

---

---

# Bluetooth

## *Part 4: Link Manager and Java ME Programming*

**Kjell Jørgen Hole**

UiB



Last updated 02.03.09

Mail: [Kjell.Hole@ii.uib.no](mailto:Kjell.Hole@ii.uib.no)

URL: [www.kjhole.com](http://www.kjhole.com)

- Definition of *Link Manager* (**LM**)
- Link setup and shutdown
- Master/Slave switch
- Multi-slot packets
- Other supported features
- More on the Java ME environment
- Java ME programming with JSR-82 APIs

# Definition of Link Manager

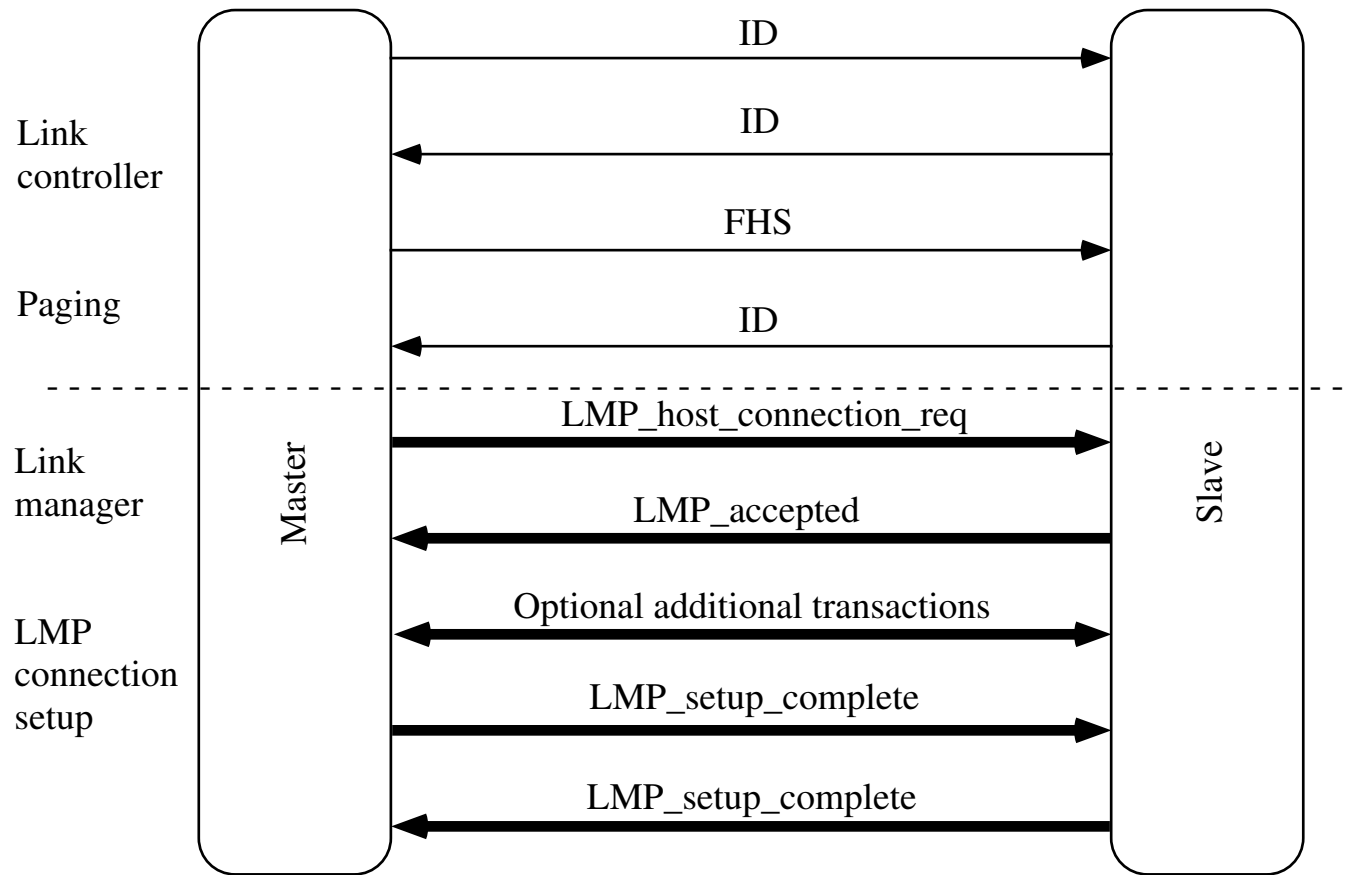
---

- LM translates commands from the *Host Controller Interface* into operations at baseband level, managing the following operations:
  - attaching Slaves to a piconet and allocating Logical Transport ADDresses (LT\_ADDRs)
  - breaking connections to detach Slaves from a piconet
  - configuring link including controlling Master/Slave switches
  - establishing ACL and SCO links
  - putting connections in low-power modes

- A LM communicates with other LMs on other devices using the *Link Management Protocol* (**LMP**)
- LMP defines a set of messages, or *Protocol Data Units* (**PDUs**), whose payload bodies contain the following fields:
  - single bit Transaction Identifier equal to 0 (1) for PDU sent from Master (Slave)
  - 7 or 15 bit *Operation Code* (OpCode) defining type of message being sent
  - message parameters

- LMP PDUs are transferred as single slot packets and are distinguished by a reserved value in the LLID field of the baseband payload header
- The PDUs are filtered out and interpreted by LM on the receiving side and are not propagated to higher layers
- LM PDUs have higher priority than user data, i.e., if the LM needs to send a message, it shall not be delayed by the L2CAP traffic
- There is no need to explicitly acknowledge LMP PDUs since LC (Link Controller) provides a reliable link

- LM establishes ACL links by controlling the baseband; then LMP messages can be used to establish SCO and eSCO links across an existing ACL connection
- LM maintains data on Slaves to which it has allocated LT\_ADDR's
- Figure 4-1 shows the messages needed to set up ACL connection
  - the optional transactions include procedures for pairing, authentication and encryption



**Figure 4-1** LMP message sequence chart for ACL link setup

# SCO Link Setup (1)

---

- Master or Slave uses an LMP SCO request (*LMP\_SCO\_req*) to initiate a SCO connection. The message has parameters:
  - timing control flags
  - $D_{SCO}$ , indicating when first SCO slot will happen
  - $T_{SCO}$ , interval which separates SCO slots
  - SCO packet type (HV1, HV2, HV3, DV)
  - air mode coding ( $\mu$ -law, A-law, CVSD)

## SCO Link Setup (2)

---

- A Master may have three active SCO links
- The more active SCO links, the less freedom the Master has to assign other time slots
- If a Slave chooses time slots, it may pick slots assigned by the Master to other devices
- When a Slave sends an *LMP\_SCO\_req*, the Master ignores the timing parameters and replies with an *LMP\_SCO\_req* with valid parameters which the Slave may acknowledge

- Any time a Master or a Slave wishes to shut down a link, it sends an *LMP\_detach* command. Possible reasons for this command are:
  - user ended connection
  - low resources
  - about to power off

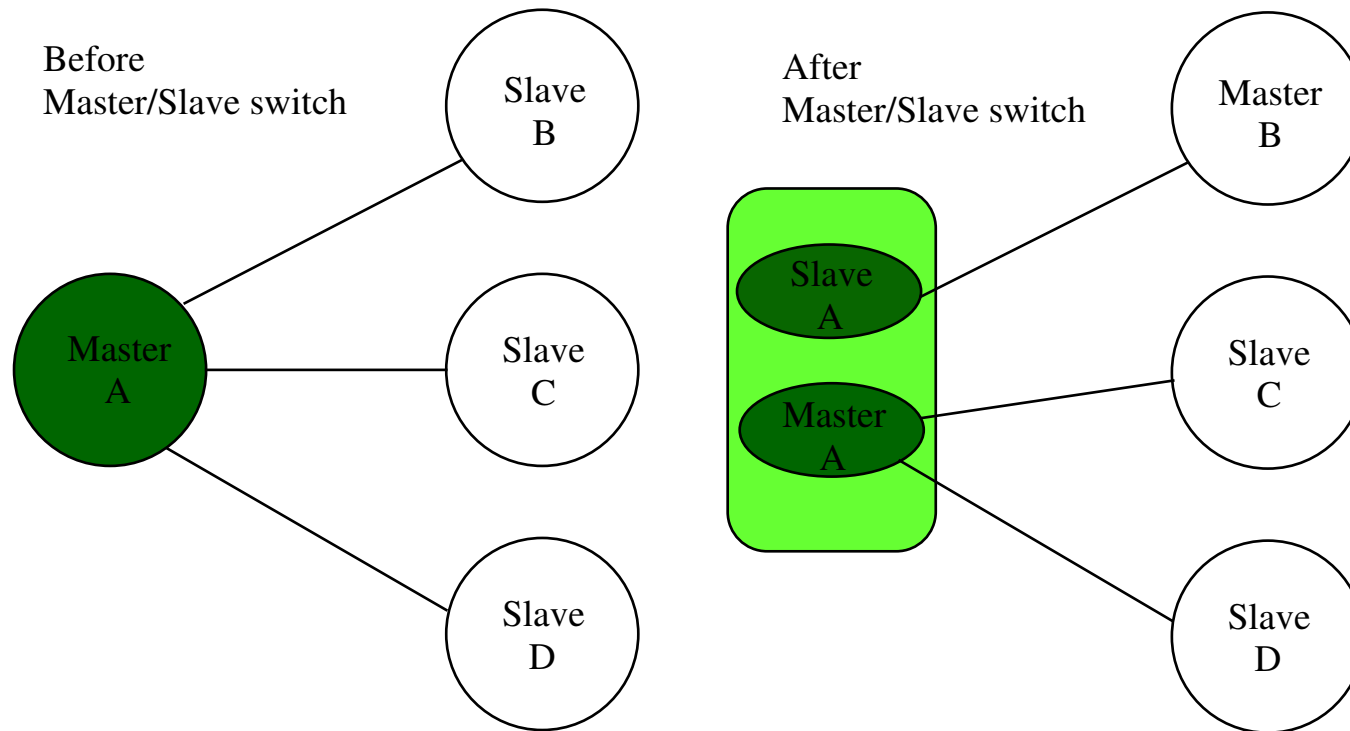
## Master/Slave Switch

---

KJhole.com

Figure 4-2 shows a Master with three Slaves. The Master accepts a Master/Slave switch from one Slave. The former Master now has a dual role as Master for two Slaves and Slave to one Master.

Note that the piconet is split into two piconets, joined in a scatternet



**Figure 4-2** Piconet configuration before and after Master/Slave switch

- Multi-slot packets make more efficient use of bandwidth
- Multi-slot packets may *not* be used
  - on noisy links because they are more likely to be corrupted by interference
  - because SCO links may not leave sufficient space between their slots
- The default for any new connection is single slot packets only. The Master and Slave may negotiate a multi-slot packet size

- If the *Receive Signal Strength Indication* (RSSI) value differs too much from the preferred value of a device, it can request an increase or a decrease of the other device's TX power
- The power adjustment requests can be made at anytime following a successful baseband paging procedure
- Not all devices support power control

# Other LM Supported Features

---

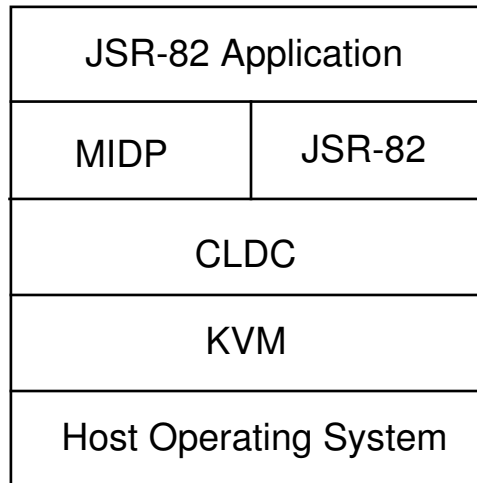
KJhole.com

- Provides mechanism for negotiating encryption mode and coordinating encryption keys
- Supports low-power connection modes
- Provides Quality of Service capabilities
- Supports test mode

# Java ME

# More on the Java ME Environment

---



**Figure 4-3** Java ME environment

- CLDC 1.1 has the following limitations:
  - *No finalization*: `Object.finalize()` method not called by the garbage collector
  - *Limited error handling*: CLDC defines only four error classes:

`java.lang.Error`

`java.lang.OutOfMemoryError`

`java.lang.VirtualMachineError`

`java.lang.NoClassDefFoundError`

- CLDC 1.1 has the following limitations:
  - No object serialization
  - No remote method invocation (allows remote objects to be used as if they were local objects)
  - No thread groups or daemon threads (threads terminated by the JVM if no user threads are running)

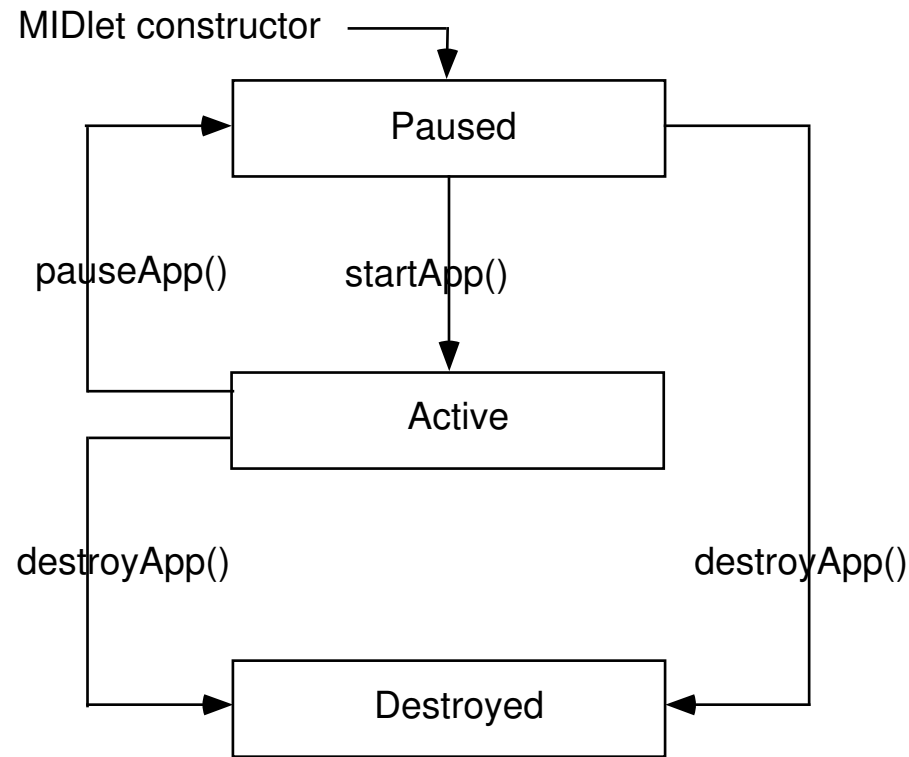
- CLDC 1.1 contains the following packages:
  - `java.io`, classes for input and output through data streams
  - `java.lang`, classes fundamental to the Java language
  - `java.lang.ref`, support for weak references
  - `java.util`, collection classes, and the date and time facilities
  - `java.microedition.io`, classes for the GCF
- The three first classes are subsets of Java SE classes

- MIDP addresses the following areas (not covered by CLDC):
  - **Application life cycle management**, defined by the `javax.microedition.midlet` package
  - **User interface and events**, defined by the `javax.microedition.lcdui` package
  - **Network connectivity**, extends GCF by providing the `HttpConnection` interface
  - **Storing data on device**, defined by the `javax.microedition.rms` package

- MIDP also has packages for game development, sound, and certificates

**MIDlet** consists of a public class definition that extends the `javax.microedition.midlet.MIDlet` abstract class. MIDlets are similar to servlets in that they facilitate a “fill-in-the-method” program structure

- Any MIDlet implements three *life cycle methods*:
  - startApp(): Initialize objects and set display
  - pauseApp(): Pause active threads, store data
  - destroyApp(): Free or close resources, store data for future use
- The above methods are called by the **application manager** to start up the MIDlet, pause it, or destroy it
  - the application manager is *preinstalled* on a MIDP device



**Figure 4-4** MIDlet state transitions

# The MIDlet Life Cycle (1)

---

KJhole.com

- A MIDlet goes through the following states:
  1. When the MIDlet is about to be run, an instance is created. The MIDlet's constructor is run, and the MIDlet is in the *Paused* state
  2. Next, the MIDlet enters the *Active* state after the application manager calls `startApp()`

## The MIDlet Life Cycle (2)

---

KJhole.com

3. While the MIDlet is Active, the application manager can suspend its execution by calling `pauseApp()`. This puts the MIDlet back in the Paused state. A MIDlet can place itself in the Paused state by calling `notifyPaused()`
4. The application manager can terminate the execution of the MIDlet by calling `destroyApp()`, at which point the MIDlet is *destroyed* and patiently awaits garbage collection. A MIDlet can destroy itself by calling `notifyDestroyed()`

- The application manager installs and runs the MIDlet to control the behavior of the MIDlet. It must be able to
  - retrieve a MIDlet suite from somewhere, possibly over a Bluetooth link
  - install the MIDlet suite on the MIDP device
  - perform version management
  - launch MIDlet from MIDlet suite
  - delete a previously installed MIDlet suite

# Definitions (1)

---

KJhole.com

**Java ARchive (JAR) file** File used to package Java classes into a compressed archive for more efficient distribution

**MIDlet suite** Two or more MIDlets that are packaged in a JAR file. These MIDlets can share resources at runtime

**JAR manifest** Defines the attributes of the MIDlets in a MIDlet suite. JAR manifest is part of the MIDlet suite

**Java Application Descriptor (JAD)** Text file containing MIDlet attributes. Since the JAR manifest is part of the MIDlet suite's JAR file, the complete JAR file must be downloaded to a device to check if there is enough space on the device for the MIDlet suite. To avoid this downloading, a JAD is created and used by the Java Application Manager to determine the space requirement

- See the User's Guide for the Sun Java Wireless Toolkit for more information

- 1) Write MIDlet
- 2) Compile source code
- 3) Preverify class file(s)\*
- 4) Package it into a JAR file
- 4.1) Sign JAR file (New and optional, MIDP 2.0)
- 5) Create JAD file
- 6) Publish the MIDlet (JAD and JAR)
- 7) Install MIDlet on the device or emulator

\*Most of the class verification in the Java SE JVM is removed from the KVM

- **Sun Java Wireless Toolkit** (formerly known as the J2ME Wireless Toolkit), [java.sun.com/products/sjwtoolkit/](http://java.sun.com/products/sjwtoolkit/)
  - supports CLDC, MIDP, and JSR-82 Java APIs for Bluetooth
  - provides software emulation of CLDC/MIDP devices
  - contains a minimal development environment, KToolBar, for compiling, packaging, and executing MIDP applications
  - use third-party editor for your Java source files

- **eclipse** and **NetBeans** IDEs
  - develop, debug, and deploy mobile applications based on Java ME
  - Both IDEs can incorporate Sun Java Wireless Toolkit
- see <http://www.eclipse.org>
- see <http://www.netbeans.org>

- The HelloClient MIDlet locates a HelloServer MIDlet and sends the text “Hello World” to the server which displays the text on its screen. Example demonstrates *MIDP UI programming*
- Needed package and import statements for *all* classes:

```
package com.jabwt.book; // Code found i textbook by Thompson et. al.  
  
import javax.microedition.io.*; // CLDC, also in MIDP javadoc  
import javax.microedition.lcdui.*; // MIDP  
import javax.microedition.midlet.*; // MIDP  
import javax.bluetooth.*; // JSR-82
```

```
public class BluetoothMIDlet extends MIDlet implements
    Runnable, CommandListener {
    public BluetoothMIDlet() {} // Empty constructor

    public void startApp() throws MIDletStateChangeException {
        new Thread(this).start(); } // Starts new thread

    public void pauseApp() {}
    public void destroyApp(boolean unconditional) {}

    public void run() {}
    public void commandAction(Command c, Displayable d) {
        notifyDestroyed(); } // Destroys MIDlet
}
```

- Utilizes `java.lang.Runnable` interface to implement multi-threading
  - defines single method `run()`
- A thread does not start running automatically at creation time. The `start()` method must be called first
  - `start()` then calls the `run()` method
- Interface `CommandListener` from package `javax.microedition.lcdui` is used to receive a high-level event
- `BluetoothMIDlet` is used by the following client and the server

```
public class HelloClient extends BluetoothMIDlet {

    public void run() {        // Creates Form and adds Exit Command
        Form f = new Form("Client");        // A screen
        f.addCommand(new Command("Exit",Command.EXIT,1));
        f.setCommandListener(this);// listener in 'this' object
        Display.getDisplay(this).setCurrent(f);// Gets Display object
            // unique to 'this' MIDlet and displays Form on screen
        try { // Retrieve connection string to connect to server
            LocalDevice local = LocalDevice.getLocalDevice();// B. manager
            DiscoveryAgent agent = local.getDiscoveryAgent();// device disc.
            String connString = agent.selectService(
                new UUID("86b4d249fb8844d6a756ec265dd1f6a3", false),
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
        }
    }
}
```

```
if (connString != null) {
    try {
        // Connect to the server and send 'Hello World'
        StreamConnection conn = (StreamConnection)//MIDP
            Connector.open(connString);

        OutputStream out = conn.openOutputStream();
        out.write("Hello World".getBytes());
        out.close();
        conn.close();

        f.append("Message sent correctly");
    } catch (IOException e) {
        f.append("IOException:" + e.getMessage());
    }
}
```

```
    } else {  
        // Unable to locate a service so just print an error  
        // message on the screen  
        f.append("Unable to locate service");  
    }  
} catch (BluetoothStateException e) {  
    f.append("BluetoothStateException: ");  
    f.append(e.getMessage());  
}  
}  
}
```

```
public class HelloServer extends BluetoothMIDlet {  
  
    // Creates a server object.  Accepts a single connection from a  
    // client and prints the data sent from the client to the screen  
    public void run() {  
        // Creates a Form and adds the Exit command to the Form  
        Form f = new Form("Server");  
        f.addCommand(new Command("Exit",Command.EXIT,1));  
        f.setCommandListener(this);  
        Display.getDisplay(this).setCurrent(f);  
    }  
}
```

```
try {
    // Make the local device discoverable for the client to locate
    LocalDevice local = LocalDevice.getLocalDevice();
    if (!local.setDiscoverable(DiscoveryAgent.GIAC)) {
        f.append("Failed to change to the discoverable mode");
        return;
    }
    // Create a server connection object to accept
    // a connection from a client
    StreamConnectionNotifier notifier =
        (StreamConnectionNotifier)
        Connector.open("btspp://localhost:" +
            "86b4d249fb8844d6a756ec265dd1f6a3");
```

```
// Accept a connection from the client
StreamConnection conn = notifier.acceptAndOpen();
// Open the input to read data from
InputStream in = conn.openInputStream();
ByteArrayOutputStream out = new ByteArrayOutputStream();

// Read the data sent from the client until the end of stream
int data;
while ((data = in.read()) != -1) { out.write(data); }

// Add the text sent from the client to the Form
f.append(out.toString());
```

```
// Close all open resources
in.close();
conn.close();
notifier.close();
} catch (BluetoothStateException e) {
    f.append("BluetoothStateException:" + e.getMessage());
} catch (IOException e) {
    f.append("IOException:" + e.getMessage());
}
}
}
```

- LM carries out the following tasks:
  - attaching Slaves to a piconet
  - breaking connections to detach Slaves from a piconet
  - configuring links including controlling Master/Slave switches
  - establishing ACL and SCO links
  - supporting QoS
  - putting connections in low-power modes
- Use eclipse or NetBeans IDE with the Sun Java Wireless Toolkit for Java/Bluetooth development