

# Sun™ SPOT Programmer's Manual

---

Release v6.0 (Yellow)

*Sun Labs*  
*November 2010*



Document Revision 2.0  
November 2010

Copyright © 2005–2010, Sun Microsystems, Inc. All rights reserved.  
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

Copyright © 2005–2010, Sun Microsystems, Inc. Tous droits réservés.  
Copyright © 2010, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. UNIX est une marque déposée concédée sous licence par X/Open Company, Ltd.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation.

# Contents

<a href="#">Part I: Introduction to the Sun™ SPOT.....</a>	<a href="#">6</a>
<a href="#">Contents of the Sun SPOT Kit.....</a>	<a href="#">6</a>
<a href="#">How to Open a Sun SPOT.....</a>	<a href="#">7</a>
<a href="#">Guided Tour of Sun SPOT Switches and LEDs.....</a>	<a href="#">8</a>
<a href="#">Powering a SPOT.....</a>	<a href="#">10</a>
<a href="#">The Tutorial: Getting Started with Sun SPOTs.....</a>	<a href="#">11</a>
<a href="#">Sun SPOT Manager Tool.....</a>	<a href="#">12</a>
<a href="#">Starting Sun SPOT Manager.....</a>	<a href="#">12</a>
<a href="#">Using the SPOTManager Tool.....</a>	<a href="#">13</a>
<a href="#">Sun SPOTs Tab.....</a>	<a href="#">13</a>
<a href="#">SPOTManager SDKs Tab.....</a>	<a href="#">15</a>
<a href="#">Solarium Tab.....</a>	<a href="#">18</a>
<a href="#">Console Tab.....</a>	<a href="#">19</a>
<a href="#">Share Tab.....</a>	<a href="#">20</a>
<a href="#">Preferences Tab.....</a>	<a href="#">21</a>
<a href="#">Solarium Tool.....</a>	<a href="#">22</a>
<a href="#">Discovering and Displaying SPOTs.....</a>	<a href="#">22</a>
<a href="#">Isolates and Jar Files.....</a>	<a href="#">26</a>
<a href="#">Example: Creating a Jar File with Multiple MIDlets.....</a>	<a href="#">26</a>
<a href="#">Manipulating Sun SPOTs in Solarium.....</a>	<a href="#">30</a>
<a href="#">Pausing, Resuming and Terminating Applications.....</a>	<a href="#">34</a>
<a href="#">Radio View.....</a>	<a href="#">35</a>
<a href="#">Managing a Network of SPOTs.....</a>	<a href="#">37</a>
<a href="#">Extending Solarium.....</a>	<a href="#">39</a>
<a href="#">Using the SPOT Emulator in Solarium.....</a>	<a href="#">41</a>
<a href="#">Sun SPOT Programming Basics.....</a>	<a href="#">48</a>
<a href="#">Debugging on a Sun SPOT.....</a>	<a href="#">48</a>
<a href="#">OTA Debugging.....</a>	<a href="#">48</a>
<a href="#">Print Debugging.....</a>	<a href="#">52</a>
<a href="#">Accessing the Sensor Board.....</a>	<a href="#">53</a>
<a href="#">Resources.....</a>	<a href="#">53</a>
<a href="#">Accelerometer.....</a>	<a href="#">55</a>
<a href="#">LEDs.....</a>	<a href="#">56</a>
<a href="#">Switches.....</a>	<a href="#">57</a>
<a href="#">Light Sensor.....</a>	<a href="#">58</a>
<a href="#">Temperature Sensor.....</a>	<a href="#">58</a>
<a href="#">Creating your own Sun SPOT Resources.....</a>	<a href="#">59</a>
<a href="#">Conditions.....</a>	<a href="#">59</a>
<a href="#">Radio Communication.....</a>	<a href="#">61</a>
<a href="#">Radiostreams.....</a>	<a href="#">61</a>

Radiograms.....	61
<b>Part II: Developing for the Sun SPOT.....</b>	<b>63</b>
<b>Building and deploying Sun SPOT applications.....</b>	<b>64</b>
Deploying and running a sample application.....	64
The MIDlet lifecycle.....	73
Manifest and resources.....	73
Managing multiple MIDlets on the SPOT.....	74
Using the Basestation.....	76
Remote operation.....	77
Managing keys and sharing Sun SPOTs.....	79
Deploying and running a host application.....	81
Configuring network features.....	83
Hardware configurations and USB power.....	84
Batch operations.....	85
<b>Developing and debugging Sun SPOT applications.....</b>	<b>85</b>
Understanding the SPOT system architecture.....	85
Overview of an application.....	88
Isolates.....	89
Threads.....	89
The Sun SPOT device libraries.....	89
The radio communication library.....	94
Conserving power.....	104
IPv6 Support.....	109
http protocol support.....	114
Configuring projects in an IDE.....	115
Debugging.....	116
<b>Part III: Working with the Basic Sun SPOT Utilities.....</b>	<b>120</b>
<b>Understanding the ant scripts.....</b>	<b>120</b>
<b>Using library suites.....</b>	<b>120</b>
Introduction.....	120
Adding user code to the library suite.....	121
Library manifest properties.....	122
Running startup code.....	123
Modifying the system library code.....	123
Recovery from a broken library.....	123
<b>Extending the ant scripts.....</b>	<b>124</b>
<b>Sending commands to SPOTs from a host application.....</b>	<b>124</b>
Implementing IUI.....	124
Construct a SpotClientCommands object.....	125
Execute commands.....	125
<b>Extending the SPOT Client.....</b>	<b>127</b>
Adding or replacing SPOT Client commands.....	127
Extending the OTACCommandProcessor.....	128

<a href="#">Hardware revision compatibility.....</a>	<a href="#">129</a>
<a href="#">Reference.....</a>	<a href="#">130</a>
<a href="#">Persistent system properties.....</a>	<a href="#">130</a>
<a href="#">Memory usage (rev 6).....</a>	<a href="#">131</a>
<a href="#">Memory usage (rev 8).....</a>	<a href="#">132</a>
<a href="#">SDK files.....</a>	<a href="#">132</a>
<a href="#">Troubleshooting.....</a>	<a href="#">138</a>
<a href="#">Software, All Platforms.....</a>	<a href="#">138</a>
<a href="#">Software, Linux.....</a>	<a href="#">140</a>
<a href="#">Sun SPOTs in General, Hardware.....</a>	<a href="#">141</a>
<a href="#">If Your Sun SPOT Needs Factory Service.....</a>	<a href="#">143</a>
<a href="#">Battery Warnings.....</a>	<a href="#">144</a>
<a href="#">Federal Communications Commission Compliance.....</a>	<a href="#">145</a>

## **Part I: Introduction to the Sun™ SPOT**

This portion of the document provides a quick introduction to the Sun SPOT (Small Programmable Object Technology) kit, for Yellow software release 6.0.

### **Contents of the Sun SPOT Kit**

A Sun SPOT kit contains the following:

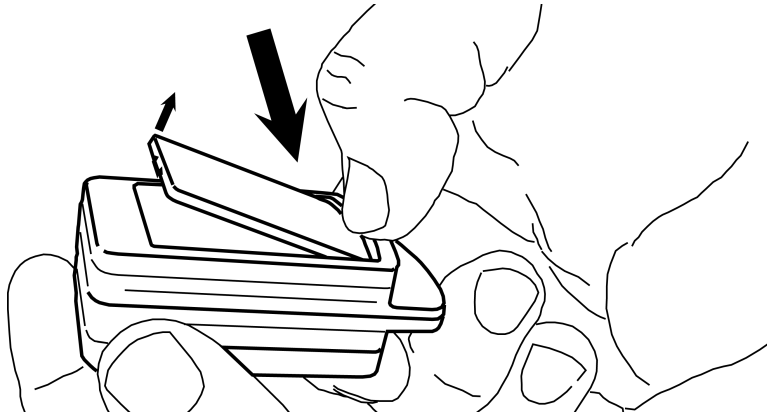
- one basestation Sun SPOT unit with USB power
- two free-range Sun SPOT units with onboard battery
- a USB cable for connection between a standard USB port and a Sun SPOT unit
- one Sun SPOT software CDROM
- two mounting brackets, each allowing a Sun SPOT unit to be wall-mounted
- one mounting bracket to allow mounting of a Sun SPOT to a circuit board

Dimensions of the Sun SPOT are 41 x 23 x 70 mm. Weight is about 54 grams.

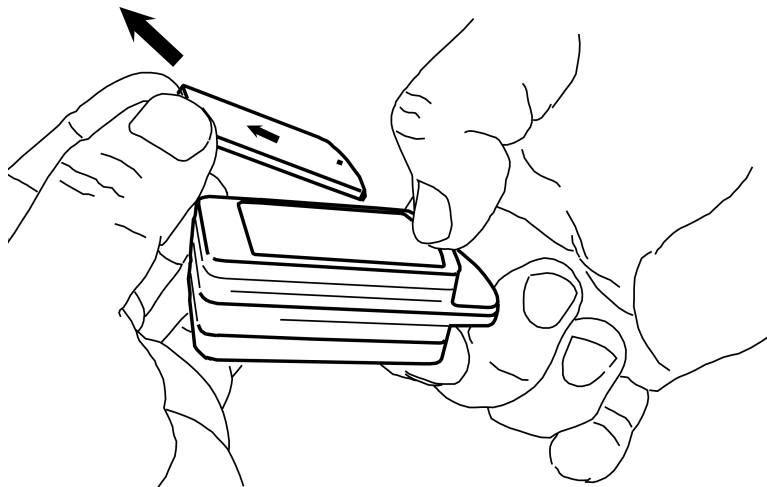
## How to Open a Sun SPOT

You must open the Sun SPOT unit lid to be able to reach the switches and LEDs on the sensor board. To open the lid, press down *firmly*, down and back, on the edge of the lid near the small raised dot. You can think of that small raised dot as the fingernail-catching dot. The closer to the edge of the lid that you press, the easier the lid will open. The opposite end of the lid pops up.

Press down firmly on the edge of the lid marked with a small raised dot.

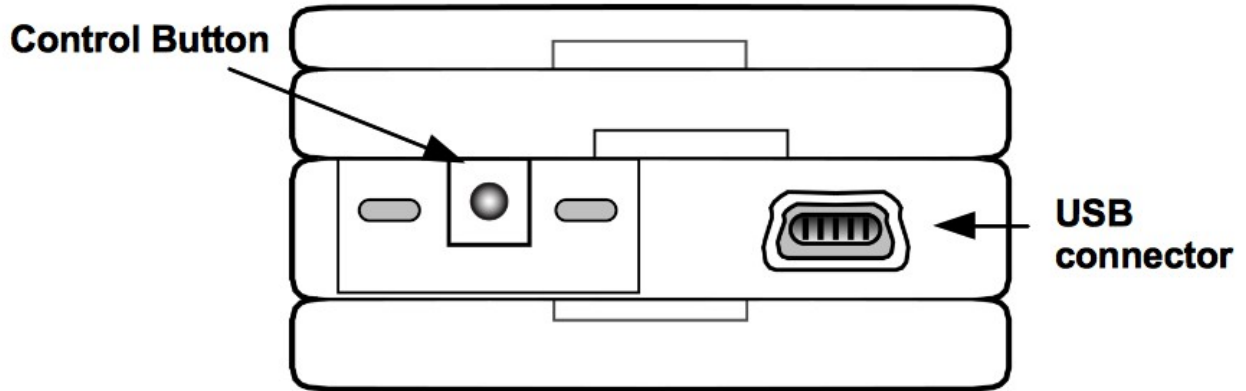


After the lid pops up, pull the lid out and away.



## Guided Tour of Sun SPOT Switches and LEDs

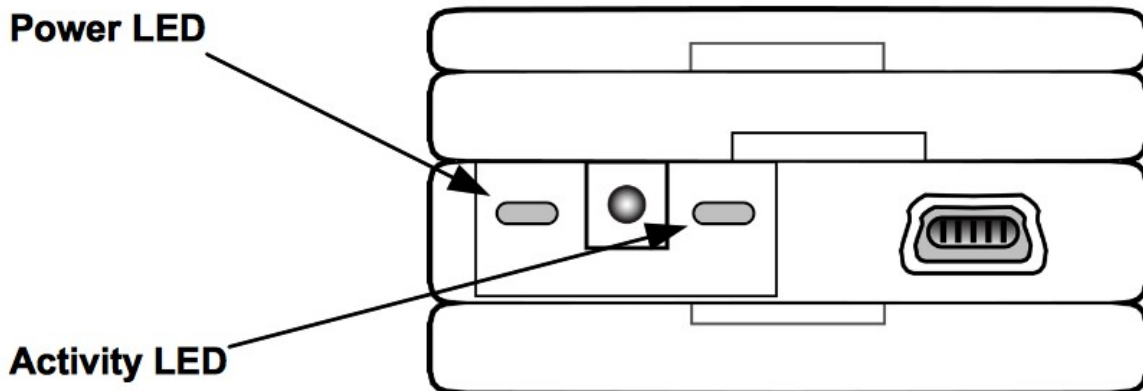
The Sun SPOT unit has one switch and one connector that are accessible without removing the case lid. These are shown below:



The connector is the micro-USB connector that allows the Sun SPOT unit to be connected to a host workstation.

The switch is the Sun SPOT unit control switch. If the Sun SPOT unit is off, pressing the switch turns the Sun SPOT unit on and causes it to boot. If the Sun SPOT unit is on, pressing the control switch causes the Sun SPOT unit to reboot. If the Sun SPOT unit is on, pressing the control switch and holding it down turns the Sun SPOT unit off.

This end of the Sun SPOT unit also has two LEDs behind the plastic casing.



The power LED is to the left of the power switch. This LED exhibits the following behaviors.

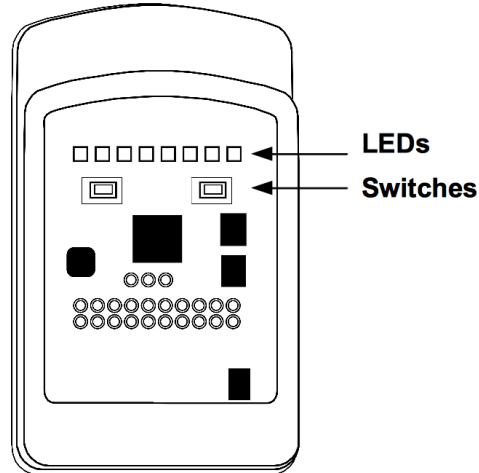


<i>Power State</i>	<i>Power LED Behavior</i>
Powering up	One bright green pulse, sharp on, soft off
Powering down	Three bright red flashes
Charging the battery while CPU is active	Slowly alternate between a dim green and a bright green on a eight second cycle
Charging the battery when CPU is asleep	Slowly alternate between off and a dim green on an eight second cycle
External power supplied, but not charging, CPU active	Steady dim green
Battery low	Steady dim red. <i>This is a change from Release 1.0.</i>
Power fault	Two short red flashes
CPU going to sleep	Short red flash, short green flash
External interrupt or alarm, including button tap.	One short green flash

The activity LED is to the right of the power switch. This LED is under Java program control and can be used in your applications, but it is usually used by the system software. Some of these uses are:

- When the Sun SPOT unit is attempting to synchronize with the USB connection to a host workstation, the activity LED flashes amber 16 times a second. This flashing lasts for two seconds or until synchronization is complete, whichever comes first.
- When the Sun SPOT unit is being used as a basestation, that is, for wireless communication between a host workstation and free-range Sun SPOT units, the green component of the activity LED changes state, i.e., switches from off to on or the reverse, for every packet transmitted by the Sun SPOT unit for the host workstation. The red LED component of the activity LED changes state for every packet received by the radio and sent to the host workstation from the Sun SPOT unit. If no packets are being sent or received, then the green activity LED blinks twice every 10 seconds to indicate that the basestation is functioning.

If the Sun SPOT unit lid has been removed, there are two switches and eight multi-color LEDs that become accessible.



The LEDs have red, blue, and green components. The switches and LEDs have no pre-defined purpose and are under Java program control.

## Powering a SPOT

The capacity of the built-in battery is 720 milliampere-hours.

The drain of the system varies with use.

Power usage for a typical Sun SPOT unit is shown in the table below.

<i>Processor board state</i>	<i>Radio</i>	<i>Sensor Board</i>	<i>Current Draw</i>
Deep sleep mode <sup>1</sup>	Off	Any	~33 microamperes
Shallow sleep <sup>2</sup>	Off	Not present	~24 milliamperes
Shallow sleep	On	Not present	~40 milliamperes
Awake, actively calculating	Off	Not present	~80 milliamperes
Awake, actively calculating	On	Not present	~98 milliamperes
Shallow sleep	Off	Present	~31 milliamperes
Shallow sleep	On	Present	~46 milliamperes
Awake, actively calculating	Off	Present	~86 milliamperes
Awake, actively calculating	On	Present	~104 milliamperes

1. In deep sleep, the processor and sensor board are both powered down.

2. Shallow sleep means devices active, but no active threads.

Changing the transmit power of the radio effects the current draw slightly. Reducing the transmission power from 0db to -25db results in a savings of about 3 milliamperes.

LEDs also use power. The power draw for a single Sun SPOT LED is specified in the table below.

<i>LED</i>	<i>Current Draw</i>
All elements, full brightness	25 milliampere
Blue element, full brightness	10 milliampere
Red element, full brightness	9 milliampere
Green element, full brightness	5 milliampere

The current draw for the LEDs is reasonably linear. An LED at half-brightness draws approximately half the current of an LED at full brightness. Reducing LED brightness to the minimum required for your situation is often a good way to conserve power. Often LED levels of 20, out of a possible 255, are reasonably visible.

The approximate length of time that a full-charged Sun SPOT unit can operate is shown below for most of the conditions of interest.

<i>Sun SPOT State</i>	<i>Battery Life Estimate</i>
Deep sleep	909 days
Shallow sleep, no radio	23 hours
Shallow sleep, radio on	15 hours
CPU busy, no radio	8.5 hours
CPU busy, radio on	7 hours
Shallow sleep, 8 LEDs on, no radio	3 hours

A power fault occurs when one of these conditions occurs:

- External power exceeds 5.5V
- $V_{\text{batt}}$  exceeds 4.9V
- $V_{\text{CC}} \pm 10\%$  of 3.0V
- $V_{\text{core}} \pm 10\%$  of 1.8V
- Battery Discharge current exceeds 500ma

A Sun SPOT may also be powered by removing the demo sensor board and supplying power to pins J1 and J2 of the CPU-board top connector. The power should be between 4.5v and 5.5v and at least 1A.

## The Tutorial: Getting Started with Sun SPOTs

The Sun SPOT SDK comes with a brief tutorial that teaches you the basics of using and programming the Sun SPOTs. We strongly suggest that you run through it before you do anything else with your

SPOTs. It takes about half an hour is well worth the time. It is the best way to get started with your Sun SPOTs.

The tutorial is a series of web pages, with the first page located at

```
[SpotSDKdirectory]/doc/Tutorial/Tutorial.html
```

where [SpotSDKdirectory] represents the directory in which the SPOT SDK was installed. On a Windows machine, this would typically be:

```
C:\Program Files\Sun\SunSPOT\sdk
```

## Sun SPOT Manager Tool

The Sun SPOT SDK comes with two important tools for managing the software on your SPOTs: SPOTManager and Solarium.

The SPOTManager is a tool for managing the Sun SPOT SDK software. You can use it to download from the Internet both new and old versions of the Sun SPOT SDK. You can use it to make one or another SDK the active SDK on your host workstation, and you can use it to download system software to your Sun SPOTs.

### ***Starting Sun SPOT Manager***

The SPOTManager tool is implemented as a Java Network Launchable Program (JNLP) file. To start the SPOTManager tool, connect to:

```
http://www.sunspotworld.com/SPOTManager/
```

and click the image of a SPOT, or, alternately, access the URL:

```
http://www.sunspotworld.com/SPOTManager/SPOTManager.jnlp
```

and the latest version of the SPOTManager application downloads to your host workstation and starts.

If you start SPOTManager with one of these URLs, you will always have the latest copy of the SPOTManager software. However, if you want to keep a cached copy of SPOTManager to operate when your host workstation is not connected to the Internet, follow these steps:

### **Launch the Java Control Panel**

In Windows, this is under *Start > Control Panel*. The Java control panel may not be visible if your control panels are organized in category view. If this is the case, select *Switch to Classic View* to gain access to the Java control panel.

**Under the General tab, go to the Temporary Internet Files panel and click the *View* button.**

This should display the Java Cache Viewer.

**Select the SPOT Manager application. Click the shortcut icon (an arrow pointing up and to the right) in the menu bar.**

This should create a local shortcut to the SPOTManager application as stored in the cache.

## **Using the SPOTManager Tool**

The SPOTManager tool, as currently configured, has eight tabs across the top:

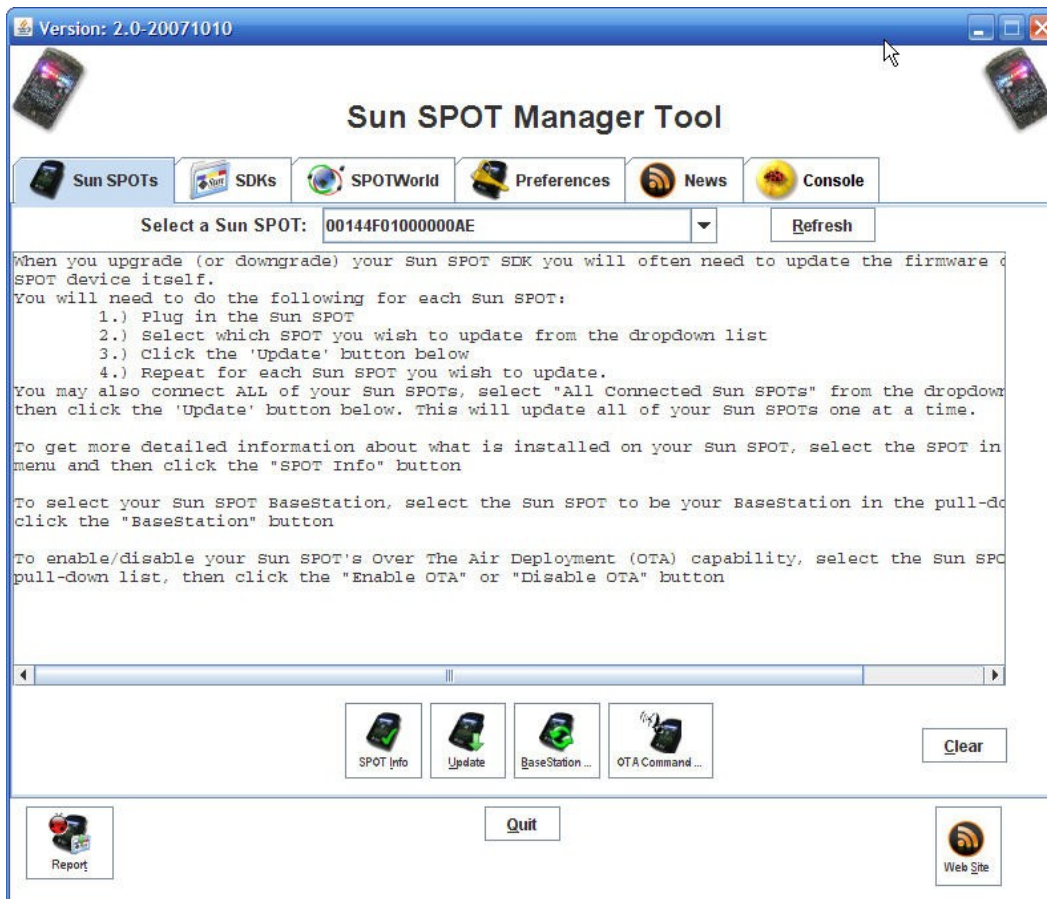
- Sun SPOTs – allows you to query the configuration of individual Sun SPOTs, download and configure the system software on them.
- SDKs – allows you to download and install versions of the Sun SPOT SDK available on the Sun SPOT website.
- Solarium – launches a tool for managing individual Sun SPOTs and the application software on them. This tool also contains a Sun SPOT emulator capable of running and testing Sun SPOT application software.
- Tutorial – allows you to view the Sun SPOT Tutorial
- Docs – allows you to browse the documentation for the currently active SDK
- Console – displays standard output for the SPOTManager tools. If there is new error output here, the type for the console tab label is red. Otherwise, it is black.
- Share – allows you to read and rate sample code snippets, plus submit your own snippets
- Preferences – configures the network and polling characteristics of the SPOT Manager tool itself.

It also has three buttons across the bottom of the window, no matter which tab is displayed. The buttons are:

- Support – allows you to report a bug in Sun SPOTs, Sun SPOT software, or the SPOTManager. If you allow, it collects and includes configuration information on the connected Sun SPOTs at the time the report is produced.
- Quit – quits the SPOTManager tool.
- Forums – opens a web browser with the URL set to the Sun SPOT Forums website, [www.sunspotworld.com/forums/](http://www.sunspotworld.com/forums/).

## **Sun SPOTs Tab**

The Sun SPOTS tab has a menu for selecting among the USB-connected Sun SPOTs, a large text output area, and eight buttons particular to this tab.



The menu above the console area is labeled “Select a Sun SPOT” and the pull-down menu lists all the Sun SPOTs that are connected by USB cables to the host workstation. Each Sun SPOT is listed by its IEEE network number. The first eight digits of this number are always 0014.4F01. The last eight digits are written on a sticker visible through the plastic surrounding the radio antenna fin on the Sun SPOT.

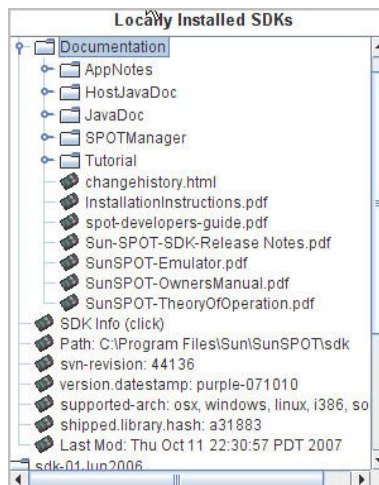
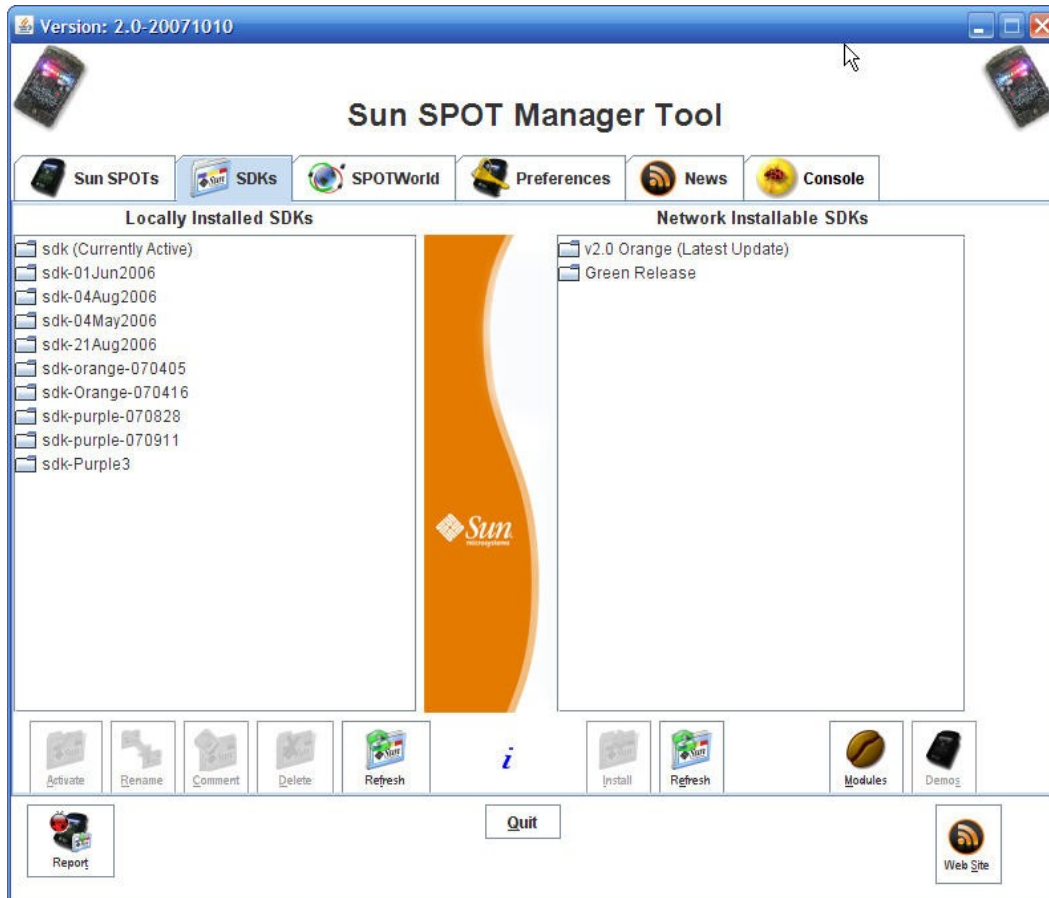
The eight buttons all take an action with respect to the selected Sun SPOT. The eight buttons are described below.

<i>Button</i>	<i>Description</i>
Upgrade	Does an <code>ant upgrade</code> on the selected Sun SPOT, loading the SPOT with the runtime software and firmware required for the current version of the SDK.
Properties	Displays the current system properties of the selected Sun SPOT and allows them to be modified.
SPOT Info	Runs <code>ant info</code> on the selected Sun SPOT and displays the information in the output area. The <code>ant info</code> command output describes the software configuration of the SPOT.
Echo	Echo output from the selected Sun SPOT
Basestation	Brings up a pop-up menu where you can enable or disable the selected SPOT as a basestation. This is ordinarily done on a Sun SPOT that does not have a battery or a demo sensor board. You are given a choice of a plain basestation (more efficient, can only talk to one free-range SPOT at a time) or a shared basestation (more overhead, can talk to multiple free-range SPOTs).
OTA Command	Enables or disables Over The Air (OTA) command processing on the selected SPOT. A pop-up menu allows you to select either Enable OTA or Disable OTA. Enabling OTA allows you to deploy, run, and debug application programs on a Sun SPOT using the radio link between the basestation and the selected Sun SPOT. <b>Note:</b> The OTA command server should only be enabled on the free-range SPOTs. It is not needed nor intended to be enabled on the basestation SPOT.
Restore	Does an <code>ant restore</code> on the selected Sun SPOT, resetting the SPOT with the runtime software and firmware it had when it came from the factory for the current version of the SDK.
Clear	Clears the output window of all text.

### ***SPOTManager SDKs Tab***

The SDKs tab shows the Sun SPOT SDKs that are available on the local workstation. It also shows the SDK versions that are currently available on the Sun SPOT website.

The left column lists the Sun SPOT SDK versions installed on the host workstation. The right column lists the Sun SPOT SDK versions available on the Sun SPOT web site. Clicking the folder icon for an SDK expands the information displayed and, in some cases, provides clickable links to documentation available as part of that SDK.





The buttons below the lists of SDKs take action on or with the SDKs. The buttons in the Locally Installed SDK menu are described below.

<i>Button</i>	<i>Description</i>
Activate	Makes the selected locally installed SDK into the active SDK; records this SDK in the user's <code>.sunspots.properties</code> file as the SDK to be used for all <code>ant</code> commands.
Rename	Allows you to rename the selected Sun SPOT SDK directory. <b>Note:</b> Under Windows it is not always possible to rename the active SDK.
Comment	Allows you to add a comment line to the SDK. This is displayed when the SDK node is expanded as shown in the figure above this table.
Delete	Deletes the selected SDK from the local disk.
Refresh	Refreshes the list of local SDKs.

There are two buttons under the Network Installable SDKs menu. They are described below.

<i>Button</i>	<i>Description</i>
Install	Downloads the SDK from the Sun SPOT website and installs it on the host workstation. Clicking this button first opens a software license window. After you accept the license terms, it opens a pop-up window where you can specify the directory where the SDK will be installed. If the SDK name is already in use, the system attempts to rename the previous directory. If it cannot rename that directory, the install fails.
Refresh	Polls the Sun SPOT website to see which Sun SPOT SDKs are available.

## Solarium Tab

This tab allows you to start an application called Solarium. Solarium makes it easier to manage a group of Sun SPOTs and the application software for those SPOTs. Solarium also contains a SPOT emulator that is useful for testing SPOT software. The use of Solarium is explained further in the section “Solarium Tool” on page 22.

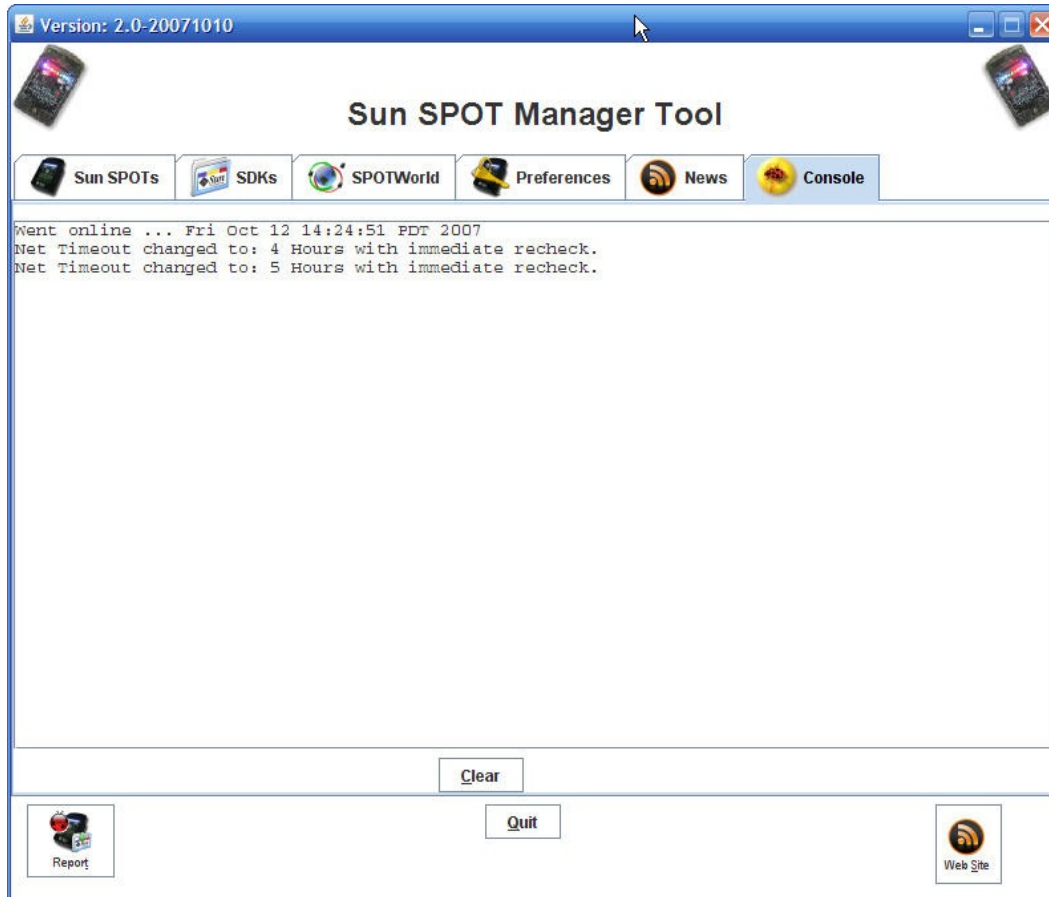
Solarium enables a basestation SPOT to communicate with any real Sun SPOTs. A basestation is not required if you only want to communicate with emulated SPOTs. A shared basestation is required if you want to have emulated SPOTs communicate over the air with real SPOTs.

If you start Solarium by using the Solarium button here, SPOTManager displays a dialog box asking you to specify which of the USB-connected SPOTs is intended to be the basestation for Solarium.



## Console Tab

The Console tab displays `System.out` and `System.err` for the process in which SPOTManager itself is running. Also, the output for any daughter processes that has not been redirected to its own output console goes to this display. For example, if you issue an Upgrade command in the Sun SPOTs tab, because the Sun SPOTs tab has an output area, the output goes to that output area. However, if you were to change the Network Timeout value under the Preferences tab, this tab does not have an output window of its own, so the output from that command goes to the Console display here.

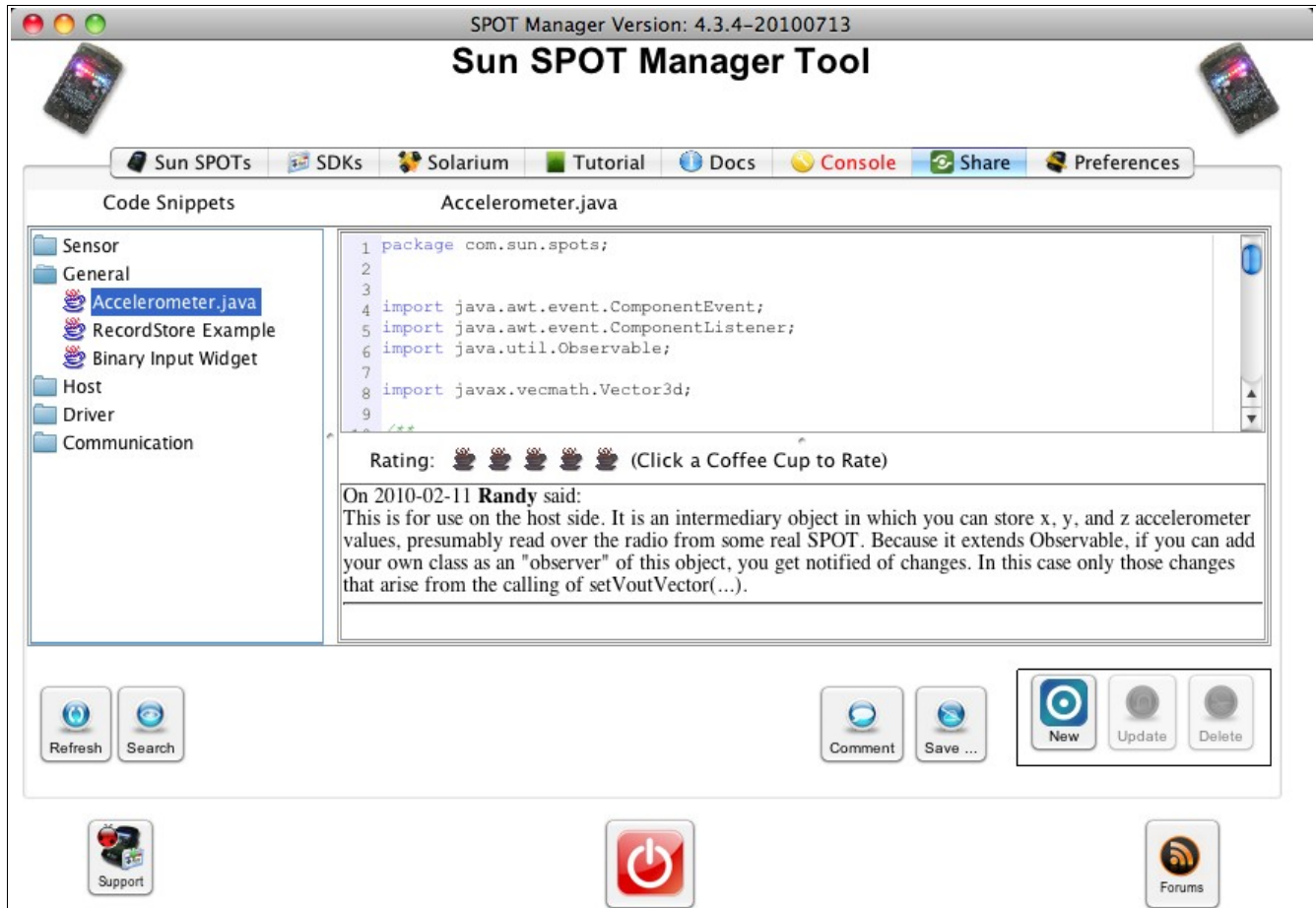


The *Clear* button empties the console display.

If you have a problem with a command, useful error messages are often displayed here.

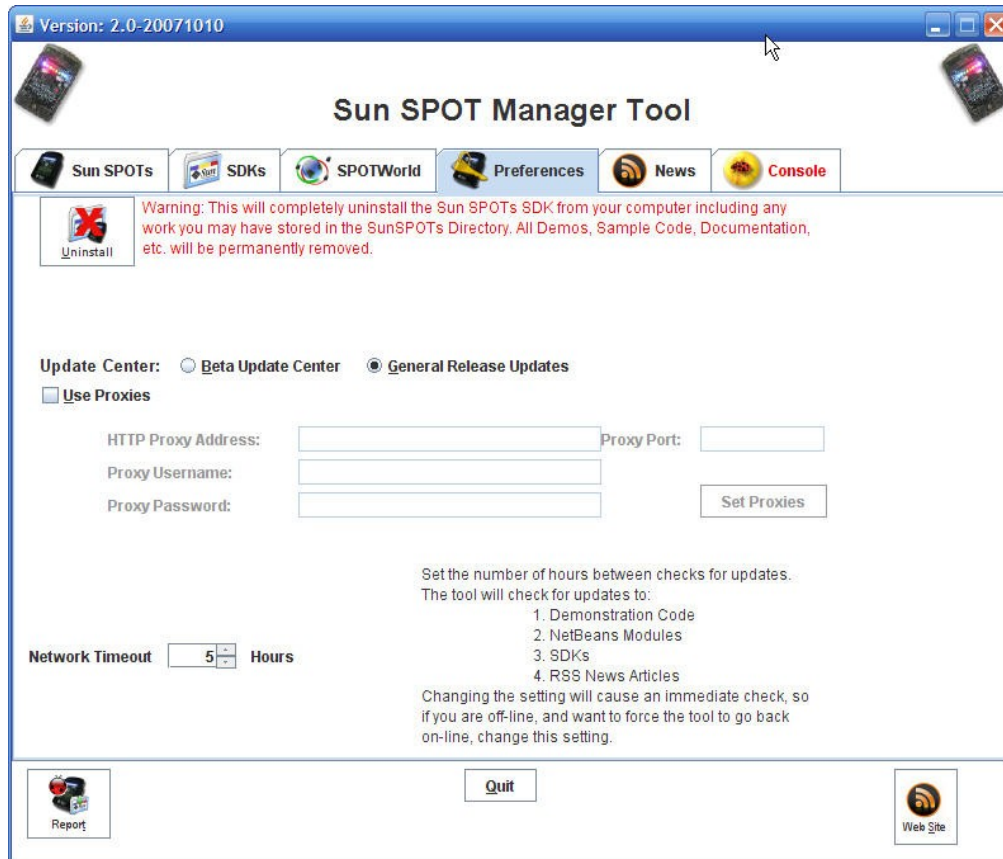
## Share Tab

The Share tab allows you to read and rate sample code snippets, plus submit your own snippets.



## Preferences Tab

The Preferences tab allows you to control the operation of the SPOTManager tool itself.



Selecting the *Uninstall* button removes all Sun SPOT SDK directories from your host workstation, including the demos and the documentation.

The Update Center settings allow you to specify which SDKs are listed as downloadable on the SDK page. If “Beta Update Center” is selected, then fully released software and beta software are shown. If “General Release Updates” is selected, then only fully released software appears.

If your Internet connection requires you to use proxies, select “Use Proxies” and specify the proxy host address, port, username, and password.

Finally, the “Network Timeout” setting controls how often the SPOTManager tool checks the Sun SPOT server for updates. Changing this setting also causes the software to refresh the listings immediately after the change is made.

# Solarium Tool

Solarium is a Java™ application that can be used to remotely manage a network of Sun SPOTs. With Solarium, you can discover nearby SPOTs and manage the complete life-cycle of applications running on those devices. Here is some of what Solarium has to offer:

**Discovering and displaying SPOTs** – Solarium can discover and display SPOTs that are connected to the desktop via USB or can be reached via radio communication.

**Interacting with SPOTs** – Solarium can be used to load and unload software on a SPOT, start/pause/resume/stop applications and query the current state of a device, e.g. memory usage, and energy statistics. A Radio View provides a way to visualize the radio connectivity between SPOTs. Using a Deployment View makes it easier to describe and manage a group of Sun SPOTs.

**Managing a Network of SPOTs** – Solarium provides a special *Deployment View* to make it easier to manage a network of Sun SPOTs. A deployment specifies what application should be loaded on each SPOT and allows you to deploy those applications with a single button press. The Deployment View also shows the current status of each SPOT. A deployment can also specify any associated host applications that need to be run.

**Emulating SPOTs** – Solarium also includes an emulator that can be used to run applications on a virtual Sun SPOT. The new Robot View enables a virtual SPOT to control a virtual iRobot Create and explore several different physical environments.

Solarium can be started from the Solarium tab in SPOT Manager by clicking the Solarium button. You can also start Solarium by opening a command line window, navigating to any SPOT project directory (containing a `build.xml` file) and executing the command `ant solarium`.

## ***Discovering and Displaying SPOTs***

Solarium can detect and display SPOTs that are connected to the desktop via USB either directly or through one or more USB hubs. It can also detect nearby SPOTs via wireless communication.

### ***Turning on the OTA Command Server***

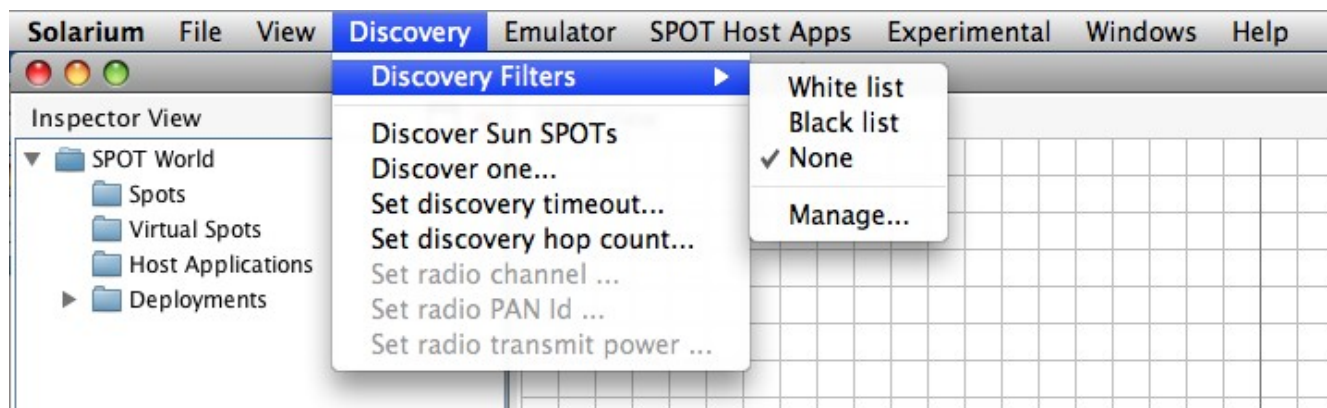
The "over-the-air" (OTA) command server is a piece of software that runs on the SPOTs and listens for management commands sent wirelessly from a host application such as Solarium.

SPOTs must have their OTA command server turned on to be discoverable via wireless communication. The OTA command server is turned on by default. If it is turned off for some reason (as indicated by `ant system-properties` showing `spot.ota.enable: false`), it can be turned on again by connecting the SPOT to the desktop via USB and using any one of the following options:

- Issuing the `ant enableota` command in a terminal window.
- Using the "Enable OTA" button in SPOT Manager

## Controlling SPOT Discovery

The **Discovery** pull-down menu in the main Solarium menu bar allows you to control the discovery process that Solarium uses to find the local SPOTs. The choices are:



- **Discovery Filters**

This command allows the discovered SPOTs to be filtered according to a white (include) or black (exclude) list. It also allows one to edit which SPOT addresses go on the white or black lists.

- **Discover Sun SPOTs**

This choice sends a broadcast discovery message, asking Sun SPOTs to identify themselves. If the broadcast message reaches a Sun SPOT running the OTA command, that SPOT will respond with basic information, e.g. its IEEE address, the version of its SDK, etc. This command also causes Solarium to scan the USB ports on the host and discover any connected SPOTs (even if they have their OTA command server turned off). Discovered SPOTs are displayed in Solarium.

- **Discover one...**

This choice allows you to send a unicast discovery message to a specific IEEE address. Solarium will attempt to find that particular SPOT. The broadcast hop count setting described below is ignored when the discovery message is sent as a unicast.

- **Set discovery time out...**

This choice specifies how long Solarium will wait for an answer to its broadcast discovery message. It defaults to three seconds.

- **Set discovery hop count...**

This choice specifies how many radio hops a broadcast discovery message may traverse before it is no longer forwarded from SPOT to SPOT. When using a shared basestation, this setting must be at least two, since the communication between the host application and the shared basestation counts as one hop. Therefore, at least two hops are required to reach any actual SPOTs. If you want Solarium to reach SPOTs beyond the direct radio range of the basestation, the hop count should be set to three or more.

- **Set radio channel...**

Allows you to specify the radio channel used for discovery. Defaults to 26.

- **Set radio PAN Id...**

Allows you to specify the PAN ID used by Solarium during discovery. Defaults to 3.

- **Set radio transmit power...**

Allows you to specify the radio transmit power used by Solarium during discovery. Defaults to full power (0).

*Note:* It is not possible to change the radio channel, PAN ID, or radio transmit power when using a shared basestation.

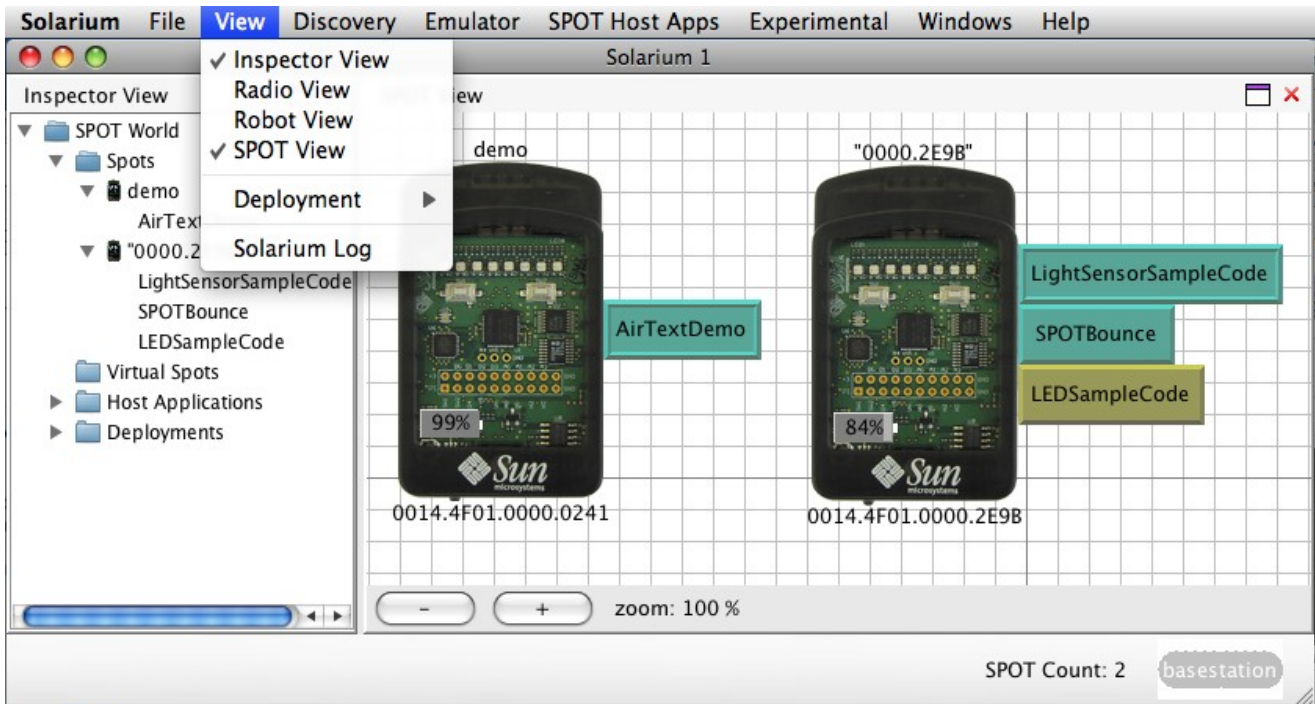
### **Controlling Which SPOTs are Displayed**

By default, Solarium displays all of the SPOTs it discovers. In an environment with lots of SPOTs, a user might want to ignore some of them to avoid cluttering the Solarium display unnecessarily. The `.solarium.properties` file in the user's home directory allows one to specify either a "blacklist" or a "whitelist" of SPOTs. If a whitelist is specified, only the SPOTs on that list are displayed, all others are ignored. If a blacklist is specified, SPOTs on that list are ignored and all others are displayed. To edit these lists, simply select the "manage..." item from the "Discovery > Discovery Filters" menu. If you wish to edit these lists without using the tool, you can (carefully) change your `.solarium.properties` file. For example, to ignore all SPOTs except those with the IEEE address of 0014.4F01.0000.020B or 0014.4F01.0000.048D insert the following line in your `.solarium.properties` file:

```
<entry key="spots.whitelist">0014.4F01.0000.020B,0014.4F01.0000.048D</entry>
```

### **Controlling How SPOTs are Displayed**

SPOTs can be displayed in several different ways by Solarium. By default Solarium starts up displaying a tree-like *Inspector View*, shown here to the left, and a 2-dimensional *SPOT View*, shown here against the quadrille background to the right. The views exposed are controlled through the **View** pull-down menu on the menu bar. Both these views also show the applications running on each SPOT.



After Solarium discovers the SPOTs, it will display all of them along with the running applications currently running on those devices.

The value of the SPOT system property name, if present, is displayed above the SPOT. If the property is not set then the low 8 characters of the SPOT's IEEE numerical address is used.



The *Deployment View*, *Radio View* and *Robot View* will be described below.

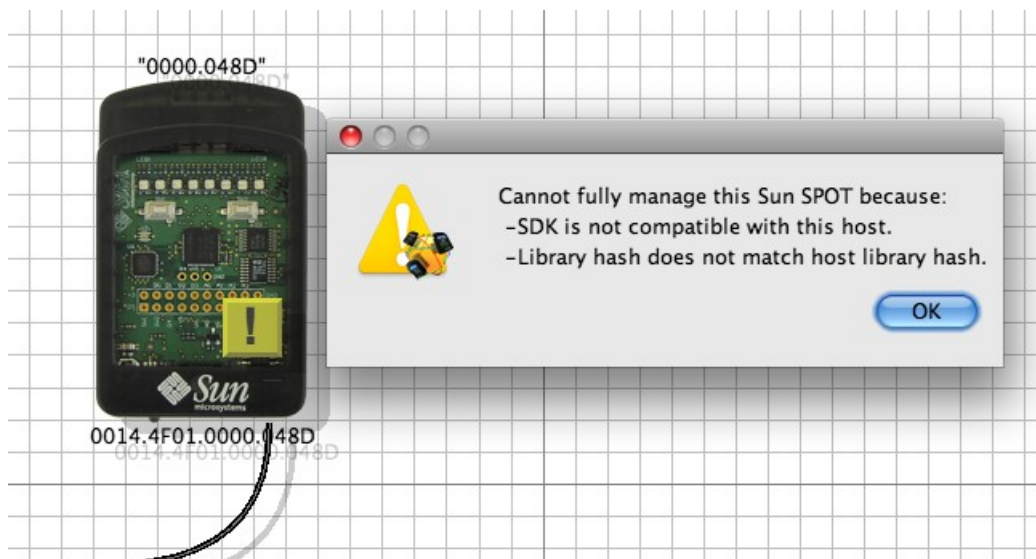
*Note:* Multiple views can be combined (tiled) in a single window. Just grab and drag the view's title bar, e.g. where it says *Inspector View*. Views can be dragged into an existing window, or dragged out onto the screen to make a new window.<sup>1</sup>

### Interpreting Visual Cues

Solarium uses a number of visual cues to indicate status information about the SPOTs:

- USB-connected SPOTs are drawn with a *tail* to indicate a USB cable.
- A yellow warning rectangle on a SPOT indicates that Solarium cannot fully manage that SPOT. Clicking on the yellow sign will bring up a panel listing all of the reasons, e.g., that SPOT may have a different owner or may have a version of the SDK that's different from the version under which Solarium is running. While SPOTs with old SDKs will be flagged, if the old SDK is still installed on the host computer and it uses the same OTA commands as the current SDK then Solarium can use the old SDK to deploy applications to the SPOT.

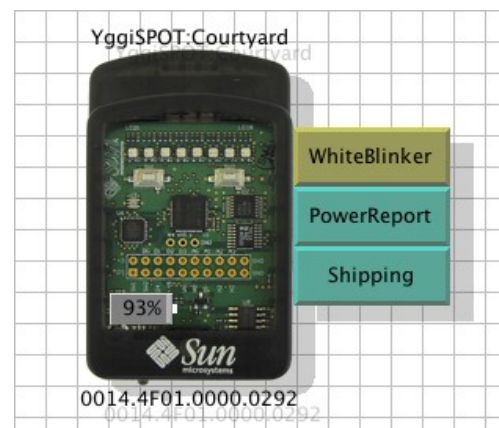
The figure below shows a USB-connected SPOT that cannot be fully managed for the reasons shown.



- A running application is drawn as a greenish-blue rectangle with a plain bezel. An application that is currently paused is drawn as a muted yellow colored rectangle.

In the figure here, the *PowerReports* and *Shipping* applications are currently running while the *WhiteBlinker* application is paused.

- Remaining battery power is shown in the small gray rectangle drawn in the bottom, left of the SPOT. In this case the SPOT has 93% of its battery capacity remaining.



<sup>1</sup>. Thanks to Benjamin Sigg, the creator of DockingFrames for his open source Java Swing docking framework, which is available from <http://dock.javaforge.com/>.

## ***Isolates and Jar Files***

Before we discuss the things that can be done with SPOTs in Solarium, we need to introduce the concepts of applications, MIDlets, isolates, and `jar` files.

In Java SE, an application consists of a static `main()` method defined in one of the loaded classes or it extends the `Applet` class if it is to run in a browser. In Java ME, an application is defined as a class that extends the `MIDlet` class. Squawk (the Java VM used by Sun SPOTs) implements Java ME, so all Sun SPOT applications extend the `MIDlet` class.

In standard Java ME, only one application can be run at a time in a Java VM, though that application may consist of many threads. Squawk allows for multiple applications to be run together in a single SPOT and uses a special *Isolate* class to prevent one application from interfering with the execution of another. Each MIDlet-based application is run in a separate isolate. While one isolate cannot directly access the instances in another, they all share the same underlying SPOT resources.

Some resources are unique, such as a radio connection on a specific port number. The first isolate to ask for that port is successfully given access to it, while any subsequent requests from other isolates fail. Other resources are truly shared such as the LEDs: one isolate might turn an LED on, and then another might turn it off or change its color.

When the SPOT is rebooted the first isolate run sets up system resources, such as the LEDs and the radio stack for the SPOT. It will then start up one or more user-specified SPOT applications, each consisting of a single MIDlet that is run in its own isolate.

Solarium allows you to start and stop multiple MIDlets on a SPOT. These MIDlets are run as separate isolates.

Java ME allows you to package up several MIDlets into a single *jar file*. The MIDlets must all be listed in a special file, `manifest.mf`. The `manifest.mf` file can be found in your project's `resources/META-INF` directory.

When you add a new MIDlet to your project you must add a new line for it in the manifest file. If you use the SPOT modules for NetBeans, you can use the pop-up menu on the package (select the *New > File/Folder* command and then select *MIDlet Class* from “Java Classes”) to create the new MIDlet. NetBeans automatically updates the manifest file. NetBeans also updates the manifest if you rename an existing MIDlet.

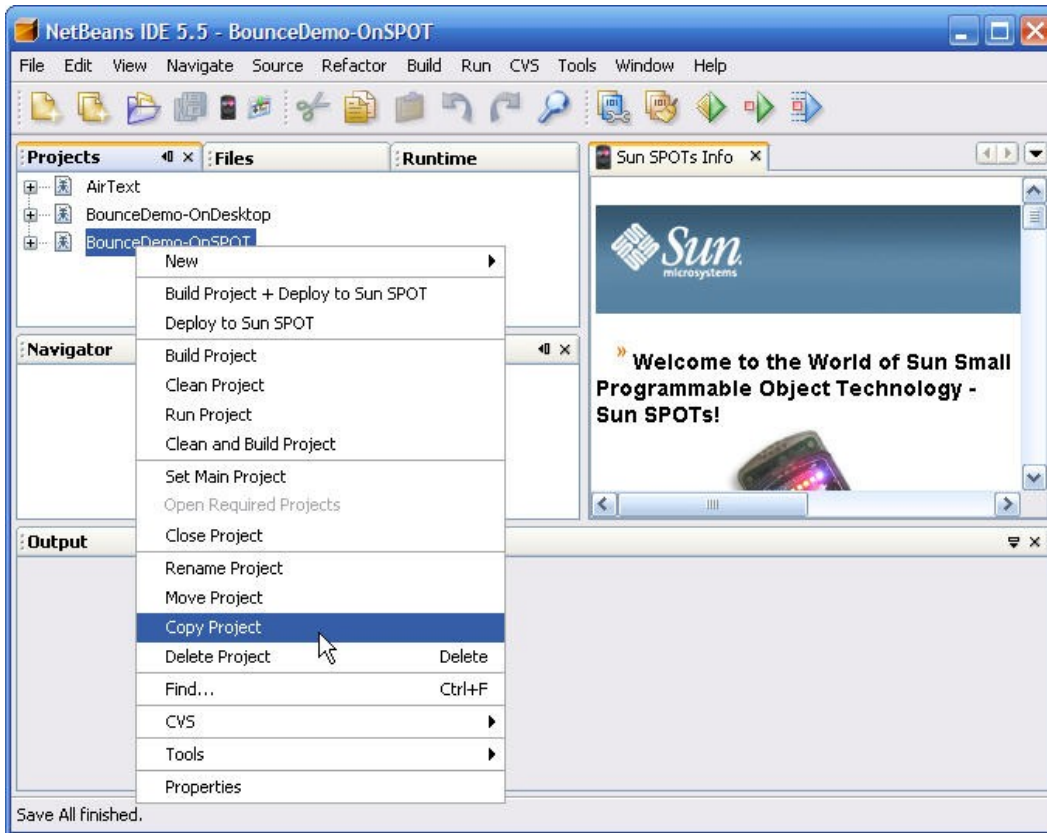
When the `jar` file is deployed to a Sun SPOT, all of the MIDlets are then available to be started from Solarium. One or more of them may be specified to be automatically run when the SPOT is rebooted. Several `jar` files may be deployed on the same SPOT.

### ***Example: Creating a Jar File with Multiple MIDlets***

With this theoretical background in place, we will briefly go through an example of the creating of a `jar` file with two MIDlets. We will use two existing demonstration projects: the Bounce demo and the AirText demo. If you have worked your way through the Sun SPOT tutorial, as we strongly suggest, you will be familiar with these demos.

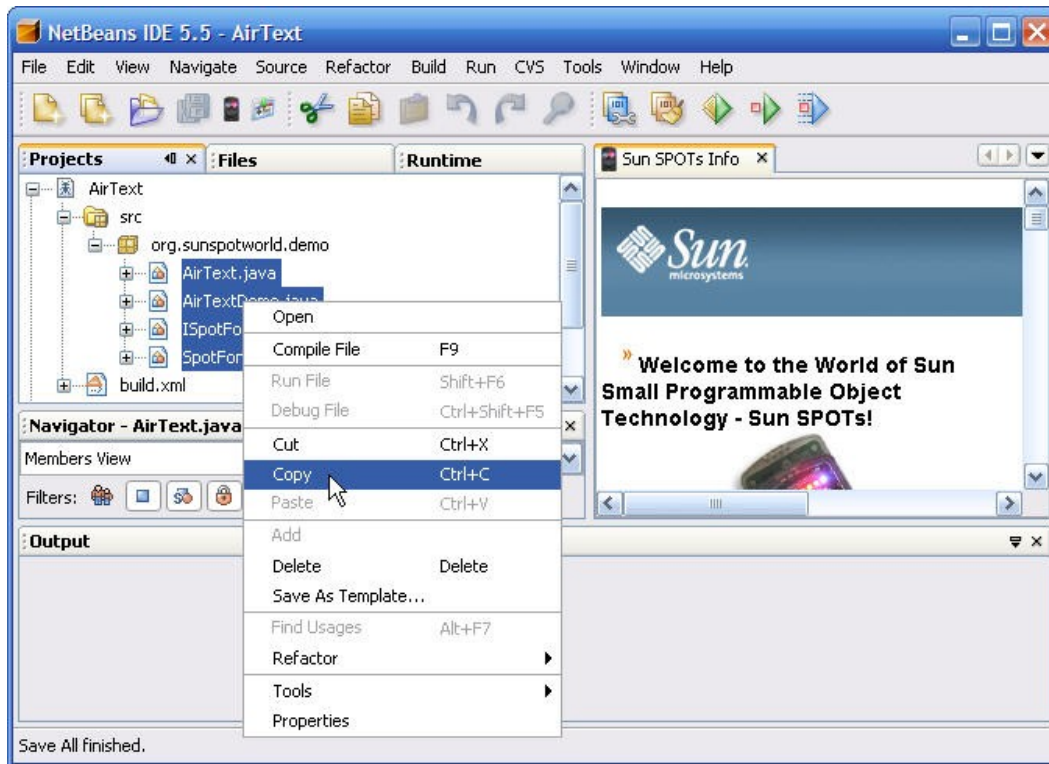
Briefly, we will copy the classes from the Bounce demo and the AirText demo into a new project, then edit the manifest file. After that, we will compile the new `jar` file and it will be available to deploy to a Sun SPOT.

1. Open NetBeans. Select the *Project* tab in the upper left, and select the “BounceDemo-OnSPOT” project. Right click the project and select *Copy Project*.



A dialog box displays, allowing you to specify the new project name. Here, we give it the name `ExampleJar`. If we were to compile this project now, we would get a `jar` file with one MIDlet in it, the Bounce demo. Now we need to add the AirText demo to the same project.

2. Select the AirText demo, click the “+” to its left, then open up the `src` selection with in it, then open the `org.sunspotworld.demo` selection within `src`. Select all four Java classes, right click, and select *Copy*.



3. Select the `ExampleJar` project, click the “+” to the left of it, then open up the `src` selection with in it, select the `org.sunspotworld.demo` node within `src`, right click `org.sunspotworld.demo`, and select *Paste*.

The AirText classes should now be copied into the `ExampleJar` project.

4. Select the Files tab in the upper left of NetBeans. Open the ExampleJar project, and the resources node within it. Open the resource META-INF and open the manifest.mf file with it.



The manifest file displays in a new window to the right:

```
MIDlet-Name: eSPOT Bounce Demo-OnSPOT
MIDlet-Version: 1.0.0
MIDlet-Vendor: Oracle
MIDlet-1: ,, org.sunspotworld.demo.SPOTBounce
MicroEdition-Profile: IMP-1.0
MicroEdition-Configuration: CLDC-1.1
```

The format of the manifest file is

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

The individual MIDlets are specified with property names of “MIDlet-1,” “MIDlet-2,” and so on. The MIDlet property value is a string of three comma-separated, arguments. The first argument is a name for the application, the second is intended to define an icon, but is not used in Sun SPOTS, and the third specifies the application’s main class.

We will edit the manifest file to give it a new MIDlet name and tell it where to find the AirText demo. We will also give the MIDlet descriptions some user-friendly names.

5. Replace the MIDlet name with “Example Jar with two MIDlets.” Edit the MIDlet-1 line to include the phrase “Bounce Demo” between the colon and the first comma. Add a second line, below it, that reads:

```
MIDlet-2: Air Text Demo, ,org.sunspotworld.demo.AirTextDemo
```

The manifest file should now read:

```
MIDlet-Name: Example Jar with two MIDlets
MIDlet-Version: 1.0.0
```

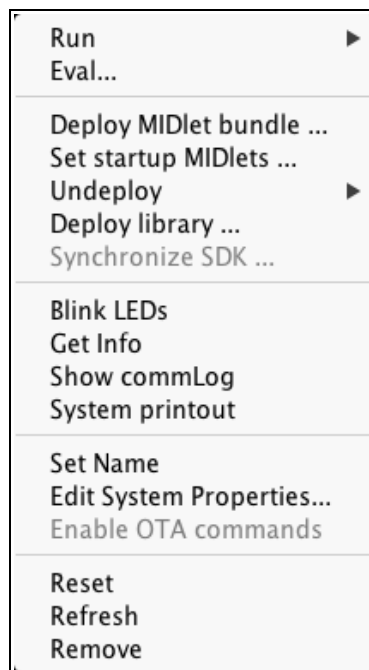
```
MIDlet-Vendor: Oracle
MIDlet-1: Bounce Demo,, org.sunspotworld.demo.SPOTBounce
MIDlet-2: Air Text Demo, ,org.sunspotworld.demo.AirTextDemo
MicroEdition-Profile: IMP-1.0
MicroEdition-Configuration: CLDC-1.1
```

## 6. Save the new version of the manifest file.

If you were to compile the project now, you would have a single jar with two MIDlets in it. We can use that jar file in Solarium.

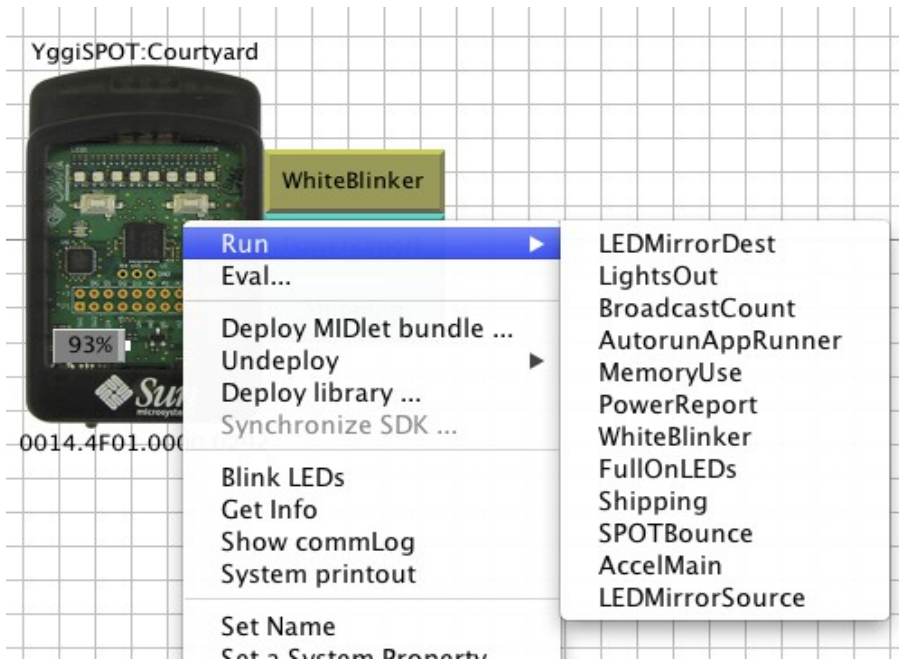
### ***Manipulating Sun SPOTs in Solarium***

If you right-click on a SPOT, the following menu will display. Some of the menu items are only enabled for USB-connected SPOTs while others require a SPOT to have its OTA command server enabled. Disabled menu items are displayed in gray.



#### • **Run**

This menu option brings up a sub-menu listing all of the MIDlets that are available in all of the deployed jar files. Selecting one causes that MIDlet to start and it adds an application object to the Solarium display for that SPOT. The application object can be moved around on the graphic display. Clicking the application object gives you a menu (described later) for that application.

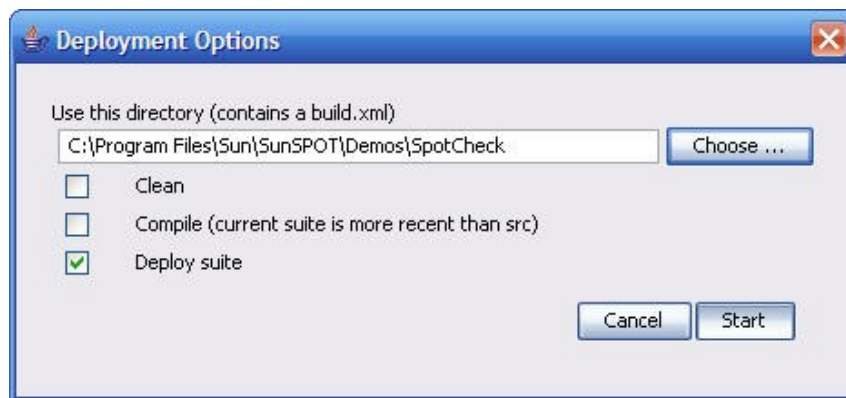


- **Eval...**

This menu option allows you to enter one or more Java statements to be evaluated by the SPOT. The submitted code is compiled, bundled into a suite, deployed to the SPOT and then run.

- **Deploy MIDlet bundle...**

Allows you to specify a project to be deployed to the selected Sun SPOT. The following dialog box displays:



By default, the project directory will be cleaned, all source code will be recompiled and the resulting suite will be installed on the SPOT. The clean and compile steps may be skipped by unchecking the appropriate boxes. This might make sense, for example, if the application suite was created recently and the same suite is being deployed on multiple SPOTs.

*Note:* Solarium can be used to deploy to SPOTs running old SDKs provided that the old SDK is still installed and is OTA-compatible with the current SDK.

- **Set startup MIDlets ...**

Allows you to select which of the installed MIDlets should be run when the SPOT is started up.

- **Undeploy**

This menu option brings up a sub-menu listing all installed suites and selecting any suite on this list will unload it from the SPOT. The command will fail if the suite is in use, e.g., one of the applications in the suite is currently active. If a suite includes multiple applications, all of them will be unloaded together.

- **Deploy library...**

Downloads to the SPOT a fresh copy of the current library (as specified by the `spot.library.name` property in the `.sunspot.properties` file in the user's home directory) on to the SPOT in the active SDK. After the library is installed, the SPOT will be reset and all previously running applications terminated.

This command is useful, for example, when a user has created a new library using the `ant library` command.

- **Synchronize SDK...**

Downloads system software from the Sun SPOT SDK currently active on the host (where Solarium is running) to the selected SPOT. This option is only accessible if the SPOT is connected to the host workstation by a USB cable.

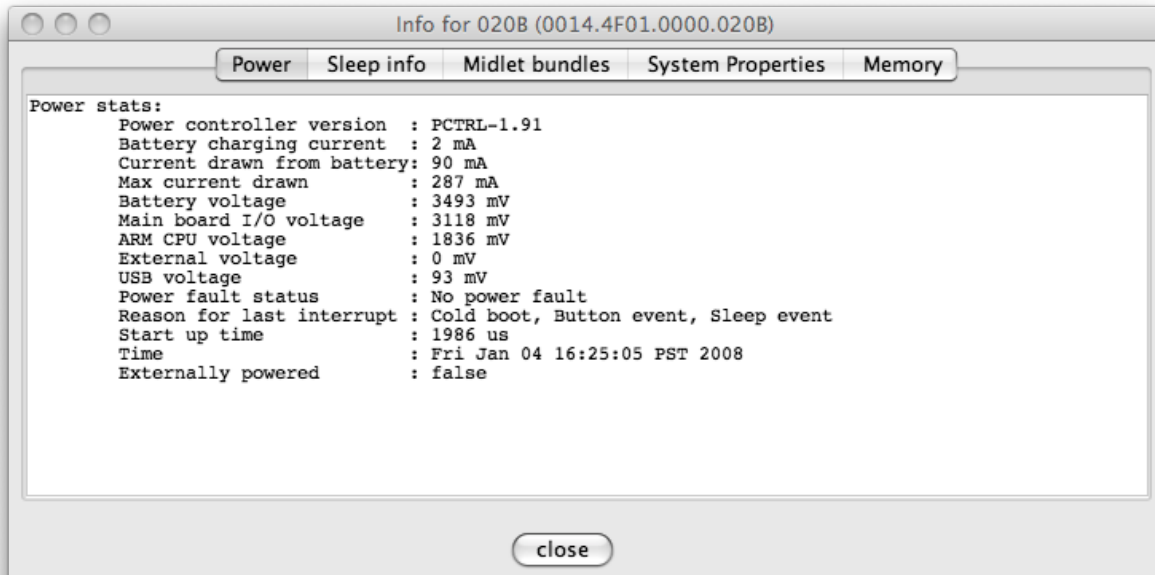
- **Blink LEDs**

Blinks all of the LEDs on the SPOT 10 times, except the power LED. This is useful for finding a particular SPOT among several.

On SPOTs equipped with the eDemo sensor board, all of its eight tricolor LEDs as well as the Activity LED on the main board are flashed. If a SPOT does not have the eDemo sensor board installed, only the activity LED is flashed.

- **Get Info**

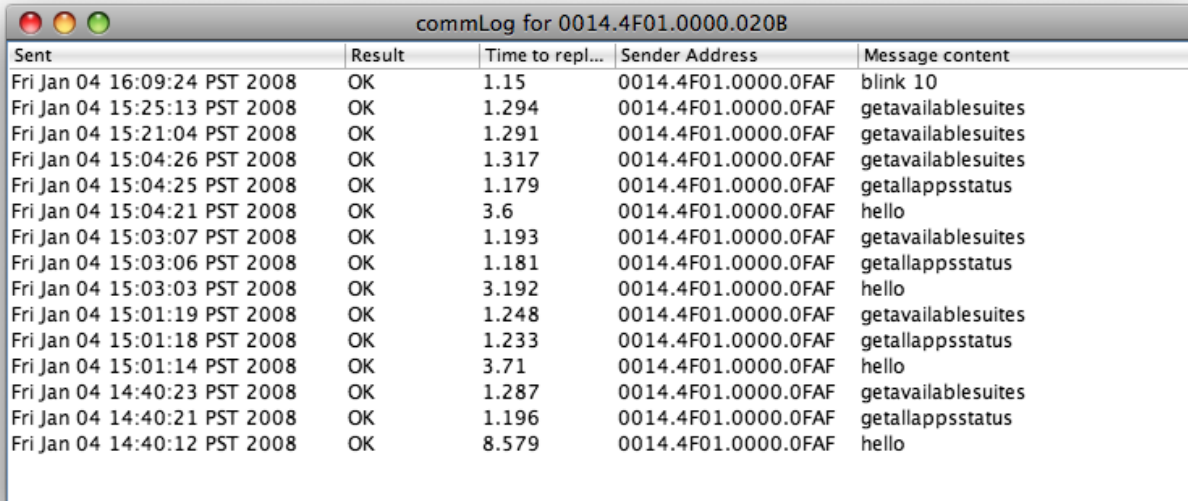
Opens a window containing multiple tabs. Each tab has information on a different aspect of the selected Sun SPOT, e.g., installed applications, system properties, and energy/memory statistics.





- **Show commLog**

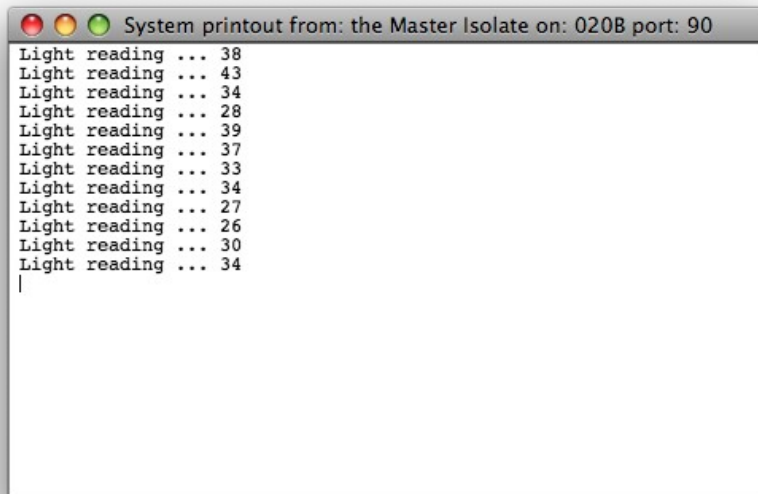
Opens a window showing a log of all the OTA commands issued to the selected SPOT.



Sent	Result	Time to repl...	Sender Address	Message content
Fri Jan 04 16:09:24 PST 2008	OK	1.15	0014.4F01.0000.0F0F	blink 10
Fri Jan 04 15:25:13 PST 2008	OK	1.294	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 15:21:04 PST 2008	OK	1.291	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 15:04:26 PST 2008	OK	1.317	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 15:04:25 PST 2008	OK	1.179	0014.4F01.0000.0F0F	getallappsstatus
Fri Jan 04 15:04:21 PST 2008	OK	3.6	0014.4F01.0000.0F0F	hello
Fri Jan 04 15:03:07 PST 2008	OK	1.193	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 15:03:06 PST 2008	OK	1.181	0014.4F01.0000.0F0F	getallappsstatus
Fri Jan 04 15:03:03 PST 2008	OK	3.192	0014.4F01.0000.0F0F	hello
Fri Jan 04 15:01:19 PST 2008	OK	1.248	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 15:01:18 PST 2008	OK	1.233	0014.4F01.0000.0F0F	getallappsstatus
Fri Jan 04 15:01:14 PST 2008	OK	3.71	0014.4F01.0000.0F0F	hello
Fri Jan 04 14:40:23 PST 2008	OK	1.287	0014.4F01.0000.0F0F	getavailablesuites
Fri Jan 04 14:40:21 PST 2008	OK	1.196	0014.4F01.0000.0F0F	getallappsstatus
Fri Jan 04 14:40:12 PST 2008	OK	8.579	0014.4F01.0000.0F0F	hello

- **System printout**

Opens a window to display standard output from the main isolate operating on that SPOT. The image below shows standard output from the LightSensor application.



```
System printout from: the Master Isolate on: 020B port: 90
Light reading ... 38
Light reading ... 43
Light reading ... 34
Light reading ... 28
Light reading ... 39
Light reading ... 37
Light reading ... 33
Light reading ... 34
Light reading ... 27
Light reading ... 26
Light reading ... 30
Light reading ... 34
|
```

- **Set Name**

The first time a SPOT displays in Solarium, the second half of its full IEEE numerical address is used as its name and displayed at the top of the SPOT image. This command allows you to specify a more meaningful name as a replacement. The name will be stored in persistent memory on the SPOT itself.

- **Edit System Properties...**

Lets the user modify the system properties stored in persistent memory on the SPOT.

- **Enable OTA Commands**

Allows the user to turn on the SPOT's OTA command server. This option is only enabled for USB-connected SPOTs that have their OTA command server turned off.

- **Reset**

Resets the SPOT. It has the same effect as pressing the Control button. It will restart all MIDlets that are specified to be run when the SPOT is started.

- **Refresh**

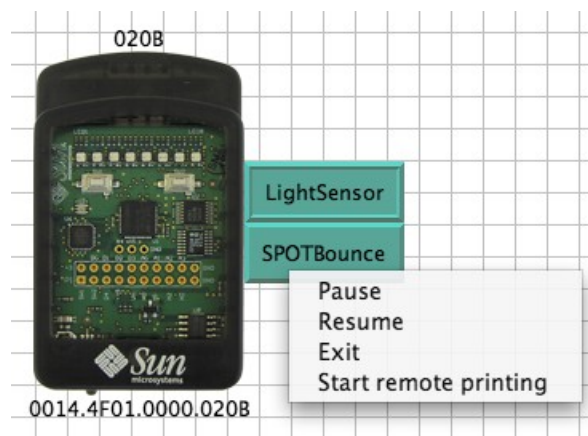
Queries the SPOT for its current list of installed and running applications and refreshes the display.

- **Remove**

Removes this SPOT from the Solarium display. Removed SPOTs can be added again through the SPOT discovery process.

### ***Pausing, Resuming and Terminating Applications***

Once an application has been launched in a child isolate, the user can pause, resume or terminate it. Clicking on the application object, brings up the menu shown below:



- **Pause**

Pauses execution of the selected application.

- **Resume**

Resumes execution of a paused application.

- **Exit**

Stops execution of the selected application.

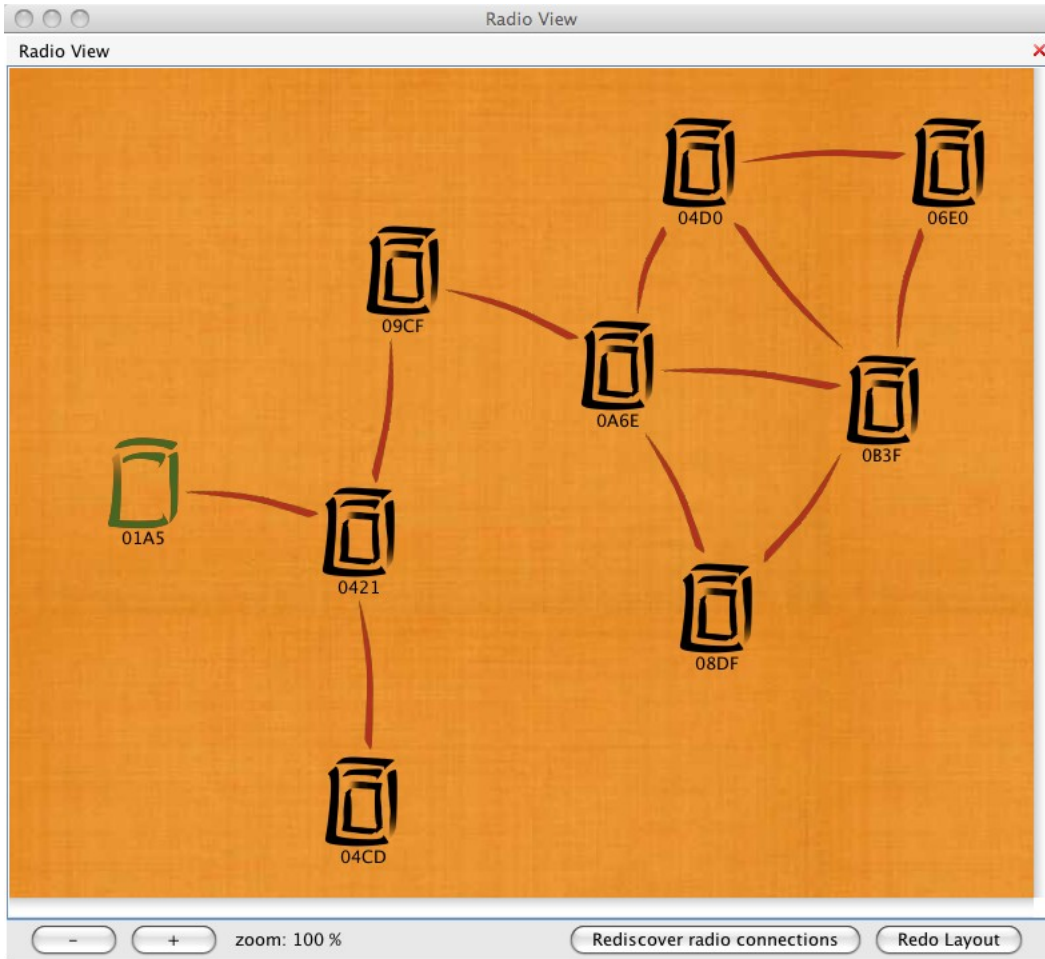
- **Start remote printing**

Opens a dialog box that directs System.out for the selected application to a window. The window can be a new window or a pane within the Solarium window.

*Note:* The current framework only supports pausing and resuming of simple applications, e.g., those that do not have currently active network connections.

## Radio View

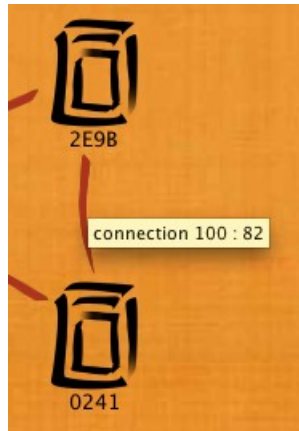
When debugging an application running on a network of SPOTs it can be quite helpful to be able to visualize the radio connectivity, i.e. which SPOTs can talk to each other. The *Radio View* provides a way for Solarium to discover and graphically depict the radio topology. Below is an example of a network of eight SPOTs and one basestation (shown in green).



When the Radio View window is first opened, Solarium sends commands to each known SPOT asking it to discover its neighbors. The red lines show which SPOTs that can exchange radio packets. After all the SPOTs have replied a graphing algorithm is used to create the best layout that shows the radio topology. The Radio View has no knowledge of where the SPOTs are located physically, so the layout algorithm can only try to minimize the number of connections that cross each other.

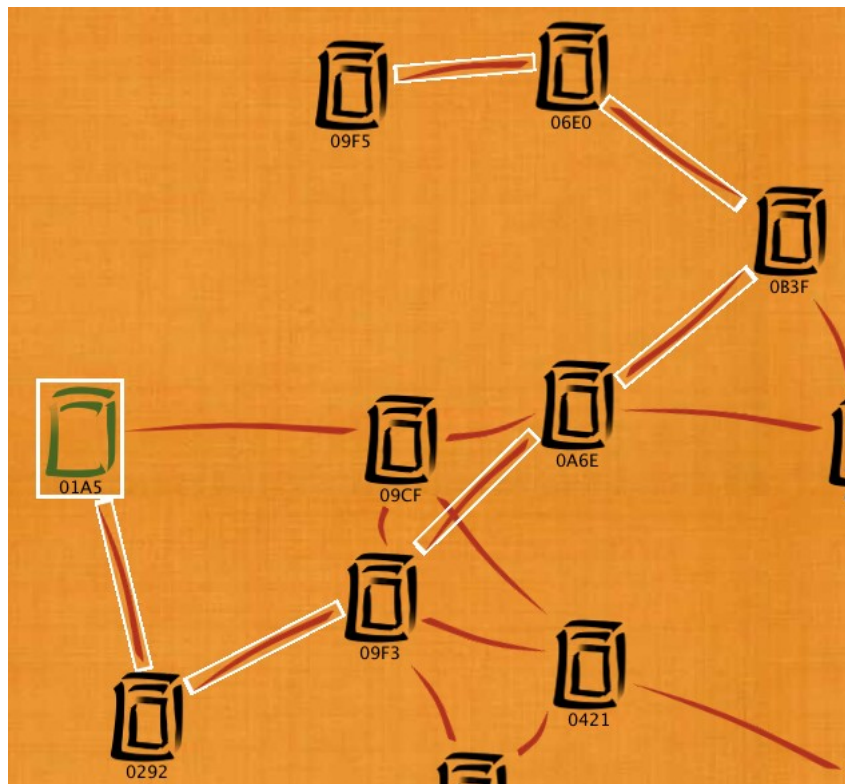
Since the radio topology can change over time, pushing the **Rediscover radio connections** button will cause Solarium to again ask each SPOT to discover its neighbors and then repeat the layout process. If the resulting layout is not clear enough, push the **Redo layout** button to try to find a better one.

If you hover the mouse over a radio connection a tooltip will appear showing the “Link Quality” in each direction as measured by the two SPOTs.



Link Quality is uniquely determined by the CORR value: CORR is a correlation measure provided directly by the radio part. Link Quality values range from 100% (the best, corresponding to a CORR value greater than 100) to 0% (the worst, which means the CORR value is less than 50).

The Radio View can also be used to show the route two SPOTs would use to talk to each other. Click on one of the SPOTs to bring up a pop-up menu where you can select the address of the SPOT to seek a route to. Below is the route from the basestation to 09F5.



*Note:* SPOTs must have OTA enabled to participate in Radio View.

## Managing a Network of SPOTs

Solarium provides a special *Deployment View* to make it easier to manage a network of Sun SPOTs. A deployment specifies what application should be loaded on each SPOT and allows you to deploy those applications with a single button press. The Deployment View also shows the current status of each SPOT. A deployment can also specify any associated host applications that need to be run.

Here is a simple deployment that describes the `SendDataDemo`.

Deployment: SendDataDemo

Name:

Map:

Description:

This demo application sends periodic sensor readings measured on one or more Sun SPOTs to an application on your laptop or PC that displays the values.

deploy only if needed

SPOTs:

Role:

Project:

Description:

This application runs on a Sun SPOT periodically sampling the built-in light sensor and broadcasting those readings over the radio.

1 IEEE Address:

Location:

Status: not connected

Host Applications:

2 Name:

Role:

Project:

Command line:

Description:

This application runs on a host computer, listens for sensor sample broadcasts and prints them.

Status: not active

The deployment information is in three main sections. First is some information about the deployment as a whole. This includes the deployment name, an optional image file to be used as a map of the

deployment (not used above), and a text description of the deployment. Note the small box with a minus sign in it to the left of the descriptive text. Clicking it with the mouse will cause the text area to expand. Below the description is the **Deploy & Run All** button that can be used to load the latest version of your SPOT applications on all the SPOTs in the deployment and start them all up. Solarium tries to determine if the code currently installed on each SPOT is up to date or not. If the **deploy only if needed** checkbox is selected then any SPOTs that have the latest version of the application on them will not be updated.

The next section specifies what SPOTs are part of this deployment, what application they should run, and their current status. Since it is often the case that many SPOTs will be running the same application, this section consists of a series of *Roles*, one for each application used, where each role includes a list of SPOTs performing that role. Each role has a short one-line name, the filename for the `build.xml` file used to build the application, a longer description, and then a list of SPOTs to be loaded with this application. Each SPOT entry consists of the SPOT's IEEE radio address, an optional short description of where it is located, and its current status. At the end of the list of SPOTs is the **Add new SPOT** button used to add another SPOT to this role. Note that the SPOT's address has a pull-down menu that lists all SPOTs currently discovered by Solarium. This makes it easy to assign a SPOT to a role. The list of SPOTs also includes a **New Virtual SPOT** menu item if you wish to select an emulated SPOT. In the above sample deployment there is one role defined, with one SPOT associated with it.

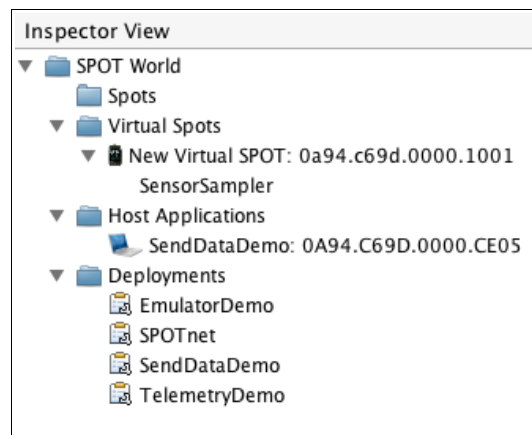
The final section specifies any host applications used by the deployment. The sample above has one host app that is used to receive the data broadcast by the SPOT and display it. Each host application has a name that the application will use to register itself when it starts up. When Solarium goes to discover SPOTs it now also will discover any host applications that have enabled OTA by calling:

```
OTACommandServer.start("SendDataDemo");
```

where "SendDataDemo" is the same as the host application name specified in the deployment. The remainder of the host application fields are similar to those for a SPOT Role with the addition of the command line to be used to start up the application. Normally this will just be `ant host-run`.

Note: Solarium must be started with the property `basestation.shared=true` in order to discover host applications.

Here is what the *Inspector View* looks like when using a virtual SPOT:



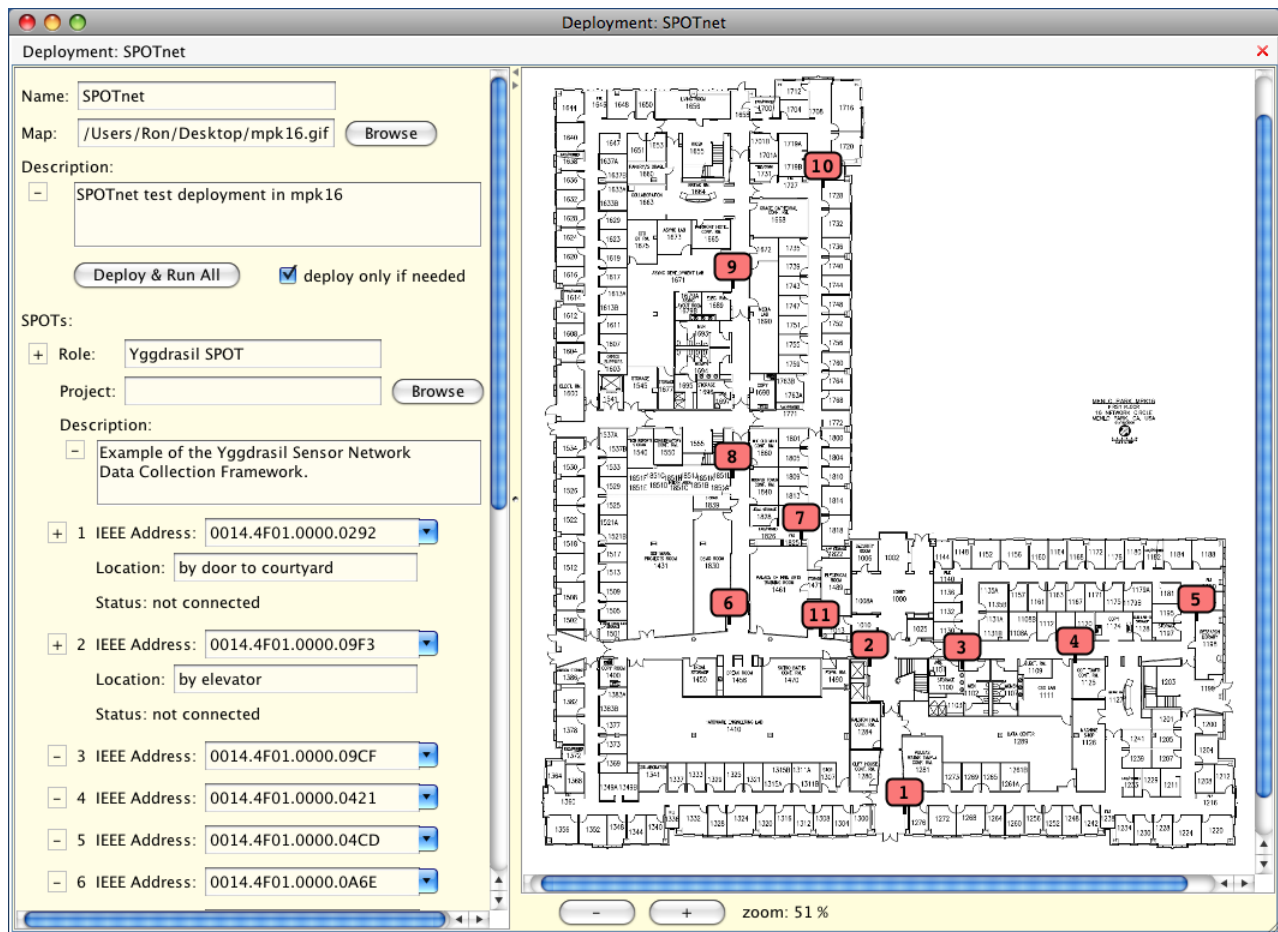
Note that the *Inspector View* lists all of the deployments that Solarium knows about. You can open one of them by right clicking on it with the mouse and selecting the **View** menu option. When you create a new deployment (**View** → **Deployment** → **New...**), a new file is created to hold the deployment information. The name you give the deployment is added to the list of deployments. The

SendDataDemo included as part of the Sun SPOT SDK now also specifies a deployment.xml file that you can open (**View** → **Deployment** → **Open...**) that is the same as the sample deployment shown above.

Here is an example of a deployment of eleven SPOTs used at Sun Labs.

This deployment specifies a map file to use that is displayed in the right half of the window. Each SPOT is shown on the map with a small balloon containing its number. You can drag the balloon to wherever the SPOT is located. While not visible in the above screenshot, this deployment has a second role defined for SPOT number 11, which is running a different application than the other ten SPOTs.

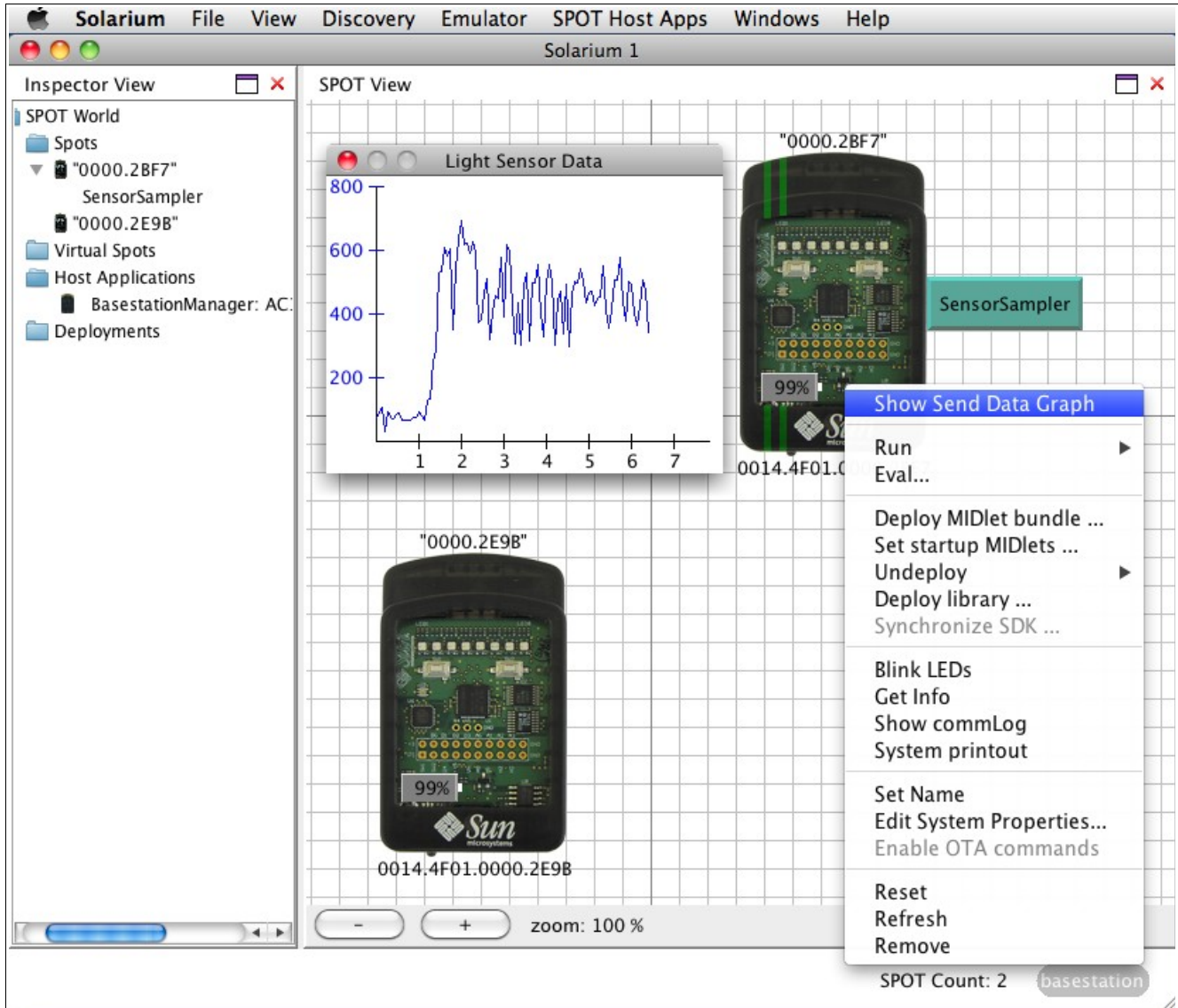
*Note:* right clicking on a SPOT or its balloon will bring up a pop-up menu allowing you to deploy the application to the SPOT, run it, remove this SPOT from the deployment, and all of the normal Solarium commands for a SPOT.



## Extending Solarium

Solarium is designed so that it is easy to extend by customizing how objects are displayed or even creating new views. New plugins can be added by just dropping them into the appropriate folder in the <SDK>/lib/SPOTWorld directory: virtualObjects or views. Two of the demos included with the SDK illustrate how to do so. The first, SolariumExtensionDemo/VirtualObjectPlugin, shows how to display a SPOT running a given application with a special image, how to add new commands to its pop-up menu, and how to display data collected by the SPOT in Solarium.

This demo builds on the SendDataDemo that sends periodic sensor readings measured on one or more Sun SPOTs to an your laptop or PC to display the values. Instead of running a special host application to gather the data, for this demo we modify Solarium to recognize a SPOT running the SendDataDemo and to be able to display the data in a Solarium window. This is done by defining a new Solarium "virtual object" to represent a SPOT running the SendDataDemo. This code is then plugged into Solarium.



The application running on the SPOT differs from the normal SendDataDemo code only in that it now notifies its OTA command server that this SPOT has a special subtype. When Solarium discovers the SPOT it checks all of the defined virtual objects to see if one matches this subtype and if so uses that virtual object's jar file to control how the SPOT is displayed—in this case with two green racing stripes—and what commands it should respond to—in this case an additional menu item to display a graph of the received data values.

Only a few files are required for this extension: `SendDataSpot.java` defines the new virtual object. It extends the regular SPOT virtual object. `PVSendDataSpot.java` and `TVSendDataSpot.java` define how to display a SendData Spot in the two dimensional SPOT View and the tree-based Inspector View. `PVSendDataSpot.java` also specifies the "Show Send Data Graph" command. The new graph is



displayed by `SendDataPanel.java` that takes care of opening a radio connection to the real SPOT, listening for light sensor values, and then displaying them in a graph. The `images` folder contains the graphic to use when displaying a SendData SPOT. The `build.xml` file has a target to create the jar file to plug into Solarium. The jar file's manifest includes attributes that tell Solarium what type/subtype of object this file applies to.

A second demo, `NewViewPlugin`, shows how to add a new view to Solarium and also how to extend the Sun SPOT Emulator so that it interacts with the new view. It demonstrates how to do everything that the Robot View in Solarium does.

## Using the SPOT Emulator in Solarium

Solarium includes an emulator capable of running a Sun SPOT application on your desktop computer. This allows for testing a program before deploying it to a real SPOT, or if a real SPOT is not available.

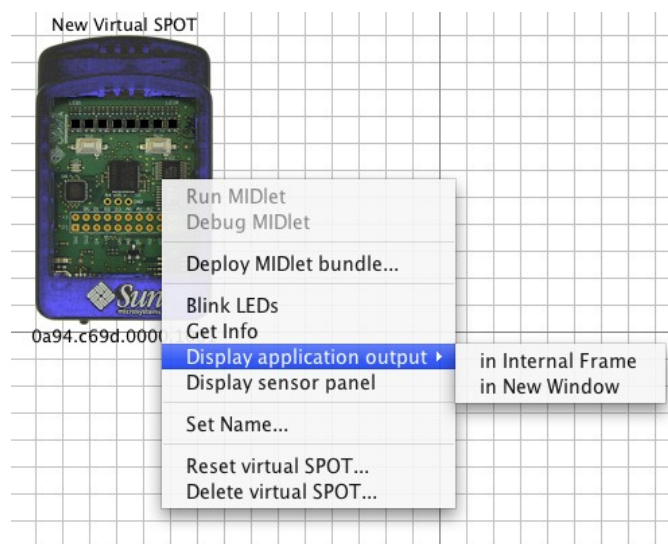
Instead of a physical sensorboard, virtual SPOTs have a sensor panel that can be used to set any of the potential sensor inputs (e.g. light level, temperature, digital pin inputs, analog input voltages, and accelerometer values). Your application can control the color of the LEDs that are displayed in the virtual SPOT image, just like it would a real SPOT. You can click with the mouse on the push button switches in the virtual SPOT image to press and release the switches.

Receiving and sending via the radio is also supported. Each virtual SPOT is assigned its own address and can broadcast or unicast to the other virtual SPOTs. If a shared basestation is available a virtual SPOT can also interact over the radio with real SPOTs.

### Manipulating virtual SPOTs

Once Solarium is running make sure that a graphic *SPOT View* is displayed (**View > SPOT View**). Then from the **File** pull-down menu in the main Solarium menu bar, select the command **New virtual SPOT**. This will create and display a new virtual SPOT. It will appear to have a blue rather than a smoke-colored plastic case. You can use the mouse to place the virtual SPOT any place in the display.

If you right-click on the virtual SPOT you will see a menu of possible commands.



- **Set Name...**

This allows you to give the virtual SPOT a descriptive name to help you identify it. Each virtual SPOT has a label above it with its name and another label below it with its IEEE radio address.

- **Deploy MIDlet bundle...**

This command lets you *deploy* a SPOT application to the virtual SPOT. It will bring up a file chooser dialog that you can use to navigate to a SPOT project directory. You can select an existing jar file created with the `ant jar-app` command or the project's `build.xml` file, in which case a process will be spawned to compile the source code, build the jar file, and then load it.

- **Run MIDlet**

Once you have loaded some MIDlets you can use this command to display a submenu listing all of the MIDlets contained in the deployed jar file and allow you to start up whatever one you want. Any running MIDlets will be displayed in a box to the right of the virtual SPOT. Clicking on a running MIDlet will display a popup menu that lets you tell the MIDlet to exit.

For example, use the **Deploy MIDlet bundle...** command to load in the `emulator_demo.jar` file located in the `Demos/EmulatorDemo` folder. Once it is loaded run the *Sawtooth* MIDlet. As it runs you will see the LEDs of the virtual SPOT be turned on, one by one, each brighter than the previous, until all are lit at which point they are all turned off and the cycle repeats. Right click on the Sawtooth application box and exit it.

- **Debug MIDlet**

This will list all of the MIDlets contained in the deployed jar file and allow you to connect an external Java Debugger to a MIDlet in order to debug it.

- **Reset virtual SPOT...**

This command will cause any running MIDlets to be killed and the Squawk VM to be restarted. If a jar file had been specified earlier, then it is automatically reloaded and you can run any of the MIDlets defined in it.

- **Display application output**

This command will display a new window where anything printed by the SPOT application to `System.out` or `System.err` will be displayed. This new window can be an Internal Frame that is displayed beneath the virtual SPOT in the main Solarium window, or it can be in a New Window. If you reset the virtual SPOT you will see messages printed when the old Squawk VM exits and the new one is started up. If the output window is covered up this command will bring it to the front.

- **Get info**

This command will bring up a new window giving some information about the virtual SPOT: its IEEE address, the jar file loaded (if any), and the names of all available MIDlets.

- **Delete virtual SPOT**

When you are done with the virtual SPOT it can be deleted using this command.

From the **Emulator** pull-down menu in the main Solarium menu bar, one can use the **Save virtual configuration...** command to write out a file that will store the state of all of the virtual SPOTs: each virtual SPOT's name and radio address, what jar file it is using, what MIDlets are running, and where the virtual SPOT is located on the grid. You can specify whether you want the current radio address kept for use when the configuration is read back in or whether you want a new address to be used. You

can also specify whether or not to automatically restart any currently running MIDlets when the configuration is read in. Along with the configuration you can include a textual description that will be displayed when the configuration is reloaded.

The **Open virtual SPOT...** pull-down menu allows you to select a previously saved configuration file and reload it. When it is reloaded any descriptive text associated with it will be displayed in a new window. You can cause this window to be redisplayed at any time using the **Display virtual configuration description** command.

Finally the **Emulator** menu has the **Delete all virtual SPOTs...** command to remove any virtual SPOTs currently defined in Solarium.

For an example of loading a predefined configuration do **Delete all virtual SPOTs...** followed by **Open virtual configuration...** and select the file *emulator\_demo.xml* from the *EmulatorDemo* in the *Demos* folder. That will create 4 virtual Spots in Solarium, and start 3 of them running various demo apps. It will also display a window with a textual description of the available MIDlets.

Note: if you start Solarium from the command line you can specify a previously saved configuration file and have it automatically loaded when Solarium starts up. To do so just set the ant property `config.file` either on the command line (e.g. `"-Dconfig.file=<path to config file>"`) or in your project's `build.properties` file. For example:

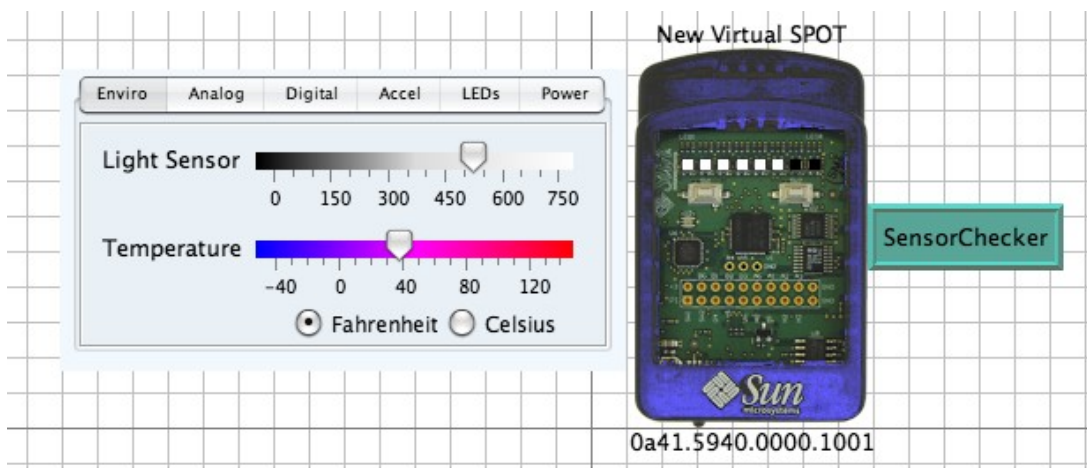
```
cd SunSPOT/sdk/Demos/EmulatorDemo
ant solarium -Dconfig.file=emulator_demo.xml
```

### Using the Sensor Panel

Virtual SPOTs do not have a physical sensorboard so an alternate mechanism is needed to feed sensor input. Choosing the **Display sensor panel** menu option brings up a window with six tabs. Each tab corresponds to a set of input pins or sensors on the virtual SPOT and allows you to determine what values will be fed to the virtual SPOT. The tabs are:

- *Enviro* - Light sensor and temperature sensor
- *Analog In* - The input levels on analog pins A0 to A5
- *Digital Pins* - The input on digital pins D0 through D4. Also displays the output to pins H0 through H3.
- *Accel* - The input from the accelerometer
- *LEDs* - The input from the two switches on the sensor board. Also displays the output from the LEDs.
- *Power* - The input voltage from the battery and charging rate.

To demonstrate how to change the various sensor values, run the *SensorChecker* demo from the already loaded *emulator\_demo.jar* file. This application uses the LEDs on the virtual SPOT to display a value



read from one of the SPOT's sensors. When it is started the light sensor reading is displayed in white. To change the light sensor value use the **Display sensor panel** command to bring up the sensor panel. On the leftmost *Enviro* tab are two sliders for controlling the value the SPOT will read for the light sensor and for the internal thermometer. As you move the light sensor slider left and right you will see the number of LEDs change appropriately.

Temperature is specified in degrees Fahrenheit or Celsius, light readings in the raw value returned from the A/D, analog inputs in volts, and acceleration in gravities (G's).

The *SensorChecker* demo has four different modes:

1. Display the light sensor reading in white
2. Display the temperature sensor reading in red.
3. Display the analog input A0 in green.
4. Display the Z acceleration in blue.

Push the left switch (SW1) by clicking on it with the mouse to advance to the next mode. Switch to the different tabs of the sensor panel to access the different sensors. As you move the slider for the current sensor you will see the LED display change.

If you go to the *Digital Pins* tab you will see the current mode shown by the application setting one of the high current output pins, H0-H3, to high. As you cycle through the different modes the SPOT application will change which pin is set to high. The digital input/output pins, D0-D4, are enabled when they are being used as an input so you can set their value. When they are being used as an output they are disabled and the SPOT application can set their value to low or high. For the *SensorChecker* demo D0 is an output, while D1-D4 are set as inputs. The application reads the value of D1 and then sets D0 to be the same. Try changing the value of D1 and watch as D0 is also changed.

*Note:* the popup menu for a virtual SPOT can also be used from the *Inspector View*. From the *Inspector View* the **Display sensor panel** command will create a new window to display the sensor panel. To locate a virtual SPOT in the *SPOT View* one can cause its LEDs to blink using the **Blink LEDs** command from its popup menu in the *Inspector View*.

### **Using the Radio**

Virtual SPOTs can communicate with each other by opening radio connections, both broadcast and point-to-point. Instead of using an actual radio these connections take place over regular and multicast sockets.

When a basestation SPOT is connected to the host computer and a shared basestation is running, virtual SPOTs can also use it to communicate with real SPOTs using the basestation's radio. The advantage of using a shared basestation is that multiple host applications can then all access the radio. One disadvantage is that communication from a host application to a target SPOT takes two radio hops, in contrast to the one hop needed with a dedicated basestation. Another disadvantage is that run-time manipulation of the basestation SPOT's radio channel, pan id or output power is not currently possible.

To always use a shared basestation add the following line to your *.sunspot.properties* file:

```
basestation.shared=true
```

Please note that some Linux distributions (e.g. SuSE) may not have multicasting enabled by default, which will prevent shared basestation operation and also any "radio" use by virtual SPOTs.

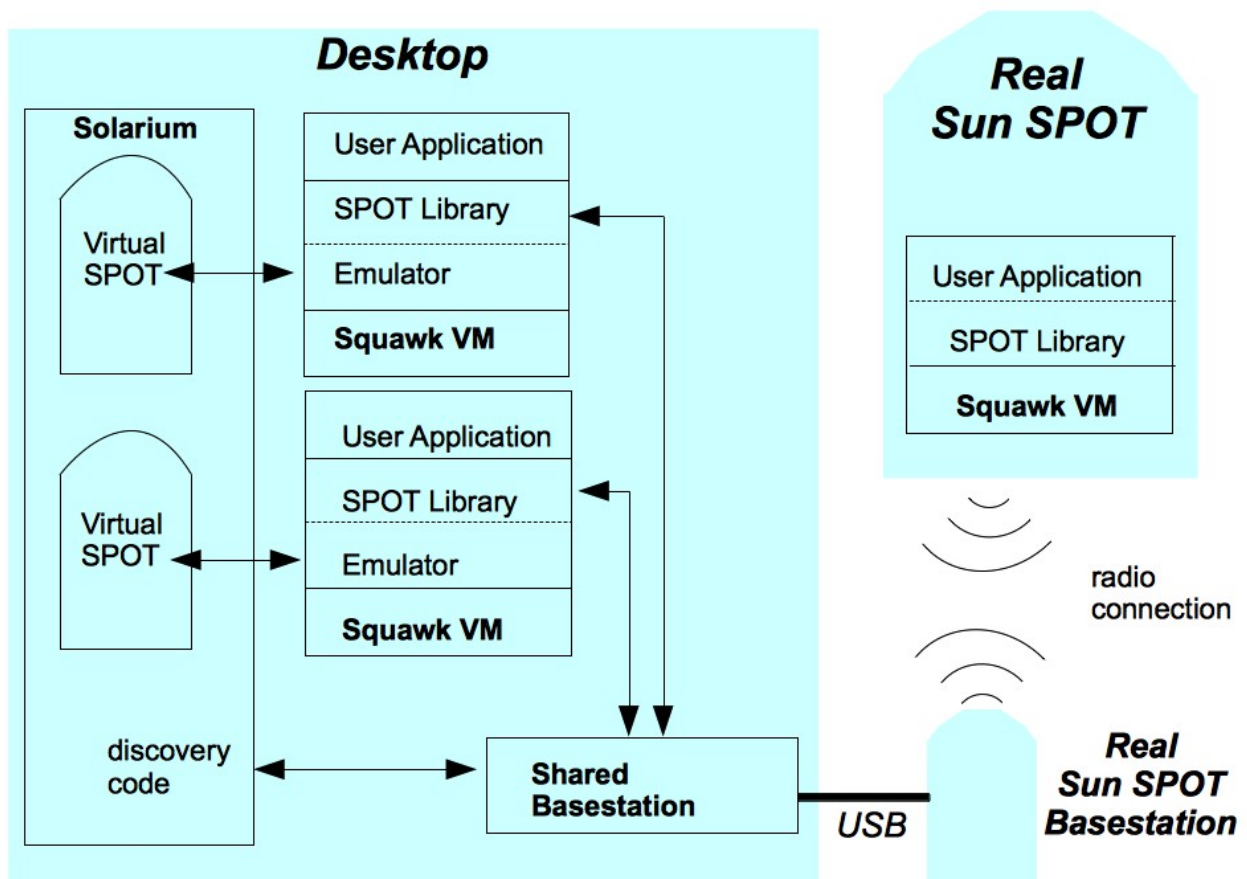
*Note:* virtual SPOTs can also communicate with SPOT host applications using the radio, but only if the host application is run with `basestation.shared` set to `true`.

The *EmulatorDemo* provides several sample MIDlets that use the radio. Start with the *BroadcastCount* demo which uses the left switch (SW1) to broadcast a message to set the color displayed in the LEDs of all receiving SPOTs and the right switch (SW2) to count in binary on the receiving SPOTs' LEDs. If a shared basestation is available then try deploying the *EmulatorDemo* to a real, physical SPOT and having it then interact with the virtual SPOTs via the radio.

### How the Emulator Works

When you create a new virtual SPOT in Solarium, a new process is started to run the emulator code in a Squawk VM. The emulator code communicates over a socket connection with the virtual SPOT GUI code in Solarium. For example when the SPOT application changes the RGB value of an LED that information is passed to the virtual SPOT GUI code that updates the display for that LED with the new RGB value. Likewise when the user clicks one of the virtual SPOT's switches using the mouse, Solarium sends a message to the emulator code that the switch has been clicked, which can then be noticed by the SPOT application.

Here's a block diagram of the Emulator architecture:



Each virtual SPOT has its own Squawk VM running in a separate process on the host computer. Each Squawk VM contains a complete host-side radio stack as part of the SPOT library, which allows the SPOT application to communicate with other SPOT applications running on the host computer, such as other virtual SPOTs, using sockets or real SPOTs via radio if a shared basestation is running.

## **Emulation vs Simulation**

The distinction between emulation and simulation is not always clear, so it is worth explaining how the two terms are used here. When a computer application is “run” in an emulator, the emulator is mimicking the behavior of a different computer system, which allows the user program to run as if it were on that other system. While the important functionality is preserved, other aspects, such as the time to do a given operation, may be quite different. Hence a program may run much slower when it is emulated.

A simulation, in comparison, is built by creating a model of a system and identifying various properties that will be accurately modeled as to how their values change. There is usually some sort of abstraction involved, for example a weather simulation does not model individual molecules but rather breaks the world up into a grid of cells of various sizes (ranging from meters to kilometers) and then characterizes several parameters for each cell.

The current Solarium implementation is primarily an emulator since it actually runs a SPOT application in a Squawk VM, just like the VM on a real SPOT. Likewise radio interaction between virtual SPOTs is emulated with data sent via packets and streams from one (virtual) SPOT to another. Only the SPOT's interaction with the environment is simulated using a simple model where the user needs to explicitly set the current sensor values. Future versions may incorporate more simulation of SPOT properties like battery level or radio range.

## **What's Missing from the Emulator?**

The initial version of the Emulator allows a SPOT application to control the LEDs and digital output pins, read various sensor inputs—switches, light level, temperature, digital input pins, analog input voltages, and accelerometer values—and send and receive radio messages. However there are other aspects of the Sun SPOT that are not currently implemented.

Like any SPOT host application using a shared basestation, a virtual SPOT cannot control the radio channel, pan id or power level. Nor is there the ability to turn the radio off and on.

Not available is various sensorboard functionality such as the UART, tone generation, servo control—including pulse width modulation (PWM), pulse generation, timing a pulse's width, and doing logical operations on the Atmega registers. These unimplemented features currently act as no-ops rather than throwing any exceptions.

There is currently no emulation of the low-level processor hardware functionality provided by the following classes and interfaces in the SPOT library: *ISpiMaster*, *IAT91\_PIO*, *IAT91\_AIC*, *IAT91\_TC*, *IProprietaryRadio*, *I802\_15\_4\_PHY*, *SpotPins*, *FiqInterruptDaemon*, *ISecuredSiliconArea*, *ConfigPage*, *ExternalBoardMap*, *ExternalBoardProperties*, *IFlashMemoryDevice*, *ISleepManager*, *ILTC3455*, *IUSBPowerDaemon*, *IAT91\_PowerManager*, *IDMAMemoryManager*, and *OTACommandServer*. Nor is there support for saving persistent properties, getting the SPOT's public key or reading the current system tick. Attempts by an emulated SPOT application to use these unimplemented features will cause a *SpotFatalException* to be thrown.

## **Future Directions for the Emulator**

While there is no schedule for when additional features will be added to the Emulator here are some likely areas for improvement in the not too distant future.

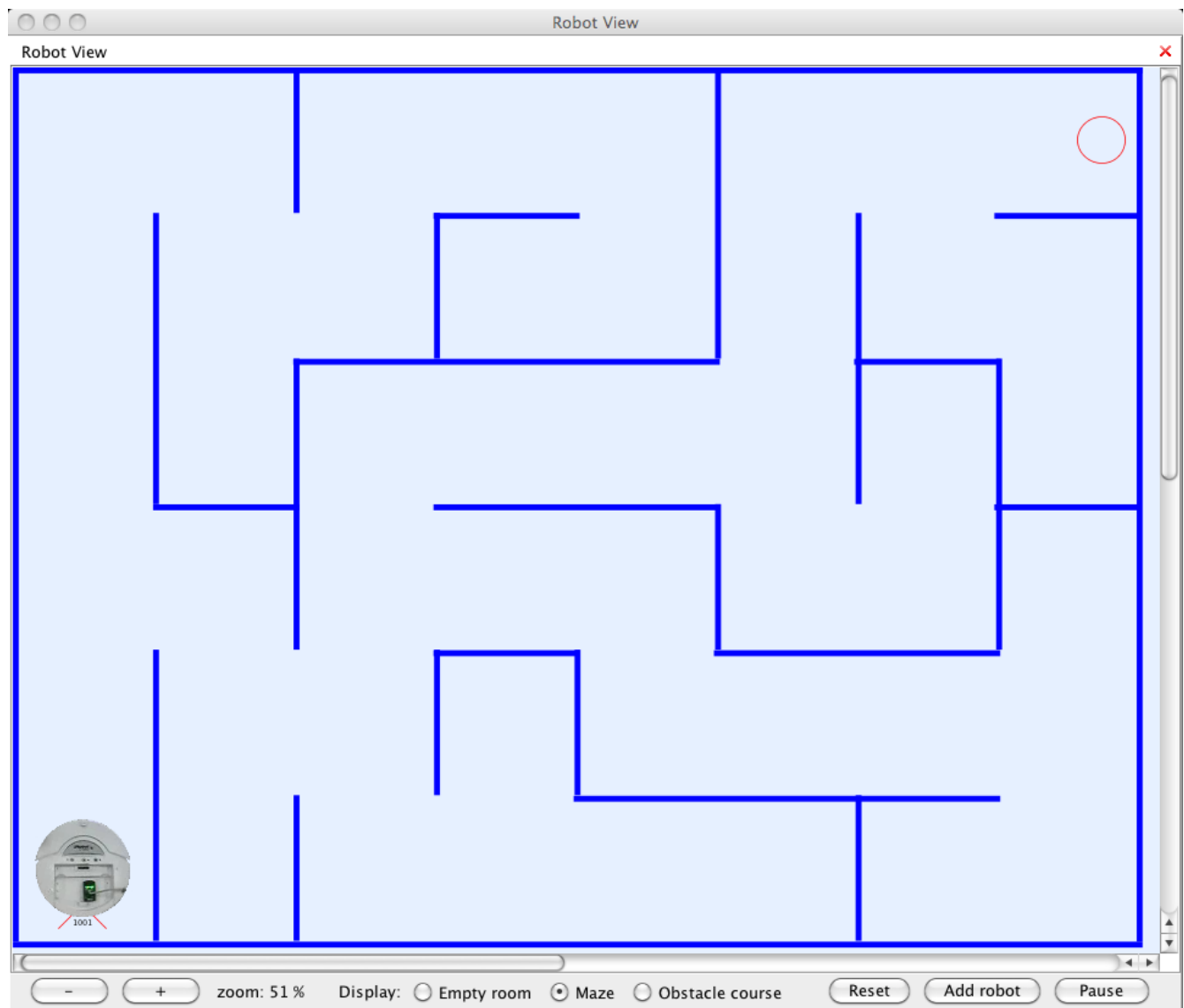
Currently no statistics or metrics are kept. Instrumenting the virtual SPOT's software stack should make it possible to report on an applications activities such as radio usage, idle time, memory usage (including number of GCs needed), etc.

The current architecture of the Emulator makes it fairly easy to add code to Solarium that can dynamically generate sensor readings for a virtual SPOT. This may take the form of a palette of virtual devices, such as a signal generator that could be hooked up to an analog input, or a way to load user written Java code into Solarium and have it control the input to a virtual SPOT, e.g. to compute the acceleration on the SPOT as it is moved.

Some more ambitious possible future directions involve adding simulation of the radio—controlling the topology of radio connections, varying the percentage of dropped packets, setting transmission power levels, or making transmissions visible in the Emulator so as to visualize the radio traffic—or of the power controller—simulating battery usage.

### ***The Robot View***

The Sun SPOT has been used to control the iRobot Create robot, which is based on the iRobot Roomba. There is even a yearly contest aimed at high school students.<sup>2</sup> To help support this activity the Emulator was extended to allow a virtual SPOT to control an emulated Create robot trying to traverse a maze.



<sup>2</sup> Go to <http://iaroc.org> for more information.

Please refer to the `iRobotCreateDemo` for more information. It uses a special library, `<SDK>/lib/create_host.jar`, that contains methods to control the Create robot in the Emulator in Solarium. Documentation for the library is in the `CreateAPIjavadoc` directory.

Note: Another library, `create_device.jar`, can be used by a real Sun SPOT to control a real iRobot Create. Since both libraries use the same API, the same SPOT application can be run in the Emulator and on a real Create.

## Sun SPOT Programming Basics

If you have not yet done so, go through the Tutorial mentioned on page 11. This will teach you how to modify an existing application then deploy it to, and run it on, your Sun SPOT. If you are using NetBeans and you have installed the Sun SPOT NetBeans modules, an easy way to create a new SPOT application is to use the *File > New Project* command and select “Sun SPOT Application” from the “General” category. NetBeans then creates all the files you need, including a simple MIDlet that you can modify, and the ancillary files and directory structure that NetBeans and the Ant scripts use.

Alternately, you can just copy a demo project. The copy gets you the ancillary files that NetBeans and the Ant scripts use and it makes sure you get the right subdirectory structure. Finally, there is a sample application in

```
[SunSPOTdirectory]/Demos/CodeSamples/SunSpotApplicationTemplate
```

You can copy that directory to get an extremely simple “Hello, World” application from which you can work. There is also a `SunSpotHostApplicationTemplate` in the same directory that contains a simple application using the basestation.

---

**Note** – If you start with one of these templates and rename the main class of the project, you must update the contents of the `[projectdirectory]/resources/META-INF/manifest.mf` file to match.

---

For a more in-depth description of the build and run process that will answer most advanced questions as well, see page 63.

Sun SPOT applications are just Java applications, so the programming language itself should be straightforward. The main issues for most programmers are (1) debugging the code and (2) determining how to get access to the peripherals on the demo sensor board. We will discuss both in the sections below.

## Debugging on a Sun SPOT

You can debug Sun SPOT applications, using either print statements in your code or using the Over The Air (OTA) debugger. The standard output for free range SPOTs can be directed, over the air, to the console of the host workstation.

### **OTA Debugging**

There are three steps to doing OTA debugging. The first step is to enable an OTA link between a Sun SPOT basestation and a free range Sun SPOT. The second is to deploy and run, in debugging mode, the application on the free-range SPOT. The final step is to attach the NetBeans debugger or your debugger of choice to the application and debug the application.



## **IEEE Extended MAC Address**

OTA communication to a Sun SPOT requires the IEEE extended MAC address for all Sun SPOTs involved. The IEEE extended MAC address is a 64-bit address, expressed as four sets of four-digit hexadecimal numbers: nnnn.nnnn.nnnn.nnnn. The first eight digits are always 0014.4F01. The last eight digits should be printed on a sticker visible through the translucent plastic on the radio antenna fin. A typical sticker would say something like "0000.0106" and that would imply an IEEE address for that SPOT of 0014.4F01.0000.0106.

You can also get the IEEE address for a Sun SPOT using the `ant info` command. To get the IEEE address this way, connect your Sun SPOT to the USB cable, open a command window on the host workstation, navigate to any Sun SPOT project directory, and execute the command:

```
ant info
```

You get output that ends with a section that looks like this:

```
[java] Sun SPOT bootloader (yellow-100901)
[java] SPOT serial number = 0014.4F01.0000.0106
[java]
[java] Startup configuration:
[java]   OTA Command Server is enabled
[java]   Configured to run midlet:
[java]     org.sunspotworld.demo.SPOTBounce in SPOT Bounce Demo
[java]
[java]   Squawk command line:
[java]     -spotsuite://library
[java]     -Xboot:268763136
[java]     -Xmxnvm:0
[java]     -isolateinit:com.sun.spot.peripheral.Spot
[java]     -dma:1024
[java]     com.sun.spot.peripheral.ota.IsolateManager
[java]
[java] Library suite:
[java]   hash=0xa31883
[java]   Installed library matches current SDK library
[java]   Installed library matches shipped SDK library
[java]   Current SDK library matches shipped SDK library
[java]
[java] Security:
[java]   Owner key on device matches key on host
[java]
[java] Configuration properties:
[java]   spot.battery.model: LP523436D
[java]   spot.battery.rating: 770
[java]   spot.external.0.firmware.version: 1.15
[java]   spot.external.0.hardware.rev: 8.1
[java]   spot.external.0.part.id: EDEMOBOARD
[java]   spot.hardware.rev: 8
[java]   spot.ota.enable: true
[java]   spot.powercontroller.firmware.version: PCTRL-2.06
[java]   spot.sdk.version: yellow-100827
[java]   spot.startup.isolates.uriids: spotsuite://Oracle/BounceDemo-OnSPOT^1
[java]
[java] Exiting
-run-spotclient-multiple-times-locally:
-run-spotclient:
BUILD SUCCESSFUL
Total time: 28 seconds
```

The IEEE extended MAC address is the number that follows “SPOT serial number:” In this case, the MAC address is 0014.4F01.0000.0106.

The IEEE address for a SPOT is also displayed when that SPOT is discovered in Solarium.

### ***Enable an OTA Link***

#### **1. Enable the OTA command server on the free-range SPOT.**

The OTA command server is enabled on Sun SPOTs direct from the factory. To test whether the OTA command server is enabled, execute the “ant info” as described directly above. The output will say either “OTA Command Server is enabled” or “OTA Command Server is disabled.”

You can enable OTA communication using the SPOTManager tool or a command line. To use the SPOTManager tool, connect the Sun SPOT to the USB cable, go to the Sun SPOTs tab in the SPOTManager, select the Sun SPOT from the pull-down menu, and then click the OTA button and select “Enable.”

To enable the OTA command server using a command line, connect your Sun SPOT to the USB cable, open a command window on the host workstation, navigate to any Sun SPOT project directory, and execute the command:

```
ant enableota
```

If you later decide to disable OTA communication, you can use the command

```
ant disableota
```

You can test whether or not the command worked by repeating the “ant info” command. Disconnect the free-range SPOT from the USB cable.

#### **2. Enable the SPOT basestation.**

Connect the basestation SPOT to the USB cable. If you execute an ant info command, in the Startup section, you will see a line that either says:

```
[java] Configured as a Basestation
```

or

```
[java] Configured to run the current application
```

“Configured to run the current application” means that the SPOT is not running in basestation mode.

To put the Sun SPOT into basestation mode, enter the command:

```
ant startbasestation
```

and the SPOT is put into basestation mode. You can confirm that it is in basestation mode with the ant info command. However, the ant info command stops the basestation application. After executing the ant info command, press the control button on the basestation to restart the basestation application

Now the free-range SPOT and the basestation are capable of communicating with each other in debugging mode.

### ***Deploy and Run the Application in Debugging Mode***

The next step is to deploy the application code to the free-range Sun SPOT and run it in debug mode. To do this:

### 1. Open a command window.

Under Windows, this is usually available from *Start > All Programs > Accessories > Command Prompt*. On a Macintosh or a Linux machine, any command line window will do.

### 2. Navigate to the project directory for the application that you wish to debug.

### 3. Deploy the application to the free-range SPOT using the command:

```
ant -DremoteId=nnnn.nnnn.nnnn.nnnn deploy
```

where *nnnn.nnnn.nnnn.nnnn* is the IEEE extended MAC address for the free-range SPOT. Ant command options are case-sensitive. The options `-DremoteID`, `-DRemoteID`, and `-DRemoteId` will not work. It must be `-DremoteId`.

### 4. Launch the application in debug mode, using the command:

```
ant -DremoteId=nnnn.nnnn.nnnn.nnnn debug
```

where *nnnn.nnnn.nnnn.nnnn* is the IEEE extended MAC address for the free-range SPOT.

The command line output rapidly scrolls through some output, then waits for 30 seconds to a minute, and then prints output that ends with:

```
-do-debug-proxy-run:  
[java] Trying to connect to VM on radio://0014.4F01.0000.0106:9  
[java] Established connection to VM (handshake took 70ms)  
[java] Waiting for connection from debugger on serversocket://:2900
```

Note the socket number indicated in the last line.

### ***Attach to the Debugger***

The final step is to attach your debugger to the application running on the free-range SPOT. The method for doing this varies with the debugger. For NetBeans, the steps are:

1. Within NetBeans, open the project that you wish to debug.
2. Ask NetBeans to attach the debugger.

You can do this either through the *Attach Debugger* command from the Run menu on the main toolbar, or you can press the *Attach Debugger* icon in the upper right. It is a blue triangle, pointing to the right, with a small red square just to the left of it.

A dialog box displays.

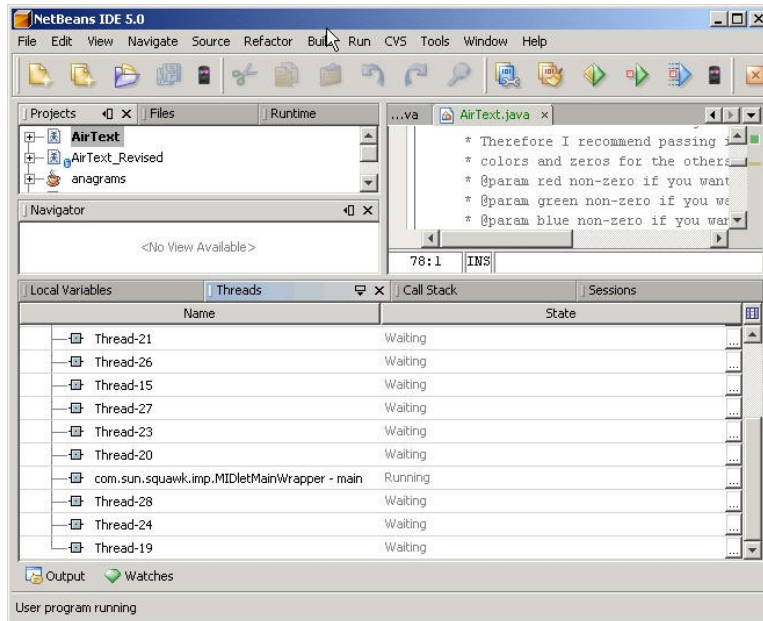


### 3. Specify the debugging attachment.

For the connector type, select *SocketAttach*. This gives you an opportunity to enter the port number for the socket. Enter the socket number that you got in command line window after doing the debug-run command. Click *OK*. After about ten or fifteen seconds, the debugger should attach to the application on your free-range SPOT.

### 4. Find the application thread and debug.

The debugger looks something like this:



The application thread is the one labeled `MIDletMainWrapper`.

It is beyond the scope of this document to explain NetBeans and debugging commands, but NetBeans has a good help system and has a typical set of debugging commands.

### **Print Debugging**

For print debugging, load the application to be debugged onto a Sun SPOT. As you debug, add calls to `System.out.println()`. For example:

```
System.out.println("Got to the Foobar method call");
```

Start the application using either the `Run` command from within NetBeans or by issuing an `ant run` command at command-line prompt from within the project directory. If the SPOT is not connected to the host workstation, the IEEE address of the SPOT must be included in the `ant` command:

```
ant -DremoteId=nnnn.nnnn.nnnn.nnnn
```

where `nnnn.nnnn.nnnn.nnnn` is the IEEE address of the SPOT.

Unless they have been redirected `System.out` and `System.err` appear on the host workstation. In NetBeans, it appears in an Output panel, normally along the bottom of the NetBeans window. In a command-line window, the debugging output appears as part of the output from the `ant run` command.

The process that runs Sun SPOT applications from the host produces a fair amount of output to the command line or output window. When debugging, be sure to scroll back through that output to see if anything important has been printed but scrolled out of sight.

For a more complete discussion of debugging, see the section on page 116.

## Accessing the Sensor Board

The sensor board includes a 3D accelerometer, a tricolor light sensor, eight tricolor LEDs, two switches, four general-purpose I/O pins, and four high current output pins. In this section, we give a rapid introduction to using these components in a Sun SPOT Java program. For more details, see the Javadoc pages in the `[SpotSDKdirectory]/doc/javadoc` directory and see the demonstration applications in the `[SpotSDKdirectory]/Demos/CodeSamples` directory. These applications are simple working applications that show the details of using one or two sensor board devices.

The simplest interfaces to sensor board devices are also described in the sections below. All of the sensor board input devices also have listener classes associated with them. These listener classes are detailed in the Javadoc.

It is important to understand the use of the `Resources` class, as it is how the programmer accesses various parts of the Sun SPOT.

### Resources

In this latest version of the system, we are introducing a new way of accessing resources, such as sensors, by asking for them by the Interface that you want to access them by. This allows the same application code to work on different devices, as long as those devices support the Resource API. For instance the new rev 8 SPOT sensor board has a different accelerometer than the older rev 6 version, however by asking for an `IAccelerometer3D` resource your application can get access to the accelerometer without needing to know about the underlying implementation class.

Central to this approach are the class `Resources` and the interface `com.sun.spot.resources.IResource`. (Earlier version of the system used another interface `IResource` from the now deprecated package `com.sun.spot.resourcesharing`, so be careful that you are using the proper interface `IResource`.)

Each of the sensors, I/O pins, and virtually anything you can access that represents a physical part of the Sun SPOT is presented to the programmer as a Java object we call a “resource,” meaning it implements the interface `IResource`. A resource object is generally shared among all applications running on the Sun SPOT.

The Sun SPOT library also includes a class `Resources` with static methods that let you access the various system resources. You can think of the `Resources` class as a repository with static methods (variants of `lookup(...)`) that help you find what you are looking for. For example, the expression

```
Resources.lookup( IAccelerometer3D.class )
```

returns the first instance it finds that implements the `IAccelerometer3D` interface. Normally of course there is only one such accelerometer, but this approach intentionally allows multiple instances of the same class (or multiple implementors of the same interface) to be stored in `Resources`. You can get an array of all resources implementing an interface by using `lookupAll(...)`, for example,

```
IResource[] all3DAccels = Resources.lookupAll( IAccelerometer3D.class );
```

## Tags

The `IResource` interface requires its implementing classes to have methods for adding, removing, and accessing “tags.” A tag is simply a `String`: users can tag objects to distinguish similar instances from each other, or to make it easy to find certain system resources, possibly those you create on your own.

Tags can be used as arguments to the lookup methods on class `Resources`. For example, by convention, all sensors and actuators on the Sun SPOT EDemoBoard have a tag `"location=eDemoboard"`, so even if there were other accelerometer objects in your `Resources`, you could find the one on the EDemoBoard by using a lookup that takes a class (or interface) *and* a tag:

```
Resources.lookup( IAccelerometer3D.class, "location=eDemoboard" )
```

This is especially useful when there are several instances of a given resource and you want to obtain a pointer to a specific one of them. For example to access switch 1 use:

```
Resources.lookup( ISwitch.class, "SW1" )
```

See the javadoc in the directory `[SpotSDKdirectory]/doc/javadoc` for the full API of `Resources` and `IResource`.

## Sensors

We want to define as best we can generic interfaces to common sensors and actuators for SPOTs. These interfaces are defined in the base spotlib, so that they can be used in SPOT applications without needing to refer to a specific implementation (e.g. `transducerlib` or user specific library for a different daughtercard). The initial set of interfaces are defined in the `com.sun.spot.resources.transducers` package.

Each sensor provides metadata that describes its accuracy, range, etc. for use by applications and to be relayed to host apps such as `sensor.network`.

Hardware specific functionality, such as interrupting on a threshold being exceeded, belongs in the implementation for the sensor, not in the generic interface.

User apps can define "conditions" to periodically check (via software polling) if a sensor's value meets some predefined criteria, e.g. exceeds a threshold, outside a specified range, etc.

### Sensor Metadata

All sensors (and actuators) implement the new `ITransducer` Interface, which extends the `IResource` Interface. Most metadata associated with a transducer should just be put into tags so that it can be searched for when doing a resource lookup. For example: name, vendor, model, version, location, latitude, longitude.

Some other metadata is defined by the API for the specific transducer. For example: units, scale, data type, etc.

Some other metadata depends on the implementing class so needs to be available via the `get` methods defined for `ITransducer`, such as a description of the sensor, its measurement range(s), and its maximum sampling rate.

Where appropriate a sensor class may implement the `IMeasurementInfo`, `IMeasurementRange` or `IMeasurementRangeVariable` interfaces. The `IMeasurementInfo` interface is used to access the min & max values, the resolution and the accuracy of the sensor. If a sensor has several measurement ranges and the application can set the measurement range of a sensor, then the sensor should implement the new interface `IMeasurementRange`, for example setting the scale used by an accelerometer to be 2 or 8 G's.

## SensorEvent class

A new `SensorEvent` class contains fields relevant to the Sensor state when an event occurs and is used to pass that sensor state to a user callback method from either a hardware interrupt or software polling.

The base `SensorEvent` class defines fields to hold the time the event occurred (a long) & a pointer to the relevant Sensor (actually an `ITransducer`). Each sensor type should define a child class to extend `SensorEvent` so that it can hold the Sensor's current value (an int, double, array, etc.), and any other Sensor state specific information (e.g. hardware threshold values, scale in use,, etc.)

The `ITransducer` Interface includes a factory method to create a new event of the proper type for the sensor, e.g. `createSensorEvent()`, and a method to store the current Sensor state into a pre-existing event, e.g. `saveEventState(SensorEvent evt)`.

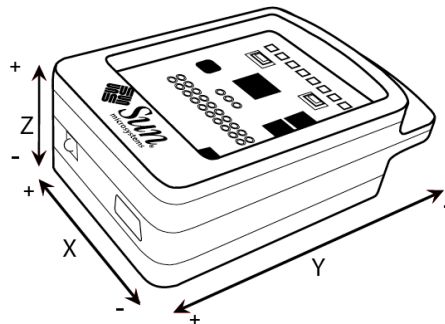
We now turn to an overview of some of the transducers on the SunSPOT sensor board.

## Accelerometer

This is a very brief introduction to the Sun SPOT accelerometer. For more detail, see the AppNote included in `[SpotSDKdirectory]/doc/AppNotes/AccelerometerAppNote.pdf`.

There are three axes on which the accelerometer measures acceleration. The Z-axis is perpendicular to the Sun SPOT boards. The X-axis is parallel to the row of LEDs on the sensor board. The Y-axis is parallel to the long edge of the sensor board.

The accelerometer's X, Y, and Z axes are illustrated below.



In the figure above, the plus (+) on the end of an axis indicates that when the device's acceleration vector increases in that direction, the associated accelerometer readings grow larger.

To use the accelerometer:

```
//Find the accelerometer interface instance
import com.sun.spot.resources.Resources;
import com.sun.spot.sensorboard.IAccelerometer3d;
IAccelerometer3D ourAccel =
    (IAccelerometer3D)Resources.lookup( IAccelerometer3D.class );
//Read from the accelerometer
double x-accel = ourAccel.getAccelX();
double y-accel = ourAccel.getAccelY();
double z-accel = ourAccel.getAccelZ();
```

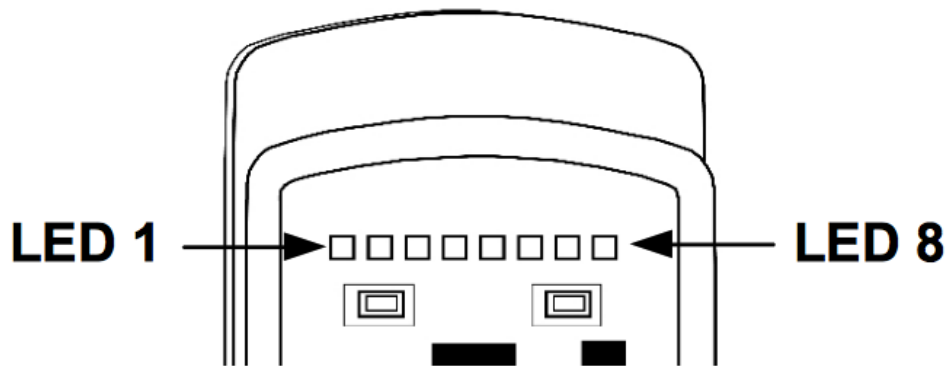
The readings are in g-force units. There is also a method `getAccel()` that returns the vector sum of the acceleration along all three individual axes.

`IAccelerometer3D` also has methods that calculate the orientation of the SPOT based on the acceleration of the SPOT. When the SPOT is at rest, this acceleration is gravity and the tilt is relative to gravity. The methods are `getTiltX()`, `getTiltY()`, `getTiltZ()`, and they return the tilt in radians.

More details are available in the Javadoc on `IAccelerometer3D` class. If you require more control than is available through this interface, look at the Javadoc for the `MMA7455LAccelerometer` and `LIS3L02AQAccelerometer` class.

## LEDs

There are eight three-color LEDs on the demo sensor board, in a row, with LED1 on the left and LED8 on the right.



Each LED has a red, a green, and a blue emitter as part of the LED. Each individual color can have an intensity from 0 to 255, with 0 being off and 255 being as bright as possible.

To use the LEDs:

### 1. Find the LED object array.

```
import com.sun.spot.resources.Resources;
import com.sun.spot.resources.transducers.ITriColorLEDArray;
ITriColorLEDArray ourLEDs =
    (ITriColorLEDArray)Resources.lookup( ITriColorLEDArray.class );
```

### 2. Set the LED color desired.

Colors are specified with the `setRGB(int red, int green, int blue)` method. Individual LEDs are referenced with `getLED(int index)`.

```
// set the LED color desired
// set the first two LEDs to bright red, the next two to bright green,
// the next two to bright blue, and the last two to white.
// First two = bright red
ourLEDs.getLED(0).setRGB(255,0,0); ourLEDs.getLED(1).setRGB(255,0,0);
// Next two = bright green
ourLEDs.getLED(2).setRGB(0,255,0); ourLEDs.getLED(3).setRGB(0,255,0);
// Next two = bright blue
ourLEDs.getLED(4).setRGB(0,0,255); ourLEDs.getLED(5).setRGB(0,0,255);
// Last two = white
ourLEDs.getLED(6).setRGB(255,255,255); ourLEDs.getLED(7).setRGB(255,255,255);
```

### 3. Turn all the LEDs on.



```
ourLEDs.setOn(); // turn all the LEDs in the array on
```

#### 4. If desired, turn the LEDs off.

```
ourLEDs.setOff(); // turn all the LEDs in the array off
```

You can also query the state of individual LEDs using the `isOn()`, `getRed()`, `getGreen()`, and `getBlue()` methods.

## Switches

The sensor board has two switches on it. These are represented in the `EDemoBoard` object as an array of type `ISwitch`. You may query the state of the switches using the `isOpen()` and `isClosed()` methods. Ordinarily you implement an event loop that checks the switches used in your application on a periodic basis, or you ask the Sun SPOT to stop and wait for the switch state to change. When you want the SPOT to wait for the state switch to change, you would use the `waitForChange()` method.

### 1. Find the two switches

```
import com.sun.spot.resources.Resources;
import com.sun.spot.resources.transducers.ISwitch;
//The switches each have a tag: "SW1" for the left, and "SW2" for the right.
ISwitch sw1 = (ISwitch) Resources.lookup(ISwitch.class, "SW1");
ISwitch sw2 = (ISwitch) Resources.lookup(ISwitch.class, "SW2");
```

### 2. Use the `ISwitchListener` interface

If you want to be notified when a switch is either pressed or released, you can "listen" to it by implementing `ISwitchListener` interface. To listen to switch 1 then, you execute

```
sw1.addISwitchListener(this);
```

Then in your class you implement the two methods

```
public void switchPressed(SwitchEvent evt) {
    //insert code to handle when sw is pressed.
}
public void switchReleased(SwitchEvent evt) {
    //insert code to handle when sw is released.
}
```

which are "callbacks". The `SwitchEvent` will contain details about the event, including which switch has changed state.

### 3. Look for a switch press.

If you wanted a switch "click" (a press and release) and were willing to wait for it:

```
if (sw1.isOpen()){ // if it is open, wait for it to close
    sw1.waitForChange();
}
// We now know it is closed -- wait for it to open.
sw1.waitForChange();
```

## Light Sensor

### Light Sensor (revision 6)

The light sensor returns an integer that ranges from 0 to 750. Zero represents complete darkness. Peak sensitivity of light sensor is at 600nm wavelength. An illustration of how the raw readings map to luminance (lx) values is given in the table below:

<i>Luminance</i>	<i>Raw Reading</i>
1000 lx	497
100 lx	50
10 lx	5

To use the light sensor:

#### 1. Find the light sensor object.

```
import com.sun.spot.resources.Resources;
import com.sun.spot.sensorboard.peripheral.ILightSensor;
ILightSensor ourLightSensor = Resources.lookup(ILightSensor.class);
```

#### 2. Get the light sensor raw reading.

```
int lightSensorReading = ourLightSensor.getValue();
```

This is fine for a constant light source. However, some light sources, specifically fluorescent light bulbs, while seeming constant to the human eye, actually vary rapidly. For these sources, it is better to use the method `getAverageValue(int n)`. The method returns the average of *n* samples taken at 1 millisecond intervals. If *n* is not specified, 17 samples are taken, spanning one sixtieth of a second, or the usual length of a power/light cycle.

```
int lightSensorReading = ourLightSensor.getAverageValue(34);
```

### TriColor Light Sensor (revision 8)

The revision 8 SPOT sensorboard replaces the simple light sensor with a tricolor one that is capable of sensing in red, green, blue, and clear channels.

## Temperature Sensor

The temperature sensor is the simplest of the sensors. There are no raw readings and no parameters to set. However, it is, inevitably, close to some heat sources in the Sun SPOT. More accurate temperature readings could be obtained with an external temperature sensor tied to the I/O pins on the sensor board.

#### 1. Find the temperature sensor object.

```
import com.sun.spot.resources.Resources;
import com.sun.spot.sensorboard.peripheral.ITemperatureInput;
ITemperatureInput ourTempSensor =
    (ITemperatureInput) Resources.lookup( ITemperatureInput.class );
```

## 2. Read the temperature.

```
// The temperature can be read in Celsius
double celsiusTemp = ourTempSensor.getCelsius();
// or in Fahrenheit
double fahrenheitTemp = ourTempSensor.getFahrenheit();
```

Note: the revision 6 SPOT has two temperature sensors, one in the A-to-D chip on the sensor board and a second on the main processor board connected to the power controller. The rev 8 SPOT only has the temperature sensor on the main processor board.

## Creating your own Sun SPOT Resources

The `Resources` class includes methods that allow you to remove or add resource objects at any time, as long as they implement the `IResource` interface. So, for example, you can add a fundamentally new kind of resource that you create on your own. The system includes a class `com.sun.spot.resources.Resource` that implements the `IResource` interface, so you can simply extend that class.

As another example, you could remove the standard system accelerometer from `Resources`, and add your own kind of accelerometer that reports a special stream of test acceleration values regardless of whether or not the physical device moves. If your “test” accelerometer implements the same interface and has the same tags as the original, then once you have made the replacement, subsequent applications that `lookup(...)` to find the accelerometer should find themselves using your “test” accelerometer.

It is important to note that resource objects are shared among Isolates. This means that resource implementors should take care to minimize interference among Isolates that use that resource. Your resource should be thread safe, and aware that some of those threads may come from other Isolates.

One specific and perhaps non-obvious warning: each Isolate gets its own copy of the static fields associated with a class. So, even though a resource instance is literally shared between two Isolates, the static fields in the class of that resource are *not* shared. When creating your own resource the field initialization code will run in each Isolate that uses your resource, and any new assignment to that field will be visible only within the Isolate that made the change. In general you will want to avoid assigning new values to static fields, and avoid changing the state of objects referenced by them.

## Conditions

Some sensor hardware can be configured to generate an interrupt if the sensor's value changes, such as when a switch is pressed, or the value exceeds some threshold. Other similar sensors may not do so or may do so based on very different tests. In order to have the SPOT periodically test a generic sensor to see if some condition is met the `Condition` class is used. For monitoring a particular sensor the user can extend the `Condition` class by overriding the `isMet()` method, which is run periodically—if it returns true then all callbacks registered to the `Condition` will be executed.

For example, here is some code to detect when the temperature goes below freezing, checking the temperature sensor every 30 seconds:

```
// locate a temperature sensor
ITemperatureInput temp =
    (ITemperatureInput) Resources.lookup(ITemperatureInput.class);

// define the callback
IConditionListener freezeListener = new IConditionListener() {
```

```

        public void conditionMet(ITransducer sensor, Condition condition) {
            System.out.println("Brrr... it's cold!");
        }
    };

    // define the condition - check temperature every 30 seconds
    Condition freeze = new Condition(temp, freezeListener, 30 * 1000) {
        public boolean isMet() {
            if (((ITemperatureInput) sensor).getCelsius() <= 0.0) {
                stop();          // stop checking once temperature goes below freezing
                return true;     // condition met: notify listeners
            } else {
                return false;    // condition not met
            }
        }
    }
    freeze.start();    // start monitoring the condition

```

Once started the `Condition` is periodically checked until either the application exits or the `stop()` method of the `Condition` is called. The example above calls `stop()` when the condition is met to prevent future checking, so the `freezeListener` will only be called once. Without the call to `stop()` the `freezeListener` would be called every 30 seconds as long as the temperature remained below 0° Celsius. A more sophisticated `isMet()` condition could be written to keep track of prior state and only call the `freezeListener` when the temperature transitioned from above to below freezing.

If the `isMet()` method is not overridden then the callbacks will be run each time the condition is tested. This is a simple way to invoke a data sampler. Here is code to report the temperature every 60 seconds:

```

// locate a temperature sensor
ITemperatureInput temp =
    (ITemperatureInput)Resources.lookup(ITemperatureInput.class);

// define the callback
IConditionListener tempReporter = new IConditionListener() {
    public void conditionMet(ITransducer sensor, Condition condition) {
        System.out.println("Current temperature is " +
            ((ITemperatureInput) sensor).getCelsius());
    }
};

// define the condition - report temperature every 60 seconds
Condition tempSampler = new Condition(temp, tempReporter, 60 * 1000);

tempSampler.start();    // start periodically reporting the temperature

```

### **The new *Task* class**

The `Condition` class is a child class of the more general `Task` class that periodically invokes its `doTask()` method. The above temperature reporter could be coded using `Task` instead of `Condition` as:

```

// locate a temperature sensor
final ITemperatureInput temp =
    (ITemperatureInput)Resources.lookup(ITemperatureInput.class);

// define the task - report temperature every 60 seconds
Task tempTask = new Task(60 * 1000) {
    public void doTask() {
        System.out.println("Current temperature is " + temp.getCelsius());
    }
};
tempTask.start();

```

```

    }
}

tempTask.start(); // start periodically reporting the temperature

```

In addition to running a `Task` periodically every `N` milliseconds, a `Task` can have a start time, expressed as a time of day: hours, minutes & seconds. A `Task` can also have an end time, also expressed as a time of day. So one can specify that a `Task` should be run everyday starting at 08:00:00 and again every 15 minutes until 17:00:00. When reaching the end time, the `Task` would be scheduled to run again the following day at its start time. Please see the `Task` Javadoc for full details.

## Radio Communication

### **Radiostreams**

The `RadiostreamConnection` interface provides a socket-like peer-to-peer radio protocol with reliable, buffered stream-based IO between two devices. This communication can be single-hop or multi-hop.

To open a connection:

```

StreamConnection conn = (StreamConnection)
Connector.open("radiostream://nnnn.nnnn.nnnn.nnnn:xxx");

```

where `nnnn.nnnn.nnnn.nnnn` is the 64-bit IEEE address of the radio, and `xxx` 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.

To establish a connection both ends must open connections specifying the same port number and complimentary IEEE addresses.

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

```

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

```

### **Radiograms**

The `RadiogramConnection` interface defines the “radiogram” protocol – the radiogram protocol is a datagram-based protocol that allows the exchange of packets between two devices.

To establish a point-to-point connection both ends must open connections specifying the same `portNo` and corresponding IEEE addresses. Port numbers between 0 and 31 are reserved for system services. Use of these ports by applications may result in conflicts.

Once the connection has been opened, each end can send and receive data using a datagram created on that connection. For example:

```

DatagramConnection conn = (DatagramConnection) Connector.open("radiogram://" +
targetIEEEAddress + ":100");
Datagram dg = conn.newDatagram(conn.getMaximumLength());
dg.writeUTF("My message");
conn.send(dg);
...
conn.receive(dg);
String answer = dg.readUTF();

```

The radiogram protocol also supports broadcast mode, where radiograms are delivered to all listeners on the given port. Because broadcast mode does not use 802.15.4 ACKs, there are no delivery guarantees.

```
DatagramConnection sendConn = (DatagramConnection)
Connector.open("radiogram://broadcast:100");
dg.writeUTF("My message");
sendConn.send(dg);
```

The radiogram protocol also supports server mode, where any radiogram sent on the given port will be received. These may be broadcast packets of packets specifically addressed to the receiving SPOT.

```
DatagramConnection recvConn = (DatagramConnection)
Connector.open("radiogram://:100");
recvConn.receive(dg);
String answer = dg.readUTF();
```

Unicast datagrams have the hop count determined as part of route discovery. Broadcast datagrams are sent for the number of hops specified by the connection. The default value is 2 hops. To make a single hop:

```
sendConn = (RadiogramConnection) Connector.open("radiogram://broadcast:100");
sendConn.setMaxBroadcastHops(1);
```

Broadcast mode is not recommended for datagrams larger than 200 bytes. Because the list of recipients is unknown, broadcast mode is inherently unreliable. With SPOTs, there is also a known problem that may result in fragments of a datagram becoming lost.

The maximum broadcast packet size is 1260 bytes of payload. An individual 802.15.4 radio packet only carries about 100 bytes of data, though the amount varies slightly depending on whether the packet has a mesh header and whether or not the fragment is the first fragment in the datagram.

Broadcast datagrams that result in two fragments are fairly reliable. Datagrams broken into three fragments (over 200 bytes of payload) are likely to experience some loss. Broadcasts datagrams broken into more than three fragments will almost certainly see some loss. The problem is related to the ability of the receiving side to empty the incoming buffer fast enough. This problem may be exacerbated by a receiver side that frequently garbage collects or has a large number of active threads.

Important data should generally be unicast via radiograms or radiostreams. The inherent ACK/retry mechanism of radiograms generally insures either delivery or notification of failure. Likewise, radiostreams provide automatic fragmentation and an additional level of assurance that fragments are reassembled in the proper order.

Alternatively, if broadcast is required, the application should attempt to limit packet size so that each broadcast results in less than 3 fragments. Single packet broadcasts result in more data space as a fragmentation header is not required. Additionally, inserting a 20ms pause between the sending of broadcast packet assists in allowing the receiver to keep up with packet reception.

## Part II: Developing for the Sun SPOT

The purpose of Part II is to aid developers of applications for Sun SPOTs. Part II is itself 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 basestation to communicate with physically remote Sun SPOTs via the radio
- Using the basestation to deploy and execute applications on remote Sun SPOTs
- Building and running your own application running on the host machine that communicates, via the basestation, with remote Sun SPOTs
- Managing the keys that secure your Sun SPOTs against unauthorized 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*

If you are working from the command line rather than an IDE, the typical process for creating and running an application 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 jar is created in the `suite` folder and is named according to the values of two properties from the application's manifest file. The name will be

`<MIDlet-Name>_<MIDlet-Version>.jar`

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

-pre-init:

-do-init:

-post-init:

-warn-jar-file:

init:

-check-for-manifest:

-set-jar-name:
    [echo] No jar.file specified.
```



```

[echo] Using "suite/SunSpotApplicationTemplate_1.0.0.jar"

-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:

-do-jar-app:
  [mkdir] Created dir: C:\MyApplication\suite
  [jar] Building jar: C:\MyApplication\suite\SunSpotApplicationTemplate_1.0.0.jar

-post-jar-app:

jar-app:

BUILD SUCCESSFUL
Total time: 2 seconds
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:
-do-init:
-post-init:
-warn-jar-file:
init:
-override-warning-find-spots:
-prepare-conditions-for-find-spots:
-find-shared-basestation:
-run-spotfinder:
-decide-whether-to-run-spotsselector:
-run-spotsselector:
    [java] Please wait while connected Sun SPOTs are examined...
-collect-spotsselector-result:
    [echo]
    [echo] Using Sun SPOT device on port COM18
-clean-up-spotsselector-output-file:
    [delete] Deleting: C:\SunSPOT\yellow-100901\temp\spotsselector-1532747104
-spotsselector-fail:
-decide-whether-to-start-basestation-manager:
-start-new-basestation-manager:
-do-find-spots:
info:
-check-run-spotclient-parameters:
-run-spotclient-once-with-remote-id:
-run-spotclient-multiple-times-with-remote-id:
-run-spotclient-once-locally:
-echo-progress-for-remote-runs:
-echo-progress-for-local-runs:
-run-spotclient-once:
    [java] SPOT Client starting...
    [java] [waiting for reset]
    [java]
    [java] Local Monitor (yellow-100901)
    [java] SPOT serial number = 0014.4F01.0000.00B7
    [java]
    [java] Startup configuration:
    [java]     OTA Command Server is enabled
    [java]     Configured to run midlet:
    [java]         org.sunspotworld.demo.SPOTBounce in SPOT Bounce Demo
    [java]
    [java] Squawk command line:
    [java]     -spotsuite://library

```

```
[java] -Xboot:268763136
[java] -Xmxnvm:0
[java] -isolateinit:com.sun.spot.peripheral.Spot
[java] -dma:1024
[java] com.sun.spot.peripheral.ota.IsolateManager
[java]
[java] Library suite:
[java]   hash=0xa37ef9
[java]   Installed library matches current SDK library
[java]   Installed library matches shipped SDK library
[java]   Current SDK library matches shipped SDK library
[java]
[java] Security:
[java]   Owner key on device matches key on host
[java]
[java] Configuration properties:
[java]   spot.battery.model: LP523436D
[java]   spot.battery.rating: 770
[java]   spot.external.0.firmware.version: 1.15
[java]   spot.external.0.hardware.rev: 8.1
[java]   spot.external.0.part.id: EDEMOBOARD
[java]   spot.hardware.rev: 8
[java]   spot.ota.enable: true
[java]   spot.powercontroller.firmware.version: PCTRL-2.06
[java]   spot.sdk.version: yellow-100901
[java]   spot.startup.isolates.uriids: spotsuite://Oracle/SPOT_Bounce_Demo^1
[java]
[java] Exiting

-run-spotclient-multiple-times-locally:

-run-spotclient:

BUILD SUCCESSFUL
Total time: 4 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 (COM18 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:

-do-init:

-post-init:

-warn-jar-file:

init:

-try-set-from-jar-name-and-deploy:

-check-for-manifest:

-set-to-jar-name:
    [echo] No to.jar.file specified.
    [echo] Using "suite/SunSpotApplicationTemplate_1.0.0.jar"

-set-from-jar-name:
    [echo] Using "suite/SunSpotApplicationTemplate_1.0.0.jar"

-pre-init:

-do-init:

-post-init:

-warn-jar-file:

init:

-set-selector-for-nonbasestation:

-override-warning-find-spots:

-prepare-conditions-for-find-spots:

-find-shared-basestation:

-run-spotfinder:

-decide-whether-to-run-spotsselector:

-run-spotsselector:
    [java] Please wait while connected Sun SPOTs are examined...

-collect-spotsselector-result:
    [echo]
    [echo] Using Sun SPOT device on port COM18

-clean-up-spotsselector-output-file:
```

```

[delete] Deleting: C:\SunSPOT\yellow-100901\temp\spotsselector-544497390
-spotsselector-fail:
-decide-whether-to-start-basestation-manager:
-start-new-basestation-manager:
-do-find-spots:
-jar-and-deploy:
-deploy-only:
-check-for-jar:
-pre-suite:
-do-suite-new:
  [java] [translating suite image [closed: false, parent: transducerlib] ...]
  [java] Romizer processed 1 classes and generated these files:
  [java]   C:\MyApplication\image.sym
  [java]   C:\MyApplication\image.suite
  [java]   C:\MyApplication\image.suite.metadata
  [java]   C:\MyApplication\image.suite.api
  [unzip] Expanding: C:\MyApplication\suite\SunSpotApplicationTemplate_1.0.0.jar
into C:\MyApplication\suite
  [move] Moving 1 file to C:\MyApplication\suite
  [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-init:
-do-init:
-post-init:
-warn-jar-file:
init:
-override-warning-find-spots:
-prepare-conditions-for-find-spots:
-find-shared-basestation:
-run-spotfinder:
-check-spotfinder-result:
-decide-whether-to-run-spotsselector:
-run-spotsselector:
-collect-spotsselector-result:
-clean-up-spotsselector-output-file:
-spotsselector-fail:
-decide-whether-to-start-basestation-manager:

```

```

-start-new-basestation-manager:

-do-find-spots:

-pre-deploy:

-set-uri:

-do-deploy:

-check-run-spotclient-parameters:

-run-spotclient-once-with-remote-id:

-run-spotclient-multiple-times-with-remote-id:

-run-spotclient-once-locally:

-echo-progress-for-remote-runs:

-echo-progress-for-local-runs:

-run-spotclient-once:
    [java] SPOT Client starting...
    [java] [waiting for reset]

    [java] Local Monitor (yellow-100901)
    [java] SPOT serial number = 0014.4F01.0000.00B7
    [java] Using target file name: spotsuite://Oracle/SunSpotApplicationTemplate
    [java] Relocating application suite to 0x10900000
    [java] About to flash to C:\MyApplication\suite/image
    [java] Writing imageapp16950.bintemp(2260 bytes) to local SPOT on port COM18
    [java] |===| 9%
    [java] |=====| 13%
    [java] |=====| 22%
    [java] |=====| 27%
    [java] |=====| 31%
    [java] |=====| 36%
    [java] |=====| 45%
    [java] |=====| 50%
    [java] |=====| 63%
    [java] |=====| 68%
    [java] |=====| 72%
    [java] |=====| 77%
    [java] |=====| 86%
    [java] |=====| 90%
    [java] |=====| 95%
    [java] |=====| 100%
    [java]
    [java]
    [java]
    [java] Exiting

-run-spotclient-multiple-times-locally:

-run-spotclient:

-post-deploy:

flashapp:

-do-deploy-only:

deploy:

-deploy-from-jar-name:

```

```
-set-from-jar-name-and-deploy:

jar-deploy:

BUILD SUCCESSFUL
Total time: 8 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:

-do-init:

-post-init:

-warn-jar-file:

init:

-set-selector-for-nonbasestation:

-override-warning-find-spots:

-prepare-conditions-for-find-spots:

-find-shared-basestation:

-run-spotfinder:

-decide-whether-to-run-spotsselector:

-run-spotsselector:
    [java] Please wait while connected Sun SPOTs are examined...

-collect-spotsselector-result:
    [echo]
    [echo] Using Sun SPOT device on port COM18

-clean-up-spotsselector-output-file:
    [delete] Deleting: C:\SunSPOT\yellow-100901\temp\spotsselector-778469344

-spotsselector-fail:

-decide-whether-to-start-basestation-manager:

-start-new-basestation-manager:

-do-find-spots:

-pre-run:

-do-run:

-check-run-spotclient-parameters:

-run-spotclient-once-with-remote-id:

-run-spotclient-multiple-times-with-remote-id:

-run-spotclient-once-locally:

-echo-progress-for-remote-runs:
```

```

-echo-progress-for-local-runs:
-run-spotclient-once:
  [java] SPOT Client starting...
  [java] [waiting for reset]

  [java] Local Monitor (yellow-100901)
  [java] SPOT serial number = 0014.4F01.0000.00B7
  [java]
  [java]
  [java] ** VM stopped: exit code = 0 **
  [java]
  [java]
  [java] Starting midlet 1 in spotsuite://Oracle/SunSpotApplicationTemplate
  [java] Hello, world
  [java] Our radio address = 0014.4F01.0000.00B7
  [java]
  [java]
  [java] ** VM stopped: exit code = 0 **
  [java]
  [java]
  [java] Exiting

-run-spotclient-multiple-times-locally:
-run-spotclient:
-post-run:
run:
BUILD SUCCESSFUL
Total time: 17 seconds
C:\MyApplication>

```

As you can see, this application just prints "Hello World" and then blinks LED1 on and off every second. It should give you a basic framework to use for writing your applications.

N.B. After your Sun SPOT has printed "Hello World" it will not exit. To force it to exit either hold down SW1 (the left push-button switch on the sensor board) or push the control button.

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 specify 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 *Using library suites* in this document.

### ***Including and excluding files from the compilation***

To include or exclude files or folders matching specific patterns from the java compilation, set the ant properties `spot.javac.exclude.src` or `spot.javac.exclude.src`, either on the command line with `-D` or in the `build.properties` file of the project. The values of the properties should be specified using standard ant wildcarding, and may be comma-separated lists of patterns.

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

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

By default, no files are excluded and all files matching the pattern `**/*.java` are included.

### ***The MIDlet lifecycle***

SPOT applications conform to the MIDlet standard. You will see from the sample application that all applications implement the three members `startApp()`, `pauseApp()` and `destroyApp()`. This along with an optional no-argument constructor is the interface that you must implement to be a MIDlet application. There are also some inherited members that you can call, notably `notifyDestroyed()`.

Your MIDlet will be constructed and then the `startApp()` member will be called when your application is to be activated. At this time your application should acquire the resources that it requires and start executing. When `destroyApp()` is called your application should release resources and stop executing.

If your application wants to exit, it must call `notifyDestroyed()`. Applications should never call `System.exit()`.

For information about pausing and other details of the lifecycle, try <http://developers.sun.com/mobility/learn/midp/lifecycle/>.

### ***Manifest and resources***

The file `MANIFEST.MF` in the `resources/META-INF` directory contains information used by the Squawk VM<sup>3</sup> 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: Oracle
MIDlet-1: AirText, , org.sunspotworld.demo.AirTextDemo
MicroEdition-Profile: IMP-1.0
```

---

<sup>3</sup> The Squawk VM is the Java virtual machine that runs on the Sun SPOT. For more details, go to <http://labs.oracle.com/projects/squawk/>.

```
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 description 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 manifest can define many MIDlets (`MIDlet-1`, `MIDlet-2`, etc.), each with their own main class. The manifest as a whole defines a MIDlet suite whose name is specified by the `MIDlet-Name` property. A side-effect of the `deploy` command is to change the configuration of the SPOT so that the deployed application will execute on the next reboot. By default it will be `MIDlet-1` that executes. You can specify a different MIDlet when deploying either on the command line

```
ant deploy -Dmidlet=2
```

or by defining the `midlet` property in your project's `build.properties` file

```
midlet=4
```

Multiple MIDlets may be specified by using a comma separated list:

```
midlet=2,4
```

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 within 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) 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:

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

These control the socket connection that underpins http access: see the section Configuring the http protocol.

## ***Managing multiple MIDlets on the SPOT***

More than one MIDlet suite can be installed on a SPOT. Each suite is identified by a uri of the form `spotsuite://xxx`. The uri can be specified by the user or generated automatically using fields from the manifest. To specify the uri for a suite use:

```
ant deploy -Duri=spotsuite://myapp
```

By default, each invocation of `ant deploy` will overwrite the existing suite with the new one. To deploy a new suite alongside the existing one use:

```
ant codeploy -Duri=spotsuite://myapp
```

Using the `codeploy` command any existing suites are retained and an additional MIDlet from the newly deployed suite is added to the SPOT's startup configuration.

The list of currently installed suites can be viewed using:

```
ant getavailablesuites
```

An unwanted suite can be removed using:

```
ant undeploy -Duri=spotsuite://xxx
```

Note that each time the library is deployed to the SPOT (using `ant flashlibrary`, or, for recovery purposes, `ant resetlibrary`) all application suites are erased.

Depending on the installed library configuration, it is possible to execute several MIDlets at the same time. Type `ant help` for details, or use the SPOT World management application.

### **Selecting the startup MIDlet(s)**

When the SPOT reboots it starts up all of the MIDlets listed in the `spot.startup.isolates.uriids` system property. Each MIDlet in the list is specified by its suite URI and its MIDlet number in that suite. If the list is empty then the built-in dummy application, which sleeps forever, is run. Use the `addstartupmidlet` and `removestartupmidlet` commands to modify the startup list. Each command requires up to two properties:

```
ant addstartupmidlet -Duri=xxx -Dmidlet=n
```

If omitted the `midlet` property defaults to 1.

### **Selecting the startup application**

The `setstartup` command allows the user to specify which application should run next time the SPOT is rebooted. The command requires up to two properties:

```
ant setstartup -Duri=xxx -Dmidlet=n
```

If omitted the `midlet` property defaults to 1.

If `uri` is set to "none" or is omitted the `midlet` property is ignored and the SPOT is configured to execute the `IsolateManager`.

### **Specifying what to do when the SPOT is restarted**

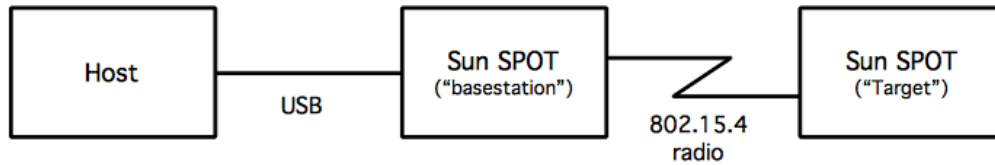
When all the running MIDlets finish or if execution is terminated by an exception that forces the Squawk VM to exit or a Watchdog timer expires, there are several different things the SPOT can do. The default behavior is for the SPOT to restart the Squawk VM and run all of the selected startup MIDlets. This is usually what is wanted if the SPOT is deployed in the field to collect data or monitor various sensors.

By setting the `spot.restart.mode` property two other modes may be selected. Setting it to the value `off` will cause the SPOT to go into deep sleep to conserve power—waking up the SPOT will require physically pushing the Control button. Alternatively setting it to the value `continue` will cause the SPOT to go into shallow sleep where it will continue to listen for OTA radio commands. Setting it to `restart` will select the default behavior of restarting all selected MIDlets.

## 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 basestation and another runs the host application. In this model, the host application has its own system-generated address, distinct from that of the basestation. 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 behavior 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 identifier or output power is not possible.

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

```
basestation.shared=false
```

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 basestation to your host computer. If you don't have a basestation you can configure a Sun SPOT to be used as the basestation and start it by issuing the following command:

```
ant startbasestation
```

This command both configures the basestation and runs it. The basestation code must be executing before you can use it. When running the basestation gives a quick double flash of its green LED every 10 seconds. If you don't see this indication press the reset button.

### ***Basestation configuration***

It is possible to select a channel, PAN identifier and transmit power for the basestation using the SPOT system properties:

```
radio.channel  
radio.pan.id  
radio.transmit.power
```

The `ant set-radio-properties` command will store these properties to the basestation and they will be used whenever the basestation is started up.

These radio properties can be changed for a single command by specifying them on the command line, e.g.

```
ant info -DremoteId= 0014.4F01.0000.0006 -Dradio.channel=11
```

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

```
RadioFactory.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 basestation.

The operations that can be performed remotely include:

- `ant deploy`
- `ant jar-deploy`
- `ant run`
- `ant fork`
- `ant debug`
- `ant info`
- `ant settime`
- `ant flashlibrary`
- `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 basestation
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 basestation***

For details, see the section Using the Basestation.

### ***Launch the SPOT Client to control a remote Sun SPOT via the basestation***

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<sup>4</sup>. 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] Local Monitor (yellow-100901)
[java] SPOT serial number = 0014.4F01.0000.02ED
```

---

<sup>4</sup> 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.

As mentioned above it is also possible to specify which radio channel and PAN identifier the basestation should use to communicate with the remote SPOT, plus the transmit power the basestation should use. To do this, set the ant properties `radio.channel`, `radio.pan.id` and `radio.transmit.power` either on the command line or in the `.sunspot.properties` file in your home 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 bytcodes
- 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 over-the-air. 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
```

or, if user B is using a different library to user A,

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

User B 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` or `flashlibrary`) 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
```

or

```
ant flashlibrary
```

### **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***

A host application is a Java program that executes on the host computer (e.g. a desktop PC). Host applications have access to libraries with a subset of the API of the libraries used by SPOT applications. This means that a host application can, for example, communicate with a SPOT via a basestation using code identical to that which you would use to communicate between two SPOTs. Note that the host application does not run on the basestation; it sends commands to the basestation over a USB connection.

## ***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 basestation as outlined in the section *Managing multiple MIDlets on the SPOT*. 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)

```
Using Sun SPOT basestation on port COM3
```

Normally, the basestation 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 basestations 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_common.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 specify multiple jars as a list separated by a classpath delimiter ("`;`" 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
```

### ***Creating an executable jar file for a host application***

It is possible to make an executable jar file of a desktop application. To do so you must manually set some Java system properties that are normally setup by the ant scripts. You need to do so before attempting to access the radio.

If you are using a shared basestation then you need to set the properties `"spot.basestation.sharing"` to `"true"` and `"SERIAL_PORT"` to `"dummyport"`. You also need to launch the shared basestation manager (via `"ant start-shared-basestation"`) before starting your application.

If your application is using a dedicated basestation, then set the properties `"spot.basestation.sharing"` to `"false"` and `"SERIAL_PORT"` to the port that the basestation is connected to, e.g. `COM2` (Windows) or `/dev/cu.usbmodem1d11` (Mac).

You also need to make sure the application's classpath includes the following files from your SDK/lib directory: `multihop_common.jar`, `spotlib_host.jar`, `spotlib_common.jar`, `squawk_common.jar`, `RXTXcomm.jar`, `spotclient_host.jar` & `signing_host.jar`. This can be done either on the command line used to launch your application or inside your application's manifest file.

Finally you need to tell Java how to find the RXTX native library: `librxtxSerial.jnilib` (Mac) or `rxtxSerial.dll` (Windows). This can be done by adding the path to your SDK/lib directory to the system property `"java.library.path"` or by placing a copy of the library in a folder that Java will look in (e.g. `/System/Library/Java/Extensions` on the Mac).

### ***A simple host application to communicate with a SPOT over USB***

Here is a very simple host application that will echo the output printed by a SPOT that is connected by USB to the host computer.

```
import com.sun.spot.client.DummySpotClientUI;
import com.sun.spot.client.SerialPortWrapper;
import java.io.*;

public class echo {

    public static void main(String[] args) {
        try {
            String port = "COM2";
            SerialPortWrapper pw =
                new SerialPortWrapper(port, new DummySpotClientUI(true));
            InputStream in =
                new SerialPortWrapper.NoTimeoutInputStream(pw.getInputStream());
            // in case we need to send characters to the SPOT
            OutputStream out = pw.getOutputStream();
            while (true) {
                int b = in.read();
                System.out.print((char) b);
            }
        } catch (IOException ex) {
            System.err.println("Error reading from USB connection: " + ex);
        }
    }
}
```

With this as a starting point it should be easy to make changes to write the data to a file, add timestamps, synchronize with the SPOT app, send text messages, etc. Note that since this host application does not use the basestation, in the project's `build.properties` file set the property `"basestation.not.required=true"`.

## ***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.

### **Network information**

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

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

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

```
[NetManagementServer] starting on port 20
```

If you have a number of SPOTs running the net management server you can issue netinfo requests like this:

```
ant netinfo -DremoteId=xxx [ -Dcommand=yyyy]
```

`remoteId` is required. Omitting the command argument causes all commands to be executed. The supported commands are:

- `stats`: retrieves the LowPanStats object from a remote node
- `routes`: retrieves the current routing table from the remote node
- `tracert`: traces the current route to the specified SPOT.

Please note that the `ant netinfo` command will only succeed if there is a basestation attached.

### **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 basestation, and operates on USB-supplied power. Apart from the lack of battery and sensor board the basestation 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 that 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.

### ***Batch operations***

When working with large numbers of SPOTs, software updating can be a lengthy process. For this reason, the properties `remoteId` and `port` may be specified as comma-separated lists and operations will then be applied to multiple SPOTs.

If two SPOTs are connected locally, then for example:

```
ant -Dport="COM3,COM4" deploy
```

And if they are accessible remotely via a basestation then

```
ant -DremoteId="0014.4F01.0000.0006,0014.4F01.0000.0007"
```

If the SPOTs are connected locally, or if a shared basestation is in use, the multiple operations will proceed in parallel. For remote SPOTs when the basestation is not shared, the operations happen in sequence.

This parallel operation speeds up the process, but be aware that the output from the processes will be interspersed, which may make it hard to interpret if errors occur. Furthermore, individual errors no longer halt the ant script, and so multiple target commands like `ant deploy run` are not recommended as the `run` target will execute regardless of whether the `deploy` succeeded.

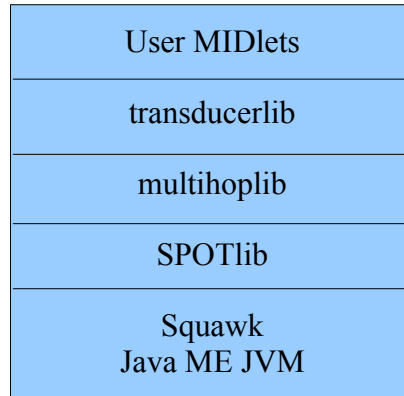
## **Developing and debugging Sun SPOT applications**

### ***Understanding the SPOT system architecture***

In order to write applications for Sun SPOTs it is first important to understand the SPOT system architecture.

#### ***Free-range SPOT architecture***

Here is a traditional stack diagram for the software running on a free-range SPOT:



**Free-range SPOT software stack**

At the top is the user-written SPOT application, which extends the Java ME MIDlet class.

At the bottom is the Squawk JVM. There is no operating system; Squawk runs on the bare metal.

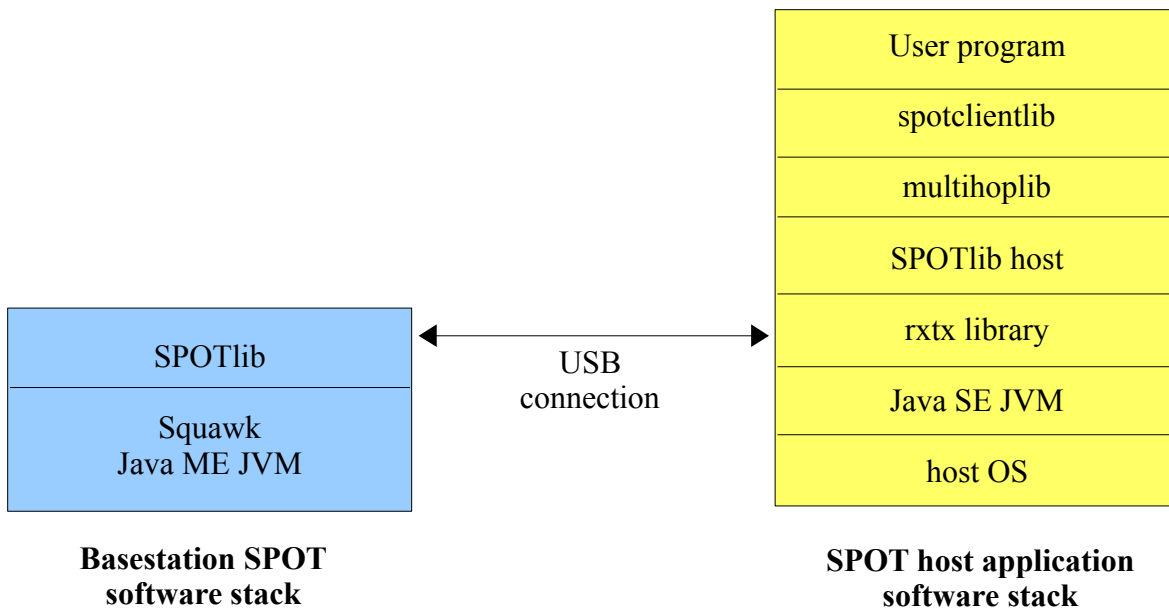
In between are the various SPOT libraries. Access to the SPOT device and basic I/O is provided by SPOTlib. This includes access to the low-level MAC radio protocol.

The multihoplib provides higher-level radio protocols, such as Radiogram and Radiostream, and also takes care of the routing of packets to SPOTs not in direct contact with this one.

The transducerlib provides a way to access the hardware on the SPOT eDemo sensor board, e.g. the accelerometer, LEDs, switches, digital I/O, analog inputs, etc.

***SPOT host application architecture***

Here is the stack diagram for a SPOT host application and how it connects to a SPOT basestation:



Again at the top is the user-written host SPOT application, which is just a regular Java SE program. It can do everything a regular Java application can do: file I/O, display Swing GUI's, etc. It can also send and receive radio messages with free-range SPOTs if a basestation is connected to the host computer.

At the bottom is the host operating system: Linux, Windows, Mac OS X or Solaris.

Just above the OS is a Java SE JVM along with all the regular Java class libraries.

In between are the various SPOT libraries. Access to the SPOT device and basic I/O is provided by the host version of SPOTlib. This includes access to the low-level MAC radio protocol, which either uses the USB connection to access the basestation's radio, or uses socket connections to communicate with other host applications.

The multihoplib again provides higher-level radio protocols, such as Radiogram and Radiostream, and also takes care of routing of packets.

The spotclientlib gives access to a number of OTA (over-the-air) commands that can be sent to a free-range SPOT. This includes the "Hello" command used to discover SPOTs within radio range.

The rtx library is used to perform serial I/O over the USB connection to the basestation.

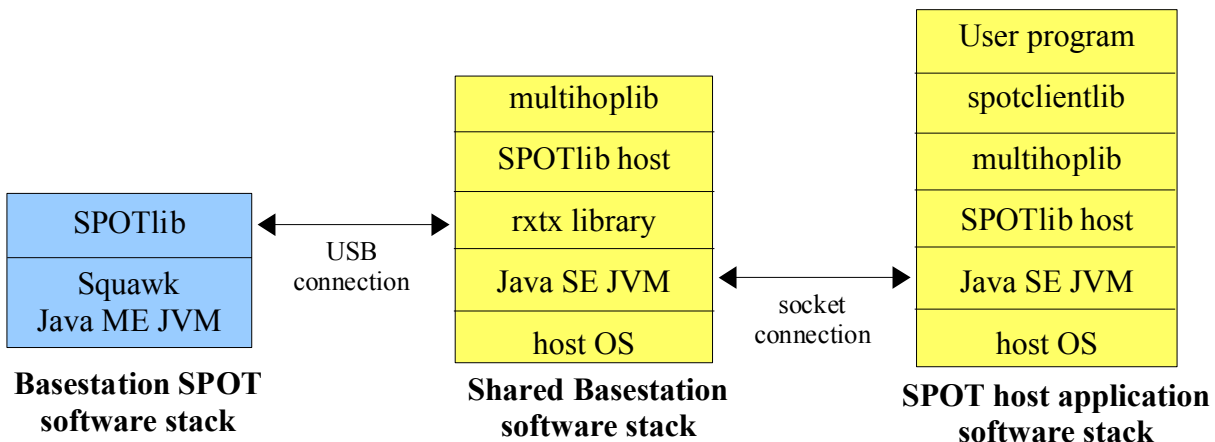
Note that Solarium and all of the SPOT SDK ant commands are SPOT host applications.

### ***SPOT basestation architecture***

The basestation provides a way for host applications to communicate with free-range SPOTs by using the basestation's radio.

Note that the host application runs solely on the host computer; no user code runs on the basestation. Packets to send are sent via USB to the basestation, which then sends them out over its radio. Likewise when the basestation receives a radio packet it forwards it to the host application.

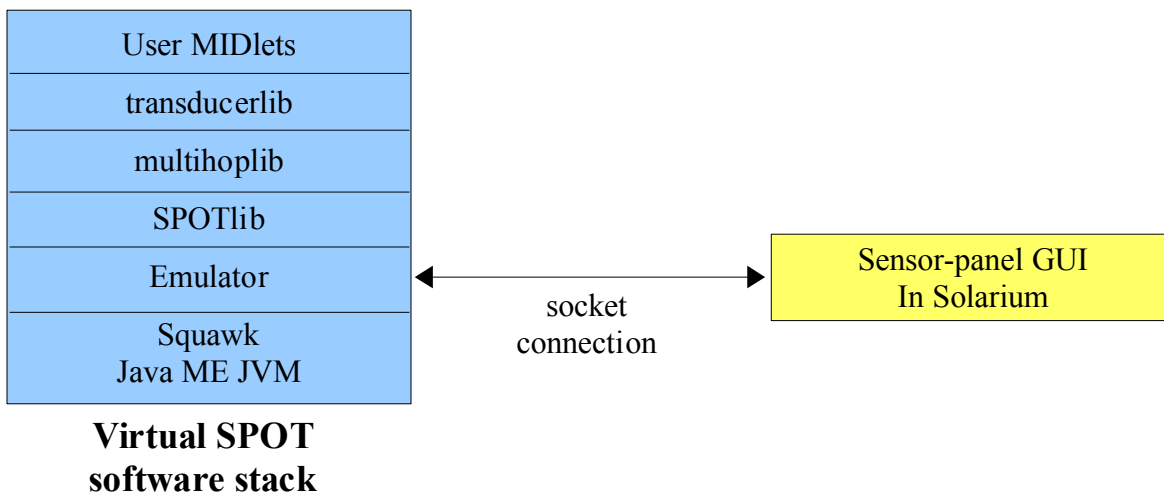
When using a shared basestation there is no user program deployed to the SPOT, instead the Basestation class in the SPOT library is run. "Radio" packets received from other host applications (via a socket connection) are relayed to free-range SPOTs using the basestation's radio.



### Virtual SPOT architecture

When you create a new virtual SPOT in Solarium, a new process is started to run the emulator code in a Squawk VM. The emulator code communicates over a socket connection with the virtual SPOT GUI code in Solarium. For example when the SPOT application changes the RGB value of an LED that information is passed to the virtual SPOT GUI code that updates the display for that LED with the new RGB value. Likewise when the user clicks one of the virtual SPOT's switches using the mouse, Solarium sends a message to the emulator code that the switch has been clicked, which can then be noticed by the SPOT application.

Here's a stack diagram of the Emulator architecture:



Each virtual SPOT has its own Squawk VM running in a separate process on the host computer. Each Squawk VM contains a complete host-side radio stack as part of the SPOT library, which allows the SPOT application to communicate with other SPOT applications running on the host computer, such as other virtual SPOTs, using sockets or real SPOTs via radio if a shared basestation is running.

### 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 The MIDlet lifecycle.

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://labs.oracle.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.



## ***Isolates***

The Squawk VM supports multiple independent execution spaces, called *Isolates*. Each isolate comprises a separate set of threads plus the objects referenced from those threads.

When the SPOT is started the first isolate runs threads that support system services as described below. Each user SPOT applications is run in its own isolate.

## ***Threads***

Developers should be aware that a number of background threads will be running in the initial Isolate 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.

### ***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 recognized.

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

## ***The Sun SPOT device libraries***

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 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:

<u>Device</u>	<u>Interface</u>
LED	<code>ILed</code>
PIO	<code>IAT91_PIO</code>
AIC	<code>IAT91_AIC</code>
Timer-Counter	<code>IAT91_TC</code>
CC2420	<code>I802_15_4_PHY</code>
MAC layer	<code>I802_15_4_MAC</code>
RadioPolicyManager	<code>IRadioPolicyManager</code>
SPI	<code>ISpiMaster</code>
Flash memory	<code>IFlashMemoryDevice</code>
Power controller	<code>IPowerController</code>
OTA Command Server	<code>OTACommandServer</code> (class)
Attention button handler	<code>FiqInterruptDaemon</code> (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 *The MIDlet lifecycle* 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 optimize 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 `VM.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. See also the section Memory usage for more information about the layout and usage of the flash memory.

### **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.

### **Using FlashFile**

Using the Record Management Store is the recommended method for accessing flash memory, because it abstracts away concerns about the physical layout of the flash memory and allocating and freeing space. However, in some cases, it may be necessary to use a lower-level mechanism. To allocate and use an arbitrary area of raw flash memory, see `com.sun.squawk.flashmanagement.FlashFile`. To read and write specific sectors, see `com.sun.squawk.flashmanagement.FlashFileOutputStream` and `FlashFileInputStream`.

## **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. The rev 8 SPOT has four separate serial lines available:

- one from the sensor board: `edemoserial://` available on pins D0 & D1 on the sensor board,
- and three from the main processor board:
  - `serial://usb` & `serial://` which both go to the USB connector and to pads on the processor board<sup>5</sup>
  - `serial://usart` & `serial://usart0` which both go to the pins RX & TX on the sensor board,<sup>6</sup>
  - `serial://usart1` which is available through the top connector with a suitable adaptor.<sup>7</sup>

The older rev 6 SPOT has only two serial lines available from the main processor board; it does not have `usart1`. Also for the rev6 SPOT `usart0` does not go to any pins on the rev6 sensor board, but is only available through the top connector with a suitable adaptor.<sup>8</sup>

Here is how to open input and output streams over the USB connection:

```
// 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());
```

---

<sup>5</sup>USART1 in the AT91SAM9G20 package.

<sup>6</sup>USART0 in the AT91SAM9G20 package.

<sup>7</sup>USART2 in the AT91SAM9G20 package.

<sup>8</sup>USART0 in the AT91RM9200 package.

The same mechanism can be used to read to and from the USART by replacing the URL "serial://usb" with "serial://usart", "serial://usart0" or "serial://usart1". Note that usart and usart0 are the same. An additional URL "serial://" is also supported that reads and writes on USB when USB is connected and in use by a host application, and otherwise to the USART pads on the main processor board.

Only one input stream may be opened to each device, and if "serial://" is opened for input then "serial://usb" cannot 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).

### ***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 specify 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  
currentIsolate.addErr("serial://usb"); // 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***

The classes for the radio stack above the Mac layer are in `multihoplib_common.jar`, and the corresponding sources can be found in `multihop_common_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<sup>9</sup>.

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("radiostream:<destAddr>:<portNo>");
```

where `destAddr` is the 64bit IEEE Address of the radio at the far end, and `portNo` is a port number in the range 0 to 255 that identifies this particular connection<sup>10</sup>. Note that 0 is not a valid IEEE address in this implementation. The connection is opened using the default radio channel and default PAN identifier (currently channel 26, PAN 3). The section Radio properties shows how to override these defaults.

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

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

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

Here's a complete example:

#### **Program 1**

```
    RadiostreamConnection conn =  
        (RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0006:100");  
    DataInputStream dis = conn.openDataInputStream();  
    DataOutputStream dos = conn.openDataOutputStream();
```

<sup>9</sup> One aspect of the current behavior 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 optimization for single hop connections.

<sup>10</sup> Ports in the range 0 to 31 are reserved for system use.

```

try {
    dos.writeUTF("Hello up there");
    dos.flush();
    System.out.println ("Answer was: " + dis.readUTF());
} catch (NoRouteException e) {
    System.out.println ("No route to 0014.4F01.0000.0006");
} finally {
    dis.close();
    dos.close();
    conn.close();
}

```

## Program 2

```

RadiostreamConnection conn =
    (RadiostreamConnection)Connector.open("radiostream://0014.4F01.0000.0007:100");
DataInputStream dis = conn.openDataInputStream();
DataOutputStream dos = conn.openDataOutputStream();
try {
    String question = dis.readUTF();
    if (question.equals("Hello up there")) {
        dos.writeUTF("Hello down there");
    } else {
        dos.writeUTF("What???");
    }
    dos.flush();
} catch (NoRouteException e) {
    System.out.println ("No route to 0014.4F01.0000.0007");
} finally {
    dis.close();
    dos.close();
    conn.close();
}

```

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

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

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

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

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

## The radiogram protocol

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

To open a server connection do:

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

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

To open a client connection do:

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

where `serverAddr` is the 64bit IEEE Address of the radio of the server, and `portNo` is a port number in the range 0 to 255 that identifies this particular connection<sup>11</sup>. Note that 0 is not a valid IEEE address in this implementation. The port number must match the port number used by the server.

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

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

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

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

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

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

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

Here's a complete example:

### Client end

```
RadiogramConnection conn =  
    (RadiogramConnection) Connector.open("radiogram://0014.4F01.0000.0006:100");  
Datagram dg = conn.newDatagram(conn.getMaximumLength());  
try {  
    dg.writeUTF("Hello up there");  
    conn.send(dg);  
    conn.receive(dg);  
    System.out.println("Received: " + dg.readUTF());  
} catch (NoRouteException e) {  
    System.out.println("No route to 0014.4F01.0000.0006");  
} finally {  
    conn.close();  
}
```

### Server end

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

---

<sup>11</sup> Ports in the range 0 to 31 are reserved for system use.



```

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");

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

RadioInputStream is = (RadioInputStream)conn.openInputStream();

```

```
byte port = conn.getLocalPort(); // now port can be accessed from 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:

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

### ***Broadcasting***

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

To perform broadcasting first open a special radiogram connection:

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

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

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

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

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

where `n` is the required number of hops.

### ***Broadcasting and basestations***

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

In dedicated mode, the host application uses the basestation's address and therefore receives this radiogram. In shared mode, the host application is still one more hop from the host process managing the basestation, and so does not receive the radiogram. Similar considerations apply to packets sent from host applications. For this reason, application developers using broadcast packets to communicate between remote SPOTs and host applications will need to consider the impact of basestation mode on the number of broadcast hops they set.

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

### **Port number usage**

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

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

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

### **Radio properties**

#### **Changing channel, PAN identifier and output power**

The `RadioPolicyManager` provides operations for changing the radio channel, the PAN identifier 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 identifier or power changes it for all connections.
- If the radio is already turned on, changing the channel or PAN identifier 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`.
- While the range of power values specified in the 802.15.4 specification is -32 to +31, the CC2420 radio chip used in the Sun SPOT only has 22 different power levels, the maximum one being 0. That's why you can set the transmit power to one value and then read back another. Also note that for channel 26 the maximum power is -3, not 0 due to FCC regulations

### ***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 system properties to adjust the radio***

The initial values of the channel, PAN identifier and transmit power can be specified using persistent system properties stored on the Sun SPOT. The properties are:

```
radio.channel  
radio.pan.id  
radio.transmit.power
```

When specified in your `.sunspot.properties` file, the project's `build.properties` file or on the command line, they will become the new default values used by a SPOT after the next `set-radio-properties` command is performed. They also apply to any host apps that use the basestation. Use `ant delete-radio-properties` to have a SPOT go back to the standard radio defaults.<sup>12</sup>

### ***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.
- **AUTOMATIC**: 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 `AUTOMATIC` 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.

---

<sup>12</sup>The old ant properties `remote.channel` & `remote.pan.id`, plus the old manifest properties `DefaultChannelNumber`, `DefaultPanId` & `DefaultTransmitPower`, will all still work for now, though they will be overruled by the `radio.*` properties if they are set.

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 `AUTOMATIC` 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 `AUTOMATIC` policy. Note that if you close the dummy connection, then the radio will turn back on, in line with the OTA command server's `AUTOMATIC` 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. The Basestation toggles the green and red processor board LEDs whenever a radio packet is received or sent. A SPOT application can do the same by setting the system property "radio.traffic.show.leds" to true and rebooting the SPOT.

Two sets of programmatic monitoring 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.getShortPacket());
```

## ***Routing policies***

When communications between Sun SPOTs are routed over more than one hop, a Link Quality Routing Protocol (LQRP) algorithm is used to determine the best route<sup>13</sup>. In this algorithm, requests for a route to a particular target SPOT (RREQs) are broadcast by a requester, and then re-broadcast by each Sun SPOT that receives them. Each Sun SPOT that knows how to route to the requested target then sends a reply back to the requester. The route that will be used is the one with the best link quality.

Each Sun SPOT can exercise some control over the part it plays in this routing through its `RoutingPolicyManager`, which implements `IRoutingPolicyManager`. The policy can be manipulated both through system properties and programmatically.

The four policies available are:

- **ALWAYS**: The Sun SPOT operates as a full routing participant. It will respond to and route RREQ and pass packets for other Sun SPOTs. Deep sleep is disabled, to ensure that the Sun SPOT will fulfill the routing role at all times.
- **IFAWAKE**: The Sun SPOT operates as a full routing participant. It will respond to and route RREQ and pass packets for other Sun SPOTs. Deep sleep is not specifically handled, so if the app otherwise permits deep sleep, the Sun SPOT may unexpectedly disappear from the network.
- **ENDNODE**: The Sun SPOT will generate RREQs when opening a connection. It will respond to them if it is the final destination. It will not repeat RREQs to others or process packets for Sun SPOTs unless it is either the ultimate sender or recipient.
- **SHAREDBASESTATION**: this policy is currently unimplemented, and behaves like ALWAYS.

The persistent property `spot.mesh.routing.enable` can be set to have the values ALWAYS, ENDNODE and IFAWAKE. It can also be set to have alternative values that controls the default policy as follows:

<b>Property value</b>	<b>Default policy</b>
True	ALWAYS
false	ENDNODE
<not set>	IFAWAKE

The policy can also be set programmatically at runtime. For example:

```
RoutingPolicy rp = new RoutingPolicy(RoutingPolicy.IFAWAKE);
RoutingPolicyManager.getInstance().policyHasChanged(rp);
```

---

<sup>13</sup>Previous versions of the SPOT radio stack used AODV (Ad-hoc On-demand Distance Vector) for computing routes.

### ***MAC-layer filtering***

Starting with the Red release one can specify MAC-layer filtering to constrain what packets are seen by the higher layers of the radio stack.

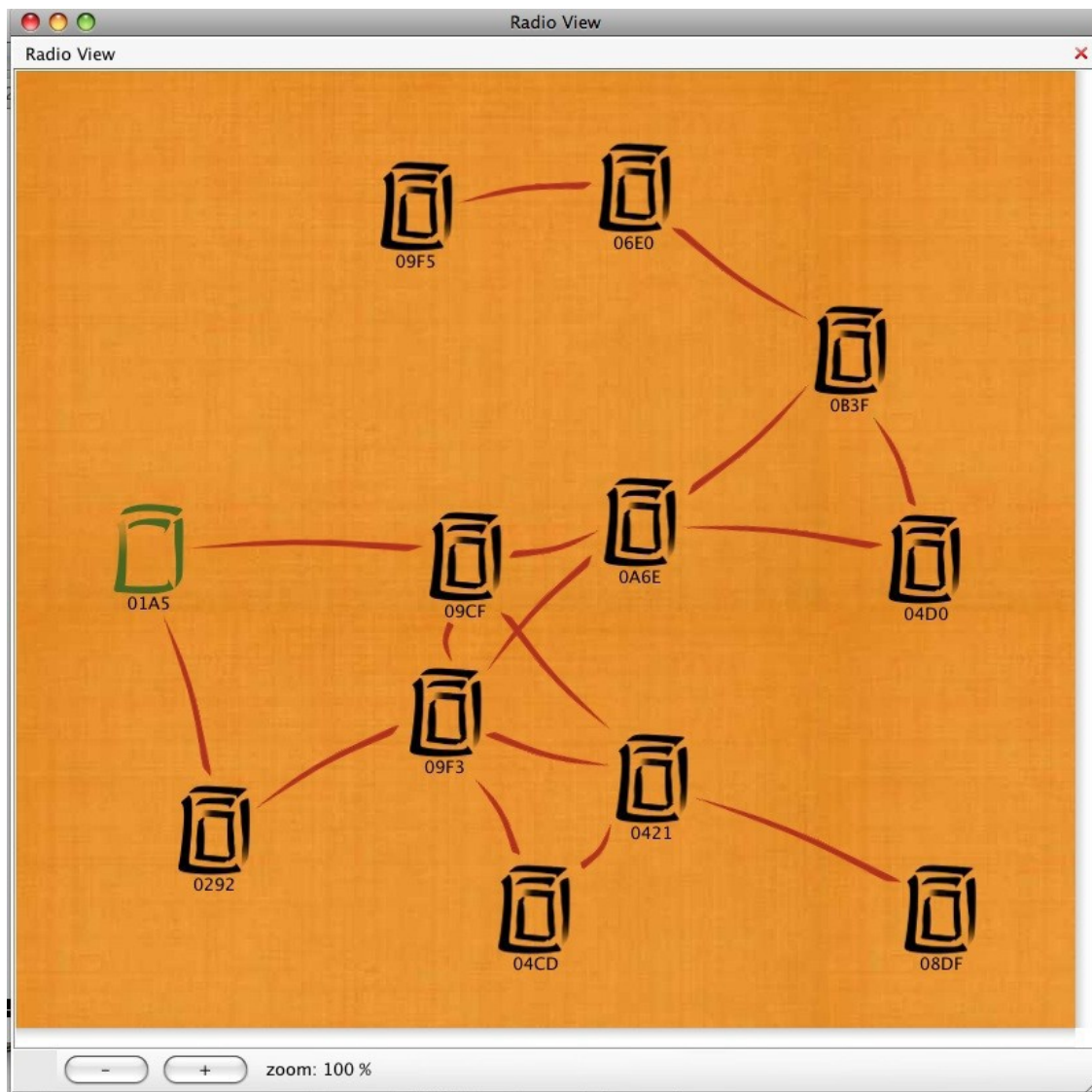
To make it easier to control the radio connectivity of a group of SPOTs, the MAC layer can now be set to throw away any packets from senders not on a white list or those on a black list. Actual routing is still done by the routing manager.

To use it set the SPOT system property "radio.filter" to true and "radio.whitelist" to a comma separated list of radio addresses, where only the low part of the addresses need to be specified, e.g. "2DFE,1234" will only receive packets from senders with an address of 0014.4F01.0000.2DFE or 0014.4F01.0000.1234. If the "radio.whitelist" property is not defined then the "radio.blacklist" property will be used to specify a list of SPOTs to ignore packets from. The properties are read when the SPOT starts up.

Note: these SPOT properties must be set before SPOT is started up; changing the properties requires a restart. Also it can cause confusion when later trying to figure out why 2 SPOTs cannot communicate.

### ***Visualizing the radio topology***

Solarium has a Radio View to show the radio connectivity between SPOTs. Here's a sample Radio View for a network of Sun SPOTs located in Sun Labs.



## Conserving power

### Turning off unused devices

By default, the SPOT's USB device controller and two of its USARTs are always enabled. A small amount of power can be saved by disabling these if they are not required. Control is via operations on `IAT91_PowerManager`. To turn off the USB support:

```
Spot.getInstance().getAT91_PowerManager().setUsbEnable(false);
```

Turning off the USB saves about 1.5mA (measured as battery draw current).

To turn off the USART support:

```
Spot.getInstance().getAT91_PowerManager().setUsartEnable(false);
```

Turning off the USARTs saves about 0.3mA (measured as battery draw current).

Note that support for USB and USARTs is not reenabled when the VM restarts; if the VM exits with support disabled it will be necessary to turn off the SPOT and turn it back on. For this reason it is best



to enclose statements that turn off these devices in a try/finally block, where the finally section turns them back on.

### **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 stopped (the CPU's power saving mode)
- Master system clocks – on
- Low-level firmware – power on
- RAM – power on but inactive
- Flash memory – power on but inactive
- CC2420 radio – power on
- AT91 peripheral clocks – on if necessary
- External board – 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 additional latency. That is, as soon as any thread becomes ready to run, the Sun SPOT wakes and carries on.

### **Reducing power consumption in shallow sleep**

Switching to a slower system clock speed during the sleep period can reduce power consumption during shallow sleep. In shallow sleep the SPOT is waiting for either an interrupt or time to pass. The problem is that the slower the clock, the bigger the interrupt latency<sup>14</sup>, and big latency can be an issue for some drivers. By default the clocks are unchanged during shallow sleep, so as to minimize latency. An operation on `IAT91_PowerManager` allows applications to specify a lower speed if they wish. *Note: currently this feature is disabled since it causes the SPOT to crash.*

The default speed for the Master Clock (MCK), the clock that controls the speed of the bus and the peripheral clocks, 60MHz.. The speeds you can choose from for shallow sleep, together with the typical latency (system idle) and battery power draw (no external board, radio off) are:

MCK in MHz	Typ. latency ( s )	Current (mA)
60.0	81	22.8
45.0	89	19.6
18.432	265	12.0
9.216	574	10.0

You will need to experiment to see what speed best suits your application. Even at 9MHz a SPOT is perfectly capable of waking from deep sleep to answer a radio request provided it is in a quiet radio environment (it takes about 4ms for the CC2420 radio chip to fill its buffer).

There is no point in taking the overhead of switching clocks if the sleep is very short. If the sleep duration is less than 20ms the clock is not adjusted.

---

<sup>14</sup> Latency is the time between the hardware requesting an interrupt and the driver beginning to service it.

There are also some caveats. Because the MCK is lower than normal, any Timer/Counters running during a shallow sleep won't keep to time. Similarly, the USART baud rates are governed by MCK, so the USARTs won't be able to receive characters during shallow sleep if you use this feature.

### **Deep Sleep**

The Sun SPOT can be programmed 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 – 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 peripheral clocks – off
- External board – main power off but standby supply available

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.

### **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.

### **Deep sleep and external batteries**

Connecting an external power source to the Sun SPOT via the USB socket or via VEXT causes different behavior for different versions of the Sun SPOT hardware. For rev 6 and earlier Sun SPOTs, this connection will inhibit true deep sleep, whereas for rev 7 and later Sun SPOTs, deep sleep will still happen.

For rev 7 and later Sun SPOTs, an external power supply does not inhibit deep sleep. The current draw in deep sleep with an external power supply will still be higher than deep sleep without the supply, because the external power supply must drive part of the power supply circuitry. Roughly speaking, battery-powered deep sleep draws around  $33\mu\text{ A}$ , whereas externally powered deep sleep draws around  $6\text{mA}$  if powered from VEXT and  $7\text{mA}$  if powered from VUSB. These figures will increase if a battery is connected that requires charging.

For rev 6 and earlier Sun SPOTs, the external power supply will inhibit true deep sleep. The Sun SPOT software will behave as if it is deep sleeping, but in fact the processor will continue to be powered but held in reset. In this mode the power draw will be either  $17\text{mA}$  from VEXT or  $25\text{mA}$  from VUSB.

There is an experimental feature allows the rev 6 Sun SPOT to enter a special low power mode:

- CPU – power on but CPU clock stopped (the CPU's power saving mode)
- Master system clocks – off (32kHz slow clock selected)
- Low-level firmware – power on
- RAM – power on but inactive
- Flash memory – power on but inactive
- CC2420 radio – power on but inactive
- AT91 peripheral clocks – off
- External board – power on

In this mode the Sun SPOT will draw around  $6.4\text{ mA}$  (as seen at the battery, no external board). *Note: currently this feature is disabled since it causes the SPOT to crash.*

### **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 behavior of the standard drivers**

<u>Device</u>	<u>Condition to permit deep sleep</u>
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 `AUTOMATIC` 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 conforming driver can be found in the class `com.sun.spot.peripheral.Led`. Note that this class is not public and hence is 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`.

### ***Debugging deep sleep problems***

If it is important that your application knows whether or not deep sleep happened, then you can 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.

Another useful feature is the ability to ask the sleep manager for the number of times deep sleep has occurred:

```
ISleepManager sleepManager = Spot.getInstance().getSleepManager();
int deepSleepCount = sleepManager.getDeepSleepCount();
```

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 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. This is also useful if you just want to see which drivers are being called.

## ***IPv6 Support***

### ***About the IPv6 Stack***

The classes for the radio stack above the Mac layer are in `ipv6lib_common.jar`, and the corresponding sources can be found in `ipv6lib_common_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 access to the IPv6 UDP and TCP protocols in a manner that is transparent to the underlying layers.

The SunSPOT SDK Demos directory contains a set of applications that provide examples of using the IPv6 network stack.

The `udp` protocol provides datagram-based communication between two devices. This protocol provides no guarantees about delivery or ordering. Datagrams could be silently lost, be delivered more than once, and be delivered out of sequence.

The `tcp` protocol provides reliable, buffered, stream-based communication between two devices. By definition, packet loss in a TCP connection is interpreted as congestion, resulting in a back-off in the rate of packet transmission. As a result, the occasional packet losses associated with low power, lossy radio links may cause a degradation in overall throughput of a connection.

### ***Addressing***

There are some important differences between using the IPv6 based services and the SPOT protocols.

Node addressing must use IPv6 format addresses. At this time, link local addressing (addresses beginning with fe80::) is supported. Globally routable IPv6 addresses are supported when using the IPv6FreenetRouter demo application provided with the SDK.

Addresses must be written using the standard for encoding IPv6 addresses in URLs (RFC2732) which specifies enclosing the address in square brackets. For example, a GCF url with an IPv6 address and port might look like:

```
udp://[fe80:0:0:0:214:4f01:0:7e0]:3000
```

For easier use with SunSPOTs, IEEE Addresses may be utilized in place of an RFC2732 encoded address. When using a SunSPOT style IEEE Address, the udp and tcp protocol handlers will convert these 64-bit addresses to properly formatted 128-bit IPv6 link local addresses. For example:

```
udp://0014.4f01.0000.7e0:3000
```

would be treated internally as the equivalent of:

```
udp://[fe80:0:0:0:214:4f01:0:7e0]:3000
```

Link local addresses begin with the prefix 'fe80'. In general, a SunSPOT with the address: 0014.4F01.0000.FACE would be converted to:

```
fe80::0214:4f01:0000:face
```

which may be abbreviated as:

```
fe80::214:4f01:0:face
```

Note that the first byte of the IEEE Address portion of the is changed from 00 to 02. The explanation for this can be found in section 2.5.1 of RFC 2373 and is described nicely in this blog post: <http://packetlife.net/blog/2008/aug/04/eui-64-ipv6/>

### **UDP support**

The *udp* protocol handler is much like the radiogram protocol handler in usage. Once a udp connection is opened, datagrams can be sent and received over the connection.

The following code fragments demonstrate the basic usage of the UDP client and server functionality. A complete example can be found in the SDK Demos directory.

#### **UDP Server example**

```
int PORT_NUMBER = 8888;
UDPConnection conn = null;
UDPDatagram dg, rdg = null;
byte connected[];

String request = null;
try {
    conn = (UDPConnection) Connector.open("udp://:" + PORT_NUMBER);
} catch (IOException ex) {
    ex.printStackTrace();
}
while (true) {
    dg = (UDPDatagram) conn.newDatagram(conn.getMaximumLength());
    dg.reset();
    try {
        conn.receive(dg);
        connected = dg.getSource();
    }
```

```

        System.out.println(dg.readUTF());
        rdg = (UDPDatagram) conn.newDatagram(conn.getMaximumLength());
        rdg.setDstAddress(connected);
        rdg.setDstPort(dg.getSrcPort());
        rdg.writeUTF("Our Response");

        conn.send(rdg);

    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

### **UDP Client example**

```

UDPConnection sendConn;
String ipv6Target = "fe80:0:0:0:214:4f01:0:7e0";

Datagram dg = null;
try {
    String url = "udp://[" + ipv6Target + "]:" + PORT_NUMBER;

    sendConn = (UDPConnection) Connector.open(url);
    dg = sendConn.newDatagram(sendConn.getMaximumLength());
    // Write the command into the datagram
    dg.writeUTF("Test String");
    // Send the command to the server
    sendConn.send(dg);
    // Wait for a response
    sendConn.receive(dg);

    // Extract the contents
    System.out.println(dg.readUTF());
    sendConn.close();
} catch (Exception e) {
    System.out.println("[APP] Unable to contact host: " + ipv6Target);
    e.printStackTrace();
}

```

### **TCP support**

When using TCP streams, the server side must be started first. At this time, the TCP server side only supports a single connection at a time. “Server socket” implementation, allowing for multiple simultaneous connections has not been completed at this time.

Client applications may receive an error when attempting to contact a TCP server that has not been started prior to running the client, just as they do in a traditional wired network. Depending on how the client side has been architected, a "Connection Refused" message may be printed. TCP clients may be coded to recover from this error and retry the connection until the server becomes available.

In general, it's good practice to run the server before the client application.

### **TCP Server example**

```

TCPConnection conn = null;

while (true) {
    try {
        conn = (TCPConnection) Connector.open("tcp://:" + PORT_NUMBER);
    } catch (IOException ex) {

```

```

        ex.printStackTrace();
        continue;
    }

    DataInputStream dis = null;
    DataOutputStream dos = null;

    try {
        dis = conn.openDataInputStream();
        dos = conn.openDataOutputStream();

        // Do the real work here
        dos.UTF("Test String");
        // Flush our output
        dos.flush();

        conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

### ***TCP Client example***

```

TCPConnection sendConn;
DataInputStream tin = null;
String ipv6Target = "fe80:0:0:0:214:4f01:0:7e0";

try {
    String url = "tcp://[" + ipv6Target + "]:" + PORT_NUMBER;

    sendConn = (TCPConnection) Connector.open(url);
    sendConn.setTimeout(3000);

    tin = sendConn.openDataInputStream();

    // do the real work here
    System.out.println(tin.readUTF());

    tin.close();
    sendConn.close();

} catch (Exception e) {
    e.printStackTrace();
}

```

### ***Demos***

The SDK Demo's directory contains a set of example programs that use the IPv6 implementation. The demos included here are:



IPSendDataDemo	This is a direct port of the SendDataDemo, using UDP broadcasts in place of radiogram broadcasts.
TCPStreamDemo	A pair of programs that act as client and server for a TCP connection. The server program should be started first.
UDPSendDataDemo	A sample client/server application using UDP transport and unicast addresses.
IPv6FreenetRouter	A sample application for creating a tunnel from the SunSPOT IPv6 network to IPv6 Internet hosts.

### **IPv6FreenetRouter**

This is a host application that make the Sun SPOT base station an IPv6 router for connected Sun SPOTs. The host application created a UDP tunnel to the Freenet6 service to obtain a globally routable IPv6 network prefix.

Use of this application requires a Freenet6 username and password. Freenet6 access is offered by GoGo6.com. After registering with GoGo6 at <http://gogonet.gogo6.com/> you can request Freenet6 credentials through the website.

The credentials used by this host application are the Freenet6 credentials, not your GoGo6 login and password. As single set of credentials may only be used to create one tunnel/router at any time. If multiple IPv6 prefixes are required simultaneously, separate credentials must be obtained.

The build.properties in the top level directory contains the basic information required to create the tunnel. To create your personal tunnel, replace the hostname, username and password with your assigned tunnel server host (i.e. montreal.freenet6.net), your userid and password.

To start the tunnel, from the Demos/IPv6FreenetRouter directory:

```
ant host-run
```

Once the tunnel has started, any free range SunSPOTs operating on the same channel and pan ID should pick up the advertised global routing prefix.

### **IPv6 Debugging**

The following properties control debugging output for various modules of the SunSPOT IPv6 stack. By default, all properties are set to *false*. Debugging is enabled by setting these values to *true*.

spot.ip.debug.icmp	Debug basic ICMP message exchanges
spot.ip.debug.ipv6	Debug core IPv6 packet handling
spot.ip.debug.nd	Debug/Decode Neighbor & Router Discovery messages
spot.ip.debug.routing	Debug IPv6 Routing lookups
spot.ip.debug.tcp	Debug TCP packet handling
spot.ip.debug.udp	Debug UDP packet handling
spot.ip.debug.ulowpan	Debug uncompressed lowpan packet handling

For example:

```
ant set-system-property -Dkey=spot.ip.debug.ulowpan -Dvalue=true
```

## ***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 basestation connected to a host running the Socket Proxy. This proxy program is responsible for communicating with the server specified in the URL. The Socket Proxy program is started by

```
ant socket-proxy
```

or

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

## ***Configuring the http protocol***

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

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

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

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

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

## **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:

<i>basestation's serial port</i>	The serial port that the basestation is connected to. This is automatically populated by the calling scripts.
<i>Radiogram port</i>	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. <ol style="list-style-type: none"><li>1. System: log all that is related to the proxy itself</li><li>2. Error: log all that is an error (exceptions)</li><li>3. Warning: log all that is a warning</li><li>4. Info: log all extra information (incoming connections, closing connections)</li><li>5. I/O: log all I/O activities</li></ol>
<i>Radio port usage</i>	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/transducer_device.jar
SDK_INSTALL_DIRECTORY/lib/ipv6lib_common.jar
SDK_INSTALL_DIRECTORY/lib/multihop_common.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_device.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
```

```
SDK_INSTALL_DIRECTORY/lib/squawk_device.jar
```

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

```
SDK_INSTALL_DIRECTORY/lib/ipv6lib_common.jar
SDK_INSTALL_DIRECTORY/lib/multihop_common.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_host.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
SDK_INSTALL_DIRECTORY/lib/squawk_common.jar
SDK_INSTALL_DIRECTORY/lib/RXTXcomm.jar
SDK_INSTALL_DIRECTORY/lib/spotclient_host.jar
SDK_INSTALL_DIRECTORY/lib/signing_host.jar
```

### **Javadoc/source configuration**

Source is included for these jars:

```
transducer_device.jar    SDK_INSTALL_DIRECTORY/src/transducer_device_source.jar
multihop_common.jar      SDK_INSTALL_DIRECTORY/src/multihop_common_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 the first four above and for `squawk_device.jar` can be found in:

```
SDK_INSTALL_DIRECTORY/doc/javadoc
```

Javadoc for `spotlib_host.jar` and the host-appropriate subset of `spotlib_common.jar` can be found in:

```
SDK_INSTALL_DIRECTORY/doc/hostjavadoc
```

### **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 basestation.

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 basestation, between the debug client and the debug agent.

To use the high-level debugger do the following:

1. Set up a standard Sun SPOT basestation.
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
```

where `xxxx` is the id of the SPOT running the application being debugged. Note that `xxxx` can be a short name, as described in the `Remote` section.

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:\spotbuild\Sandbox>ant debug -DremoteId=0014.4F01.0000.0D27
Buildfile: build.xml

...

-collect-spotsselector-result:
    [echo]
    [echo] Using Sun SPOT basestation on port COM45

...

-do-debug-proxy-run:
    [java] Starting echo...
    [java] Quit called: ignoring command.
    [java] Echo running, starting proxy...

    [java] Trying to connect to VM on radiostream://0014.4F01.0000.0D27:9
    [java] Established connection to VM (handshake took 94ms)
    [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, PAN identifier or transmit power**

It is not possible for an application being debugged to select dynamically a different channel, PAN identifier or transmit power. Instead the required channel, PAN identifier or transmit power must be specified using persistent system properties (see *Using system properties to adjust the radio*). Then the required channel or PAN identifier can be specified like this:

```
ant debug -DremoteId=spot1 -Dradio.channel=11 -Dradio.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`.

There are several Sun SPOT-specific plugins for NetBeans 6.0 that aid in developing Sun SPOT applications. These are:

1. SunSPOTApplicationTemplate — for creating a new Sun SPOT Application NetBeans project
2. SunSPOTHostApplicationTemplate — for creating a new Sun SPOT Host Application NetBeans project
3. SpotRenameRefactoring — for automatically updating the manifest file for a Sun SPOT application when a MIDlet is renamed or added
4. Sun SPOT Info — the Sun SPOT 'welcome' and Tutorial module

To install these plugins into NetBeans first select `Tools > Plugins` and click on the `Settings` tab. Then push the `Add` button and for the Name enter “Sun SPOTs Update Center” and for the URL enter `http://www.sunspotworld.com/NB6/updates.xml` and then press `OK`. Next go to the `Available Plugins` tab, and select the plugins in the SunSPOTs category and press the `Install` button.

Finally, it will improve the debugging experience if you associate source code with the various Sun SPOT SDK jar files. This is done automatically when you create a new Sun SPOT Application or Sun SPOT Host Application NetBeans project using the above plugins. For projects that you create manually you need to modify your NetBeans `project.xml` file so that it reads in the various SPOT properties files. Add into the `<general-data>` section right after the `<name>` tags:

For SPOT apps:

```
<properties>
  <property-file>${user.home}/.sunspot.properties</property-file>
  <property-file>build.properties</property-file>
  <property-file>${sunspot.home}/default.properties</property-file>
</properties>
```

For SPOT host apps:

```
<properties>
  <property-file>${user.home}/.sunspot.properties</property-file>
  <property-file>build.properties</property-file>
  <property-file>${sunspot.home}/default.properties</property-file>
  <property-file>${netbeans.user}/build.properties</property-file>
</properties>
```

and then set up the classpath information using those properties. In the `<java-data>` section modify the `<compilation-unit>`:

For SPOT apps:

```
<compilation-unit>
  <classpath mode="boot">${sunspot.bootclasspath}</classpath>
```

```

    <classpath mode="compile">${sunspot.classpath}</classpath>
    <built-to>build</built-to>
    <source-level>1.4</source-level>
</compilation-unit>

```

For SPOT host apps:

```

<compilation-unit>
  <classpath mode="compile">${hostagent.compile.classpath}</classpath>
  <built-to>build</built-to>
  <source-level>1.5</source-level>
</compilation-unit>

```

Note: if your host application uses any additional jar files, such as the absolute or free-design layout managers, then include them in the classpath:

```

<classpath mode="compile">${hostagent.compile.classpath}:
    ${libs.absolutelayout.classpath}:
    ${libs.swing-layout.classpath}</classpath>

```

(Note: the above example is all one line.)

### **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/transducer_device.jar
SDK_INSTALL_DIRECTORY/lib/multihop_common.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_device.jar
SDK_INSTALL_DIRECTORY/lib/spotlib_common.jar
SDK_INSTALL_DIRECTORY/lib/squawk_device.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:

```

transducer_device.jar    SDK_INSTALL_DIRECTORY/src/transducer_device_source.jar
multihop_common.jar     SDK_INSTALL_DIRECTORY/src/multihop_common_source.jar
spotlib_common.jar      SDK_INSTALL_DIRECTORY/src/spotlib_source.jar
spotlib_host.jar        SDK_INSTALL_DIRECTORY/src/spotlib_host_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.

To attach Javadoc, double click the Javadoc location for each jar and use the browse button to go to either the `doc/javadoc` (or the `doc/hostjavadoc` for `spotlib_host`) folder inside your SDK installation.

## Part III: Working with the Basic Sun SPOT Utilities

### Understanding the ant scripts

The properties that control the ant scripts may be defined in several places. In order of precedence, these are

1. On the command line
2. `.sunspot.properties` in the user's home directory
3. `build.properties` in the current directory
4. `default.properties` in the SDK installation

Some property values are also defined explicitly within ant targets. The first two items in the list above will always take precedence over such definitions, but ant targets can take precedence over the third and fourth items by setting properties before the `init` target is executed.

### 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}  
${sunspot.lib}/multihop_common.jar${path.separator}${  
{sunspot.lib}/transducer_device.jar
```

The first line specifies the name of the library suite file. 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

- `multihop_common.jar` (the standard communications stack)
- `transducer_device.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 initialization 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 initial isolate. It is run after all normal `spotlib` initialization and before the first application is launched. If required, this feature can be disabled by setting the system property `spot.start.manifest.daemons` to `false`. This is done automatically when selecting a SPOT to be a basestation, and the property is deleted when deploying an application or when “`ant selectapplication`” is performed.

## **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_common_source.jar` contains the radio communications stack; and `transducer_device_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).

## **Recovery from a broken library**

The library suite contains the code that deals with commands like `deploy` and `flashlibrary`. One of the perils of library development is that it is possible to flash a library that does not execute properly, and may even not be able to execute these commands. Thereafter, it is not possible to replace the library using `flashlibrary`. There is therefore a specialized recovery command for writing the library through the bootloader:

```
ant resetlibrary
```

Because this command works through the bootloader, it is not available for remote Sun SPOTs.

## Extending the ant scripts

The ant scripts supplied with the SDK can be supplemented by:

- scripting written in a project-specific build.xml file, or
- an xml file specified in the *user.import.paths* property

Protected targets begin with "-pre" or "-post". They exist as hooks for users to add behavior to the ant scripts and are intended to be overridden.

Private targets are all other targets that begin with the character "-", and should not be executed directly from user extensions to the ant scripts. They should not be replaced by implementations in a user extension.

Public targets are all other targets. They can be executed from either the command line or from user extensions. They should not be replaced by implementations in a user extension. Private ant targets are subject to change with no notice in future releases. Public and protected targets will be deprecated for at least one release before being retired in a subsequent release.

The existing ant scripts provide useful infrastructure for starting the SpotClient and other related tasks – see the contents of the build.xml file in the root of your SDK installation, and the various component files in the ant sub-directory. Once you have written your scripts, you may wish to make them available to all your Sun SPOT projects. They can be included in the standard ant scripts by defining the ant variable:

```
user.import.paths=/the/path/to/my-extra-scripts.xml
```

This variable must be defined either in `.sunspot.properties` or on the command line to be effective. If you have more than one set of scripts to include, you may set the value of `user.import.paths` to be a comma-separated list of paths. If your scripts have help information, you should also set another variable to specify the target that displays the help information: once again, this may be a comma-separated list of targets.

```
user.help.targets=my-extra-scripts-help-target
```

## Sending commands to SPOTs from a host application

For normal application development, a Sun SPOT 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.

This section explains the programmatic interface to the SPOT Client software, which is provided to allow development of tools that provide 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
- Construct a `SpotClientCommands` object.
- Have the development tool execute various of the commands provided by the SPOT Client code.

### Implementing IUI

This interface defines various methods that the SPOT Client 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.

### **Construct a `SpotClientCommands` object**

At startup, a development tool should create its UI and a `SpotClientCommands` object. The required code depends on whether the SPOT is directly connected to the host via USB or is to be accessed remotely via a basestation.

This code shows the general style for accessing a directly connected SPOT:

```
IUI ui = new MyUIClass();
String pathToApp = ".";
Properties p = new Properties();
String userHome = System.getProperty("user.home");
p.load(new FileInputStream(new File(userHome, ".sunspot.properties")));
String sunspotHome = p.getProperty("sunspot.home");
File sunspotArmDir = new File(sunspotHome, "arm");
String libPath = sunspotHome + File.separator + "arm" +
                File.separator + p.getProperty("spot.library.name");
String keystorePath = userHome + File.separator + "sunspotkeystore";
String commPort = "com3";
SpotClientCommands commandRepository =
    new SpotClientCommands(ui,
                          pathToApp,
                          libPath,
                          sunspotArmDir,
                          keystorePath,
                          commPort);
```

The `SpotClientCommands` constructor for remote access has two additional parameters:

```
...
String remoteSpotAddress = " 0014.4F01.0000.02FF";
SpotClientCommands commandRepository =
    new SpotClientCommands(ui,
                          pathToApp,
                          libPath,
                          sunspotArmDir,
                          keystorePath,
                          commPort,
                          remoteSpotAddress,
                          ISpotAdminConstants.MASTER_ISOLATE_ECHO_PORT);
```

The `SpotClientCommands` object provides a repository containing one instance of `ISpotClientCommand` for each of the commands available.

### **Execute commands**

The general pattern for invoking a command, using the simplest approach, is:

```
commandRepository.execute(<command name> [, <args>]);
```

The command name is a string; the number of arguments and their types depends on the command. The full set of commands available will vary according to which extension modules, if any, are installed.

The default set of commands is:

Name	Implementation class	USB only
blink	BlinkCommand	
deletepublickey	DeletePublicKeyCommand	
deletesystemproperty	DeleteSystemPropertyCommand	
flashapp	FlashAppCommand	
flashbootloader	FlashBootloaderCommand	Y
flashimage	FlashManufacturingImageCommand	Y
flashlib	FlashLibCommand	
flashprops	FlashConfigCommand	
flashvb	FlashBootstrapCommand	Y
flashvm	FlashVMCommand	Y
getfat	GetFATCommand	Y
getstartup	GetCmdLineParamsCommand	
getsystemproperties	GetSystemPropertiesCommand	
hello	HelloCommand	
quit	QuitCommand	
readconfigpage	ReadConfigPageCommand	
readsector	ReadSectorCommand	Y
resetconfigpage	ResetConfigPageCommand	
resetfat	ResetFATCommand	Y
resetlib	ResetLibCommand	Y
resetsystemproperties	ResetSystemPropertiesCommand	
run	StartAppCommand	
setpublickey	SetPublicKeyCommand	
setslowstartup	SetSlowStartupCommand	
setstartup	SetCmdLineParamsCommand	
setsystemproperty	SetSystemPropertyCommand	
settime	SetTimeCommand	
synchronize	SynchronizeCommand	

Please see the javadoc (in doc/hostjavadoc) for the command's class for full details.

For all commands except "hello", a session must be started with execution of the "synchronize" command. Then as many commands as desired can be sent. The session is ended with the "quit" command. Note that only one session may be open to a remote SPOT at a time.

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.

A tool can get slightly more control by getting the command object from the `SpotClientCommands` using `getCommand(commandName)`. The `ISpotClientCommand` interface provides methods for determining the required arguments.

## Extending the SPOT Client

It is possible to extend the SPOT Client with additional commands or replace the default implementation of most commands. This requires extending both the host-side SPOT Client and the `OTACCommandProcessor` on the remote SPOT.

The SPOT Client communicates with the `OTACCommandProcessor` either via the radio or via USB. Either way, the nature of the interaction is as follows:

1. The SPOT Client establishes communication with the `OTACCommandProcessor`.
2. When a command is invoked, the SPOT Client signs the command and sends it to the `OTACCommandProcessor`.
3. The `OTACCommandProcessor` checks the command's signature (remote access only – for directly connected access the signature is ignored) and sends back a string to indicate that the result of the check.
4. Assuming the signature was ok, the `OTACCommandProcessor` looks up the implementation of the command and invokes it.
5. The command is executed, and ends by sending back a string to indicate the result.

### ***Adding or replacing SPOT Client commands***

On construction the `SpotClientCommands` object checks for any registered extension classes (see below). For each class that it finds, it constructs an instance using a no-argument constructor and then calls `editCommandRepository` on it with the current instance of `SpotClientCommands` as an argument. The extension should modify the `SpotClientCommands` instance by adding or replacing any commands it wishes to. You may also wish to add to the ant scripts so that your new commands are exposed through that interface alongside the built-in commands.

To write an extension, create a class that implements `ISpotClientExtension`, and that implements a no-argument constructor. Then, build this into a jar file along with any supporting classes. If you are launching the SPOT Client in the normal way – from ant – then both the jar file and the class itself must be registered with `SpotClient` by setting up ant properties. For example:

```
spotclient.addin.classes=com.sun.spot.client.command.crypto.MySpotClientExtension
spotclient.addin.jars=${sunspot.lib}/myspotclientextension.jar
```

These properties may be conveniently defined in your `.sunspot.properties` file. If you require two extensions, then the `spotclient.addin.classes` variable may be a comma-separated list, and the `spotclient.addin.jars` a standard path variable.

If you are not launching the SPOT Client from ant you should define the extension class(es) in the system property `spotclient.addin.classes`, and add your extension jar to the classpath.

When the `SpotClientCommands` is instantiated, your extension class should be instantiated and its `editCommandRepository(...)` member called. Your class should then add/replace commands, something like this:

```
public class MySpotClientExtension implements ISpotClientExtension {
    public void editCommandRepository(SpotClientCommands commandRepository) {
        commandRepository.addCommand(new MyCommand());
    }
}
```

```
}
```

You may have commands that are only appropriate to remote or locally connected SPOTs, or commands – such as the built-in “hello” command – that don’t apply to a specific SPOT. You may add such commands conditionally, for example:

```
public class MySpotClientExtension implements ISpotClientExtension {
    public void editCommandRepository(SpotClientCommands commandRepository) {
        if (commandRepository.getMode() == SpotClientCommands.MODE_LOCAL) {
            SpotClientCommands.LOCALCommandRepository.addCommand(new LocalOnlyCommand());
        }
    }
}
```

For examples of how to implement commands, see the contents of `.../src/client_source.jar` in the SDK. A class that implements a command should extend `AbstractClientCommand`.

You should then create ant targets for each of your commands. See the section *Extending the ant scripts* for details.

## **Extending the OTACCommandProcessor**

For the remote Sun SPOT to handle your new SPOT Client commands, the `OTACCommandProcessor` must be extended on the remote Sun SPOT. At startup, the `OTACCommandProcessor` checks for registered extension classes. For each that it finds, it constructs an instance using a no-argument constructor and calls `configureCommands` on it. The extension adds or replaces commands in the `OTACCommandProcessor`’s repository. Each time a command is received, the `OTACCommandProcessor` looks up the command and invokes the defined implementation.

To make this happen, first create a class that implements `IOTACCommandProcessorExtension` and has a no-argument constructor. Then, build this into a jar file – using `ant jar-app` – along with any supporting classes. The jar file must contain a standard SPOT library manifest that defines a manifest property like this:

```
spot-ota-extension-crypto: my.package.MyOTACCommandProcessorExtension
```

The prefix “spot-ota-extension-” is required – the remainder of the property name should be specific to your extension.

This jar should then be added to the library using the standard mechanism – see the section *Adding user code to the library suite*.

When the `OTACCommandProcessor` is instantiated, your extension class should be instantiated and its `configureCommands(...)` method called. Your class should then add/replace commands, something like this:

```
public class MyOTACCommandProcessorExtension implements IOTACCommandProcessorExtension {
    public void configureCommands(IOTACCommandRepository repository) {
        repository.addCommand("mycmd", new MyCommand());
    }
}
```

The command object can implement one or many commands because the command name is always passed to it as an argument.

```
public class MyCommand implements IOTACCommand {
    private static final int MY_CMD_SECURITY_LEVEL = 5;

    public boolean processCommand(String command, DataInputStream params,
        IOTACCommandHelper helper) {
```



```

        // ... insert specific code to perform command here ...
        if (commandFailed) {
            helper.sendErrorDetails("The command failed because...");
        } else {
            helper.sendPrompt(); // indicate it all worked ok
        }
        return true; // because we don't want the OTACmdProcessor to close down
    }

    public int getSecurityLevelFor(String command) {
        return MY_CMD_SECURITY_LEVEL;
    }
}

```

Your class will be passed an instance of `IOTACmdHelper` each time that it is asked to process commands. This provides access to some generally useful routines. See the Javadoc for more details. The source code to `OTADefaultCommands` (in `.../src/spotlib_source.jar`) is a good source of examples of command processing.

Generally your routine should handle exceptions internally and use `helper.sendErrorDetails()` to inform the SPOT Client of failures. You should throw an `IOException` only if you get an exception during communication with the remote SPOT Client.

## Hardware revision compatibility

Version 6 (yellow) of the SDK is compatible with SPOT main boards that are hardware revision 4 through 8. Older versions of the SDK (e.g. red or blue) are compatible with SPOT main boards that are hardware revision 4 through 6; they are not compatible with hardware revision 8 SPOTs.

To set the maximum hardware revision with which an SDK is compatible, add the following property to the `version.properties` file, which is located in the base folder of the SDK installation:

```
max.hardware.rev=n
```

where `n` is the hardware revision number.

# Reference

## Persistent system properties

Property name	Meaning
radio.channel radio.pan.id radio.transmit.power	These three properties specify the default radio parameters.
radio.traffic.show.leds	Toggle the green/red processor board LED whenever a radio packet is received/sent
radio.filter radio.whitelist radio.blacklist	Used to specify MAC-layer packet filtering.
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.diagnostics	If true, SPOT outputs additional information about various operations: sometimes useful for support purposes. Defaults to false.
spot.log.connections	Log to System.out each time a radiostream or radiogram connection is made. Defaults to true. See section Radio properties.
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.routing.enable	Can be set to "true", "endnode", "always" or "ifawake". Selecting "true" selects the default (currently "ifawake").
spot.mesh.management.enable	If true, enable all mesh management facilities, such as route tracing. 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.
spot.remote.print.disabled	Inhibit remote echoing of output from a SPOT communicating with a remote basestation. Defaults to false. See section Ensure that the remote Sun SPOT is executing the OTA Command Server.
spot.restart.mode	Controls what happens when all MIDlets finish. Recognized values are: "restart" (reboot SPOT = current default), "off" (SPOT goes into deep sleep) or "continue" (SPOT goes into shallow sleep & continues to listen for OTA radio commands).
spot.startup.isolates.uriids	A list of the MIDlets to be run when the SPOT starts up. Consists of a series of value pairs: the suite URI and the number(s) of the MIDlets in that suite to be run.
spot.start.manifest.daemons	If true or absent run all startup code defined in the library manifest.

## Memory usage (rev 6)

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:

Sector numbers	Start address	Space	Use
0-3	0x10000000	32Kb	Bootloader and associated data
4	0x10008000	8Kb	Config page
5	0x1000A000	8Kb	Flash File Allocation Table (FAT)
6	0x1000C000	8Kb	Persistent properties
7	0x1000E000	8Kb	Reserved for TrustManager library extension
8-11	0x10010000	256Kb	VM executable
12-19	0x10050000	512Kb	Squawk bootstrap suite bytecodes

20-69	0x10140000	3,200Kb	FlashFile area: dynamically allocated between library and application suites, the RMS store, and any application-managed FlashFiles
70-71	0x103F0000	16Kb	MMU level 1 table
72-77	0x103F4000	48Kb	Available for library extensions

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

## Memory usage (rev 8)

The Sun SPOT flash memory runs from 0x10000000 to 0x10800000 (8M bytes), and is organized as 8 x 8Kb followed by 127 x 64Kb sectors. The flash memory is allocated as follows:

Sector numbers	Start address	Space	Use
0-3	0x10000000	32Kb	Bootloader and associated data
4	0x10008000	8Kb	Config page
5	0x1000A000	8Kb	Flash File Allocation Table (FAT)
6	0x1000C000	8Kb	Persistent system properties
7	0x1000E000	8Kb	Reserved for TrustManager library extension
8-11	0x10010000	256Kb	VM executable
12-19	0x10050000	512Kb	Squawk bootstrap suite bytecodes
20-133	0x10140000	7,200Kb	FlashFile area: dynamically allocated between library and application suites, the RMS store, and any application-managed FlashFiles
134	0x107F0000	16Kb	MMU level 1 table

The Sun SPOT external RAM is mapped to run from 0x20000000 to 0x20100000 (1M 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 directory:

ant	Holds ant scripts
arm	Directory holding binary files specific to the Sun SPOT
bin	Host-specific executables
Demos	Demo applications and code samples
doc	Documentation
external-src	Modified sources of open-sourced items
lib	Jar files that are of interest to developers (typically because they will be needed to configure an IDE).
SPOT Utilities	Utility programs
src	Library source code
temp	Used to hold temporary files
tests	Test programs

upgrade	Files used to upgrade firmware during “ant upgrade”
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

### Jar naming conventions

Jar files in the SDK fall into three categories:

- Those that do not contain java classes, such as source jars.
- Those that are private to the SDK and contain classes. These jars reside in the bin directory.
- Those that are useful to SPOT developers and contain classes. These jars reside in the lib directory.

A naming convention is applied to these to indicate their purpose as follows:

- <name>\_device.jar: containing code that can only be executed on the Sun SPOT
- <name>\_host.jar: containing code that can only be executed on the host computer
- <name>\_common.jar: containing code that can be executed either on the Sun SPOT or on the host computer.

### 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 bin directory:

debugger_classes.jar	Sun SPOT high-level debugger classes. Used by the debug proxy.
debugger_proxy_classes.jar	The classes of the Sun SPOT debugger proxy.
debuggerproxylauncher.jar	The classes for the program that launches the debug proxy.
emulator.suite	The built suite for the SPOT emulator
emulator.suite.metadata	Metadata for the emulator suite
emulator_classes.jar	The classes for the emulator
emulator_j2se.jar	The J2SE classes for the emulator
hosted-support_classes.jar	
imp_classes.jar	The classes for the IM Profile
preverify.exe	The J2ME pre-verifier program. File extension may vary.
romizer_classes.jar	The classes of the Squawk suite creator. Used by the debug proxy.
spotfinder.exe	The program that identifies which COM ports map to Sun SPOTs. File extension may vary.
spotselector.jar	Program that allows the user to select a connected SPOT from a list

squawk.exe	The Squawk executable for the host. File extension may vary.
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_device_classes.jar	The Squawk classes used to create suites and for debugging.
squawk_host_classes.jar	
translator_classes.jar	The classes of the translator. Used by the debug proxy.

### Contents of the Demos directory:

CodeSamples	A subdirectory containing simple examples of how to use the radio and some of the basic SPOT eDemo board sensors.
AirStore	A data repository that Sun SPOTs (and Sun SPOT Host Applications) can share via their radios. The goal is to enable simple, one line "gets" and "puts" with which distributed applications can store and access data, while enabling its use as a "blackboard" style coordination mechanism, similar to a Tuple Space.
AirText	An application that uses the linear array of tricolor LEDs on the eDemoboard and "persistence of vision" to display text in thin air when the SPOT is moved back and forth.
BounceDemo	A demo to pass a "ball" displayed in the LEDs between 2 or more SPOTs.
BuiltInSensorsDemo	A demo of the Sun SPOT "eDemoBoard" built-in devices (light detector, temperature, accelerometer, LEDs, switches).
CryptoDemos	A collection of demos showing how to use cryptographic functionality on Sun SPOTs.
DatabaseDemo	This demo application stores periodic sensor readings measured on one or more Sun SPOTs into a database and runs a few simple SQL queries on the stored data.
EmulatorDemo	A series of sample MIDlets to demonstrate the new SPOT Emulator.
GeigerDemo	Reports radio traffic in an audible manner like a Geiger counter.
HostQuerySpotsDemo	This demo is a host application that demonstrates how to locate nearby SPOTs and then send them commands over-the-air (OTA). It makes use of the spotclient library that is also used by both Solarium and ant commands.
HTTPEdemo	This demo application uses the Sun SPOT's built-in HTTP networking capability to interact with the Twitter service.
InfraRedDemo	This is a simple application to demonstrate the use of the InfraRed capabilities on the new rev8 Sun SPOT. Turns your SPOT into a remote volume control for your tv.
iRobotCreateDemo	A simple demo to bounce an emulated Create robot off the walls in the Robot View in Solarium.
RadioStrength	A simple application for 2 SPOTs. Each SPOT broadcasts 5 packets per second and listens for radio broadcasts from the other SPOT. The radio signal strength of the packets received is displayed in the SPOT's LEDs.

SendDataDemo	This demo application sends periodic sensor readings measured on one or more Sun SPOTs to an application on your laptop or PC that displays the values.
SensorDotNetworkDemo	This demo is an extension of the SendDataDemo. It sends periodic sensor readings from one or more Sun SPOTs to a web-based service called sensor.network where that data can be easily searched, shared, visualized and analyzed.
SolariumExtensionDemo	This demo shows two ways to extend the Solarium application. The first, VirtualObjectPlugin, modifies Solarium to display a SPOT that is running the SendDataDemo with a unique icon and with a modified popup menu that includes displaying the data being collected within Solarium. The second, NewViewPlugin, shows how to add a new view to Solarium and also how to extend the Emulator to work with the new view.
SPOTWebDemo	This demo is a host application that starts up a simple web server and lets remote users interact with a collection of SPOTs in the vicinity of the attached basestation using a standard web browser. Authorized remote users can monitor the state of sensors, applications and other system statistics. They can also install, start, pause, resume, stop and remove applications.
TelemetryDemo	This is a demo application that provides the basic framework for collecting data on a remote SPOT and passing it over the radio to a host application that can record and display it.
YggdrasilDemo	This demo uses the Yggdrasil library ( <a href="http://yggdrasil.dev.java.net">http://yggdrasil.dev.java.net</a> ) to provide an easy way to get sensor data from a Sun SPOT to a host. Yggdrasil provides support for basic Wireless Sensor Network applications, where the nodes are able to sleep and mesh. Yggdrasil also supports a simple mechanism of sending the data through the basestation to <a href="http://sensor.network.com">http://sensor.network.com</a> .

### Contents of the doc directory:

AppNotes	A sub-directory containing various application notes
hostjavadoc	A sub-directory containing Javadoc for the part of the SPOT base library that is appropriate for host apps, and for the host-specific SPOT Client and host-agent code.
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.
README.txt	A pointer to the index.html file in the SDK directory.
ReleaseNotes.html	Release Notes
SPOTManager	A sub-directory containing help for the SPOT Manager tool
SunSPOT-Programmers-Manual.pdf	This manual
SunSPOT-TheoryOfOperation.pdf	Details about the hardware
Tutorial	A sub-directory containing the Quick Start tutorial (start at Tutorial.html)

### **Contents of the lib directory:**

junit.jar	The well-know Java testing framework, supplied here for use in host applications, NOT the Sun SPOT.
ipv6lib_common.jar	The classes of the Sun SPOT IP stack library.
multihoplib_common.jar	The classes of the Sun SPOT comms stack library.
networktools_host.jar	The host-side classes for trace route and other network facilities
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)
sdcard_device.jar	The SD Card external board driver
signing_host.jar	Classes that are used to security sign suites and commands.
socket_proxy_host.jar	The host-side proxy for the http connection support.
spotclient_host.jar	The classes used to send commands to SPOTs from the host.
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 basestation.
spottestframework_device.jar	The classes for the on-device test framework
SPOTWorld	A sub-directory containing the classes to run solarium
squawk_common.jar	The Squawk classes used by both host and SPOT apps.
squawk_device.jar	The Squawk bootstrap classes stripped to contain just those parts of Squawk available to Sun SPOT applications. Sun SPOT applications compile against this.
transducer_device.jar	The classes of the Sun SPOT eDemo board library.

### **Contents of the SPOT-Utilities directory:**

CalibrateAccelerometer	A SPOT application to manually calibrate the accelerometer on the eDemo board.
PacketSniffer	A SPOT application to listen to all packets received by the radio and to print them out.
SpotCheck	A diagnostic application that runs on the Sun SPOT device to check power and sensor functionality.

### **Contents of the src directory:**

emulator_j2se_source.jar	Source code for the j2se parts of the emulator
emulator_source.jar	Source code for the main part of the emulator
ipv6lib_common_source.jar	Source code for the IP stack.
multihop_common_source.jar	Source code for the radio communications stack.
sdcard_device_source.jar	Source code for the driver for the SD card external board



<code>sdproxylaucher_source.jar</code>	Source code for the debugger proxy program.
<code>solarium_source.jar</code>	Source code for the Solarium tool.
<code>spotclient_source.jar</code>	Source code for the SPOT Client program.
<code>spotlib_host_source.jar</code>	Source code for the Sun SPOT base library (the part that is specific to host applications).
<code>spotlib_source.jar</code>	Source code for the Sun SPOT base library.
<code>spotsselector_source.jar</code>	Source code for the program that selects which SPOT to use with ant commands.
<code>spotsocketproxy_source.jar</code>	Source code for the host-side socket proxy.
<code>spottestframework_source.jar</code>	Source code for the on-device test framework
<code>transducerlib_source.jar</code>	Source code for the Demo Sensor Board library.

### **Contents of the upgrade directory:**

<code>demosensorboardfirmware.jar</code>	Current version of the firmware for the eDemo board.
<code>oobd.jar</code>	Current version of the out-of-the-box-demo that comes preinstalled on each SPOT.
<code>pctrlfirmware.jar</code>	Current version of the firmware for the main board power controller.

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](#)).

# Troubleshooting

## Software, All Platforms

---

### Problem – But I don't want to use NetBeans!

---

One of the things that NetBeans does is set some needed environmental variables. If you don't use NetBeans, do the following:

- Modify your `PATH` to include the bin subdirectory for your Java Development Kit (JDK).
- Modify your `PATH` to include bin subdirectory for the Ant directory.
- Set `JVM_DLL` to the location of your Java virtual machine (Windows only).
- Set `JAVA_HOME` to the location of your top level Java SDK directory (Windows and Linux).

NetBeans also provides a user interface for the several `Ant` commands used to deploy and run code on the Sun SPOTs from the host workstation. Read through the Sun SPOT Developer's Guide and make sure you understand how to use the `Ant` commands described there.

---

### Problem – You get the following error message any time you attempt to communicate with a Sun SPOT over the USB cable, including any of the many `ant` commands:

---

```
[java] WARNING: RXTX Version mismatch
[java] Jar version = RXTX-2.1-7
[java] native lib Version = RXTX-2.1-7pre17
```

---

The host workstation uses a serial communication package called RXTX to communicate with the Sun SPOTs. The version in use on the host workstation and on the Sun SPOT must match. The correct version of the RXTX library is installed on the host workstation when the SDK is installed, but if another version is on the load path, a mismatch can occur. Look for an old version of the RXTX library somewhere in your load path. The name of the RXTX file varies with the operating system of the host workstation, as shown in the table below.

<i>Operating System</i>	<i>RXTX File Name</i>
Windows	rxtxSerial.dll
Macintosh	librxtxSerial.jnilib
Linux	librxtxSerial.so

Change your `PATH` variable to avoid the other version of RXTX or remove the excess version of RXTX.

---

**Problem** – My Sun SPOT has got into a state where it continually restarts and I get an error whenever I try to deploy to it. How can I recover?

---

First you need to get the SPOT into a state where it is listening to commands from the host rather than continually restarting. To do that, follow this procedure:

- Disconnect the Sun SPOT from the USB cable.
- Kill all the `ant` and `java` processes listening on the port.
- Hold the control button in for a few seconds until a double red flash indicates that the Sun SPOT has powered down.
- Type this command at a command prompt:  

```
ant -Dport=COMnn info
```

substituting the correct communication port/device name for `COMnn`.

If you don't know the correct port name just enter a dummy value, e.g., "x", and complete the procedure. The output should list the correct port name. Then repeat the procedure with the correct port.

As soon as the `ant` script starts to complain that the port isn't available, saying something like:

```
[java] Port COM31 unavailable...
[java] Available ports: COM1 COM2 COM3 LPT1
[java] retrying...
```

immediately plug in the Sun SPOT.

The `ant info` command should now operate correctly. When it has finished, you can take the necessary recovery action, normally to reinstall the system software using `ant upgrade` and then redeploying your (possibly corrected) application.

---

**Problem** – When I try to create a suite (via `ant suite` or `ant deploy`) I get the following error:  
Unable to locate tools.jar Expected to find it in C:\...

---

Most likely your system environment variables are not setup correctly. Please make sure your `PATH` variable has your JDK directory as its `FIRST` value. This also ensures that Windows is not using the `java.exe` commonly located in `C:\Windows\System32`. Also ensure that `JAVA_HOME` is set correctly.

---

**Problem** – How do I modify the `sunspot.home` property for the `ant` build system?

---

You need to modify the file `.sunspot.properties` located in your home directory. In Windows, this is `C:\documents and settings\`. On the Macintosh, this is `/Users/`.

---

**Problem** – Is there an API to query the ID of a SPOT at run-time?

---

```
System.getProperty("IEEE_ADDRESS") .
```

---

**Problem** – When I do an OTA ant deploy, I get a “No route found” error.

---

If your attempt to do an OTA ant deploy shows console output that looks like:

```
-run-spotclient-once:
[java] SPOT Client starting...
[java] Error: No route found
[java] The SPOT client will now exit
BUILD FAILED.
```

then you probably have the OTA command server enabled on your basestation. It should not be so enabled. Do an `ant disableota` with the basestation as the target and retry your OTA deploy.

## Software, Linux

---

**Problem** – (*Linux*): When trying to deploy an application to a Sun SPOT, I get error messages from RXTX, saying that the device is not available, or that it doesn't have permission to access it or the lock file.

---

Make sure that you followed the instructions in the section “Adjust Permissions (Linux)” on page 29 of the *Installation Instructions*. In particular don't forget to log out after performing the changes.

If you still cannot deploy to the Sun SPOT, make sure that the `/var/lock` directory exists, and that it has read, write, and execute permissions for the group to which you added the user. If it doesn't exist, you should create it. The recommended permissions are `755`, and the group should be `lock`. Also check the permissions and group of the device file. The name of the device should be in the error message; common names are `/dev/ttyACMx` or `/dev/usbmodem`.

If this doesn't fix the problem, verify that there is no link to `/var/lock` or `/var/spool/lock` as this may cause RXTX to detect the lock file twice and to fail.

---

**Problem** – (*Linux*): When trying to deploy an application to a Sun SPOT I get an error message: “No Sun SPOT devices found. ....” or similar.

---

In addition to the obvious things, like making sure that a Sun SPOT is actually connected to the USB port, it may be that your Linux installation doesn't have the `cdc_acm` driver that is required to access the Sun SPOT. Try calling `dmesg | grep usb` in a command window. There should be a message that looks something like `usb ...: New full speed USB using ...`. Furthermore there should also be a message referring to `cdc_acm`. If you see the general messages but you do not see any `cdc_acm` messages, you probably have to install the `cdc_acm` driver onto your system.

If you see the `cdc_acm` messages, and still get a "No Sun SPOT devices found..." error, it may be that the `spot-finder` script has failed to detect the device. This may occur if the `hal_device` command is not available and the device gets assigned a name not matching the expected `/dev/ttyACM*` pattern. We recommend you install `hal_device` on your system in this case. Alternatively, you may specify the appropriate device name manually in the `ant port` property. You define this property in the `build.properties` for a project or in the command line to `ant`.

---

**Problem – (Linux):** When trying to deploy an application to a Sun SPOT, I get an error message: `Port Error: An SELinux policy prevents this sender from sending this message to this recipient unavailable...` or something similar.

---

This means that the SELinux policy is too restrictive and is not allowing access to the serial device. This is, for example, the case in the default settings for Fedora Core 5.

If you don't require the additional security, SELinux allows you to either disable it or set it to a less restrictive setting. For example, in Fedora Core 5, changing the SELinux setting to permissive solves this problem. You can change the SELinux setting in the `System/Administration/Security Level and Firewall` menu. If you don't want to give up this security level, you need to define a SELinux policy that gives RXTX permission to the device.

## Sun SPOTs in General, Hardware

---

**Problem –** SPOT doesn't power up. Neither the power LED nor the activity LED are lit.

---

Check the battery connector.

Plug the SPOT unit into a powered USB host. This can be a powered USB hub, a USB port on Computer, or a USB mini Type B charger. Make sure the connectors are seated properly and the host device is powered. If the SPOT is not a basestation SPOT and if power LED starts to slowly flash green, then the battery is charging; allow the battery to charge for three hours.

Push the attention button, do not hold it in.

### ***Additional Troubleshooting***

If you have a digital voltmeter and are comfortable using it on small circuits, make the following voltage measurements. Unscrew the main retaining screw (Phillips head) and remove the eDEMO board to expose the main board. Leave the battery installed and plug the USB port into a powered source. Set the DVM to measure volts and connect the leads of the DVM to common and volts. Connect the common (-) lead to gold ring around the screw retention hole. From the top of the board with the antenna at the top, find a via marked VSTBY. This is above and to the right of the 30 pin white connector. This should measure  $+3.0V \pm 5\%$  when any power source is connected. Find the left most ceramic capacitor next to 3V. This is on the lower left side of the board. Put the positive (+) lead on the bottom of this capacitor. This should measure  $+3.0V \pm 5\%$  when power is on. Find the capacitor below the 3.0V marked 1.8V. On the top lead of the capacitor measure  $1.8V \pm 5\%$ . If the SPOT is not a basestation perform the following measurement. In the lower right hand corner is a through hole pad marked V+, this should measure between 3.2V and 4.7V. If you remove the USB, it should not drop below 3.2V. If it does, the battery is defective or dead.

---

**Problem** – The Sun SPOT powers up but doesn't boot. The power LED comes on green but the activity LED doesn't flash.

---

Connect the Sun SPOT to the USB port of your host workstation and run `ant info` to see if the problem is a faulty activity LED.

The Sun SPOT has either a hardware fault or a corrupt ARM bootloader. This requires factory service.

---

**Problem** – The Sun SPOT powers up and the power LED turns constant bright red. This may occur during upgrading of power controller firmware.

---

This is corrupted firmware for the power controller. Try to upgrade without power cycling the SPOT. This may require factory service to restore.

---

**Problem** – The Sun SPOT comes on and boots (activity LED flashes green) but host workstation can't find the Sun SPOT as a USB device.

---

Check the USB cable. Check for potential interfering drivers on the host workstation (try on a different computer). Exit any application that is trying to communicate to the Sun SPOT, disconnect the USB cable, hold the attention button for > 3 seconds, then tap the attention button to restart the SPOT. Plug the cable back in and try again.

*For Windows only:* Look in the Device Manager to see if there is an "unknown" USB device. If there is, see if it appears and disappears when you connect and disconnect the Sun SPOT. If the unknown device appears to be the SPOT, connect it, ask Windows to uninstall it, then disconnect the SPOT, then reconnect it. You should now get the "new device" dialog and be able to reinstall the driver.

---

**Problem** – The Sun SPOT comes on and starts to boot, but the activity light flashes red and not green.

---

The ARM9 memory test has failed. Retry, but if there are continued failures this indicates bad memory and requires factory service.

---

**Problem** – The Sun SPOT comes on and boots. The host workstation USB finds the SPOT but says the device is busy.

---

There is probably an active process still connected to the SPOT. Disconnect the Sun SPOT, kill all the processes that talk to the SPOT, reboot the SPOT and reconnect.

---

**Problem** – The Sun SPOT continually reboots.

---

The Sun SPOT may have encountered an exception that causes it to continually reboot. There may be a problem in the deployed SPOT application. Do `ant echo` and look at any output from the SPOT to see what the problem is. Try deploying a known demo to the SPOT and see if that works. If not try reloading the SPOT system code by running `ant upgrade` on the SPOT.

---

**Problem** – The Sun SPOT communicates with the host workstation USB but the power LED flashes red.

---

Connect the Sun SPOT to the host workstation USB. Load the application in the demo folder called `SpotCheck`. This prints out the voltage and current usage of the SPOT along with any fault issues with the SPOT.

---

**Problem** – The Sun SPOT communicates with the host workstation USB but the eDEMO board is not working properly.

---

Load and run the application in the `demo` folder called `SpotCheck`. Cycle through the tests to check the light sensor, temperature sensor, accelerometer, and LEDs on the eDemo board.

---

**Problem** – The Sun SPOT communicates with the host workstation USB but doesn't communicate with the radio.

---

Check for potential interference and shielding of the antenna fin. Antenna (thin part of the SPOT box) must be clear of metal objects. Testing requires two SPOTs with eDEMO boards. Load and run an application in the demo folder called `RadioStrength`. Each SPOT then takes turns transmitting and receiving data packets and the signal strength is displayed as a bar graph on the Sun SPOT LEDs.

## If Your Sun SPOT Needs Factory Service

If your Sun SPOT requires factory service, please send an email to [info@sunspotworld.com](mailto:info@sunspotworld.com) with "Service request" in the subject line of the message. Please summarize the problem with your Sun SPOT in the body of the message.

# Battery Warnings

Do not short-circuit battery. A short-circuit may cause fire, explosion, and/or severe damage to the battery.

Do not drop, hit or otherwise abuse the battery as this may result in the exposure of the cell contents, which are corrosive.

Do not expose the battery to moisture or rain. Keep battery away from fire or other sources of extreme heat. Do not incinerate.

Exposure of battery to extreme heat may result in an explosion.

No other battery substitutions or different chemistry batteries should be used.

Do not bypass the battery protection circuit.

Dispose of batteries properly. Do NOT throw these batteries in the trash. Recycle your batteries, if possible.



# Federal Communications Commission Compliance

**NOTE:** This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try and correct the interference by one or more of the following measures: Reorient or locate the receiving antenna. Increase the separation between the equipment and receiver. Connect the equipment into an outlet on a circuit different from that to which the receiver is connected. Consult the dealer or an experienced radio/TV technician for help.

The Sun SPOTs are supplied with a shielded USB cable. Operation with a non-shielded cable could cause the Sun SPOTs to not be in compliance with the FCC approval for this equipment. The antenna used with this transmitter must not be co-located or operated in conjunction with any other antenna or transmitter; to do so could cause the Sun SPOTs to not be in compliance with the FCC approval for this equipment. Any modifications to the Sun SPOTs themselves, unless expressly approved, could void your authority to operate this equipment.

FCC Declaration of Compliance:

Responsible Party:

Oracle America, Inc.

500 Oracle Parkway

Redwood Shores, CA 94065

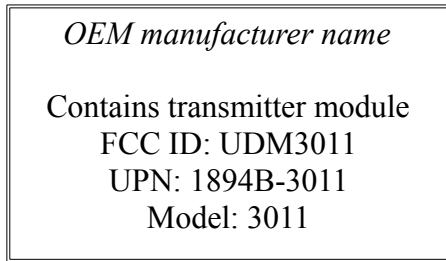
Phone: US +1.650.506.7000; International +1.650.506.7000

FCC IDENTIFIER: UDM3011

This device complies with Part 15 of the FCC Rules. Operation is subject to the following conditions: this device may not cause harmful interference and this device must accept any interference received, including interference that may cause undesired operation.

This device can be used as is (stand-alone) or as a module (part of a final host product). If the device will be used as a module these rules must be followed:

**Integrator must place a label outside their product similar to the example shown:**



**Caution: Exposure to Radio Frequency Radiation**

To comply with FCC RF exposure compliance requirements, a separation distance of at least 20 cm must be maintained between the antenna of this device and all persons. This device must not be co-located or operating in conjunction with any other antenna or transmitter.

Module 3011 and antenna tested with must be integrated in the end product in such a way that the end user cannot access the either the module, cables, or antennas.

The installer of this radio equipment must ensure that the antenna is located or pointed such that it does not emit RF field in excess of Health Canada limits for the general population; consult Safety Code 6, obtainable from Health Canada's website [www.hc-sc.gc.ca/rpb](http://www.hc-sc.gc.ca/rpb).