via what is referred to as the Bluetooth Control Center (BCC), which typically are a set of control panels that serves as the central authority for local Bluetooth device settings. The following shows the BCC control panels for Bluetooth on P900:

Figure 6. The P900 Bluetooth Control Panels





The control panels provide options to set the readable device name that other Bluetooth enabled devices will see, to enable/disable Bluetooth, and to specify if the device shall be searchable by other Bluetooth devices. Other features include scan for other Bluetooth devices in the neighborhood, specify connection permissions, and connect or disconnect (i.e. pair or remove paired) devices.

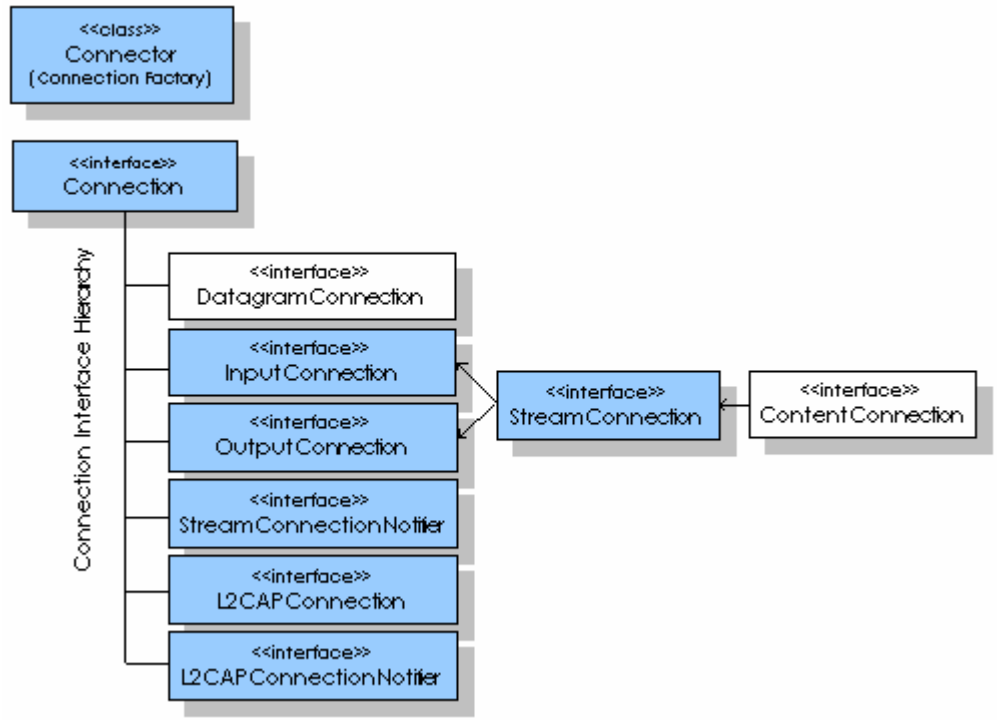
5.6. Overview of the Connection APIs

With the Java APIs for Bluetooth you can create Bluetooth connections that use the Serial Port Profile (RFCOMM), L2CAP and OBEX. As OBEX is considered optional (and not supported in P900/P908), it is not covered in this paper.

5.6.1. Bluetooth Connection Types

The Java APIs for Bluetooth are based on the CLDC 1.0 Generic Connection Framework (GCF). Figure 7 illustrates the relationship between the GCF and the Bluetooth APIs. The Java APIs for Bluetooth appear in the blue boxes:

Figure 7. Bluetooth and the Generic Connection Framework.



The Connector class is the Connection factory. In support of RFCOMM and OBEX connectivity, the Bluetooth API leverages existing GCF connection types `StreamConnection` (and indirectly `InputConnection` and `OutputConnection`) and `StreamConnectionNotifier`. For L2CAP connectivity, the API introduces two new connection types the L2CAPConnection, and L2CAPConnectionNotifier.

`StreamConnection` is a subinterface of both `InputConnection` and `OutputConnection`. `InputConnection` and `OutputConnection` returns input and output streams with methods that allow you to read and write (formatted) data. The `StreamConnectionNotifier` is an interface that represents a server side stream connection. `StreamConnectionNotifier` defines a single method, `acceptAndOpen()` that waits for and opens (returns) incoming stream connections. For more information about these aforementioned interfaces, please refer to the MIDP 1.0 Javadoc.

L2CAPConnection is a subinterface of both Connection and methods to read and write raw data, and to discover the send and receive Maximum Transmission Unit (MTU). A MTU defines the maximum number of bytes that can be sent or received without loosing data. The L2CAPConnectionNotifier is very similar to `StreamConnectionNotifier`, but in this case it represents a server side L2CAP connection. L2CAPConnectionNotifier defines a single method, `acceptAndOpen()` that waits for and opens (returns) incoming L2CAP connections.

5.6.2. Creating a Connection

Creating Bluetooth (server and client) connections is done similarly to creating other types of GCF connections: by using the Connector connection factory class. To create a connection, use the `Connector.open()` method and provide an appropriate URL that describes the Bluetooth connection type to create. Three forms of the `open()` method are defined:

- `open(String url)`
- `open(String url, int mode)`

- `open(String url, int mode, boolean timeout)`

Where:

- **`url`** is the URL that describes the connection type to create (examples follow).
- **`mode`** is the connection mode to use - `READ`, `WRITE`, or `READ_WRITE` (the default).
- **`timeout`** is a flag to indicate that the caller wants to be notified about timeout exceptions (`InterruptedIOException`), if the underlying connection implements notification. (The default is false.)

The simple form, `open(String url)`, is the most widely used. The connection URL, which is used to describe the connection type to create, has the following form:

```
scheme://host:port;parameters
```

Where:

- **`scheme`** describes the connection type to use. See Table 3.
- **`host`** for client connections. It specifies the remote address to connect to, or `localhost` for server connections.
- **`port`** for client connections. It specifies the protocol service multiplexor. For server connections, it specifies the service UUID.
- **`parameters`** specify optional parameters, such as friendly name and MTU sizes.

The following table summarizes the URL scheme and GCF connection type to use for RFCOMM and L2CAP Bluetooth connections:

Table 3. RFCOMM and L2CAP Bluetooth Connections.

Bluetooth Connection	URL Scheme	Client Connection	Server Connection
Serial Port Profile (RFCOMM)	`Btspp`	`StreamConnection`	`StreamConnectionNotifier`, `StreamConnection`
L2CAP	`btl2cap`	L2CAPConnection	`L2CAPConnectionNotifier`, `L2CAPConnection`

- To create a Bluetooth *client connection* specify a remote device address and channel in the URL. For example:

To create a RFCOMM client connection, specify a `btspp` scheme, the remote device address, and the service channel:

```
StreamConnection con = (StreamConnection) Connector.open("btspp://0050C000321B:5");
```

To create a L2CAP client connection, specify a btl2cap scheme, along with the address and protocol service multiplexor for the remote device:

```
L2CAPConnection con = (L2CAPConnection) Connector.open("btl2cap://0050C000321B:1000");
```

To discover the client connection service URL, use the `ServiceRecord.getConnectionURL()` method.

- To create a Bluetooth *server connection*, specify `localhost` and the service UUID and name. For example:

To create a RFCOMM (SPP) server connection, specify a `btspp` scheme, `localhost` as the host, the service UUID, and a friendly name:

```
StreamConnectionNotifier cn = (StreamConnectionNotifier) Connector.open("btspp://
```

```
localhost:" + MY_SERVICE_NUMBER);
```

Being MY_SERVICE_NUMBER a String representing the service UUID and service name, for example "3B9FA89520078C303355AAA694238F08;name=mySPPSrv".

To create a L2CAP server connection, specify a btl2cap scheme, the service UUID, and a friendly name:

```
L2CAPConnectionNotifier = cn (L2CAPConnectionNotifier)
Connector.open("btl2cap://localhost:" + MY_SERVICE_NUMBER);
```

Being MY_SERVICE_NUMBER a String representing the service UUID and service name, for example "3B9FA89520078C303355AAA694238F08;name=myL2CAPSrv".

Before creating the Bluetooth connection as shown above, get the local device, and make it to discoverable:

```
LocalDevice local = LocalDevice.getLocalDevice();
local.setDiscoverable(DiscoveryAgent.GIAC);
```

5.6.3. Waiting For a Connection

Waiting for connections is accomplished by:

1. Creating a server connection using the appropriate connect notifier:
   - For RFCOMM a `StreamConnectionNotifier`
   - For L2CAP an `L2CAPConnectionNotifier`
2. Waiting, accepting and opening new connections using the `acceptAndOpen()` method.

Listing 1. Handling Incoming L2CAP Connections.

```
L2CAPConnectionNotifier server = null;
byte[] data = null;
int length;
:
:
try {
    // Get the local device, make it discoverable
    LocalDevice local = LocalDevice.getLocalDevice();
    local.setDiscoverable(DiscoveryAgent.GIAC);
    // Create a L2CAP connection notifier
    server = (L2CAPConnectionNotifier)
Connector.open("btl2cap://localhost:1020304050d0708093a1b121d1e1f100");
    while (!done) {
        L2CAPConnection conn = null;
        // Wait for incoming L2CAP connections
        conn = server.acceptAndOpen();
        // Read the incoming data
```

```
        length = conn.getReceiveMTU();
        data = new byte[length];
        length = conn.receive(data);
        :
        :
    }
} catch (Exception e) {
    ... Handle Exception
}
```

Listing 2. Handling Incoming Stream Connections.

```
StreamConnectionNotifier server = null;
byte[] data = new byte[256];
int length;
:
:
try {
    // Get the local device, make it discoverable
    LocalDevice local = LocalDevice.getLocalDevice();
    local.setDiscoverable(DiscoveryAgent.GIAC);
    // Create a stream connection notifier
    server = (StreamConnectionNotifier)Connector.open(
        "btspp://localhost:11111111111111111111111111111111");
    while (!done) {
        StreamConnection conn = null;
        // Wait for incoming stream connections
        conn = server.acceptAndOpen();
        // Read the incoming data
        InputStream in = conn.openInputStream();
        length = in.read(data);
        :
        :
    }
} catch (Exception e) {
    ... Handle Exception
}
```

5.6.4. Sending and Receiving Data

Sending and receiving stream data is accomplished by using a `StreamConnection`. Sending and receiving L2CAP data is accomplished by using an L2CAPConnection.

Listing 3. Sending Stream Data Using L2CAPConnection.

```
String url = "...";
int index = 0;
L2CAPConnection con = null;
transmitBuffer[] temp = null;
byte[] data = ...;

try {
    // Open the connection to the server
    con = (L2CAPConnection)Connector.open(url);

    // The the transmit MTU (the maximum amount of data that can be sent)
    int MTUSize = con.getTransmitMTU();
    // Allocation a buffer of that (MTU) size
    transmitBuffer = new byte[MaxOutBufSize];
    :
    :

    while (index < data.length) {
        //  Send the data... move MTUSize bytes from data
        //  buffer to transmit buffer
        if ((data.length - index) < MTUSize) {
            System.arraycopy(data, index, transmitBuffer, 0, data.length - index);
        } else {
            System.arraycopy(data, index, transmitBuffer, 0, MTUSize);
        }
        // Send the data, and adjust data index
        con.send(transmitBuffer);
        index += MTUSize;
        //  Reset the transmit buffer
        for (int=0; i<MTUSize; i++) transmitBuffer[i] = 0;
    }
    // Close the output connection and stream when done
    con.close();
} catch (Exception e) {
    ... Handle Exception
}
```

Listing 4. Sending Stream Data Using `StreamConnection`.

```
String url = "…";
OutputStream os = null;
StreamConnection con = null;
:
:
try {
   // Open the connection to the server
   con =(StreamConnection)Connector.open(url);
   // Open the output stream
   os = con.openOutputStream();
   // Sends stream data to remote device
   os.write(data.getBytes());
   // Close the output connection and stream when done
   os.close();
   con.close();
} catch (Exception e) {
    ... Handle Exception
}
```

5.7. Overview of the Device Management APIs

At the center of device management are the `javax.bluetooth.LocalDevice` and `javax.bluetooth.RemoteDevice`, and `javax.bluetooth.DeviceClass` classes. These classes provide the device management capabilities that are part of the Generic Access Profile (GAP).

5.7.1. Local Device

The local Bluetooth device is represented by `javax.bluetooth.LocalDevice`. This class provides methods to manage and retrieve local device and information about it such as its Bluetooth address, device class, and discovery agent. Some of the methods provided by this class include:

- `static LocalDevice getLocalDevice()` – static method to retrieve the `LocalDevice` object that represents the local Bluetooth device.
- `java.lang.String getBluetoothAddress()`– retrieves the Bluetooth address of the local device. A Bluetooth address is represented as `java.lang.String` that represents a 12 characters long hexadecimal value.
- `java.lang.String getFriendlyName()`– retrieves the name of the local device.
- `DiscoveryAgent getDiscoveryAgent()`– returns the discovery agent for this device.
- `boolean setDiscoverable(int mode)` – sets the discoverable mode of the device.
- `static java.lang.String getProperty(java.lang.String property)` – retrieves Bluetooth system properties.

- `ServiceRecord getRecord(javax.microedition.io.Connection notifier)` – retrieves the service record corresponding to the passed (`btspp`, `btl2cap`, or `btgoep`) notifier.

For a complete list of the available methods please refer to the Javadoc.

Listing 5. Using the `LocalDevice` methods.

```
import javax.microedition.io.*;
import javax.bluetooth.*;

LocalDevice localDevice; // The local device
String localAddress; // The local device's Bluetooth address
String localName; // The local device's name
DiscoveryAgent agent; // The DiscoveryAgent for the local Bluetooth device
:
:
try {
    //  Get the local device
    localDevice = LocalDevice.getLocalDevice();
    //  Make local device discoverable
    localDevice.setDiscoverable(DiscoveryAgent.GIAC);
    // Get the local device's Bluetooth address
    localAddress = localDevice.getBluetoothAddress();
    // Get the local device's name
    localName = localDevice.getFriendlyName();
     // Get the DiscoveryAgent object for device and service discovery
    agent = localDevice.getDiscoveryAgent();
} catch (Exception e) {
    ... Handle Exception
}
```

5.7.2. Remote Device

The remote local Bluetooth device is represented by `javax.bluetooth.RemoteDevice`. This class provides methods to retrieve the `RemoteDevice` object associated with a Bluetooth connection, methods to learn the address and name of the remote device, and security-related methods. Some of the methods provided by this class include:

- `static RemoteDevice getRemoveDevice(javax.microedition.io.Connection)` – static method to retrieve the `RemoteDevice` object associated with the passed Connection.
- `java.lang.String getBluetoothAddress()` – retrieves the Bluetooth address of the remote device. A Bluetooth address is represented as `java.lang.String` that represents a 12 characters long hexadecimal value.
- `java.lang.String getFriendlyName()`– retrieves the name of the remote device.
- `boolean authenticate()` – attempts to authenticate the remote device.

- `boolean isAuthenticated()` – determines if this `RemoteDevice` has been authenticated.
- `boolean isEncrypted()` – determines if data exchanges with this `RemoteDevice` are currently being encrypted.

For a complete list of the available methods please refer to the Javadoc.

Listing 6. Using the `RemoteDevice` methods.

```
import javax.microedition.io.*;
import javax.bluetooth.*;

String url = "...";
StreamConnection con;
RemoteDevice remoteDevice; // The local device
String remoteAddress; // The local device's Bluetooth address
String remoteName; // The local device's name
:
:
try {
    con = (StreamConnection) Connector.open(url);
    remoteDevice = RemoteDevice.getRemoteDevice(con);
    // Get the local device's Bluetooth address
    remoteAddress = remoteDevice.getBluetoothAddress();
    // Get the local device's name
    remoteName = remoteDevice.getFriendlyName();
    if (!remoteDevice.isEncrypted()) {
        // If connection not encrypted, try to turn encryption on...
        if (!remoteDevice.authenticate() || !remoteDevice.encrypt(con, true)) {
            // Error... unable to turn on encryption
            return;
        }
    }
} catch (Exception e) {
    ... Handle Exception
}
```

5.7.3. Device Class

Class `DeviceClass` represents a class of device (CoD) as specified in the Bluetooth specification. Devices classes are identified using a major, minor and service class. `DeviceClass` defines the following methods:

- `int getMajorDeviceClass()` – retrieves the major device class.
- `int getMinorDeviceClass()` – retrieves the minor device class.
- `int getServiceClasses()` – retrieves the major service classes

Listing 7. Using the `DeviceClass` methods.

```
static final NLDMSC = 0x22000; // Networking, Limited Discoverable Major Service Class
static final PHONE_MAJOR_CLASS = 0x200;
static final CELLULAR_MINOR_CLASS = 0x04;
:
:
LocalDevice localDevice;
DeviceClass deviceClass;
:
:
try {
    // Get local device
    localDevice = LocalDevice.getLocalDevice();
    // Get class of device for the local device
    deviceClass = localDevice.getDeviceClass();
    // Do appropriate processing depending on the class of device
    if (deviceClass.getMajorDeviceClass() == PHONE_MAJOR_CLASS) {
        if (deviceClass.getMinorDeviceClass() == CELLULAR_MINOR_CLASS) {
            :
            :
        }
    }
} catch (Exception e) {
    ... Handle Exception
}
```

5.7.4. Gathering Bluetooth Properties

The Java APIs for Bluetooth defines a set of properties for the local device that can be discovered by using `LocalDevice.getProperty()` method:

```
LocalDevice locaDevice = LocalDevice.getLocalDevice();
String apiVer = localDevice.getProperty("bluetooth.api.version");
```

The following table summarizes the Strings for the Java APIs for Bluetooth properties:

Table 4. Local Device Properties.

Property	Description
`bluetooth.api.version`	The version of the Java APIs for Bluetooth wireless technology that is supported.
`bluetooth.l2cap.receiveMTU.max`	The maximum `ReceiveMTU` size in bytes supported in L2CAP.
`bluetooth.connected.devices.max`	The maximum number of connected devices supported (will include

	parked devices).
`bluetooth.connected.inquiry`	Is inquiry allowed during a connection?
`bluetooth.connected.page`	Is paging allowed during a connection?
`bluetooth.connected.inquiry.scan`	Is inquiry scanning allowed during connection?
`bluetooth.connected.page.scan`	Is page scanning allowed during connection?
`bluetooth.master.switch`	Is master/slave switch allowed?
`bluetooth.sd.trans.max`	Maximum number of concurrent service discovery transactions.
`bluetooth.sd.attr.retrievable.max`	Maximum number of service attributes to be retrieved per service record.

5.8. Registering a Service

Service registration is automatically taken care of by the Java Bluetooth API implementation. Calling `Connector.open()` automatically creates a service record for the service. A subsequent call to `StreamConnectionNotifier` or L2CAPConnetionNotifier `acceptAndOpen()` method adds the previously created service record to the Service Discovery Database (SDDB), making the device connectable and able to respond to connection attempts by clients.

5.9. Overview of the Device and Service Discovery APIs

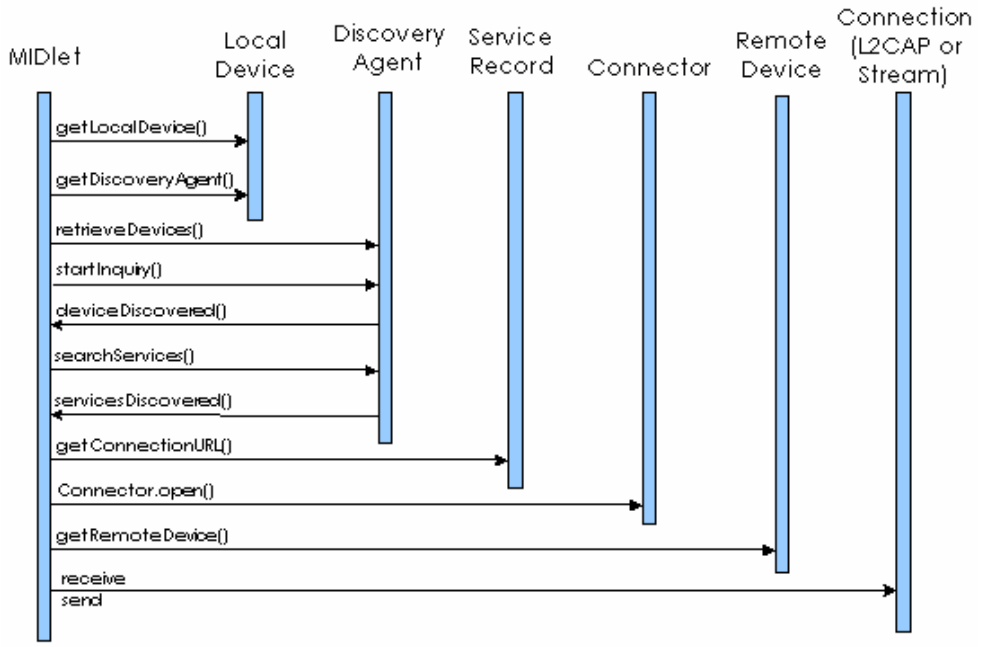
Device and service discovery is probably the most complex and involved part of the Java APIs for Bluetooth. The main discovery API consists of the following classes and interfaces: the `DiscoverAgent` class, the `DiscoveryListener` interface, the `RemoteDevice`, and the `ServiceRecord` interface:

5.9.1. The MIDlet

A JSR-82 capable MIDlet uses the `DiscoveryAgent`, which provides methods to perform device and service discovery. The MIDlet would also implement the `DiscoveryListener` interface to be notified when devices and services are discovered. To retrieve the device's `DiscoveryAgent`, the MIDlet invokes the static method `LocalDevice.getDiscoveryAgent()`.

A MIDlet begins the discovery phase by invoking the `DiscoveryAgent.startInquiry()` method, which places the device in inquiry mode. As devices and/or services are discovered, the `DiscoveryAgent` notifies the MIDlet by invoking the MIDlet's discovery callback methods `deviceDiscovered()` and `servicesDiscovered()`. Because inquiries are time consuming, before initiating an inquiry a MIDlet typically invokes the `DiscoveryAgent` methods `retrieveDevices()` and `searchServices()` to search the local cache for previously discovered devices and or services. Figure 8 shows a basic discovery sequence diagram:

Figure 8. Sequence Diagram - Using the Discovery API.



5.9.2. The DiscoveryAgent

The `DiscoveryAgent` provides methods to perform device and service discovery. The following methods are defined:

- `boolean cancelInquiry(DiscoveryListener listener)` – removes the device from inquiry mode.
- `boolean cancelServiceSearch(int transID)` – cancels the service search transaction that has the specified transaction ID.
- `RemoteDevice[] retrieveDevices(int option)` – returns an array of Bluetooth devices that have either been found by the local device during previous inquiry requests or been specified as a pre-known device, depending on the argument.
- `int searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev, DiscoveryListener discListener)` – searches for services on a remote Bluetooth device that have all the `UUIDs` specified in `uuidSet`.
- `java.lang.String selectService(UUID uuid, int security, boolean master)` – attempts to locate a service that contains `uuid` in the `ServiceClassIDList` of its service record.
- `boolean startInquiry(int accessCode, DiscoveryListener listener)` – places the device into inquiry mode.

5.9.3. The DiscoveryListener

The `DiscoveryListener`, which provides callbacks that are invoked when devices and services are discovered, is implemented by the MIDlet. The following callbacks are defined:

- `void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)` – called when a device is found during an inquiry.
- `void inquiryCompleted(int discType)` – called when an inquiry is completed.
- `void servicesDiscovered(int transID, ServiceRecord[] servRecord)` – called when service(s) are found during a service search.

- `void serviceSearchCompleted(int transID, int respCode)` – called when a service search is completed or was terminated because of an error.

5.9.4. The ServiceRecord

The `ServiceRecord` describes a remote service or a `RemoteDevice`. It provides methods to get the service attributes, its connection URL, the host remote device, as well as ways to change the Service Discovery Database:

- `int[] getAttributeIDs()` – returns the service attribute IDs whose value could be retrieved by a call to `getAttributeValue()`.
- `DataElement getAttributeValue(int attrID)` – returns the value of the service attribute ID, provided it is present in the service record. Otherwise, this method returns null.
- `java.lang.String getConnectionURL(int requiredSecurity, boolean mustBeMaster)` – returns a String, including any optional parameters that a client can use to connect to the service described by this `ServiceRecord`.
- `RemoteDevice getHostDevice()` - returns the remote Bluetooth device that populated the service record with attribute values.
- `boolean populateRecord(int[] attrIDs)` – retrieves the values by contacting the remote Bluetooth device. The values returned are a set of service attribute IDs for a service that is available on the Bluetooth device.
- `boolean setAttributeValue(int attrID, DataElement attrValue)` – modifies this `ServiceRecord` to contain the service attribute defined by the attribute-value pair (`attrID`, `attrValue`).
- `void setDeviceServiceClasses(int classes)` – used by a server application to indicate the major service class bits that should be activated in the server's `DeviceClass` when this `ServiceRecord` is added to the SDDB.

The following listing shows segments of a MIDlet that uses the discover API.

Listing 8. Example Using the Discovery API.

```
public class MyMIDlet implements DiscoveryListener {

    LocalDevice localDevice = LocalDevice.getLocalDevice();
    DiscoveryAgent discoveryAgent = localDevice.getDiscoveryAgent();
    RemoteDevice[] devList;
    Vector deviceList = new Vector(); // vector of discovered devices
    ServiceRecord serviceRecord;
    :
    :
    /** Helper method to beging the discovery phase of remote services and devices */
    private void discover() {
        // Retrieve devices found on a recent inquiry.
        devList = discoveryAgent.retrieveDevices(DiscoveryAgent.CACHED);
        if (devList != null) {
            serviceRecord = searchServices(devList);
```

```
        if (serviceRecord == null) {
            // Retrieve pre-known devices .
            devList = discoveryAgent.retrieveDevices(DiscoveryAgent.PREKNOWN);
            if (devList != null) {
                serviceRecord = searchServices(devList);
            }
        }
    }
    if (serviceRecord == null) {
        serviceAgent.startInquiry(DiscoveryAgent.GIAC, this);
    }
}

/* Searches for services of interested on remoted devices */
private boolean searchServices(RemoteDevice[] devList) {

    UUID[] searchList = new UUID[2];

    // Add the UUID for L2CAP to make sure that the service record
    // found will support L2CAP.  This value is defined in the
    // Bluetooth Assigned Numbers document.
    searchList[0] = new UUID(0x0100);

    // Add the UUID for our service of interest.
    searchList[1] = new UUID(MY_SERVICE_UUID, false);

    //  Search the device list for our service of interest.
    for (int i = 0; i < devList.length; i++) {
        try {
            int trans;
            trans =
               discoveryAgent.searchServices(null, searchList, devList[i], this);
        } catch (BluetoothStateException e) {
        }
    }
}

/*********************************/
/** DiscoveryListener Callbacks **/
/*********************************/
```

```
    /** Invoked by the implementation when services have been discovered */
    public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {
        if (serviceRecord != null) return;
        // Service of interest found. Just use the first one...
        serviceRecord = servRecord[0];
    }

    /** Invoked by the implementation when devices have been discovered */
    public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
        deviceList.addElement(btDevice); // keep track of discovered devices
    }

    /** Invoked by the implementation when service serach has been completed */
    public void serviceSearchCompleted(int transID, int respCode) {
        :
    }

    /** Invoked by the implementation when inquiry phase has been completed */
    public void inquiryCompleted(int discType) {
        :
    }
}
```

5.10. Security

The Java APIs for Bluetooth supports secure client and server connections. Server connections can be specified as secure before or after the connection is established.

To specify a secure connection before it gets established, a connection URL that specifies the appropriate security parameters must be built. For example, the following URL creates a RFCOMM secure server connection that must be authenticated and encrypted:

```
String url = "btspp://localhost:3B9FA89520078C303355AAA694238F07;
authenticate=true;encrypt=true";
```

To change the security settings for an already established connection, use the following `RemoteDevice` methods:

- `remoteDevice.authenticate()` - Attempts to authenticate this `RemoteDevice`.
- `remoteDevice.encrypt(con, true)` - Attempts to turn encryption on or off for an existing connection.
- `remoteDevice.authorize(con)` - Determines if this `RemoteDevice` should be allowed to continue to access the local service provided by the Connection.

To test if a connection is secure, use the following `RemoteDevice` methods:

- `boolean isAuthenticated()` – determines if this `RemoteDevice` has been authenticated.

- `boolean isAuthorized(javax.microedition.io.Connection conn)` – determines if this `RemoteDevice` has been authorized previously by the BCC of the local device to exchange data related to the service associated with the connection.
- `boolean isEncrypted()` – determines if data exchanges with this `RemoteDevice` are currently being encrypted.
- `boolean isTrustedDevice()` – determines if this is a trusted device according to the BCC.

The client can also indicate encrypt and authenticate parameters in the `Connector.open()` connection URL string. In addition, if the client is connecting to a remote *service*, the client can use the `ServiceRecord.getConnectionURL()` method to get a properly formed connection URL for the remote service, including a URL that indicates a secure connection. The definition of `getConnection()` is a follows:

- `java.lang.String getConnectionURL(int requiredSecurity, boolean mustBeMaster)` – Returns a String including optional parameters that can be used by a client to connect to the service described by this `ServiceRecord`.

The following code snippet shows how to use `getConnectionURL()` to get a URL connection with security parameters:

```
ServiceRecord serviceRecord;
:
:
// Once the ServiceRecord for the remote servce has been retrieved,
// use getConnectionURL to get the connection URL.
String url= serviceRecord.getConnectionURL(ServiceRecord.AUTHENTICATE_ENCRYPT,
false);
```

Finally, in terms of security, a JSR-82 capable MIDlet must be signed as trusted code in order to access the handset's Bluetooth stack.

6. The P900 J2ME SDK

You can use the Sony Ericsson P900 J2ME SDK to develop and test your P900 MIDP Bluetooth-enabled applications. The P900 J2ME SDK extends the Sun Wireless Toolkit version 2.0 or later, with P900 skins and properties files, the Bluetooth API, and sample code. You can find the Sony Ericsson's P900 J2ME SDK in the Sony Ericsson's Java docs and tools.

7. Resources

7.1. Abbreviations

Acronym	Meaning

API	Application Programming Interface
CBS	Cell Broadcast Service
CLDC	Connected Limited Device Configuration
IDE	Integrated Development Environment
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JCP	Java Community Process
JTWI	Java Technology for the Wireless Industry
JRE	Java Runtime Environment
MIDP	Mobile Information Device Profile
SDK	Software Development Kit
SIG	Special Interest Group
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

7.2. Further Information and Links

Sony Ericsson Developer World – www.sonyericsson.com/developer
JSR 82 specification Java Community Page - www.jcp.org/en/jsr/detail?id=82
Bluetooth official web sites– bluetooth.com and bluetooth.org
Sony Ericsson's Java docs and tools - http://www.sonyericsson.com\developer\site\global\docstools\java\p_java.jsp
Java Tips, Tricks & Code - Controlling Bluetooth Mini Race Car from the P900, included with this training material - www.sonyericsson.com/developer/site/global/techsupport/tipstrickscode/java/p_tips_java_1202.jsp
Java Tips, Tricks & Code - Bluepad, a Bluetooth shared sketch board for P900 phones included with this training material - www.sonyericsson.com/developer/site/global/techsupport/tipstrickscode/java/bluepad+-+a+bluetooth+shared+sketch+board+for+the+p900+phones.jsp
J2ME MIDP 2.0 specification – http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html
Sony Ericsson P900 J2ME SDK - www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Sony Ericsson Java Developer's Guidelines –

www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Sony Ericsson P900 Java MIDP 2.0 Kit - www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Developing MIDlets with Sony Ericsson J2ME™ SDK and Sun ONE Studio, Borland JBuilder or Metrowerks CodeWarrior - www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Adapting your MIDlets to the Sony Ericsson T610/618 - www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Optimizing J2ME applications for the T610 and Z600 Sony Ericsson series– www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Wireless Application Programming with J2ME and Bluetooth – http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/

7.3. The Bluetooth Car Application 

Included with this training material is the Bluetooth Car sample application, which can also be found in the Sony Ericsson's Java Tips, Tricks & Code website.

The Bluetooth Car demo application, written in J2ME MIDP 2.0 and the JSR-82 Bluetooth API, allows you to control the Sony Ericsson Bluetooth Car-100 Mini Race Car from a Bluetooth enabled cellphone such as the P900. To build the Bluetooth Car demo application you need the MIDP 2.0 Software Development Kit for the P900 that includes the P900 device emulator for Sun Wireless Toolkit 2.0, the Java APIs for Bluetooth, and Developers' Guidelines describing P900 MIDP 2.0 support.

The Bluetooth Car demo consists of the following Java classes:

- `BluetoothCar` - The main program that contains the standard MIDP `startApp()`, `pauseApp()`, and `destroyApp()` methods. The `startApp()` method creates the Car API, GUI, and `SPPServer` objects.

- Car API - This class allows the `CarConnection` and GUI to communicate with each other. It provides getter and setter methods that allow the GUI class to control the Bluetooth car and check its status.

- `CarConnection` - This starts a communication connection with the car, retrieves user commands and sends them to the car, and maintains the status of the car. The `processEvents()` method reads the stream of user commands and converts it into a stream of control commands required by the Bluetooth car.

- `GUI` - The Graphic User Interface (GUI) provides the user with graphics and sound. The screen consists of a steering wheel for turning the car, and a gear-shift for changing the speed and direction. The steering wheel allows the user to turn the car hard-left, left, straight, right, and hard-right and the gear-shift changes the speed between first, second, third gear, neutral and reverse.

- `SPPServer` - This class uses the JSR-82 APIs. Method `defineService()` creates an RFCOMM SPP (Serial Port Profile) server connection, waits for the client to connect and then creates a connection to the car.

7.4. Bluepad - a Bluetooth shared sketch board for P900 phones

Included with this training material is the Bluepad sample application, which can also be found in the Sony Ericsson's Java Tips, Tricks & Code website.

The BluePad application is an example Bluetooth MIDlet that was designed for the Sony Ericsson P900 phone. The application requires two phones with support for JSR-82 and a touch screen. You should run the MIDlet on two P900 phones. Connect one phone as the server and the other as the client. After the phones have connected, anything that you draw on either screen will be displayed on both phones. The Yellow/Red icon in the top left corner of the screen indicates the status of the Bluetooth connection.

Before you start the application, make sure that you have Bluetooth switched ON in the control panel.

The Bluetooth Pad demo application was written in J2ME MIDP 2.0 with the JSR-82 Bluetooth API and contains the following parts:

- `BluePadClient` - starts a Bluetooth client connection and attempts to connect to the server on the specified remote device. It handles all client-specific functionality.

- `BluePad` - creates a canvas which is a scratch pad or white board when connected to another phone running this same MIDlet. Anything you draw on one screen will automatically appear on the other screen. The mobile phone must support touch screen and the Bluetooth API.

- `BluePadServer` - starts a Bluetooth server and waits for a client connection to arrive. It handles the server-specific funcionality.

- `BluePad Canvas` - extends canvas and implements support for the drawing methods. This MIDlet creates a canvas which is a scratch pad or white board when connected to another phone running this same MIDlet. Anything you draw on one screen will automatically appear on the other screen. The mobile phone must support touch screen and the Bluetooth API.