



Lab session 3:

Radio Communication (part 1).

The aim of this 3rd lab session is to get in touch with the radio communication elements and the different options for spot communication that are available for Sun SPOTs. For using the different elements related to radio communication, you will eventually need to import the following packages:

- `import com.sun.spot.io.j2me.radiogram.*;`
- `import com.sun.spot.io.j2me.radiostream.*;`
- `import com.sun.spot.peripheral.NoRouteException;`
- `import com.sun.spot.peripheral.TimeoutException;`
- `import com.sun.spot.peripheral.radio.*;`
- `import java.io.DataInputStream;`
- `import java.io.DataOutputStream;`
- `import java.io.IOException;`
- `import javax.microedition.io.Connector;`
- `import javax.microedition.io.Datagram;`
- `import javax.microedition.io.DatagramConnection;`

This document is based on the documentation included at the Sun SPOT Developer's Guide ('Red' release version).¹

Introduction

J2ME uses a Generic Connection Framework (GCF) to create connections (such as HTTP, datagram, or streams) and perform I/O. The current version of the Sun SPOT SDK uses the GCF to provide radio communication between SPOTs, routed via multiple hops if necessary, using a choice of two protocols.

The radiostream protocol provides reliable, buffered, stream-based communication between two devices.

The radiogram protocol provides datagram-based communication between two devices. This protocol provides no guarantees about delivery or ordering. Datagrams sent over more than one hop could be silently lost, be delivered more than once, and be delivered out of sequence. Datagrams sent over a single hop will not be silently lost or delivered out of sequence, but they could be delivered more than once.²

The protocols are implemented on top of the MAC layer of the 802.15.4 implementation.

¹ Also available at <http://www.sunspotworld.com/docs/Red/spot-developers-guide.pdf>

² One aspect of the current behaviour that can be confusing occurs if a SPOT that was communicating over a single hop closes its connection but then receives a packet addressed to it. In this case the packet will be acknowledged at the MAC level but then ignored. This can be confusing from the perspective of another SPOT sending to the now-closed connection: its packets will appear to be accepted because they were acknowledged, but they will not be processed by the destination SPOT. Thus connections working over a single hop have slightly different semantics to those operating over multiple hops; this is the result of an important performance optimisation for single hop connections.

The radiostream protocol

The radiostream protocol is a socket-like peer-to-peer protocol that provides reliable, buffered stream-based IO between two devices.

To open a connection do:

```
RadiostreamConnection conn =  
(RadiostreamConnection)Connector.open("radiostream:<destAddr>:<portNo>");
```

where `destAddr` is the 64bit IEEE Address of the radio at the far end, and `portNo` is a port number in the range 0 to 255 that identifies this particular connection.³ Note that 0 is not a valid IEEE address in this implementation. The connection is opened using the default radio channel and default PAN identifier (currently channel 26, PAN 3). The section Radio properties shows how to override these defaults.

To establish a connection both ends must open connections specifying the same `portNo` and corresponding IEEE addresses.

Once the connection has been opened, each end can obtain streams to send and receive data, for example:

```
DataInputStream dis = conn.openDataInputStream();  
DataOutputStream dos = conn.openDataOutputStream();
```

Here's a complete example:

Program 1

```
RadiostreamConnection conn =  
(RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0006:100");  
DataInputStream dis = conn.openDataInputStream();  
DataOutputStream dos = conn.openDataOutputStream();  
  
try {  
    dos.writeUTF("Hello up there");  
    dos.flush();  
    System.out.println ("Answer was: " + dis.readUTF());  
} catch (NoRouteException e) {  
    System.out.println ("No route to 0014.4F01.0000.0006");  
}  
  
} finally {  
    dis.close();  
    dos.close();  
    conn.close();  
}
```

Program 2

```
RadiostreamConnection conn =  
(RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0007:100");  
DataInputStream dis = conn.openDataInputStream();  
DataOutputStream dos = conn.openDataOutputStream();  
  
try {  
    String question = dis.readUTF();  
    if (question.equals("Hello up there")) {  
        dos.writeUTF("Hello down there");  
    } else {  
        dos.writeUTF("What???");  
    }  
}
```

³ Ports in the range 0 to 31 are reserved for system use.

```

    }
    dos.flush();

} catch (NoRouteException e) {
    System.out.println ("No route to 0014.4F01.0000.0007");
} finally {
    dis.close();
    dos.close();
    conn.close();
}

```

Data is sent over the air when the output stream buffer is full or when a `flush()` is issued.

The `NoRouteException` is thrown if no route to the destination can be determined. The stream accesses themselves are fully blocking - that is, the `dis.readUTF()` call will wait forever (unless a timeout for the connection has been specified – see below).

There is no automatic handshaking when opening a stream connection, so if Program 2 is started after Program 1 it may miss the *"Hello up there"* message.

Behind the scenes, every data transmission between the two devices involves an acknowledgement. The sending stream will always wait until the MAC-level acknowledgement is received from the next hop. If the next hop is the final destination then this is sufficient to ensure the data has been delivered. However if the next hop is not the final destination then an acknowledgement is requested from the final destination, but, to improve performance, the sending stream does not wait for this acknowledgement; it is returned asynchronously. If the acknowledgement is not received despite retries a `NoMeshLayerAckException` is thrown on the next stream write that causes a send or when the stream is flushed or closed. A `NoMeshLayerAckException` indicates that a previous send has failed – the application has no way of knowing how much data was successfully delivered to the destination.

Another exception that you may see, which applies to both `radiostream` and `radiogram` protocols, is `ChannelBusyException`. This exception indicates that the radio channel was busy when the SPOT tried to send a radio packet. The normal handling is to catch the exception and retry the send.

The radiogram protocol

The radiogram protocol is a client-server protocol that provides datagram-based IO between two devices.

To open a server connection do:

```

RadiogramConnection conn =
    (RadiogramConnection) Connector.open("radiogram://:<portNo>");

```

where `portNo` is a port number in the range 0 to 255 that identifies this particular connection. The connection is opened using the default radio channel and default PAN identifier (currently channel 26, PAN 3). The section Radio properties shows how to override these defaults.

To open a client connection do:

```

RadiogramConnection conn =
    (RadiogramConnection)Connector.open("radiogram://<serveraddr>:<portNo>");

```

where `serverAddr` is the 64bit IEEE Address of the radio of the server, and `portNo` is a port number in the range 0 to 255 that identifies this particular connection.⁴ Note that 0 is not a valid IEEE address in this implementation. The port number must match the port number used by the server.

Data is sent between the client and server in datagrams, of type `Datagram`. To get an empty datagram you must ask the connection for one:

```
Datagram dg = conn.newDatagram(conn.getMaximumLength());
```

Datagrams support stream-like operations, acting as both a `DataInputStream` and a `DataOutputStream`. The amount of data that may be written into a datagram is limited by its length. When using stream operations to read data the datagram will throw an `EOFException` if an attempt is made to read beyond the valid data.

A datagram is sent by asking the connection to send it:

```
conn.send(dg);
```

A datagram is received by asking the connection to fill in one supplied by the application:

```
conn.receive(dg);
```

Here's a complete example:

Client end

```
RadiogramConnection conn =
    (RadiogramConnection)Connector.open("radiogram://0014.4F01.0000.0006:100");
Datagram dg = conn.newDatagram(conn.getMaximumLength());

try {
    dg.writeUTF("Hello up there");
    conn.send(dg);
    conn.receive(dg);
    System.out.println ("Received: " + dg.readUTF());
} catch (NoRouteException e) {
    System.out.println ("No route to 0014.4F01.0000.0006");
} finally {
    conn.close();
}
```

Server end

```
RadiogramConnection conn =
    (RadiogramConnection)Connector.open("radiogram://:100");
Datagram dg = conn.newDatagram(conn.getMaximumLength());
Datagram dgreply = conn.newDatagram(conn.getMaximumLength());

try {
    conn.receive(dg);
    String question = dg.readUTF();
    dgreply.reset(); // reset stream pointer
    dgreply.setAddress(dg); // copy reply address from input

    if (question.equals("Hello up there")) {
        dgreply.writeUTF("Hello down there");
    } else {
        dgreply.writeUTF("What???");
    }
}
```

⁴ Ports in the range 0 to 31 are reserved for system use.

```

    }
    conn.send(dgreply);
} catch (NoRouteException e) {
    System.out.println ("No route to " + dgreply.getAddress());
} finally {
    conn.close();
}

```

There are some points to note about using datagrams:

- Only datagrams obtained from the connection may be passed in send or receive calls on the connection. You cannot obtain a datagram from one connection and use it with another.
- A connection opened with a specific address can only be used to send to that address. Attempts to send to other addresses will result in an exception.
- It is permitted to open a server connection and a client connection on the same machine using the same port numbers. All incoming datagrams for that port will be routed to the server connection.
- Currently, closing a server connection also closes any client connections on the same port.

Using system allocated ports

Most applications use perhaps one or two ports with fixed numbers (remember that each SPOT can open many connections on the same port, as long as they are to different remote SPOTs). However, some applications need to open variable numbers of connections to the same remote SPOT, and for such applications, it is convenient to ask the library to allocate free port numbers as required. To do this, simply exclude the port number from the url. So, for example:

```

RadiogramConnection conn =
    (RadiogramConnection)Connector.open("radiogram://0014.4F01.0000.0006");

```

This call will open a connection to the given remote SPOT on the first free port. To discover which port has been allocated, do

```

byte port = conn.getLocalPort();

```

You can open server radiogram connections in a similar way:

```

RadiogramConnection serverConn =
    (RadiogramConnection)Connector.open("radiogram://");

```

The same port allocation process works with radiostream connections. However, for radiostream connections, the port number is allocated only after either an input or output stream is opened. So, for example

```

RadiostreamConnection conn =
    (RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0006");

// byte port = conn.getLocalPort(); // this would throw an exception as the
// port has not been allocated yet.

RadioInputStream is = (RadioInputStream)conn.openInputStream();
byte port = conn.getLocalPort(); // now port can be accessed from connection...
port = is.getLocalPort(); // ...or from the input stream.

```

Adjusting connection properties

The `RadiostreamConnection` and `RadiogramConnection` classes provide a method for setting a timeout on reception. For example:

```
RadiostreamConnection conn =
    (RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0006:100");
conn.setTimeout(1000); // wait 1000ms for receive
```

There are some important points to note about using this operation:

- A `TimeoutException` is generated if the caller is blocked waiting for input for longer than the period specified.
- Setting a timeout of 0 causes the exception to be generated immediately if data is not available.
- Setting a timeout of -1 turns off timeouts, that is, wait forever.

Broadcasting

It is possible to broadcast datagrams. By default, broadcasts are transmitted over two hops, so they may be received by devices out of immediate radio range operating on the same PAN. The broadcast is not inter-PAN. Broadcasting is not reliable: datagrams sent might not be delivered. SPOTs ignore the echoes of broadcasts that they transmitted themselves or have already received.

To perform broadcasting first open a special radiogram connection:

```
DatagramConnection conn =
    (DatagramConnection)Connector.open("radiogram://broadcast:<portnum>");
```

where `<portnum>` is the port number you wish to use. Datagrams sent using this connection will be received by listening devices within the PAN.

Note that broadcast connections cannot be used to receive. If you want to receive replies to a broadcast then open a server connection, which can be on the same port.

If you wish broadcasts to travel for more or less than the default two hops, do

```
((RadiogramConnection)conn).setMaxBroadcastHops(n);
```

where `n` is the required number of hops.

Broadcasting and basestations

As described in the section `Managing multiple MIDlets on the SPOT`, basestations may operate in either shared or dedicated mode. Imagine a remote SPOT, out of range of the basestation, which broadcasts a radiogram with the default two hops set. An intermediate SPOT that is in range of the basestation then picks up the radiogram, and relays it. Because the basestation receives the packet after two hops, it does not relay it.

In dedicated mode, the host application uses the basestation's address and therefore receives this radiogram. In shared mode, the host application is still one more hop

from the host process managing the basestation, and so does not receive the radiogram. Similar considerations apply to packets sent from host applications. For this reason, application developers using broadcast packets to communicate between remote SPOTs and host applications will need to consider the impact of basestation mode on the number of broadcast hops they set.

These considerations will also impact anyone that works directly with the 802.15.4 MAC layer or lower levels of the radio communications stack. At the MAC layer, all packets are sent single hop: using a dedicated basestation these will reach remote SPOTs, using a shared basestation they will not.

Port number usage

The valid range of port numbers for both the radio and radiogram protocols is 0 to 255. However, for each protocol, ports in the range 0 to 31 are reserved for system use; applications should not use port numbers in this range.

The following table explains the current usage of ports in the reserved range.

Port number	Protocol	Usage
8	radiogram://	OTA Command Server
9	radiostream://	Debugging
10	radiogram://	http proxy
11	radiogram://	Manufacturing tests
12	radiostream://	Remote printing (Master isolate)
13	radiostream://	Remote printing (Child isolate)
14	radiogram://	Shared basestation discovery
20	radiogram://	Trace route server

Radio Communication (part 2): HTTP support

The http protocol is implemented to allow remote SPOT applications to open http connections to any web service accessible from a correctly configured host computer.

To open a connection do:

```
HttpConnection connection =
    (HttpConnection)Connector.open("http://host:[port]/filepath");
```

Where:

- host is the Internet host name in the domain notation, e.g. www.upc.edu or a numerical TCP/IP address
- port is the port number, which can usually be omitted (and then the default of 80 applies)
- filepath is the path/name of the resource being requested from the web server

Here's a complete example that retrieves the source html of the home page from the <http://www.sunspotworld.com> website:

```
HttpConnection connection =
    (HttpConnection)Connector.open("http://www.sunspotworld.com/");
connection.setRequestProperty("Connection", "close");

InputStream in = connection.openInputStream();
StringBuffer buf = new StringBuffer();
int ch;
while ((ch = in.read()) > 0) {
    buf.append((char)ch);
}
System.out.println(buf.toString());

in.close();
connection.close();
```

In order for the http protocol to have access to the specified URL, the device must be within radio reach of a basestation connected to a host running the Socket Proxy. This proxy program is responsible for communicating with the server specified in the URL. The Socket Proxy program is started by `ant socket-proxy` or `ant socket-proxy-gui`.

Configuring the HTTP Protocol

The http protocol is implemented as an HTTP client on the Sun SPOT using the socket protocol. When an http connection is opened, an initial connection is established with the Socket Proxy over radiogram on port 10 (or as specified in the MANIFEST.MF file of your project). The Socket Proxy then replies with an available radio port that the device connects to in order to start streaming data.

By default, a broadcast is used in order to find the basestation that will be used to open a connection to the Socket Proxy. In order to use a specific basestation add the following property to the project's MANIFEST.MF file:

```
com.sun.spot.io.j2me.socket.SocketConnection-BaseStationAddress: <IEEE address>
```

By default, radiogram port 10 is used to perform the initial connection to the Socket Proxy. This can be overridden by including the following property in the project's MANIFEST.MF file

```
com.sun.spot.io.j2me.socket.SocketConnection-BaseStationPort: <radiogram port>
```

If the host computer running the Socket Proxy is behind a proxy server, the device can still access the Internet if the following property is specified in the MANIFEST.MF:

```
com.sun.squawk.io.j2me.http.Protocol-HttpProxy: <proxyaddress>:<port>
```

where proxyaddress is the hostname or IP address of the HTTP proxy server and port is the proxy's TCP port.

Socket Proxy GUI mode

The SocketProxy program actually has two modes it can run in. The default as shown earlier, which presents a headless version. There is also a GUI version that allows you to view log information and see what is actually going on. To launch the proxy in GUI mode, issue the following command

```
ant socket-proxy-gui
```

This will present you with the following window:

- Basestation's serial port

The serial port that the basestation is connected to. This is automatically populated by the calling scripts.

- Radiogram port

The radiogram port used to establish the initial connection. This panel lets the user select the logging level to use. The log level affects which logs are to be displayed in the log panel.

1. System: log all that is related to the proxy itself
2. Error: log all that is an error (exceptions)
3. Warning: log all that is a warning
4. Info: log all extra information (incoming connections, closing connections)
5. I/O: log all I/O activities

- Radio port usage

The SocketProxy reserves a radio port for every Sun SPOT connected to it. This counter lets you monitor that activity. Whenever the SocketProxy runs out of radio ports, it will say so in here and stop accepting new connections until some ports get freed up.

The log level can be set with the socketproxy.loglevel property in the SDK's default.properties file. The log levels are the same as in the GUI mode but they are set in a complementary way. So if warning is selected as a level, the level will actually be system, error and warning. The default radiogram port used for the initial connection can be set with the socketproxy.initport property in the SDK's default.properties file.

Exercises (part I)

For doing these exercises you can use the code sample to be found at `[SpotSDKdirectory]/Demos/CodeSamples/RadioBroadcastSampleCode` and at the directory `[SpotSDKdirectory]/Demos/CodeSamples/RadioInputStreamSampleCode` as starting point or template.

- 1 Test (i.e., build, deploy and run) and study the source code of the demo included at `[SpotSDKdirectory]/Demos/CodeSamples/RadioBroadcastSampleCode`
- 2 Test (i.e., build, deploy and run) and study the source code of the demo included at `[SpotSDKdirectory]/Demos/CodeSamples/RadioInputStreamSampleCode`
- 3 Binary tennis

Using the two Sun SPOTs we provided you, make a program for making them play *binary tennis*. In binary tennis, one of the two spots is the leader and starts the game by displaying a zero in its LEDs, and sending also this number to the other spot. The game proceeds in the following way: every time a spot receives a number, it increments it, displays it in its LEDs, and sends the new number to the other spot. The leader can be chosen manually by the user (e.g., by pressing one of the switches of one of the spots), or it can be the result of an agreement between the spots.

- 4 Follow the leader of the class

For this exercise, all the spots in the class will be needed, and you must contribute to it with the two Sun SPOTs we provided you. One of them will be the elected leader but, a priori, a spot does not know whether it will be the leader or not. The chosen leader should broadcast messages to the other spots, and the others, as good servants, should echo what the leader says.

The broadcasted message could be information you like, e.g. a binary number (and thus they could also display it in the LEDs), or any string (and thus they could print it on the computer screen).

Exercises (part II)

5 Counting tags

Using your Sun SPOTs, retrieve the source HTML code of any web site you like and count the number of appearances of a concrete tag.

6 Show me where to go

Using your Sun SPOTs, retrieve the address you will find at the file <http://albcom.lsi.upc.edu/axs/dummy.txt> and use it to launch a browser with the location of that address on the [Google Maps](#). The address is composed by a street name and number, the name of a city, and the name of a country (e.g., Jordi Girona 1 Barcelona Spain).

The URL used in Google Maps for showing an specific location shares the following prefix:

```
http://maps.google.es/maps?f=q&source=s_q&hl=ca&geocode=&q=
```

followed by the words composing the address, which are joined using the symbol '+'. For example, for locating the address 'Jordi Girona 1 Barcelona Spain', a valid URL would be the following:

```
http://maps.google.es/maps?  
f=q&source=s\_q&hl=ca&geocode=&q=jordi+girona+1+Barcelona+Spain
```

7 Show me how to go

Using your Sun SPOTs, retrieve the address you will find at the file <http://albcom.lsi.upc.edu/axs/dummy2.txt> and use it to launch a browser with the route from this location to the previous one (i.e., the one in the dummy.txt file from the previous exercise) using also the [Google Maps](#).

The URL used in Google Maps for showing an router shares the following prefix:

```
http://maps.google.es/maps?f=d&source=s_d
```

followed by the indication of the source address and the destination address. The former is prefixed with the tag '&saddr=', while the latter is prefixed with the tag '&daddr='. For example, for going from 'Pau Gargallo 15 Barcelona Spain' to 'Jordi Girona 1 Barcelona Spain', a valid URL would be the following:

```
http://maps.google.es/maps?  
f=d&source=s\_d&saddr=Pau+Gargallo+15+Barcelona&daddr=Jordi+Girona+1+Barcel  
ona
```

By default, the route is calculated for going by car. If you want to show the router by foot, then the suffix '&dirflg=w' must be added (e.g., at the end of the whole URL).