The Rolling Pins Developers Guide

Arnþór Magnússon

1. Version, October 2012

Contents

Part I

Getting Started

Introduction

The Rolling Pins are modular robotic devices, that can output light in different colors, vibration and audio. They can be interacted through movement (accelerometer and gyroscope), other pins and other devices containing wireless module.

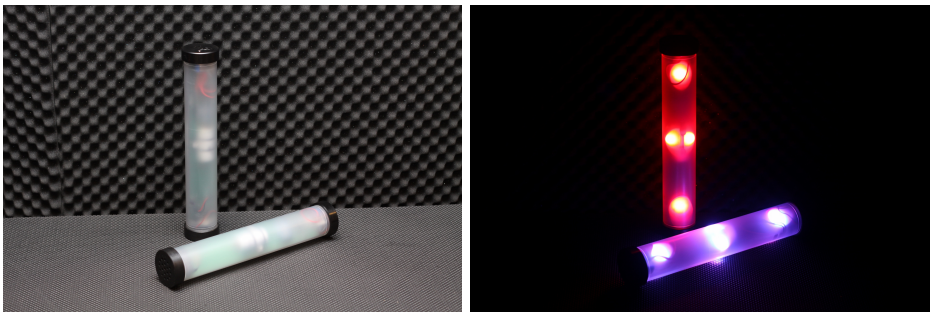

Figure 1: The Rolling Pins.

The Rolling Pins are made out of sturdy translucent plastic material which forms a cylinder. A black lid is screwed on each end of the cylinder, the top end contains the power connector and the JTAG programming connector, while the bottom end contains the speaker.

Internal hardware

- One 8bit RISC Atmel ATMega 1280 processor.
- Eight RGB LEDs
  Each with 16 levels of intensity per color, thus able to display up to 4096 different colors.
- One vibrator.
- One Speaker and a audio module.
  Audio can be played from a SD memory card through a speaker.
- One accelerometer and a gyroscope.
  To measure movement and orientation of the pin.
- One XBee communication chip.
  The XBee allows for communication between pins, as well as other devices containing XBee modules.

Chapter 1

Software Installation

This chapter describes the useful and in most cases necessary software installations for developing applications for the Rolling pins. We recommend setting up all the following software:

1.1 Source code

1.2 Windows Installation (32bit)

Source Code

The source code for the pins is located at Github: `https://github.com/Playware/Pins`. To download the code, one must have a Git client, one such is TortioiseGit:

1. Download a Git client and install, TortioiseGit:
   `http://code.google.com/p/tortoisegit/`
2. Create a new folder that is supposed to contain the source code.
3. Right click on the new folder and select TortioiseGit->Clone and enter the Github url: `https://github.com/Playware/Pins`, continue and the program fetches the source code and places it in the folder.

X-CTU Software (optional)

X-CTU is a stand-alone tool for configuring XBee modules.

1. Download the installation file from:
   `ftp://ftp1.digi.com/support/utilities/Setup_XCTU_5260.exe`
2. Browse to where the file is stored.
3. Double-click on the installer file and follow the X-CTU Setup Wizard.
4. When asked if you would like to check Digi's website for firmware updates, click Yes.

5. After the firmware updates are complete, click Close.
6. Start X-CTU by double-clicking on the X-CTU icon placed on your desktop or by selecting: Start > Programs > Digi > X-CTU

FTDI Driver for XBEE

In many cases this driver will be installed automatically when the XBee module is plugged into a windows computer. If not, follow these steps:

1. Download the installation file from:
   Win98 - XP: `http://ftp1.digi.com/support/driver/40002636_A.zip`
   Vista/Win7: `ftp://ftp1.digi.com/support/driver/ftdi_WindowsVista_7_Drivers.zip`
2. Unzip the file to a folder and point the windows driver installation to look for the driver in that folder.
3. Windows should set up the driver automatically from there

Atmel AVR Studio (optional)

The Atmel AVR Studio is a great tool for JTAGing and debugging. It also includes the driver for the JTAG ICE MkII programmer/debugger used at Center For Playware.

1. Download version 4 of the program and all service packs if available from:
   `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725`
   or version 5 from:
   `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=17212&source=avr_5_studio_overview`
   In both cases a registration might be required.
2. Install the program and all service packs in order by running the executable files.

Win AVR (AVR GCC) Compiler Suite

The Win AVR Compiler Suite includes the avr gcc compiler needed for compiling code for the pins from Eclipse.

1. Download the latest version of the program from:
   `http://sourceforge.net/projects/winavr/files/WinAVR/`
2. Double click the installer file and follow the wizard.

Eclipse (Eclipse IDE for C/C++ Developers) (32-bit) (optional)

Eclipse is a Java based programming environment which has proven to be very good for creating C code.

1. Download the latest version of the C/C++ version of the program from:
   `http://www.eclipse.org/downloads/`
2. Unzip the file to a desired folder (No installation required).

AVR GCC Plugin for Eclipse (optional)

AVR GCC Plugin is necessary for compiling software through Eclipse

- See further instructions following the link:
  `http://avr-eclipse.sourceforge.net/wiki/index.php/Plugin_Download`

Arduino environment

The Rolling Pins uses the Arduino framework for convenience.

1. Download the latest version of the program from:
   `http://arduino.cc/en/Main/Software`
2. Unzip the downloaded file when it finishes, make sure to preserve the folder structure

1.3 Linux Installation

Source Code

The source code for the pins is located at Github: `https://github.com/Playware/Pins`. To download the code, one must have a Git client, which is available in the Linux package manager.

1. To get Git, open a terminal and type:

```
apt-get install git
```

2. Go to a directory you want to place the source code:

```
cd /path/to/source/code/
```

3. Pull the source code from Github:

```
git clone https://github.com/Playware/Pins
```

AVR Libc and AVR-GCC compiler

AVR-GCC and Libc are required in order to develop and compile the source for the AVR microchip. In many cases this software can be installed from a package manager within your Linux distribution. For example in Debian based distro:

```
apt-get install gcc-avr avr-libc
```

Otherwise the software is available at `http://www.nongnu.org/avr-libc/`.

Eclipse (Eclipse IDE for C/C++ Developers) (optional)

Eclipse is a Java based programming environment which has proven to be very good for creating C code.

- Install via package manager:

```
apt-get install eclipse-cdt
```

or

1. Download the latest version of the C/C++ version of the program from: `http://www.eclipse.org/downloads/`
2. Unpack program and run eclipse.

AVR GCC Plugin for Eclipse (optional)

AVR GCC Plugin is necessary for compiling software through Eclipse

- See further instructions following the link: `http://avr-eclipse.sourceforge.net/wiki/index.php/Plugin_Download`

AVRDude

avrdude is used to upload a compiled program to a individual pin. It can be installed using a package manager:

```
apt-get install avrdude
```

or by visit avrdude webpage `http://www.nongnu.org/avrdude/` and follow their instructions.

Arduino environment

The Rolling Pins uses the Arduino framework for convenience.

- Install via package manager:

  ```
  apt-get install arduino
  ```

or

1. Download the latest version of the program from:
   `http://arduino.cc/en/Main/Software`
2. Unzip the downloaded file when it finishes, make sure to preserve the folder structure

1.4 Setup and Configuration

In which environment to program the Rolling Pins is a matter of opinion. The author of this document prefers to use *vim* to program and when finished, one connects the Rolling Pin to the JTAG ICE MkII programmer/debugger which in turn is connected to the computer. The code is compiled using the Make build utility which needs to be configured, by setting the path to previously installed Arduino environment in the file `Makefile` at the `RollingPins` directory. Find the variable `ARDUINO_DIR` and set it to the Arduino install directory. Then all you need is to go to the `RollingPins` directory and run the following command in terminal:

```
make upload
```

and the make script compiles and uploads the program to the pin. The make script is supported by many programming environments and are easily configured to use the provided `Makefile`. If one prefers to use Eclipse environment, it would be wise to follow the instructions in the Tile manual.

1.5 The Rolling Pin Folder Structure

The programing framework is divided into several directories depending on functionality.

- RollingPins
  - **Controller** *(Here you can find various functions for timers, events and communication protocol.)*
  - **Drivers** *(Here are the drivers for the various modules of the pins such as Light, Speakers, Accelerometer etc.)*
  - **Applications** *(Here all the applications reside, where each application has it's own folder.)*
  - **Libraries** *Here you can find the various libraries, for instance for the XBee module.*
  - **Bin**
  - makefile

The important files

The important files that you need to look at are:

- **Application.h**
  This file resides in the application folder and is a header file with all applications made for the Rolling Pins. The application to be run needs to be defined as the DEFAULT_APPLICATION. When a new program is made, a new line with an extern Application needs to be added. REMEMBER that the name of the application in this file can not be exactly the same as the *namespace* in the application itself.

- **Application.cpp**
  This file resides in the application folder and is a container file with all applications made for the Rolling Pins. When a new program is made, the new application needs to be added to the **apps* array.

- **RPApi.h**
  This file holds some global configurations and some system functions. It is a good idea to include this file into every application made for convenience.

- **Makefile**
  This file is a part of a automatic build system, to build the program into hex file and uploading to a pin. This file might need editing, to set the path to Arduino environment.

- **HelloWorld.cpp**
  This file resides in the HelloWorld folder in the application folder and is a template for a new program. When making a program it is best to make a copy of this file and start from there.

1.6 Compiling and Uploading Programs to the Pins

First the code needs to be compiled, which is done using the Make build system, by going into the `RollingPins` directory in a terminal or command line window, and type

```
make
```

which creates a hex file into the `bin` directory.

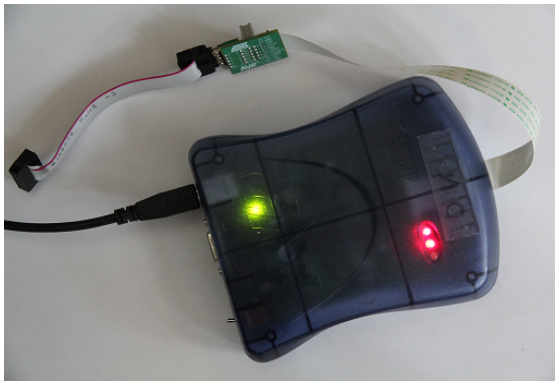

Figure 1.1: A JTAG ICE MkII.

To upload this program to a pin, a external programmer with JTAG interface is required, such as JTAG ICE MkII (see figure 1.1). The programmer is plugged both to the computer via USB and to one pin, where the pin is connected with a regular cable directly to the JTAG connector which is located on top of the Rolling Pin, as seen in figure 1.2.

Figure 1.2: Programmer connected to a pin through the JTAG plug.

When the program has been compiled it is ready to be uploaded to a pin, which can be done using the command:

```
make upload
```

Part II

Programming the Pins

Chapter 2

Your First Program

2.1 "Hello World"

For this example application we will make a pin version of the famous Hello World example. In our case it means that we will turn on all LEDs for the pin for one second and then off again for the same time.

First we start creating a directory called *HelloWorld* in the *application* directory and create and open a file called *HelloWorld.cpp*. Then we can fill it with a bare bone application

```
#include "RPApi.h"

namespace HelloWorld
{
  static void setup();
}

Application_t helloWorld = {"HelloWorld", HelloWorld::setup};

static void HelloWorld::setup()
{
}
```

First to notice is the declaration of the variable `helloWorld`

```
Application_t helloWorld = {"HelloWorld", HelloWorld::setup};
```

which defines an object containing a reference name for our application *HelloWorld* and a starting point, function called `setup`. From the function `setup()` we can setup the flow of our program, for instance create loops or timers to execute some code with certain intervals or how to handle incoming packets or messages. For our application we want to create a timer that executes a function we create with 1 sec. interval. Lets call our function `onTimerFired()`.

```
  static void onTimerFired()
  {
    // Our code will be put here
```

```
}

static void HelloWorld::setup()
{
  timerManager.createPeriodic(0, 1000, onTimerFired);
}
```

In order to turn on and off all the LEDs we use the functions

```
light.setAll(200, 200, 200) // turn on
light.update()
// and
light.clearAll() // turn off
light.update()
```

where `setAll(red, green, blue)` sets all LEDs to a color depending on the values given (from 0 to 255), while `clearAll()` turns all of them off. The `update()` function needs to be issued every time lights should be changed. Then we can create some logic to turn on and off our LEDs

```
static bool isOn = false;

static void onTimerFired()
{
  if (isOn)
    light.clearAll();
  else
    light.setAll(LIGHT_WHITE);
  light.update();
  isOn = !isOn;
}
```

Next step is to register our new application as a part of the pins software and make it start when the pin is turned on. To register the application, open the file *Application.h* and add the line

```
extern Application_t helloWorld;
```

then add `&helloWorld` (hint: which was defined in HelloWorld.cpp) to the array `*apps[]` in *Application.cpp*. To tell the compiler that we want our application to run at startup, open the file *Application.h* and find the line

```
#define DEFAULT_APPLICATION &someApplicationName
```

and change `&someApplicationName` to `&helloWord`.

Finally compile and upload the program to a pin and to see your application in action.

Chapter 3

Digging Deeper

Now that we have a working program that can be compiled and uploaded to the Rolling Pins, we can start creating more complex code.

3.1 The Leds

Each pin has eight RGB leds with 8 bit resolution per color. That means that you can set the value of each channel between 0 and 255, the higher value the stronger intensity of the color is added to the mix.

The functions for manipulating the LEDs should already be included in your application through the `RPApi.h` file, but are defined in **drivers/Light.h**. The packet has four functions worth to mention:

- `void setAll(uint8_t red, utin8_t green, utin8_t blue)`
  This function takes in values for the red, green, and blue part of the color and sets that value for all the LEDs.
- `void set(uint8_t idx, uint8_t red, utin8_t green, uint_t blue)`
  This function takes in values for the red, green, and blue part of the color and sets that value for the corresponding LED (0-7).
- `void clearAll()`
  This function turns all LEDs off or more specifically, sets the value for red, green and blue to 0.
- `void update()`
  After changing the color of any LED, one must call this function to actually update the changes.

3.2 Vibrator

A vibrator is present in each pin, to interact with a user. Three functions can be utilized and are available through the **RPApi.h** file. There are three function that can manipulate the vibrator:

- `vibrator.on()`
  This function turns on the vibrator.
- `vibrator.off()`
  This function turn off the vibrator.
- `vibrateUntil(uint16_t milliSec)`
  This function turns the vibrator on for `milliSec` milliseconds and then off again.

3.3 Sound

In order to play a sound or music, a micro-SD memory card must be present in the pin, containing files on the ADPCM (.ad4) format and be named accordingly **0000.ad4**, **0001.ad4** ... **0511.ad4**. Wave and MP3 files can be converted to this format using software provided by 4D Systems, which is found at this URL: `http://www.4dsystems.com.au/prod.php?id=73`

Functions are provided to play the sound files present on the memory card, these functions are accessed through the `sound` object that is available through the **RPApi.h** file. Below is a list of available functions and their descriptions:

- `play()`
  Plays/pauses a song.
- `playSong(uint8_t idx)`
  Starts playing the song with the specified index (`idx`), the index is a reference to a file on the memory card having a filename containing the index number.
- `stop()`
  Stops playing.
- `next()`
  Skips to the next sound file with a greater index number.
- `prev()`
  Skips to the previous sound file with a smaller index number.
- `setVol(uint8_t level)`
  Sets the volume level, there are 7 levels from 1 to 7.
- `incVol()`
  Increases the volume level by one.
- `decVol()`
  Decreases the volume level by one.
- `getVol()`
  Returns the current volume level.
- `getNowPlaying()`
  Returns the current song playing.
- `isPlaying()`
  Returns true or false depending if a song is playing.

3.4 Movement

Each pin has a accelerometer and a gyroscope that allows to detect movements of 6 degrees of freedom (DoF). The object `movement` is available through the **RPApi.h** file which acts as a interface to the accelerometer and gyroscope, the available functions are:
Accelerometer:
The accelerometer values are given in multiples of the gravity ($n \times G$, where G is the acceleration of the gravity).

- `Rx()`
  Returns the magnitude of the acceleration in X direction.
- `Ry()`
  Returns the magnitude of the acceleration in Y direction.
- `Rz()`
  Returns the magnitude of the acceleration in Z direction.
- `R()`
  Returns the overall magnitude of the acceleration.

Angle position:
The rotation of the pin can be calculated relative to the gravity, when the pin is kept still. Three functions are available that return the angle in degrees:

- `Ax()`
  Returns the angle of the x axis relative to gravity.
- `Ay()`
  Returns the angle of the y axis relative to gravity.
- `Az()`
  Returns the angle of the z axis relative to gravity.

Gyroscope: The gyroscope measures the rotation around the pin axes:

- `Gyz()`
  Returns the rotation around the x axis (in quid/sec).
- `Gxz()`
  Returns the rotation around the y axis (in quid/sec).
- `Gxy()`
  Returns the rotation around the z axis (in quid/sec).

3.4.1 Calibration

Both the gyroscope and the accelerometer can be calibrated with the `movement.calibrate()` function, the pin should be standing upright and still during calibration. There shouldn't be any need for calibration, as calibration values are loaded on default when a pin is turned on (with `RP_loadCalibration()`). In order to calibrate the pin and overwrite the default calibration values, one can utilize the `RP_calibrate()` function.

3.5 Timers

A convenient way to create timers is to use the `timerManager` object, which is available through the **RPApi.h** file. A timer is created in order to execute a specific function after a specific time or periodically with a given interval. Three functions exists to create timers:

- `createOneShot(unsigned long start, void (*callback)())`
  Creates a timer to executes a function passed in as `callback()` after `start` milliseconds.

- `createNShot(unsigned long start, unsigned long interval, int shots, void (*callback)())`
  Creates a timer to execute a function periodically passed in as `callback()`, starting after `start` milliseconds, with an `interval` and is executed number of `shots`.

- `createPeriodic(unsigned long start, unsigned long interval, void (*callback)())`
  Creates a timer to executes a given function `callback()` periodically with an `interval`.

3.6 Communication

Two pins are able to communicate with each other with radio communications through a included XBee chip. A event-driven messaging system is present that allows programmers to define their own message structures, the only requirement is that the beginning of the message is a byte size message type (`msgType`). The message type is used to route the messages to subscribers.

Lets create an example of a pair of two pins, when one of the pin is shaken, the other one vibrates and changes color. First, create a message structure, where the first byte is the message type `msgType` and the second is the intensity of the shake.

```
#include "RPApi.h"
enum Message_type {MSG_SHAKEN}

struct Shaken_msg
{
  uint8_t msgType;
  uint8_t intensity;
} msgShaken;
```

The first line defines a enumerate[1] which declares the constant `MSG_SHAKEN` as the number 0, this will be our new message type number. Application defined message types can have numbers from 0 to 49. The second element defines the message structure and a object called `msgShaken`. Next we will take the message object we defined and assign some values to it and send it.

[1] For more info on Enumerates, take a look at C documentation

```
void onShakeDetected()
{
  msgShaken.msgType = MSG_SHAKEN;
  msgShaken.intesity = 1;
  comm.send((uint8_t*)&msgShaken, sizeof(struct Shaken_msg));
}
```

First we create the function `onShakeDetected()` which we will call when the pin is shaken. The function assigns the message type to the `msgShaken` and we will start with a intensity of 1. The last line, `comm.send(...)` sends the message to the other pin.

Now we need the other pin to receive the message and act upon it, first we need to subscribe to the message type `MSG_SHAKEN` in the setup function.

```
static void handleShakenMsg(Message_t* msg)
{
  Shaken_msg* message = (Shaken_msg*)msg->data;
  // Our logic goes here ...
}

static void Shaken::setup()
{
  comm.subscribe(MSG_SHAKEN, handleShakenMsg);
}
```

If we start looking at the line in `setup()`, `comm.subscribe(...)`, we are subscribing to all messages received with message type `MSG_SHAKEN` and the function `handleShakenMsg(...)` is executed with the message data as an argument.

In the `handleShakenMsg()` function the message data is converted into `Shaken_msg` which we can then use to access the message.

Now we have gone through both sending and receiving messages, here is then the complete example, which should be uploaded to both pins:

```
#include "RPApi.h"
enum Message_type {MSG_SHAKEN}

struct Shaken_msg
{
  uint8_t msgType;
  uint8_t intensity;
} msgShaken;

void onShakeDetected(uint8_t intesity)
{
  msgShaken.msgType = MSG_SHAKEN;
  msgShaken.intesity = intesity;
  comm.send((uint8_t*)&msgShaken, sizeof(struct Shaken_msg));
```

```
}

void onSampleTimer()
{
  movement.update();
  if (*movement.R() > 1.5)
  {
    float value = (*movement.R()-1.5)/(2);
    if (value < 0)
      value = 0
    if (value > 1)
      value = 1;
    onShakeDetected(value*255);
  }
}

static void setLight(uint8_t intesity, uint8_t red, uint8_t green, uint8_t
{
  float intesity = intesity/255.0;
  light.setAll(intesity*red, intesity*green, instesity*blue);
  light.update(); // Apply the light changes
}

uint8_t colorIdx = 0;

static void handleShakenMsg(Message_t* msg)
{
  Shaken_msg* message = (Shaken_msg*)msg->data;
  switch(colorIdx)
  {
    case 0:
      setLight(message->intesity, LIGHT_RED);
      break;
    case 1:
      setLight(message->intesity, LIGHT_GREEN);
      break;
    case 2:
      setLight(message->intesity, LIGHT_BLUE);
      break;
  }
  colorIdx = (colorIdx + 1) % 3;
}

static void Shaken::setup()
{
  timerManager.createPeriodic(50, 50, onSampleTimer);
  comm.subscribe(MSG_SHAKEN, handleShakenMsg);
}
```

Summary of Communication

- Define a message structure:

```
struct Message_msg
{
  char msgType;
  uint8_t command;
  uint8_t value;
};
```

- Send a message: Set the message type and send with `comm.send(uint8_t* data, uint8_t length)`.
- Receive a message: Subscribe to a message with `comm.subscribe(uint8_t msgType, void (*callback)())` where the message will be delivered to function pass in `callback()`.

Chapter 4

Questions and Answers

- **Running `jtagiceii.exe` gives me the error message "Failed to load .cac file. Unable to determine supported devices".**
  This error message appears on Windows 7, when it's not ran as administrator. To run as administrator, right click on the program and select "Run as administrator" in the drop down list.