Sun™ Small Programmable Object Technology (Sun SPOT) Developers' Guide

This version of the Guide refers to V2.0
(updated 16-April-2007)

Contents

Introduction

The purpose of this guide is to aid developers of applications for Sun SPOTs. The guide is divided into two sections.

Building and deploying Sun SPOT applications provides information about how to build, deploy and execute Sun SPOT applications. The topics covered include:

- Building and deploying simple applications
- Deploying applications you've received as jars from other developers
- Including properties and external resources through the manifest
- Setting up a base station to communicate with physically remote Sun SPOTs via the radio
- Using the base station to deploy and execute applications on remote Sun SPOTs
- Building and running your own application running on the host machine that communicates, via the base station, with remote Sun SPOTs
- Managing the keys that secure your Sun SPOTs against unauthorised access
- Sharing keys to allow a workgroup to share a set of Sun SPOTs.

Developing and debugging Sun SPOT applications provides information for the programmer. This includes

- A quick overview of the structure of Sun SPOT applications
- Using the Sun SPOT libraries to
    - Control the radio
    - Read and write persistent properties
    - Read and write flash memory
    - Access streams across the USB connection
    - Use deep sleep mode to save power
    - Access http
- Debug applications
- Configure your IDE
- Modify the supplied library
- Write your own host-side user interface for controlling Sun SPOTs

This guide does **not** cover these topics:

- The libraries for controlling the demo sensor board
- Installation of the SDK.

Building and deploying Sun SPOT applications

Deploying and running a sample application

The normal process for creating and running an application (assuming you are working from the command line rather than an IDE) is:

- Create a directory to hold your application.
- Write your Java code.
- Use the supplied ant script to compile the Java and bind the resulting class files into a deployable unit.
- Use the ant script to deploy and run your application.

In this section we will describe how to build and run a very simple application supplied with the SDK. Each step is described in detail below.

1. The directory `Demos/CodeSamples/SunSpotApplicationTemplate` contains a very simple Sun SPOT application that can be used as a template to write your own.

   The complete contents of the `template` directory should be copied to the directory in which you wish to do your work, which we call the *root directory* of the application. You can use any directory as the root of an application. In the examples below we have used the directory `C:\MyApplication`.

   All application root directories have the same layout. The root directory contains two files - `build.xml` and `build.properties` - that control the ant script used to build and run applications. The root directory also contains three sub-directories. The first, named `src`, is the root of the source code tree for this application. The second, named `nbproject`, contains project files used if your IDE is Netbeans. The third, named `resources`, contains the manifest file that defines the application plus any other resource files that the application needs at run time. Other directories will appear during the build process.

2. Compile the template example and create a jar that contains it by using the "`ant jar-app`" command. The created jar is called `imlet.jar` and is created in the `suite` folder.

```
C:\MyApplication>ant jar-app
Buildfile: build.xml

-pre-init:

-init-user:

-init-system:

-do-init:

-post-init:

init:

-set-jar-name:

-pre-clean:

-do-clean:
```

```
   [delete] Deleting directory C:\MyApplication\build
   [delete] Deleting directory C:\MyApplication\suite
   [delete] Deleting directory C:\MyApplication\j2meclasses

-post-clean:

clean:

-pre-compile:

-do-compile:
    [mkdir] Created dir: C:\MyApplication\build
    [javac] Compiling 1 source file to C:\MyApplication\build

-post-compile:

compile:

-pre-preverify:

-make-preverify-directory:
    [mkdir] Created dir: C:\MyApplication\j2meclasses

-unjar-utility-jars:

-do-preverify:

-post-preverify:

preverify:

-pre-jar-app:

-find-manifest:

-do-jar-app:
    [mkdir] Created dir: C:\MyApplication\suite
      [jar] Building jar: C:\MyApplication\suite\imlet.jar

-post-jar-app:

jar-app:

BUILD SUCCESSFUL
Total time: 1 second
C:\MyApplication>
```

If you have any problems with this step you need to ensure your Java JDK and Ant distributions are properly installed, as per the instructions in the Installation Guide.

3. Connect the Sun SPOT to your desktop machine using a mini-USB cable.

4. Check communication with your Sun SPOT using the "`ant info`" command, which displays information about the device.

```
C:\MyApplication>ant info
Buildfile: build.xml

-pre-init:

-init-user:

-init-system:

-do-init:
```

```
-post-init:

init:

-override-warning-find-spots:

-main-find-spots:
     [echo] Using Sun SPOT device on port COM45

-do-find-spots:

slots:

-run-spotclient-with-optional-remoteId:

-run-spotclient-with-script-contents:

-run-spotclient:
     [java] Waiting for target to synchronise...
     [java] (please reset SPOT if you don't get a prompt)
     [java] [waiting for reset]

     [java] Sun SPOT bootloader (1514-20060824)
     [java] SPOT serial number = 0014.4F01.0000.011D

     [java] Application slot contents:
     [java] 0: C:\arm9\BounceDemo-OnSPOT/suite/image (Thu Aug 24 12:51:22 BST 2006)
     [java]    28196 bytes at 0x10140000
     [java] 1: /home/Syntropy/SunSPOT/sdk-21Aug2006/tests/spottests/suite/image
(Thu Aug 24 16:39:14 BST 2006)
     [java]    115452 bytes at 0x101a0000 (current)

     [java] OTA Command Server is enabled
     [java] Not ignoring application suite at startup
     [java] Squawk startup command line:
     [java]    -Xmx:470000
     [java]    -Xmxnvm:128
     [java]    -isolateinit:com.sun.spot.peripheral.Spot
     [java]    -MIDlet-1
     [java] Library suite hash:
     [java]    0x50b227

     [java] Exiting
   [delete] Deleting: C:\SunSPOT\dev\temp\spot-temp-1808154274

info:

BUILD SUCCESSFUL
Total time: 3 seconds
C:\MyApplication>
```

If you don't see the expected output, try pressing the Sun SPOT's control button.

You will notice that the communication port has been automatically detected (COM45 in this example). If you have more than one Sun SPOT detected, the ant scripts will present you with a menu of connected Sun SPOTs and allow you to select one.

You may not wish to use the interactive selection process each time you run a script. As an alternative, you can specify the port yourself on the command line:

```
ant -Dspotport=COM2 info
```

or

```
ant -Dport=COM2 info
```

The difference between these two commands is that the "`spotport`" version will check that there is a Sun SPOT connected to the specified port, whereas the "`port`" version will attempt to connect to a Sun SPOT on the specified port regardless. You should normally use the "`spotport`" version. If you prefer, you may specify the port in the `build.properties` file of the application:

```
spotport=COM2
```

or

```
port=COM2
```

On Unix-based systems, including Mac OS X, if you see an `UnsatisfiedLinkError` exception, this means that you need to create a spool directory for the communications driver RXTX to use, as locks are places in that directory. See the section *notes on the RXTX driver* in the Installation Guide for guidance on how to set up your spool directory.

5. To deploy the example application, use the "`ant jar-deploy`" command.

```
C:\MyApplication>ant jar-deploy
Buildfile: build.xml

-pre-init:

-init-user:

-init-system:

-do-init:

-post-init:

init:

-set-jar-name:

-check-for-jar:

-override-warning-find-spots:

-main-find-spots:
     [echo] Using Sun SPOT device on port COM45

-do-find-spots:

-pre-suite:

-do-suite:
     [exec] [loaded object memory from
'file://C:/SunSPOT/dev/lib/translator.suite']
     [exec] [loaded object memory from
'file://C:/SunSPOT/dev/arm/transducerlib.suite']
     [exec] [Loaded squawk.imlet.Startup]
     [exec] [Including resource: META-INF/MANIFEST.MF]
     [exec] [Adding property key: Manifest-Version value: 1.0]
     [exec] [Adding property key: Ant-Version value: Apache Ant 1.6.5]
     [exec] [Adding property key: Created-By value: 1.5.0_06-b05 (Sun Microsystems
Inc.)]
     [exec] [Adding property key: MIDlet-Name value: Test]
     [exec] [Adding property key: MIDlet-Version value: 1.0.0]
```

```
     [exec] [Adding property key: MIDlet-Vendor value: Sun Microsystems Inc]
     [exec] [Adding property key: MIDlet-1 value: Spottests,,
squawk.application.Startup]
     [exec] [Adding property key: MIDlet-2 value: TestMIDlet label,,
com.sun.spot.TestIMlet]
     [exec] [Adding property key: MicroEdition-Profile value: IMP-1.0]
     [exec] [Adding property key: MicroEdition-Configuration value: CLDC-1.1]
     [exec] [Including resource: res1.txt]
     [exec] Created suite and wrote it into image.suite


     [exec] ------------------------------------------------------------------
     [exec] Hits   -   Class:98.07%  Monitor:88.74%  Exit:100.00%  New:98.68%
     [exec] GCs: 3 full, 0 partial
     [exec] ** VM stopped: exit code = 0 **
     [move] Moving 1 file to C:\MyApplication\suite
     [move] Moving 1 file to C:\MyApplication\suite
   [delete] Deleting: C:\MyApplication\image.suite.api

-post-suite:

-pre-deploy:

-do-deploy:

-run-spotclient-with-optional-remoteId:

-run-spotclient-with-script-contents:

-run-spotclient:
     [java] Waiting for target to synchronise...
     [java] (please reset SPOT if you don't get a prompt)
     [java] [waiting for reset]

     [java] Sun SPOT bootloader (1321-20060816)
     [java] SPOT serial number = 0014.4F01.0000.012E
     [java] About to flash to slot 1
     [java] Writing imageapp35015.bintemp(2065 bytes) to COM45
     [java] ..................
     [java] Download operation completed successfully
     [java] Writing Configuration(1080 bytes) to COM45
     [java] .........
     [java] Download operation completed successfully

     [java] Exiting
   [delete] Deleting: C:\SunSPOT\dev\temp\spot-temp-472249501

-post-deploy:

jar-deploy:

BUILD SUCCESSFUL
Total time: 4 seconds
C:\MyApplication>
```

If this step fails, this is most likely to be because your system does not know the location of the JavaVM. You can find out how to configure your system by running the following ant command and following the instructions given in the output.

```
C:\MyApplication>ant environment
Buildfile: build.xml

-pre-init:

-init-user:

-init-system:
```

```
-do-init:

-post-init:

init:

environment:

     [java] To configure the environment for Squawk, try the following command:

     [java]     set JVMDLL=C:\jdk1.5.0_06\jre\bin\client\jvm.dll


BUILD SUCCESSFUL
Total time: 0 seconds
C:\MyApplication>
```

6. Run the application. To run the application, use the "`ant run`" command.

```
C:\MyApplication>ant run
Buildfile: build.xml

-pre-init:

-init-user:

-init-system:

-do-init:

-post-init:

init:

-override-warning-find-spots:

-main-find-spots:
     [echo] Using Sun SPOT device on port COM45

-do-find-spots:

-pre-run:

-do-run:

-run-spotclient-with-optional-remoteId:

-run-spotclient-with-script-contents:

-run-spotclient:
     [java] Waiting for target to synchronise...
     [java] (please reset SPOT if you don't get a prompt)
     [java] [waiting for reset]

     [java] Sun SPOT bootloader (1321-20060816)
     [java] SPOT serial number = 0014.4F01.0000.012E

     [java] Squawk VM Starting (2006-08-16:13:21)...
     [java] Detected hardware type eSPOT-P2
     [java] PCTRL-1.67
     [java] EDEMOBOARD_REV_0_2_0_0
     [java] Starting OTACommandServer
     [java] My IEEE address is 0014.4F01.0000.012E
     [java] Registering protocol manager 'radiogram'
     [java] Adding radiogram connection for Server on port 8
     [java] Hello world!
```

```
     [java] ------------------------------------------------------------------
     [java] Hits   -   Class:95.74%  Monitor:92.38%  Exit:100.00%  New:98.43%
     [java] GCs: 2 full, 0 partial
     [java] ** VM stopped: exit code = 0 **


     [java] Exiting
   [delete] Deleting: C:\SunSPOT\dev\temp\spot-temp-674916937

-post-run:

run:

BUILD SUCCESSFUL
Total time: 10 seconds
C:\MyApplication>
```

As you can see, this application just prints "Hello world!" However, it gives you a framework to use for your applications.

N.B. After your Sun SPOT has printed "Hello world!" it probably will not exit immediately. Instead, you will have to push the control button to force it to exit. This is because by default, Sun SPOTs run a background thread which listens for over-the-air commands. You can disable this behaviour if you wish. For more details, see the section *Ensure that the remote Sun SPOT is executing the OTA Command Server*.

As a shortcut, the ant command "`deploy`" combines `jar-app` and `jar-deploy`. Thus we could build, deploy and run the application with the single command line:

```
ant deploy run
```

Deploying a pre-existing jar

To deploy an application that has already been built into a suitable jar (by using the "`ant jar-app`" command described earlier), use the "`ant jar-deploy`" command with a command line option to specify the path to the jar:

```
ant jar-deploy -Djar.file=myapp.jar
```

Incorporating utility classes into your application

You can include code from pre-existing jar files as part of your application. We refer to jars used in this way as *utility jars*. A utility jar is built in the normal way using "`ant jar-app`". To include a utility jar as part of your application specify it using "`-Dutility.jars=<filename>`", e.g.

```
ant deploy -Dutility.jars=util.jar
```

You can specifiy multiple utility jars as a list separated by a classpath delimiter (";" or ":"). Note that you may need to enclose the list in quotes. Also, the classes in the utility jars must all be preverified. One way to ensure this is to create the jar using

```
ant jar-app
```

Resource files in utility jars will be included in the generated jar, but its manifest is ignored.

If you have code that you want to include as part of all your applications you might consider building it into the system library – see the section *Advanced topics* in this document.

Excluding files from the compilation

To exclude files or folders matching a specific pattern from the java compilation, set the ant property `spot.javac.exclude.src`, either on the command line with –D or in the build.properties file of the project. The value of the property should be specified using standard ant wildcarding.

For example, to exclude all source files in all "unittests" folders, use:

```
spot.javac.exclude.src=**/unittests/*
```

Manifest and resources

The file `MANIFEST.MF` in the `resources/META-INF` directory contains information used by the Squawk VM[1] to run the application. In particular it contains the name of the initial class. It can also contain user-defined properties that are available to the application at run time.

A typical manifest might contain:

```
MIDlet-Name: Air Text demo
MIDlet-Version: 1.0.0
MIDlet-Vendor: Sun Microsystems Inc
MIDlet-1: AirText, , org.sunspotworld.demo.AirTextDemo
MicroEdition-Profile: IMP-1.0
MicroEdition-Configuration: CLDC-1.1
SomeProperty: some value
```

The syntax of each line is:

```
<property-name>:<space><property-value>
```

The most important line here is the one with the property name "`MIDlet-1`". This line has as its value a string containing three comma-separated arguments. The first argument is a string that provides a name for the application and the third defines the name of the application's main class. This class must be a subclass of `javax.microedition.midlet.MIDlet`. The second argument defines an icon to be associated with the MIDlet, which is currently not used.

The application can access properties using:

```
myMidlet.getAppProperty("SomeProperty");
```

All files within the resources directory are available to the application at runtime. To access a resource file:

```
InputStream is = getClass().getResourceAsStream("/res1.txt");
```

This accesses the file named "`res1.txt`" that resides at the top level with the resources directory.

Other user properties

For properties that are not specific to the application you should instead use either

- persistent System properties (see section *Persistent properties*) for device-specific properties
- properties in the library manifest (see section *Library manifest properties*
- *Each library* extension must contain a file named

[1] The Squawk VM is the Java virtual machine that runs on the Sun SPOT. For more details, go to http://research.sun.com/projects/squawk/.

`resources/META-INF/MANIFEST.MF`

within its root folder. The adderlib `extension` has such `a` file, whose content is

FavouriteSnake: Viper

This defines a property whose value will be available to all applications in a similar fashion to application-specific manifest properties. The `addertest` application demonstrates this by displaying the value of this property. The library suite is built to contain all the properties defined by the manifests of all its input jars. For more details on accessing these properties, see the section *Manifest and resources.*

- Running startup code
- Some library extensions require initialisation to be performed at startup (for example, to create daemon threads). To specify startup code that must be run, add one or more lines to the manifest properties of the library extension with the following format:

```
spot-startup-xxxx: my.fully.qualified.Classname
```

where `xxxx` is a string that is provided as the single argument to the static `main()` method of `Classname`.

Startup code is run only in the master isolate. It is run after all normal spotlib initialisation is completed but before the OTACommandServer (if configured) is started and before the application is launched.

- Modifying the system library code) for properties that are constant for all your Sun SPOTs and applications.

Built-in properties

There are a number of properties with reserved names that the libraries understand. These are:

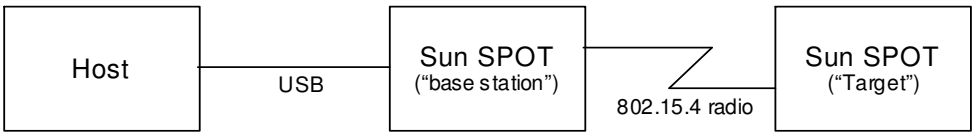
```
DefaultChannelNumber
DefaultPanId
DefaultTransmitPower
com.sun.spot.io.j2me.socket.SocketConnection-BaseStationAddress
com.sun.spot.io.j2me.socket.SocketConnection-BaseStationPort
```

The first three control the radio's operation; see section *Using manifest properties to adjust the radio*. The last two control the socket connection that underpins http access: see the section *Configuring the http protocol*.

Using the Basestation

Overview

The purpose of the Sun SPOT Basestation software is to allow applications running on the Host to interact with applications running on Targets. The physical arrangement is:



The Host can be any of the supported platforms (e.g. Windows PC, Mac). The Host application is a J2SE program. The Target application is a Squawk Java program.

In the SPOT SDK and documentation the 64-bit addresses that identify SPOTs are expressed as sixteen hexadecimal digits subdivided into four groups that are separated by dots for readability.

The Basestation may run in either dedicated or shared mode. In dedicated mode, it runs within the same Java VM as the host application and can only be used by that application. In this model, the host's address is that of the basestation.

In shared mode, two Java virtual machines are launched on the host computer: one manages the base station and another runs the host application. In this model, the host application has its own system-generated address, distinct from that of the base station. Communication from host application to the target is therefore over two radio hops, in contrast to one hop in the dedicated case.

The main advantage of shared mode is that more than one host application can use the same basestation simultaneously. Shared mode also allows multiple host processes to communicate with each other using the radio communication stack, which makes it possible to simulate the communication behaviour of a group of SPOTs using host applications (this simulation has different timing characteristics).

The disadvantage of shared mode is that run-time manipulation of the basestation SPOT's radio channel, pan id or output power is not possible.

By default, host applications will use the base station in dedicated mode. To switch to shared mode, insert this line into the `.sunspot.properties` file in your user root folder:

```
multi.process.basestation.sharing=true
```

Host applications that send or receive broadcasts, or that interact directly with the lower layers of the radio stack, will behave differently in the two modes. This issue is discussed in more detail in the section *Broadcasting and basestations*.

Set up

Connect a base station to your host computer. If you don't have a base station you can configure a Sun SPOT to be used as the base station by issuing the following command:

```
ant selectbasestation
```

Then press the Sun SPOT's control button and the Sun SPOT will act as a base station.

Base Station configuration

It is possible to select a channel and pan id for the base station using command line properties in conjunction with `ant host-run`. The properties are:

```
-Dremote.channel=nn
-Dremote.pan.id=nn
```

Alternatively, if you are operating in dedicated mode, the `IRadioPolicyManager` interface provides operations to adjust the output power of the radio, the PAN Id and the channel number. To access the policy manager from your host program do:

```
Spot.getInstance().getRadioPolicyManager()
```

Remote operation

Introduction

Until now, in this manual, we have worked with Sun SPOTs connected directly to the host computer. In this section we show how it's possible to carry out some, but not all, of the same operations on a remote Sun SPOT communicating wirelessly with the host computer via a base station.

The operations that can be performed remotely include:

- `ant deploy`
- `ant jar-deploy`
- `ant run`
- `ant fork`
- `ant debug`
- `ant info`
- `ant settime`
- `ant deletepublickey`
- `ant set-system-property`
- `ant system-properties`
- `ant delete-system-property`

In each case, the procedure is the same:

1. ensure that the remote Sun SPOT is executing the OTA ("over the air") Command Server
2. connect a Sun SPOT base station
3. specify the remote Sun SPOT's ID, either on the ant command line (using the `-DremoteId=<IEEE address>` switch) or in the application's build.properties file (`remoteId=<IEEE address>`)

If you wish, you may also carry out a fourth step, which is:

4. program the remote Sun SPOT application to take suitable actions during over-the-air downloads.

Each of these four is now considered in more detail

Ensure that the remote Sun SPOT is executing the OTA Command Server

The remote Sun SPOT must run a thread that listens for commands. To check whether or not the command server is enabled on a SPOT use the `ant info` command. Factory-fresh SPOTs have the command server enabled by default (except for basestations).

To configure the SPOT so that this thread is started each time the SPOT starts issue this command via a USB connection:

```
ant enableota
```

The SPOT remembers this setting in its configuration area in flash memory.

To configure the SPOT so that the OTA Command Server is not started each time the SPOT starts issue this command:

```
ant disableota
```

Although the OTA Command Server thread runs at maximum Thread priority, parts of the radio stack run at normal priority. This means that if an application runs continuously in parallel with the OTA Command Server, it should not run at a priority greater than normal, otherwise OTA Command Server commands may not be processed.

Connect a Sun SPOT base station

For details, see the section *Using the Basestation.*

Launch the spot client to control a remote Sun SPOT via the base station

To deploy an application use:

```
ant -DremoteId=<IEEE_ADDRESS> deploy
```

To run the deployed application use:

```
ant -DremoteId=<IEEE_ADDRESS> run
```

Unless the feature has been explicitly disabled, any output generated by the application using `System.out` or `System.err` will be redirected over-the-air and displayed in the command window exactly as if the SPOT was connected by USB[2]. This feature is disabled by setting the system property:

```
spot.remote.print.disabled=true
```

In these commands, `<IEEE_ADDRESS>` is the 64-bit IEEE address of the form `xxxx.xxxx.xxxx.xxxx`. By default this is the same as the serial number printed on each Sun SPOT. Alternatively you can execute an ant command to a locally connected Sun SPOT such as

```
ant info
```

which will print the serial number, for example:

```
...
[java] Sun SPOT bootloader (orange-20061120)
[java] SPOT serial number = 0014.4F01.0000.02ED
...
```

It is also possible to specify which radio channel and pan id the base station should use to communicate with the remote SPOT. To do this, set the ant properties `remote.channel` and

[2] The streams are not forwarded to the host until handshaking is complete. As a result, any output generated by your application before this point will not be displayed. Handshaking is usually completed within 100ms of start-up.

`remote.pan.id` either on the command line or in the `.sunspot.properties` file in your user root folder.

Using short names for SPOTs

As a shortcut, it is possible to use short names for SPOTs instead of the full 16-digit IEEE address. To do this create a file called "spot.names" in your user home directory. The file should contain entries of the form:

```
<short-name>=<IEEE_ADDRESS>
```

for example

```
my-spot-1=0014.4F01.0000.0006
```

Note that these short names are used for all connections opened from host applications to remote SPOTs, but are not available on SPOTs themselves.

Take suitable actions during over-the-air downloads

During over-the-air downloads, an application should suspend operations that use radio or flash memory, or that are processor intensive.

To do this, you need to write a class that implements the interface `com.sun.spot.peripheral.ota.IOTACommandServerListener`, and attach it to the `OTACommandServer` with code something like this:

```
OTACommandServer otaServer = Spot.getInstance().getOTACommandServer();
IOTACommandServerListener myListener = new MyListener();
otaServer.addListener(myListener);
```

Your listener object will then receive callbacks `preFlash()` and `postFlash()` around each flash operation.

Managing keys and sharing Sun SPOTs

Background

When you update your Sun SPOT with a new library or application, or with a new config page, the data that you send is cryptographically signed. This is for two reasons:

- to ensure that the code executing on your Sun SPOT contain valid bytecodes
- to prevent remote attackers from downloading dangerous code to your Sun SPOT via the radio.

By default, each user of each SDK installation has their own public-private key pair to support this signing. This key pair is created automatically when that user first requires a key (for example, when deploying an application for the first time).

Factory-fresh Sun SPOTs are not associated with any owner and do not hold a public key. The first user to deploy an application to that device (either via USB or over-the-air) becomes its owner. During this process, the owner's public key is automatically transferred to the device. Only the owner is allowed to install new applications or make configuration changes. This is enforced by digitally signing the application or config page with the owner's private key, and verifying the signature against the trusted public key stored on the device.

Even if you aren't concerned about security, you need to be aware of this if you want to be able to use Sun SPOTs interchangeably amongst two or more SDK installations. See the section *Sharing Sun SPOTs*.

Changing the owner of a Sun SPOT

Once set, only the owner can change the public key remotely, although anyone who has physical access to the Sun SPOT can also change the public key. If user B wishes to use a Sun SPOT device previously owned by user A, they can become the new owner in one of two ways:

- If user B does not have physical access to the device, user A can use the command

  ```
  ant deletepublickey
  ```

  to remove their public key from the Sun SPOT. User A can also use this procedure remotely, for example

  ```
  ant deletepublickey -DremoteId=0014.4F01.0000.0006
  ```

  User B can then deploy an application to the remote spot using a command like

  ```
  ant deploy -DremoteId=0014.4F01.0000.0006
  ```

  and will become the new owner automatically. During the time that the device has no owner (after user A has executed `deletepublickey` and before user B has executed `deploy`) the Sun SPOT will be exposed to attackers (a third user C could become its owner before user B). For this reason, if security is critical, we recommend replacing the public keys only via USB.

- If user B has physical access to the device, they can connect the device via USB and execute

  ```
  ant deploy
  ```

In both cases, if a customised library has been flashed to the Sun SPOT, it must be re-flashed by user B so that the library is signed using user B's private key. This means that user B must also execute

```
ant flashlibrary
```

This command cannot be executed remotely. Note that this procedure is not necessary if the library has not been customised, as verification for the factory-installed library is handled differently.

Sharing Sun SPOTs

If you want to share Sun SPOTs between two or more SDK installations or users, you have to ensure that the SDK installations and users share the same key-pair. To do this, start by installing each SDK as normal. Then, copy the key-pair from one "master" user to each of the others. You can do this by copying the file `sdk.key` from the `sunspotkeystore` sub-directory of the "master" user's home directory and replacing the corresponding file in each of the other user's `sunspotkeystore` directories.

You then have to force the master's public key onto each of the Sun SPOTs associated with the other installations. The simplest way to do this is to re-deploy the application via USB

```
ant deploy
```

for each Sun SPOT.

What is protected?

Applications and customized libraries are always verified and unless the digital signature can be successfully verified using the trusted public key, the application will not be executed. Extra security is provided for over-the-air deployment. In this case, all updates to the configuration page are verified before the page is updated. This prevents a number of possible attacks, for example a change to the trusted public key, or a denial of service where bad startup parameters are flashed.

Generating a new key-pair

If you wish to generate a new key-pair – for example, if you believe your security has been compromised – just delete the existing `sdk.key` file. The next time you deploy an application or a library to a Sun SPOT a new key will be automatically created. Again, if you are using a customized library, you will need to update the signature on the library by executing

```
ant flashlibrary
```

Limitations

This security scheme has some current limitations. In particular:

- There is no protection against an attacker who has physical access to the Sun SPOT device.
- The SDK key pair is stored in clear text on the host, and so there is no protection against an attacker with access to the host computer's file system.

Deploying and running a host application

Example

The directory `Demos/CodeSamples/SunSpotHostApplicationTemplate` contains a simple host application. Copy this directory and its contents to a new directory to build a host application.

To run the copied host application, first start the base station as outlined in the section *Using the Basestation*. Run the example on your host by using these commands:

```
ant host-compile
ant host-run
```

If the application works correctly, you should see (besides other output)

```
Base station initialized
```

Normally, the base station will be detected automatically. If you wish to specify the port to be used (for example, if automatic detection fails, or if you have two base stations connected) then you can either indicate this on the command line, by adding the switch "`-Dport=COM3`", or within your host application code:

```
System.setProperty("SERIAL_PORT", "COM3");
```

If your application doesn't require a basestation you may add the following switch to the command line:

```
ant host-run -Dbasestation.not.required=true
```

When the switch is specified a basestation will still be correctly identified if one is available, but if no basestation is available the `port` property will be set to `dummyport`.

Your own host application

You should base your host application on the template provided. Your host application can open connections in the usual way. The classes required to support this are in `spotlib_host.jar`, `spotlib_common.jar` and `squawk_classes.jar` (all in the `lib` sub-directory of the SDK install directory). These are automatically placed on the build path by the ant scripts but you might want to configure host projects in your IDE to reference them to remove compilation errors.

If your application requires non-standard Java VM parameters, then you can add these like this:

```
ant host-run -Dhost.jvmargs="-Dfoo=bar -Xmx20000000"
```

Incorporating pre-existing jars into your host application

You can include code from pre-existing jar files as part of your application. To do this add the jars to your user classpath property, either in build.properties or on the command line using "`-Duser.classpath=<filename>`", e.g.

```
ant host-run -Duser.classpath=util.jar
```

You can specifiy multiple jars as a list separated by a classpath delimeter (";" or ":"). Note that you may need to enclose the list in quotes.

You can create a jar suitable for inclusion in a host application with the ant target "`make-host-jar`", e.g.:

```
ant make-host-jar -Djar.file=util.jar
```

Configuring network features

Mesh routing

Every SPOT can act as a mesh router, forwarding packets it receives on to other SPOTs. It begins doing this automatically as soon as your application opens a `radiostream` or `radiogram` connection (see below). You do not need to perform any extra configuration for this to happen. However there are two special situations:

1. You want the SPOT to act as a mesh router but the application running it does not use the radio.
2. You want the SPOT to act just as a mesh router, with no application running on it.

For situation 1 above you should set the system property `spot.mesh.enable` to true, like this:

```
ant set-system-property -Dkey=spot.mesh.enable -Dvalue=true
```

With this property set the SPOT will turn on the radio at start-up and immediately begin routing packets. It will also ensure that the radio stays on, regardless of the policies set for application connections. Therefore a SPOT with this option set will never go into deep sleep.

For situation 2 above you should configure the SPOT as a dedicated mesh router, using the command:

```
ant selectmeshrouter
```

When the SPOT is reset (by pressing the attention button or by an `ant run` command) it will begin acting as a dedicated router. A SPOT set in this configuration does not need the `spot.mesh.enable` property to be set.

Trace route

A SPOT can optionally run a trace route server. If a SPOT is running the trace route server it can participate in route tracing requests. The default is for the server not to run. To enable the server set the system property `spot.mesh.traceroute.enable` to true, like this:

```
ant set-system-property -Dkey=spot.mesh.traceroute.enable -Dvalue=true
```

You should then see the server indicate its presence at start-up, like this:

```
[TraceRouteServer] starting
```

If you have a number of SPOTs running the trace route server you can issue trace route requests, like this:

```
ant tracert -DremoteId=0014.4F01.0000.0006
```

which will trace the current route to the specified SPOT.

Logging

As a diagnostic aid, you can enable display of all network route discovery and route management activity. To do that, set the system property `spot.mesh.route.logging` to true, like this:

```
ant set-system-property -Dkey=spot.mesh.route.logging -Dvalue=true
```

To see the same diagnostics for host applications, use:

```
ant host-run -Dspot.mesh.route.logging=true
```

See also the section below “Adjusting Log Output” for information on controlling whether opening and closing connections will be logged.

Hardware configurations and USB power

The SPOTs in the development kit come in two configurations:

- SPOT + battery + demo sensor board
- SPOT only

The SPOT-only package is intended for use as a radio base station, and operates on USB-supplied power. Apart from the lack of battery and sensor board the base station SPOT is identical to other SPOTs.

SPOTs are expected to work in a battery-less configuration (powered by USB power) if they do not have any other boards (such as the demo sensor board) fitted. If other boards are fitted the power consumption may exceed the maximum permitted by the USB specification. This is especially

critical during the USB enumeration which occurs when plugging in a new device. During this phase the SPOT may only draw 20% (100mA) of its full power requirements. It is known that a SPOT with the demo sensor board requires more power than this during startup and therefore does not work on USB power alone. For the hardware configurations in the kit this is not an issue, but for custom configurations this constraint should be taken into account.

Developing and debugging Sun SPOT applications

Overview of an application

A Sun SPOT application is actually implemented as a MIDlet. This should not concern most developers greatly: the simplest thing is to copy a template, as described above, and start editing the copy. The most significant implications of this are covered in the section *Manifest and resources*.

The Sun SPOT application runs on top of a number of other software components within the Sun SPOT. These include

- a bootloader (which handles the USB connection, upgrades to the SDK components, launching applications, and much of the interaction with ant scripts).
- a Config page (containing parameters that condition the operation of the bootloader).
- a Squawk Java VM (The Squawk VM is the Java virtual machine that runs on the Sun SPOT. For more details, go to http://research.sun.com/projects/squawk/).
- a bootstrap suite (containing the standard JME Java classes).
- a library suite (containing the Sun SPOT-specific library classes).

The operation of these will generally be transparent to application developers.

Threads

Developers should also be aware that a number of background threads will be running as a result of using the Sun SPOT libraries. These include various threads which manage sleeping (limiting power use where possible), monitor the state of the USB connection, and threads within the radio communication stack. All of these threads run as daemon threads.

One other thread to be aware of is the OTA command server thread: see section *Remote operation*. This thread may or may not be running according to the configuration of the Sun SPOT, but if it is, it will inhibit applications from exiting normally even after all user threads have stopped running.

Thread priorities

Through the standard contract of the Thread class, Squawk allows threads to be assigned priorities in the range Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10). Special interfaces allow threads to be given higher priorities known as "system priorities", but these are not intended for application use.

These rules should be adhered to by application developers to ensure correct operation of the libraries:

1. Applications should not use system priorities. If they do, the library may not work.
2. Application threads that are compute-bound should run at a lower priority than Thread.NORMAL. Otherwise they may interfere with the correct operation of the library.
3. Application threads at Thread.MAX_PRIORITY will not normally compete with any library threads, which all run at lower priorities except as detailed below. Note that if library threads are prevented from running then library operation may be affected. In particular, this may either degrade radio performance or cause incoming broadcast traffic to be missed.

The libraries only use system priorities in three cases:

1. To guarantee that application threads don't run during a system operation during which the system may be in an inconsistent state.
2. To guarantee short uninterruptible periods of operation for critical hardware interaction.
3. To ensure that the attention button is recognised.

Threads are only assigned a permanent system priority to achieve 1. or 3.

The Sun SPOT device libraries

Introduction

This section describes the contents of the Sun SPOT base library, `spotlib_device.jar` plus `spotlib_common.jar` (the source code for both is in `spotlib_source.jar`).

Sun SPOT device library

The library contains drivers for:

- The on-board LED
- The PIO, AIC, USART and Timer-Counter devices in the AT91 package
- The CC2420 radio chip, in the form of an IEEE 802.15.4 Physical interface
- An IEEE 802.15.4 MAC layer
- An SPI interface, used for communication with the CC2420 and off-board SPI devices
- An interface to the Sun SPOT's flash memory

It also contains:

- The basestation support code (in `com.sun.spot.peripheral.basestation`)
- The over-the-air (OTA) control support code (in `com.sun.spot.peripheral.ota`)
- The radio policy manager
- A simple on-SPOT test framework, based on junit (in `com.sun.spot.testFramework`)
- A framework for inter-isolate communication using a remote procedure call style (in `com.sun.spot.interisolate`)
- A handler for the attention button on the edge of the device

For details about using these devices and features see the respective Java interface:

Device	**Interface**
LED	`ILed`
PIO	`IAT91_PIO`
AIC	`IAT91_AIC`
Timer-Counter	`IAT91_TC`
CC2420	`I802_15_4_PHY`
MAC layer	`I802_15_4_MAC`
RadioPolicyManager	`IRadioPolicyManager`
SPI	`ISpiMaster`
Flash memory	`IFlashMemoryDevice`
Power controller	`IPowerController`
OTA Command Server	`OTACommandServer (class)`
Attention button handler	`FiqInterruptDaemon (class)`

Each physical device is controlled by a single instance of the respective class, accessed through the interfaces listed above. The single instance of the Sun SPOT class creates and manages access to the device driver singletons. The singletons are created only as required.

For example, to get access to the on-board green LED, do:

```
Iled theLed = Spot.getInstance().getGreenLed();
```

To turn the LED on and off:

```
theLed.setOn();
theLed.setOff();
```

Persistent properties

The sections Manifest and resources and Library manifest properties explain how to define application-specific and library-specific properties. Sometimes it is useful to define persistent properties for an individual SPOT device, and that is the topic of this section. Such properties are always available on the given SPOT regardless of which application is being run.

An 8k byte area of flash is reserved for persistent System properties that can be read and written from SPOT applications and by using ant commands on the host. All properties have a String key and a String value.

We distinguish between *user-defined* properties, set either using `ant set-system-property` or from within a SPOT application, from other System properties, such as those defined by Squawk.

Accessing properties from SPOT applications

To obtain the value of a System property do:

```
System.getProperty(<propName>);
```

This call returns null if the property has not been defined. All system properties, including user-defined properties, can be accessed this way.

To set the value of a user-defined property do:

```
Spot.getInstance().setPersistentProperty(<propName>, <propValue>);
```

A property can be erased by setting its value to null. Setting or erasing a persistent property takes about 250ms. If you wish to set several properties at once, you can optimise performance like this:

```
Properties propsToWrite = new Properties();
propsToWrite.setProperty("key1", "value1");
…
propsToWrite.setProperty("key99", "value99");
Spot.getInstance().setPersistentProperties(propsToWrite);
```

You can also get the value of a user-defined system property by using:

```
Spot.getInstance().getPersistentProperty(<propName>);
```

This call is much slower to execute than `System.getProperty`, because it re-reads the flash memory. Therefore you should normally use `System.getProperty` to access user-defined properties. However, user-defined properties are loaded from the flash when an isolate starts which means that you will need to use the slower method of access in two situations:

- there is a possibility that another isolate has written a new value for the property since the current isolate started
- your application has overwritten the user-defined property's value in memory but not in flash (by executing `System.setProperty`).

All user-defined properties can be accessed using:

```
Spot.getInstance().getPersistentProperties();
```

Accessing properties from the host

To view all user-defined System properties stored in a SPOT do:

```
ant system-properties
```

To set a property do:

```
ant set-system-property -Dkey=<key> -Dvalue=<value>
```

To delete a property do:

```
ant delete-system-property -Dkey=<key>
```

Overriding the IEEE address

When it starts up the SPOT loads the system properties from flash memory as described above and then checks whether the property `IEEE_ADDRESS` has been set. If not, it sets this property to the string representation of the device's serial number. The IEEE 802.15.4 MAC layer uses this property to determine the address that should be adopted by this device. If you wish to set an address different from the serial number then use the facilities described above to set the `IEEE_ADDRESS` property; this entry will take precedence.

Accessing flash memory

Two mechanisms are provided for reading and writing flash memory.

Using IFlashMemoryDevice

Read and write access to the Sun SPOT's flash memory is via an object conforming to the `IFlashMemoryDevice` interface. To obtain that object:

```
IFlashMemoryDevice mem = Spot.getInstance().getFlashMemoryDevice();
```

The `IFlashMemoryDevice` interface provides low-level access to the whole of the flash memory. A safer way of accessing that part of the flash memory available to applications is to read and write using streams:

```
IFlashMemoryDevice mem = Spot.getInstance().getFlashMemoryDevice();
int startSector = mem.getFirstAvailableSector();
DataOutputStream dos = new DataOutputStream(mem.getOutputStream(startSector, 2));
dos.writeUTF("hello there");
dos.flush();
DataInputStream dis = new DataInputStream(mem.getInputStream(startSector, 2));
String s = dis.readUTF();
```

The call to open a stream takes two parameters: the first specifies the sector number of the first sector to be read or written. The second parameter specifies the number of contiguous sectors to be allocated to this stream. Opening an output stream erases the data in all the allocated sectors. In the

example above two sectors are allocated (and thus erased), starting with the first sector available to applications.

Using the Record Management Store

The Record Management Store (RMS) that is part of the Java IM Profile provides a simple record-based mechanism for access to persistent data.

Here is a very simple example of using the record store:

```
RecordStore rms = RecordStore.openRecordStore("TEST", true);
byte[] inputData = new byte[]{12,13,14,15,16};
int recordId = rms.addRecord(inputData, 0, inputData.length);
byte[] outputData = rms.getRecord(recordId);
rms.closeRecordStore();
```

See `javax.microedition.rms.RecordStore` for full details.

Configuration

Although read streams can be opened on any area of flash memory, output streams can only be opened on sectors allocated for that purpose. By default sectors 39 to 46 inclusive (8 * 64Kb) are allocated for stream-based access.

By default sectors 47 to 69 inclusive (23 * 64Kb) are allocated for RMS use.

The number of sectors allocated for stream use can be adjusted by setting the system property `spot.sectors.reserved.for.streaming` to the number of required sectors. The first RMS sector always follows the last streaming sector.

Using input and output streams over the USB and USART connections

There is a mechanism provided for Sun SPOT applications to access input and output streams that access the USB and USART connections. A standard Sun SPOT can only access the USB connection. However, with a suitable adaptor, the Sun SPOT can access a USART connected to its top connector[3].

```
// for input from USB
InputStream inputStream = Connector.openInputStream("serial://usb");
int character = inputStream.read();

// for output from USB
OutputStream outputStream = Connector.openOutputStream("serial://usb");
outputStream.write("[output message goes here\n]".getBytes());
```

The same mechanism can be used to read to and from the USART by replacing the URL `"serial://usb"` with `"serial://usart"`. A third URL `"serial://"` is also supported. This reads and writes on USB when USB is connected and in use by a host application, and otherwise on the USART.

Only one input stream may be opened to each device, and if `"serial://"` is opened for input then neither `"serial://usb"` nor `"serial://usart"` can be opened for input. Multiple output streams may be opened, and output will be interspersed in an unpredictable fashion.

By default, `System.out` and `System.err` are directed to `"serial://"` (because the Sun SPOT is conformant with CLDC 1.1, there is no `System.in` on the Sun SPOT).

[3] The USART is USART0 in the AT91RM9200 package.

Configuring USART parameters

The URL for USART access can uniquely be extended with additional parameters to configure the USART if required. For example:

```
InputStream inputStream = Connector.openInputStream(
      "serial://usart?baudrate=57600&databits=7&parity=even&stopbits=1.5");
```

databits may be 5, 6, 7 or 8, parity can be none, even, odd, space or mark, and stop bits can be 1, 1.5 or 2. You need specified only those options you wish to change. Because a call of this nature will disrupt any ongoing communications, this kind of call can only be made while there are no existing output or input streams connected to `"serial://usart"`.

As we said above, `System.out` and `System.err` are by default connected to `"serial://"` and by implication to `"serial://usart"`. Thus, if it is required to adjust the USART settings in the fashion shown above, then `System.out` and `System.err` must be diverted like this:

```
Isolate currentIsolate = Isolate.currentIsolate();
currentIsolate.clearOut();
currentIsolate.clearErr();
currentIsolate.addOut("serial://usb"); // not required but useful for development
currentIsolate.addErr("serial://usb"); // not required but useful for development
OutputStream serialOutput = Connector.openOutputStream("serial://usart?baudrate=9600");
```

Note that none of this is required if the default settings for the USART are acceptable (these are baudrate=115,200, databits=8, parity=none, stopbits=1).

Limitations on the use of the USART

When the Sun SPOT boots it will send some characters over the USART using the default settings unless USB is connected and in use by a host application. These characters are necessary to allow the SPOT to talk to the SPOT Client program (the program that underpins many of the ant commands) via a RS232 connection. Your application should take steps to ignore these characters.

USART input uses a 512 byte input ring buffer, loaded using an interrupt handler. There is no protection for overflow of the ring buffer. It is up to the application to ensure that data is read quickly enough, given the baud rate and rate of transmission.

Transmission is not interrupt driven: each transmit request is a synchronous call to VM native code. Whilst transmitting the SPOT cannot do anything else, so it is wise to use the highest baud rate compatible with reliable reception.

The radio communication library

About the radio stack

The classes for the radio stack above the Mac layer are in `multihoplib_rt.jar`, and the corresponding sources can be found in `multihoplib_source.jar`.

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[4].

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

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("radio://<destinationAddr>:<portNo>");
```

where `destinationAddr` 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. Note that 0 is not a valid IEEE address in this implementation. The connection is opened using the default radio channel and default PAN Id (currently channel 26, PAN 3). The section *Radio properties* shows how to override these defaults.

To establish a bi-directional 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("radio://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();
}
```

[4] 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.

Program 2

```
RadiostreamConnection conn = (RadiostreamConnection)
     Connector.open("radio://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???");
          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).

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 Id (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. 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://0014.4F01.0000.0006:10");
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://:10");
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???");
     }
     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://0014.4F01.0000.0006:10");

// 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 the port can be accessed from the 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:

- If setting a timeout it must be set before opening streams or creating datagrams.
- 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(3);
```

Broadcasting and basestations

As described in the section *Using the Basestation,* 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 I802.15.4 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)
20	radiogram://	Trace route server

Radio properties

Changing channel, pan id and output power

The `RadioPolicyManager` provides operations for changing the radio channel, the PAN Id and the transmit power. For example:

```
IRadioPolicyManager rpm = Spot.getInstance().getRadioPolicyManager();
int currentChannel = rpm.getChannelNumber();
short currentPan = rpm.getPanId();
int currentPower = rpm.getOutputPower();
rpm.setChannelNumber(11);  // valid range is 11 to 26
rpm.setPanId((short) 6);
rpm.setOutputPower(-31);  // valid range is -32 to +31
```

There are some important points to note about using these operations:

- The most important point is that changing the channel, Pan Id or power changes it for all connections.
- If the radio is already turned on, changing the channel or Pan Id forces the radio to be turned off and back on again. The most significant consequence of this is that remote devices receiving from this radio may detect a "frame out of sequence" error (reported as an `IOException`). The radio is turned on when the first `RadiogramConnection` is opened, or when the first input stream is opened on a `RadioConnection`.

Adjusting log output

It is possible to disable the log messages displayed via `System.out` each time a connection is opened or closed. To do this set the system property `spot.log.connections` to false. To do this using an ant command:

```
ant set-system-property -Dkey=spot.log.connections -Dvalue=false
```

To do this from within an application:

```
Spot.getInstance().setProperty("spot.log.connections", false);
```

For host programs, use:

```
ant host-run -Dspot.log.connections=false
```

Using manifest properties to adjust the radio

The initial values of the channel, Pan Id and transmit power can be specified using properties in the application manifest. The properties are:

```
DefaultChannelNumber
DefaultPanId
DefaultTransmitPower
```

Turning the receiver off and on

The radio receiver is initially turned off, but turns on automatically when either a radiogram connection capable of receiving is opened (that is, a non-broadcast radiogram connection) or when an input stream associated with a radio connection is opened. The receiver is turned off automatically when all such connections are closed.

Connections can be associated with one of three policies:

- `ON`: the default policy, where the existence of the connection forces the radio receiver to be on.
- `OFF`: where the connection does not require the radio receiver to be on.
- `WEAK_ON`: where the connection is prepared to handle incoming radio traffic but is happy to acquiesce if another connection wants the radio off.

To set a connection's policy, do, for example:

```
myRadiogramConnection.setRadioPolicy(RadioPolicy.OFF);
```

In the presence of multiple connections the radio is always kept on as long as one or more connections have set their policy to be `RadioPolicy.ON`. If no connections are set to `ON` and one or more connections are set to `OFF` then the radio is turned off, and if all the current connections are set to `WEAK_ON` the radio is turned on. If all connections are removed, the radio is turned off.

Note that all the above applies to the radio receiver. Even when the receiver is off, outgoing traffic can still be transmitted. When transmission happens, the receiver is turned on briefly, so that link-level acknowledgements can be received where necessary.

If your application does not use the radio, but has the OTA command server running, then because the OTA command server's connection has its policy set to WEAK_ON, the radio will be on. So, if you want to turn the radio off to conserve power, or to allow deep sleep to happen, you should create a dummy connection and set its policy to off:

```
RadiogramConnection conn = (RadiogramConnection) Connector.open("radiogram://:42");
conn.setRadioPolicy(RadioPolicy.OFF);
```

This will overrule the OTA command server connection `WEAK_ON` policy. Note that if you close the dummy connection, then the radio will turn back on, in line with the OTA command server's `WEAK_ON` policy.

Allowing the radio stack to run

Some threads within the radio stack run at normal priority. So, if your application has threads that run continuously without blocking, you should ensure that these run at normal or lower priority to permit the radio stack to process incoming traffic.

Signal strength

When using the Radiogram protocol it is possible to obtain various measures of radio signal quality captured when the datagram is received.

RSSI (received signal strength indicator) measures the strength (power) of the signal for the packet. It ranges from +60 (strong) to -60 (weak). To convert it to decibels relative to 1 mW (= 0 dBm) subtract 45 from it, e.g. for an RSSI of -20 the RF input power is approximately -65 dBm.

CORR measures the average correlation value of the first 4 bytes of the packet header. A correlation value of ~110 indicates a maximum quality packet while a value of ~50 is typically the lowest quality packet detectable by the SPOT's receiver.

Link Quality Indication (LQI) is a characterization of the quality of a received packet. Its value is computed from the CORR, correlation value. The LQI ranges from 0 (bad) to 255 (good).

These values are obtained using:

```
myRadiogram.getRssi();
myRadiogram.getCorr();
myRadiogram.getLinkQuality();
```

Monitoring radio activity

It is sometimes useful to monitor radio activity, for example when investigating errors. Two sets of facilities are provided. One allows the last ten radio packets received to be displayed on demand, along with information about the state of the radio chip at the time the packets were read. The second facility provides a count of key errors in the MAC layer of the radio stack.

To view the last ten radio packets, do

```
IProprietaryRadio propRadio = RadioFactory.getIProprietaryRadio();
propRadio.setRecordHistory(true);

… // radio activity including receiving packets

propRadio.dumpHistory(); // prints info to system.out
```

To see counts of MAC layer errors, do

```
IProprietaryRadio propRadio = RadioFactory.getIProprietaryRadio();
propRadio.resetErrorCounters(); // set error counters to zero

… // radio activity provoking errors

System.out.println("RX overflows since reset " + propRadio.getRxOverflow());
System.out.println("CRC errors since reset " + propRadio.getCrcError());
System.out.println("Channel busy stopped TX " + propRadio.getTxMissed());
System.out.println("Short packets received " + propRadio.geShortPacket());
```

Conserving power using deep sleep mode

Shallow Sleep

A thread is idle if it is executing `Thread.sleep()`, blocked on a synchronization primitive or waiting for an interrupt from the hardware. Whenever all threads are idle the Sun SPOT drops into a power saving mode ("shallow sleep") to reduce power consumption and extend battery life. Decisions about when to shallow sleep are taken by the Java thread scheduler inside the VM. This is transparent to applications – no special work on the part of the programmer is required to take advantage of it.

While considerable power can be saved during shallow sleep, it is still necessary to power much of the Sun SPOT hardware:

- CPU – power on but CPU clock off (the CPU's power saving mode)
- Master system clocks – power on
- Low level firmware – power on
- RAM – power on but inactive
- Flash memory – power on but inactive
- CC2420 radio – power on
- AT91 peripherals – power on

Because power is maintained to all the devices the Sun SPOT continues to react to external events such as the arrival of radio data. The Sun SPOT also resumes from shallow sleep without any latency. That is, as soon as any thread becomes ready to run, the Sun SPOT wakes and carries on.

Deep Sleep

The Sun SPOT can be programming to use a deeper power-saving mode called “deep sleep”. In this mode, whenever all threads are inactive, the Sun SPOT can switch off its primary power supply and only maintain power to the low level firmware and the RAM.

- CPU – power off
- Master system clocks – power off
- Low level firmware – power on
- RAM – main power off, RAM contents preserved by low power standby supply
- Flash memory – power off
- CC2420 radio – power off
- AT91 peripherals – power off

The Java thread scheduler decides when to deep sleep. It takes some time to wake from deep sleep, so the scheduler may choose to shallow sleep if the sleep duration will be too short to make deep sleep worthwhile. Because deep sleep involves switching off the power to peripherals that may be active it is necessary to interact with the device drivers to determine if deep sleep is appropriate. If a deep sleep cannot be performed safely the scheduler will perform a shallow sleep instead.

Activating deep sleep mode

Deep sleep is enabled by default. You can disable deep sleep mode using the SleepManager:

```
ISleepManager sleepManager = Spot.getInstance().getSleepManager();
sleepManager.disableDeepSleep();
```

If deep sleep is disabled the Sun SPOT will only ever use shallow sleep. Once deep sleep is enabled, the Sun SPOT may choose to deep sleep if appropriate.

The minimum idle time for which deep sleeping is allowed can be found from the SleepManager. For example, the following code can only ever trigger a shallow sleep:

```
Thread.sleep(sleepManager.getMinimumDeepSleepTime() - 1);
```

The following code *may* deep sleep, but only if a number of preconditions (detailed later) are met:

```
Thread.sleep(sleepManager.getMinimumDeepSleepTime());
```

The minimum deep sleep time includes a time allowance for reinitializing the hardware on wake up so that user code resumes at the correct time. Any part of the allowance that is not needed is made up using a shallow sleep.

If it is important that your application knows whether or not deep sleep happened, then you can alternatively execute this code:

```
ISleepManager sleepManager = Spot.getInstance().getSleepManager();
SleepManager.ensureDeepSleep(5000);
```

This code will either deep sleep for the specified time of 5,000 milliseconds, or throw an `UnableToDeepSleepException` to explain why deep sleep was not possible. In this case no sleeping will happen, deep or shallow. The section *Preconditions for deep sleeping* explains the possible reasons for a failure to deep sleep. Note that this feature is only available if you are executing in the master isolate.

USB inhibits deep sleep

USB hosts require that any device that is plugged into a USB port is able to answer requests on the USB bus. This requirement prevents the Sun SPOT from deep sleeping when it is attached to a USB host. A Sun SPOT on USB uses shallow sleep instead. Connecting a simple external battery via the USB socket will not inhibit deep sleep.

Preconditions for deep sleeping

A number of preconditions must be met in order for the Java thread scheduler to decide to deep sleep:

- Deep sleep mode must not have been deactivated using the `SleepManager` (as shown above).
- All threads must be blocked either on synchronization primitives or on a timed wait with a time to wake up at least as long as the minimum deep sleep time (including the wake time allowance)
- At least one thread must be on a timed wait (that is, the Sun SPOT will not deep sleep if all threads are blocked on synchronization primitives).
- All device drivers must store any necessary persistent state in RAM (to protect against the associated hardware being switched off) and agree to deep sleep. A driver may veto deep sleep if switching off its associated hardware would cause problems. In this case all drivers are set up again and a shallow sleep is performed instead.

Deep sleep behaviour of the standard drivers

Device	Condition to permit deep sleep
CC2420	Radio receiver must be off
Timer-counter	The counter must be stopped
PIO	All pins claimed must have been released
AIC	All interrupts must be disabled
PowerManager	All peripheral clocks must be off

Note that the appropriate way for an application to turn off the radio receiver is to give your connections the `WEAK_ON` or `OFF` policy.

The deep sleep/wake up sequence

The device drivers for the Sun SPOT are written in Java, allowing them to interact with the SleepManager when it is determining if a deep sleep should be performed. The full sequence of activities follows.

1. The Java thread scheduler discovers all threads are blocked, with at least one performing a `Thread.sleep()`.

2. Of all threads executing a `Thread.sleep()` it determines which will resume soonest. If the sleep interval is less than the minimum deep sleep time it shallow sleeps.
3. If the sleep interval is at least the minimum deep sleep time and deep sleeping is enabled, it sends a request to the SleepManager thread.
4. The SleepManager checks whether the Sun SPOT is connected to USB. If it is, a shallow sleep is performed.
5. The SleepManager requests each driver to "tear down", saving any state that must be preserved into RAM and releasing any resources such as interrupt lines and PIO pins it has acquired from other drivers. Finally, each driver returns a status indicating whether it was able to do this successfully. If any driver failed, all drivers are reinitialized and a shallow sleep is performed.
6. If all drivers succeed, a deep sleep is performed. The low level firmware is programmed to wake the Sun SPOT up at the requested time and main power is turned off.

When the firmware restores main power to the Sun SPOT it resumes execution of the Java VM. The SleepManager then requests each driver to reinitialize its hardware using the state it stored before deep sleeping. Finally, the SleepManager does a brief shallow sleep until the end of the allotted driver wake up time allowance and resumes execution of the user program. The deep sleep appears to be transparent, except that external events such as radio packets arriving may have been missed during the deep sleep.

Writing a device driver

In deep sleep mode the CPU will not be able to receive interrupts from peripherals. Furthermore, the peripherals themselves are switched off, so they cannot detect external events or trigger interrupts. Only the low level firmware's real time clock continues to operate, allowing the Sun SPOT to wake up after a predetermined interval. Therefore, if the Sun SPOT deep sleeps it could miss external events unless drivers are written appropriately.

If you write your own device driver and you want to support deep sleep your driver must implement the `IDriver` interface. This has two methods "`tearDown`" and "`setUp`" which will be called indirectly by the `SleepManager` when the Sun SPOT deep sleeps and wakes, respectively. The `tearDown` method should only return true if the driver approves the request to deep sleep. A driver also has a method "name" that is used to identify the driver in messages to the user. The Sun SPOT has a `DriverRegistry` with which the driver should register itself with in order to participate in the tear down/set up sequence. A simple example of a confirming driver can be found in the class `com.sun.spot.peripheral.Led`. Note that this class is not public and hence in not described in the javadoc but its source can be found in `<sdk>/src/spotlib_source.jar`.

One constraint on device driver authors is that device drivers may receive interrupts while processing `tearDown` and `setUp`. The radio driver handles this by vetoing deep sleep unless an application has previously disabled the radio and consequently its interrupts. An alternative strategy is to turn off interrupts at the beginning of `tearDown`, and reenable them at the end of `setUp`.

It can prove difficult to debug `tearDown()` and `setup()` methods as these are only normally invoked when the USB port is not connected to a host computer, which means that diagnostic messages cannot be seen. To overcome this method, you can execute code like this:

```
Spot.getInstance().getSleepManager().enableDiagnosticMode();
```

The effect of this will be that the `SleepManager` will go through the full process of tearing down even though USB is enumerated. It will then enter a shallow sleep for the requisite period, and then set up the drivers as though returning from deep sleep.

http protocol 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.hut.fi` or a numerical TCP/IP address. `port` is the port number, which can usually be omitted, in which case 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 base station 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 SPOTusing 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 base station that will be used to open a connection to the Socket Proxy. In order to use a specific base station 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>
```

HTTP Proxy

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:

base station's serial port	the serial port that the base station 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.

Configuring projects in an IDE

This section includes general-purpose notes for setting up all IDEs. We assume that the reader is familiar with his or her own IDE, and in particular, incorporating the various ant scripts that this guide refers to.

Classpath configuration

The classpath for an application that runs on a Sun SPOT needs to include the following jars in this order:

```
SDK_INSTALL_DIRECTORY/lib/transducerlib_rt.jar
SDK_INSTALL_DIRECTORY/lib/multihoplib_rt.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_device.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
SDK_INSTALL_DIRECTORY/lib/squawk_rt.jar
```

The classpath for a host application that uses the base station needs to include the following jars in this order:

```
SDK_INSTALL_DIRECTORY/lib/multihoplib_rt.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_host.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
SDK_INSTALL_DIRECTORY/lib/squawk_classes.jar
```

Javadoc/source configuration

Source is included for these jars:

```
transducerlib_rt.jar       SDK_INSTALL_DIRECTORY/src/transducerlib_source.jar
multihoplib_rt.jar         SDK_INSTALL_DIRECTORY/src/multihoplib_source.jar
spotlib_device.jar         SDK_INSTALL_DIRECTORY/src/spotlib_source.jar
spotlib_common.jar         SDK_INSTALL_DIRECTORY/src/spotlib_source.jar
spotlib_host.jar           SDK_INSTALL_DIRECTORY/src/spotlib_host_source.jar
```

Javadoc for all the above and for `squawk_rt.jar` can be found in:

```
SDK_INSTALL_DIRECTORY/doc/javadoc
```

Debugging

The Squawk high-level debugger uses the industry-standard JDWP protocol and is compatible with IDEs such as NetBeans and Eclipse. Note that it is not currently possible to debug a SPOT directly connected via USB, but instead, the target SPOT must be accessed remotely via a base station.

The Squawk high-level debugger comprises the following:

- A debug agent (the *Squawk Debug Agent*, or SDA) that executes in the Sun SPOT being debugged. The agent is responsible for controlling the execution of the installed application.
- A debug client, such as JDB, NetBeans or Eclipse.
- A debug proxy program (the *Squawk Debug Proxy*, or SDP) that executes on the desktop. The proxy communicates, over the air via a Sun SPOT base station, between the debug client and the debug agent.

To use the high-level debugger do the following:

1. Set up a standard Sun SPOT base station.
2. Build and deploy your application as normal, either via USB or over-the-air.
3. Ensure that the SPOT running your application has the OTA Command Server enabled:

   ```
   ant enableota
   ```

4. In the root directory of your application do

   ```
   ant debug -DremoteId=xxxx -Dbasestation.addr=yyyy
   ```

where xxxx is the id of the SPOT running the application being debugged and yyyy is the id of your base station. Note that xxxx can be a short name, as described in the *Remote* section, but yyyy cannot. If you regularly use the same basestation you can define the basestation.addr property in the .sunspot.properties file in your user root folder and then omit it from the command line.

The SPOT will be restarted with the SDA running; your application will not start. Then the SDP will be started and will communicate with the SPOT. This takes a few seconds. The output should look something like this:

```
C:\arm9\AirText>ant debug -DremoteId=spot1
Buildfile: build.xml

. . .

-main-find-spots:
     [echo] Using Sun SPOT basestation on port COM7

 . . .

-run-spotclient:
     [java] My IEEE address is 0000.0000.0000.0001
     [java] Waiting for target to synchronise...
     [java] (please wait for remote SPOT spot1 to respond)
     [java] (if no response ensure SPOT is running OTACommandServer)

     [java] Remote Monitor (1443-20060717)
     [java] SPOT serial number = 0014.4F01.0000.0080
     [java] Writing Configuration(1080 bytes) to remote Spot
     [java] ............


     [java] Exiting
   [delete] Deleting: C:\SunSPOT\dev\temp\spot-temp-1157051385

. . .

-do-debug-proxy-run:
     [java] Starting hostagent...
     [java] My IEEE address is 0000.0000.0000.0001
     [java] Done starting hostagent
     [java] Trying to connect to VM on radio://spot1:9
     [java] Established connection to VM (handshake took 62ms)
     [java] Waiting for connection from debugger on serversocket://:2900
```

5. Start a remote debug session using your preferred debug client. The process for doing this varies from client to client, but you need to use a socket connection on port 2900. When the connection from the debug client to the SDP is closed at the end of the debug session the SDP will exit and return to the command prompt. Output generated by the application using `System.out` or `System.err` will be displayed in the proxy window.
6. To take the remote SPOT out of debug mode so that it just runs your application do:

```
ant selectapplication
```

Limitations

The current version of the debugger has some limitations that stem from the fact that when using the debugger, the application runs in a child isolate.

Using non-default channel or pan id

It is not possible for an application being debugged to select dynamically a different channel or pan id. Instead the required channel or pan id must be selected using manifest properties (see section *Using manifest properties to adjust the radio*). Then the required channel or pan id can be specified like this:

```
ant debug -DremoteId=spot1 -Dremote.channel=11 -Dremote.pan.id=99
```

Unexpected exceptions

Some code which runs correctly in a standalone application will cause an exception when running under the debugger. To aid debugging, we recommend that you always set a breakpoint on `SpotFatalException` so that you can at least see when a conflict occurs. Instructions for doing this with some IDEs are shown below.

Conflicts will occur for applications that:

- Interact directly with the MAC or PHY layers of the radio stack. Applications that use the radio via the radiostream: and radiogram: protocols should work correctly under the debugger.

- Interact directly with the hardware. For example, applications that manipulate IO pins directly and applications that manipulate the LEDs on the front of the Sun SPOT processor board (applications that manipulate the LEDs on the demo sensor board should work correctly under the debugger).

Configuring NetBeans as a debug client

Select the "Run" menu item and the sub-item "Attach Debugger…" Enter 2900 as the port number and 5000ms as the timeout. The connector field should be set to "socket attach".

We recommend that you set a breakpoint for `SpotFatalException` (see section *Unexpected exceptions* for more information). To do this, select "Run…", then "New Breakpoint…". In the dialog, set the breakpoint type to be "Exception, the package to be `com.sun.spot.peripheral` and the class name to be `SpotFatalException`.

Configuring Eclipse as a debug client

From the "Run" menu select the "Debug…" option. Create a new "Remote Java Application" configuration and set the port number to 2900, with the "Standard (socket attach)" connection type.

We recommend that you set a breakpoint for `SpotFatalException` (see section *Unexpected exceptions* for more information). To do this, select "Run…" and then "Add Java Exception Breakpoint…" and then select `SpotFatalException`.

Finally, it will improve the debugging experience if you associate source code with the various Sun SPOT SDK jar files. To do this, select your Eclipse project, then from the right button menu select "Properties…". Select the "Java Build Path" on the left of the Properties dialog and then "Libraries" tab on the right. If you haven't already done so, add the five key jar files here:

```
SDK_INSTALL_DIRECTORY/lib/transducerlib_rt.jar
SDK_INSTALL_DIRECTORY/lib/multihoplib_rt.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_device.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
SDK_INSTALL_DIRECTORY/lib/squawk_rt.jar
```

Expand each of the first four of these, then select and edit the “Source attachment”. Point this to the relevant source jar as follows:

```
transducerlib_rt.jar       SDK_INSTALL_DIRECTORY/src/transducerlib_source.jar
multihoplib_rt.jar         SDK_INSTALL_DIRECTORY/src/multihoplib_source.jar
spotlib_common.jar         SDK_INSTALL_DIRECTORY/src/spotlib_source.jar
spotlib_host.jar           SDK_INSTALL_DIRECTORY/src/spotlib_source.jar
```

You may need to create a new Eclipse classpath variable to do this. After this, you should see source for the Sun SPOT library code as you debug.

Advanced topics

Using library suites

Introduction

The earlier section *Deploying and running a sample application* shows the process of building user application code into a suite file, deploying that suite file to a Sun SPOT, and executing the application. In fact, each Sun SPOT actually contains three linked suite files:

- a bootstrap suite: which contains the base J2ME and CLDC libraries, and other system-level Java classes
- a library suite: which contains Sun SPOT-specific library code supplied in this release
- an application suite: which contains application code.

For simple application development, the existence of the bootstrap and library suites can be ignored. However, if you develop a substantial body of code that, over time, becomes stable and slow-changing, you can add this code to the library suite. This makes application suites smaller, and hence quicker to build and quicker to deploy. The disadvantage of this is that when the code within the library suite does change, two suites must be re-built and re-deployed, which takes longer.

Alternatively, you may wish to modify the supplied code in the library suite, to experiment with alternatives to system components. In this instance, you might even move code from the library suite to the application suite.

The library suite is constructed from a number of `.jar` files. The process for building a library suite is as follows:

1. Build any new `.jar` files containing extensions to the library.
2. Rebuild any of the existing `.jar` files which you wish to modify
3. Combine the various `.jar` files into a library suite
4. Deploy the new library suite to your Sun SPOTs
5. Build and deploy applications that use the library suite as usual.

The next section works through an example of adding user code into the library suite.

Adding user code to the library suite

Locate the folder `LibraryExtensionSampleCode` in the `CodeSamples` directory. This contains a tiny library extension consisting of a single class `adder.Adder`, that has a single static method `add(int x, int y)` that adds two numbers together. In this example, we will rebuild the library suite to include this extension, install it to a Sun SPOT, and then deploy an application that uses our extension without including the `Adder` class within the application.

Start by copying `LibraryExtensionSampleCode` to a working area. This contains two sub-directories: `adderlib` and `addertest`. `adderlib` contains the library. You should find a sub-directory containing the library Java source: you can add to or edit this as you see fit.

1. Build any new .jar files containing additions to the library

With `adderlib` as your current directory, execute the command

```
ant jar-app
```

to create a jar file containing the library extension. You can check the correct execution of this command by looking in your SDK installation, where you should now find `adderlib_rt.jar` in the `lib` directory. The name of the `.jar` file is controlled from the file `build.properties` in the root directory of the `adderlib` directory.

2. Rebuild any of the existing .jar files which you wish to modify

In this example, we don't plan to modify the supplied library code, and so we skip this step. However, see the section *Modifying the system library code* for an explanation of what to do if you do wish to make such modifications.

3. Combine the various .jar files into a library suite

Identify the file `.sunspot.properties` in your user root folder. Add these two lines

```
spot.library.name=adderlib

spot.library.addin.jars=${sunspot.lib}/adderlib_rt.jar${path.separator}${sunspot.lib}/mul
tihoplib_rt.jar${path.separator}${sunspot.lib}/transducerlib_rt.jar
```

The first line specifies the name of the library suite file, and should replace an existing line that defines the same property. This can be any legal filename you wish (the actual file will be created with a "`.suite`" suffix).

The second line specifies some of the `.jar` files will be combined to create a library jar. The three files listed in `.sunspot.properties` are

- `multihoplib_rt.jar` (the standard communications stack)
- `transducerlib_rt.jar` (the standard library for the demo sensor board)
- `adderlib_rt.jar` (the sample library extension we're working with).

In fact, all libraries contain two further `.jar` files, which do not need to be specified in the `spot.library.addin.jars` property:

- `spotlib_device.jar` (core library classes that run on the SPOT device)
- `spotlib_common.jar` (core library classes that run on both the SPOT device and in host applications. For example, the high level interface to radio communication).

Once you have modified `.sunspot.properties`, execute this command:

```
ant library
```

This should create a library suite named `adderlib.suite` in the `arm` sub-directory of your SDK installation. Because we have defined the properties that control the "`ant library`" command in `.sunspot.properties`, this command can be executed from any folder that contains a valid `build.xml`.

Note that by default, the library is built without line number information to save space in flash memory. For debugging purposes, you may find it useful to build it with line number information, which will put line number information in stack traces. To do this, do

```
ant library -DKEEPSYMBOLS=true
```

4. Deploy the new library suite to your Sun SPOTs

Use the command

```
ant flashlibrary
```

to flash the new library onto your Sun SPOTs. This command always flashes the library whose name is defined in `.sunspot.properties`, regardless of where it is executed.

5. Build and deploy applications that use the library suite as usual

Change directory to the `addertest` folder, and deploy and run the application as usual using the command

```
ant deploy run
```

Library manifest properties

Each library extension must contain a file named

```
resources/META-INF/MANIFEST.MF
```

within its root folder. The `adderlib` extension has such a file, whose content is

```
FavouriteSnake: Viper
```

This defines a property whose value will be available to all applications in a similar fashion to application-specific manifest properties. The `addertest` application demonstrates this by displaying the value of this property. The library suite is built to contain all the properties defined by the manifests of all its input jars. For more details on accessing these properties, see the section *Manifest and resources*.

Running startup code

Some library extensions require initialisation to be performed at startup (for example, to create daemon threads). To specify startup code that must be run, add one or more lines to the manifest properties of the library extension with the following format:

```
spot-startup-xxxx: my.fully.qualified.Classname
```

where `xxxx` is a string that is provided as the single argument to the static `main()` method of `Classname`.

Startup code is run only in the master isolate. It is run after all normal spotlib initialisation is completed but before the OTACommandServer (if configured) is started and before the application is launched.

Modifying the system library code

It is also possible to modify the supplied library code if you wish. To do this, you should expand step 2 in the process outlined in the section *Adding user code to the library suite* as follows.

The source code for the libraries is supplied in three parts: `spotlib_source.jar` contains the code that supports the SPOT main board; `multihop_source.jar` contains the radio communications stack; and `transducerlib_source.jar` contains the code that supports the Demo Sensor Board.

This example shows the process for rebuilding the library with a modified version of the source in `spotlib_source.jar`. The process for modifying any or all of the other source `.jars` is similar. The example uses Windows commands, but the process is similar for other operating systems.

`cd c:\spot_libraries`	Or wherever you want to base the temporary directories

`mkdir spotlib`
`cd spotlib`
`copy <install>/src/spotlib_source.jar .`	

`jar xvf spotlib_source.jar`	You should now make changes to the source
`ant jar-app`	Recreates `spotlib_common.jar` and `spotlib_device.jar` in `<install>/lib`
`ant library`	Rebuilds your library using the modified jars

You must also execute "`ant flashlibrary`" to install the library on your Sun SPOT and "`ant deploy`" to install your application (this last step is required even if the application has not changed, to ensure that the application is compatible with the new library).

Using the spot client

For normal application development, a Sun SPOT connected to a serial or USB port is accessed via ant scripts (see most other sections of this guide for examples). The ant scripts in turn drive a command line interface that is supplied as part of the Sun SPOT SDK. This command line interface is found in spotclient.jar, along with the classes that provide the functions behind that command line interface.

The purpose of this section is to explain the interface between the spot client software and the user interface, to allow developers of development tools to build user interfaces other than the command line interface. To create such a tool, there are three essential steps:

- Write a class that implements the interface IUI. An instance of this will be used by the Spot client code to provide feedback during its operation
- Wire an instance of this class together with a number of other objects at start-up.
- Have the development tool execute various of the commands provided by the spot client code.

Implementing IUI

This interface defines various methods that the `SpotClient` calls to provide unsolicited feedback, which consists of various kinds of progress information, and the console output from the target Sun SPOT. The `IUI` developer needs to implement these to convey this information to their user appropriately.

Wire objects together

At system startup, a development tool should create and wire together various objects. This code shows the general style:

```
String keyStorePath = "/foo/bar"; // path to user's key store
SpotManager spotManager = new SpotManager(keyStorePath);
MyUIClass ui = new MyUIClass();
SerialPortTarget spt = new SerialPortTarget();
spt.initialise("COM3", ui); //port to which Spot is attached
spotManager.setTarget(spt);
SpotClientCommands commandRepository = new SpotClientCommands(ui, spotManager, appPath,
libFile, sysBinPath);
```

The `SpotClientCommands` object provides a repository containing one instance of `SpotClientCommand` objects for each of the commands available to the UI: the UI can retrieve and execute these command objects. Alternatively, the following is an optional further step.

```
SpotClient spotClient = new SpotClient(ui, spotManager, appPath, libFile, sysBinPath);
```

The SpotClient class provides the same functions as the individual SpotClientCommands but presented as methods on a single class. This alternative API is more compatible with previous releases.

Execute commands

The following code samples show how to flash an application to the Spot using the two alternative APIs:

```
…
commandRepository.getFlashAppCommand().execute("/myapp/suite/image.suite");
…
```

```
…
spotClient.flashApp("/myapp/suite/image.suite");
…
```

The execution of commands may throw various kinds of unchecked exception. Tool developers may choose to catch any or all of these:

- `SpotClientException`: abstract superclass for these unchecked exceptions:
  - `SpotClientArgumentException`: failure due to invalid parameters in API call
  - `SpotClientFailureException`: other non-fatal failure during a `SpotClient` API call. This has these subclasses:
    - `ObsoleteVersionException`: the target is running the wrong version of the bootloader or config page for the spot client executing on the host. Assuming that the host is running the latest spot client, then the solution is to flash the target with the latest bootloader before continuing
    - `SpotSerialPortInUseException`
    - `SpotSerialPortNotFoundException`
    - `SpotSerialPortException`: other exception in serial port comms
  - `SpotClientFatalFailureException`: fatal failure in execution of an `SpotClient` API call. Callers should normally exit, or at least not reuse the instances of `SpotClient`, `SpotClientCommands`, `SpotManager` and `Itarget`.

Reference

Persistent system properties

Property name	Meaning
spot.hardware.rev spot.powercontroller.firmware.version spot.sdk.version spot.external.0.part.id spot.external.0.hardware.rev spot.external.0.firmware.version spot.external.1.part.id spot.external.1.hardware.rev spot.external.1.firmware.version	The first three of these properties define the hardware, power controller firmware and SDK software versions installed on the SPOT. The items starting spot.external.0 define the first external board's part id (what kind of board it is), hardware and firmware versions. The items starting spot.external.1 define the same information for the second external board. The information about external boards is updated as a SPOT boots, so the information will not be correct after boards are added/removed and before the SPOT is rebooted.
spot.remote.print.disabled	Inhibit remote echoing of output from a SPOT communicating with a remote base station. Defaults to false. See section *Ensure that the remote Sun SPOT is executing the OTA Command Server*.
spot.sectors.reserved.for.streaming	There are 30 64k sectors of flash memory available for application data storage in the SPOT. This property controls the division of these between flash memory available to stream over (see section *Using IFlashMemoryDevice*) and the RMS system (see section *Using the Record Management Store*). By default the property has the value 8, leaving 22 sectors for the RMS system.
spot.log.connections	Log to System.out each time a radiostream or radiogram connection is made. Defaults to true. See section *Radio properties*.
spot.diagnostics	If true, SPOT outputs additional information about various operations: sometimes useful for support purposes. Defaults to false.
spot.mesh.enable	If true, a SPOT that otherwise has no radio connections still functions as mesh routing router. Defaults to false. See section *Configuring network features*.
spot.mesh.route.logging	If true, output detailed information about mesh route discovery. Defaults to false. See section *Configuring network features*.
spot.mesh.traceroute.enable	If true, participate in traceroute requests. Defaults to false. See section *Configuring network features*.
spot.ota.enable	If true, the OTACommandServer is run at startup. See section *Ensure that the remote Sun SPOT is executing the OTA Command Server*.

Memory usage

The Sun SPOT flash memory runs from 0x10000000 to 0x10400000 (4M bytes), and is organized as 8 x 8Kb followed by 62 x 64Kb followed by 8 x 8Kb sectors. The flash memory is allocated as follows:

Start address	Space	Use
0x10000000	64Kb	Bootloader
0x10010000	256Kb	VM executable
0x10050000	512Kb	Squawk bootstrap suite bytecodes
0x100D0000	448Kb	Library suite bytecodes
0x10140000	384Kb	Application slot 1
0x101A0000	384Kb	Application slot 2
0x10200000	2Mb less 16Kb	Available for data storage
0x103FC000	16Kb	Persistent properties and config page

The Sun SPOT external RAM is mapped to run from 0x20000000 to 0x20080000 (512K bytes).

SDK files

The SDK installer places a number of files and directories into the SDK directory specified during installation. This section explains the purpose of each file and directory.

sunspot-sdk

Ant	[Private - holds ant scripts]
Arm	Directory holding binary files specific to the Sun SPOT
Bin	Host-specific executables
Doc	Documentation
Lib	Various jar files
upgrade	Files used to upgrade firmware during “ant upgrade”
Src	Library source code
Tests	Test programs
build.xml	The master ant build script
Default.properties	Default property settings for the master ant build script
index.html	Index into supplied documentation
SunSPOT.inf	[Private – a copy of the Windows USB device information file, which should not be needed by the user]
version.properties	The version of the installed SDK

Contents of the arm directory:

bootloader-spot.bin	The ready-to-flash version of the bootloader for the Sun SPOT device.
spotlib.suite	The base Sun SPOT device library suite.
squawk.suite	The bootstrap suite used when creating Sun SPOT application suites.
transducerlib.suite	A Sun SPOT library suite containing the base library, the comms stack and the eDemo board library.
vm-spot.bin	The ready-to-flash version of the VM executable.

Contents of the lib directory:

debugger_classes.jar	The classes of the Sun SPOT high-level debugger. Used by the debug proxy.
desktop_signing.jar	Classes that are used to security sign suites and commands.
j2se_classes.jar	The J2SE classes used by Squawk on the desktop. Used by the debug proxy.
junit.jar	The well-know Java testing framework, supplied here for use in host applications, NOT the Sun SPOT.
multihoplib_rt.jar	The classes of the Sun SPOT comms stack library.
networktools.jar	The host-side classes for trace route and other network facilities
romizer_classes.jar	The classes of the Squawk suite creator. Used by the debug proxy.
RXTXcomm.jar	Classes that bind to the RXTX serial comms library to provide access to a serial port from Java host programs.
rxtxSerial.dll	Run-time library for rxtx (file name will vary with operating system)
sdp_classes.jar	The classes of the debug proxy.
sdproxylauncher.jar	The classes for the program that launches the debug proxy.
singlehoplib_rt.jar	The classes of the old Sun SPOT single-hop only comms stack library.
socket_proxy.jar	The host-side proxy for the http connection support.
spotclient.jar	The debug client application. Type *ant debug* to see how to run it directly.
spotlib_common.jar	The classes of the Sun SPOT base library that are common to both the host and device usage.
spotlib_device.jar	The classes of the Sun SPOT base library that are specific to execution on the Sun SPOT.
spotlib_host.jar	The classes needed to build host applications that use radio connections via the base station.
squawk.jar	One of the set of files needed to run Squawk on the host.
squawk.suite	The bootstrap suite used with Squawk on the host.
squawk.sym	Symbolic information for the bootstrap suite.
squawk_classes.jar	The Squawk bootstrap classes in standard J2SE form. Host applications have this in their classpath.
squawk_rt.jar	The Squawk bootstrap classes stripped to contain just those parts of squawk available to Sun SPOT applications. Sun SPOT applications compile against this.
transducerlib_rt.jar	The classes of the Sun SPOT eDemo board library.
translator.suite	The bytecode translator suite used by Squawk when building suites.
translator_classes.jar	The classes of the translator. Used by the debug proxy.

Contents of the bin directory:

preverify.exe	The J2ME pre-verifier program. File extension may vary.
spotfinder.exe	The program that identifies which COM ports map to Sun SPOTs. File extension may vary.
squawk.exe	The Squawk executable for the host. File extension may vary.

Contents of the src directory:

multihoplib_source.jar	Source code for the radio communications stack.
sdproxylauncher_source.jar	Source code for the debugger proxy program.
singlehoplib_source.jar	Source code for the old single-hop-only radio communications stack.
spotclient_source.jar	Source code for the SpotClient program.
spotlib_host_source.jar	Source code for the Sun SPOT base library (the part that is specific to host applications).
spotlib_source.jar	Source code for the Sun SPOT base library.
spotselector_source.jar	Source code for the program that selects which SPOT to use with ant commands.
transducerlib_source.jar	Source code for the Demo Sensor Board library.

Contents of the doc directory:

spot-developers-guide.pdf	This manual
README.txt	A pointer to the index.html file in the SDK directory.
javadoc	A sub-directory containing Javadoc for the Squawk base classes (a superset of the J2ME standard), the demo sensor board library and for the SPOT base library.

Contents of the upgrade directory:

demosensorboardfirmware.jar	Current version of the firmware for the eDemo board.
pctrlfirmware.jar	Current version of the firmware for the main board power controller.

Contents of the tests directory:

spottests.jar

BasestationTests.jar

RadioStackTests.jar

The installer also creates a `.sunspot.properties` file and a `sunspotkeystore` folder in your user home directory (whose location is operating-system specific). The main purpose of `.sunspot.properties` is to specify the location of the SDK itself, but you can edit `.sunspot.properties` and insert user-specific property settings. Any such settings override settings in an application's `build.properties` file and those in the `default.properties` file. `sunspotkeystore` contains the public-private key pair used to sign library and application suites flashed onto your SunSPOTs (for more information, see the section *Managing keys and sharing Sun SPOTs*).