Radiocrafts

Embedded Wireless Solutions

SPR Tutorial 2: Send Periodic Network Data

v1.0, August 2, 2018

Table of Contents

List of Tables

No table of figures entries found.

List of Figures

No table of figures entries found.

Abbreviations

Abbreviation	Description
SPR	Radiocrafts Software-Programmable RF Module
RIIoT	Radiocrafts Industrial Internet of Things

1 Introduction

In this tutorial, you will develop an application that joins to a network, sends periodic data, and responds to message from the coordinator.

You will learn:
- How to configure a network (frequency, channel, data rate)
- How to configure the module to start the auto-join process when it starts up
- How to be notified of network state changes (e.g. OFFLINE to ONLINE)
- How to send a message to the coordinator
- How to react to message received from the coordinator

This tutorial is a continuation of tutorial 1 where the basics of creating an application is introduced. So if you haven't, it is recommended to start there.

2 Hardware Setup

For this tutorial, you will need:
- RC1880 Sensor Board
- Button that connects to GPIO_8
- LED that connects to GPIO_0 (already on board)
- LED that connects to GPIO_1

You will also need a coordinator. You can use any IEEE 802.15.4g coordinator that supports 868 MHz. We recommend using Radiocraft's RC1880-GPR module with RIIoT Network Controller in a linux gateway.

3 Behavior

The behaviors we will be implementing are:
- Network settings: 868 MHz, 50kbps
- Automatically looks for a network to join when powering up
- a LED indicating the state of the network. ON = network online.
- Every 30 seconds, send a data packet to the coordinator. The first byte of the data packet is an incremental counter. The data packet is also sent when the button is pressed
- Use the first byte of the message to turn on or off a LED.

4 Coding

The tutorial will build up the application code step by step, but you can skip to see to the full source code in 4.10. You can also grab the full source code in the `example` folder, in the file `tutorial2_network_periodic_data.c`.

4.1 Include the SPR header file

Edit the source file `app.c`. Erase previous codes in the file and start again.

Needed for every app source, include the header file `spr_app.h` where all the platform APIs are declared.

```
#include "spr_app.h"
```

4.2 Map the Button and LED to GPIOs

Next we create constant macros to map the button and LED to the correct GPIO pins.

```
/********* Constants ***********/
#define BUTTON_0 GPIO_8
#define LED_0 GPIO_0
#define LED_1 GPIO_1
```

4.3 Declare the timer

We will declare a variable for our timer that tracks when to report data to the coordinator.

```
/********* File-scope Variables ****************/
static TimerId reportTimer;
```

4.4 The Setup()

Next, you write the function `Setup()`, which is called whenever the module starts.

Some of the code will look familiar from the previous tutorial. You will see new code that configures the network. See the code below and the commentaries.

```
void Setup()
{
  Debug.printf("My Setup\r\n");

  // Configure GPIO for Button 0
  GPIO.setDirection(BUTTON_0, INPUT);
  GPIO.setPull(BUTTON_0, PULL_UP);
  GPIO.setHandler(BUTTON_0, FALLING_EDGE, button0Handler);

  // Configure GPIO for LED 0
  GPIO.setDirection(LED_0, OUTPUT);
  GPIO.setValue(LED_0, LOW);

  // Configure GPIO for LED 1
  GPIO.setDirection(LED_1, OUTPUT);
  GPIO.setValue(LED_1, LOW);

  // Configure the network
  Network.setFreqBand(FREQ_868_MHZ); (1)
  Network.setDataRate(DATA_RATE_50_KBPS); (2)
  uint8_t channelMask[CHANNEL_BITMAP_SIZE] = {0x00,}; (3)
  channelMask[0] = 0xFF; //scans channels 0-7
  Network.setChannelMask(channelMask);

  Network.setNetworkStateChangeHandler(networkStateChangedHandler); (4)
  Network.setReceivedMessageHandler(receivedMessageHandler); (5)
  Network.setAutoJoin(true); (6)

  // create the timer to report data
  reportTimer = Timer.create(PERIODIC, 30*SECOND, reportData); (7)
}
```

(1) Choose between FREQ_868_MHZ and FREQ_915_MHZ
(2) Choose between DATA_RATE_5_KPBS and DATA_RATE_5_KBPS. DATA_RATE_5_KBPS is not part of the IEEE 802.15.4 standard, but it provides longer range.
(3) The channel mask determines the channels to scan when joining a network. Channels 0-33 are available when using 863 MHz; channels 0-128 are available when using 915 MHz. Each bit in the channel mask represents a channel. In this example, only the first 8 channels are scanned. So it initializes the whole array to 0, and only enables the first 8 bit.
(4) When the network state changes, it will call the function `networkStateChangedHandler`, which you will implement later.
(5) When a message is received, it will call the function `receivedMessageHandler`
(6) Enables auto-join, which will start the join process whenever the module powers up or whenever the network state becomes offline. If the join fails, the module periodically retries. Alternatively, if you don't enable auto-join, you can call `Network.join()` based on an user input (e.g. button push).
(7) Creates the timer that will call the function `reportData` every 30 seconds.

4.5 Handler for Network State Changes

We will now add the implementation for `networkStateChangdHandler()`, the function we registered within `Setup()` to be called whenever the network state changes.

In the handler, we will turn LED 0 on or off, start or stop the report timer according to the network state.

```
static void networkStateChangedHandler(NetworkState state)
{
   Debug.printf("Network State: %s\r\n", NetworkStateString[state]); (1)

   if (ONLINE == state) {
      Debug.printline("PanId=%2x Ch=%d ShortAddr=%2x", Network.getPanId(),
                      Network.getChannel(), Network.getShortAddress()); (2)
      GPIO.setValue(LED_0, HIGH); (3)
      Timer.start(reportTimer); (4)
   } else {
      GPIO.setValue(LED_0, LOW);
      Timer.stop(reportTimer);
   }
}
```

(1) `NetworkStateString[state]` returns a string for the network state enum. E.g. ONLINE
(2) This shows how to get the PAN ID, the channel, and node's short address after it joined the network.
(3) Turns LED on if network state is online
(4) Start the report timer if network state is online

The possible network states are `OFFLINE`, `JOINING`, `ONLINE`, `REJOININ`

4.6 Report Data Function

We will now implement the function that reports the data to the coordinator. This function is called by the timer every 30 seconds. It will also be called when the button is pressed.

Add the following code after `Setup()`.

```
static void reportData(void)
{
    Debug.printline("Report Data cnt=%d", dataCounter);
    uint8_t data[] = {dataCounter,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF}; (1)
    Network.send(sizeof(data), data); (2)
    dataCounter += 1; (3)
}
```

(1) Prepares a 7 byte data packet, where the first byte is the counter

(2) Note that you don't need a destination address. This is a star topology network. All data packets go to the coordinator.
(3) Increments the counter.

4.7 Button Handler

Implement the button handler, so that when it is pressed, it also calls `reportData()` to send a data packet to the coordinator.

```
static void button0Handler(enum GPIO_Pin pin, enum GPIO_InterruptEdge edge)
{
   Debug.printf("Button 0 Pressed!\r\n");
   NetworkState networkState = Network.getNetworkState();

   if (ONLINE == networkState) {
      reportData();
   }
}
```

4.8 Handle Received Message

Implement `receivedMessageHandler()`, where we registered in `Setup()` to handle received messages. The function takes 2 parmeters as input: the length of the message, and the message byte array.

It sets the state of LED 1 based on the first byte of the received message.

```
static void receivedMessageHandler(uint8_t len, uint8_t message[])
{
    Debug.printf("Received message len=%d data= ", len);
    Debug.printArray(len, message); (1)
    // set the LED 1 according to the 1st byte of the message
    if (message[0] == 0x01)
       GPIO.setValue(LED_1, HIGH);
    else
       GPIO.setValue(LED_1, LOW);
}
```

(1) This is a special debug print function to print a byte array in hex.

4.9 Declaring the Handlers

The compiler needs all the functions to be declared at the top of the file. So add the following declarations before `Setup()`

```
/********* Private Function Declarations ****************/
static void button0Handler(enum GPIO_Pin pin, enum GPIO_InterruptEdge edge);
static void networkStateChangedHandler(NetworkState state);
static void receivedMessageHandler(uint8_t len, uint8_t *message);
static void reportData(void);
```

4.10 Full Source Code

The full source code should look like below. You can also grab the full source code in the `example` folder, in the file `tutorial2_network_periodic_data.c`.

```
#include "spr_app.h"

/********* Constants ***********/
#define BUTTON_0 GPIO_8
#define LED_0 GPIO_0
#define LED_1 GPIO_1

/********* Private Variables ****************/
static TimerId reportTimer;
static uint8_t dataCounter = 0;

/********* Private Function Declarations ****************/
// These are event handlers that will be implemented later in the file, but
// are declared here so Setup() can reference them when registering events.
static void button0Handler(enum GPIO_Pin pin, enum GPIO_InterruptEdge edge);
static void networkStateChangedHandler(NetworkState state);
static void receivedMessageHandler(uint8_t len, uint8_t *message);
static void reportData(void);

/********* Public Functions ****************/

/**
* Setup() is called by the framework on startup
*/
void Setup()
{
   Debug.printf("User Setup\r\n");

   // Configure GPIO for Button 0
   GPIO.setDirection(BUTTON_0, INPUT);
   GPIO.setPull(BUTTON_0, PULL_UP);
   GPIO.setHandler(BUTTON_0, FALLING_EDGE, button0Handler);

   // Configure GPIO for LED 0
   GPIO.setDirection(LED_0, OUTPUT);
   GPIO.setValue(LED_0, LOW);

   // Configure GPIO for LED 1
   GPIO.setDirection(LED_1, OUTPUT);
   GPIO.setValue(LED_1, LOW);

   // Configure the network
   Network.setFreqBand(FREQ_868_MHZ);
   Network.setDataRate(DATA_RATE_50_KBPS);
   uint8_t channelMask[CHANNEL_BITMAP_SIZE] = {0x00,};
   channelMask[0] = 0xFF; //scans channels 0-7
   Network.setChannelMask(channelMask);
   Network.setNetworkStateChangeHandler(networkStateChangedHandler);
   Network.setReceivedMessageHandler(receivedMessageHandler);
   Network.setAutoJoin(true);

   // create the timer to report data
   reportTimer = Timer.create(PERIODIC, 30*SECOND, reportData);
}

/********* Private Functions ****************/
static void networkStateChangedHandler(NetworkState state)
{
   Debug.printline("Network State: %s", NetworkStateString[state]);

   // set the LED 0 according to network state
   if (ONLINE == state) {
      Debug.printline("PanId=%2x Ch=%d ShortAddr=%2x", Network.getPanId(),
                       Network.getChannel(), Network.getShortAddress());
      GPIO.setValue(LED_0, HIGH);
      Timer.start(reportTimer);
   } else {
      GPIO.setValue(LED_0, LOW);
      Timer.stop(reportTimer);
   }
}

static void reportData(void)
{
    Debug.printline("Report Data cnt=%d", dataCounter);
    uint8_t data[] = {dataCounter,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF};
    Network.send(sizeof(data), data);
    dataCounter += 1;
}

static void button0Handler(enum GPIO_Pin pin, enum GPIO_InterruptEdge edge)
{
   Debug.printline("Button 0 Pressed!");
   NetworkState networkState = Network.getNetworkState();

   if (ONLINE == networkState){
      reportData();
   }
}

static void receivedMessageHandler(uint8_t len, uint8_t message[])
{
    Debug.printf("Received message len=%d data= ", len);
    Debug.printArray(len, message);
    // set the LED 1 according to the 1st byte of the message
    if (message[0] == 0x01)
       GPIO.setValue(LED_1, HIGH);
    else
       GPIO.setValue(LED_1, LOW);
}
```

5 Compile and Test

Compile and flash the application using the same steps introduced in the Getting Started Guide.

Open a serial terminal so you can see the debugging statements.

Start a network on the coordinator. Open permit join on the network. Then restart your SDM module to check if it joins!

6 Revision History

Revision	Date	Changes
V1.0	August 2, 2018	For release

Radiocrafts Support:

Knowledge base: https://radiocrafts.com/knowledge-base/
Application notes library: https://radiocrafts.com/resources/application-notes/
Whitepapers: https://radiocrafts.com/resources/articles-white-papers/
Technology overview: https://radiocrafts.com/technologies/
RF Wireless Expert Training: https://radiocrafts.com/resources/rf-wireless-expert-training/

Contact Radiocrafts

Sales requests: https://radiocrafts.com/contact/