Small Bird Geolocating Radio Tag and Receiver System

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Ruoxuan Xu, Xiaoxing Yan

MEng Field Advisor: Joseph Skovira

MEng Outside Advisor: David Ward Winkler

Degree Date: December 2018

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Small Bird Geolocating Radio Tag and Receiver System

Author: Ruoxuan Xu(rx65), Xiaoxing Yan(xy363)

Abstract: This project is to implement communication functions for bird tags in a Small Bird Geolocating Radio Tag and Receiver System, which contains multiple bird tags and one base station. Wildlife tags attached to birds transmit birdID and twilight signals to base station in order to track the birds' locations without recapturing them. To implement this communication process, we change the chip from SI1060 to CC1310 Micro Controller Unit(MCU), which consumes less energy, with a RF core as the built-in chip for the bird tag to interact with the base station. We have developed the communication protocol, implemented the code for cc1310. We arrange those modes of reading, transmitting and sleeping as multiple threads and schedule them in the TI Real time operating system. By experimenting between finished tags and base station, we have the testing result for transmission accuracy with limited power consumption and varying distance.

Executive Summary

1. We concluded the communication pattern of birdtag geolocating system needed by Taber team, and redesigned the state-changing implementation in the context of TI RTOS.

2. We learned the functions and scheduling mechanism of TI RTOS on the chip cc1310, and implemented the program via TI RTOS environment.

3. On the cc1310 Launchpad, we write programs for 2 tasks and register them in RTOS to be controlled by the hardware interrupts. We substitute the Real Time Clock with buttons on the Launchpad to simulate the hardware interrupts. Once button1 is pressed, task 1 of receiving RF packet will be running; once button 0 is pressed, task 2 of reading the data from light sensor MAX44009 will be running. After tasks finish, the system goes back to standby mode.

4. We programmed to realize i2c initialization and i2c data transferring between cc1310 MCU and the light sensor.

5. We went on with long-term testing on the RF transmitting accuracy of cc1310 with different distance.

Contents

1. Introduction
2. Project Specifications
   2.1 project problems
   2.2 project requirements
3. Project Design
   3.1 Communication Protocol
   3.2 Tag State Diagram
4. Project Implementation
5. Testing and Results
6. Conclusions
   6.1 Achievements
   6.2 Acknowledgements
7. References
8. Appendix
   8.1 System Installation
   8.2 Code

1. Introduction

For this project, we are working with TABER team, which is a research group working on animal technology to reconstruct bird migration paths using a small, light-weight tag attached to them. These bird tags are supposed to periodically record the light signal and transmit twilight data with the unique bird ID to the neighboring base station, if any is met on their routes. From the collected signals the biologists can distinguish the twilight information and thus infer the whole migration routes of birds. As for the power reduction part, which managed by another group in TABER team, this tag can be powered by solar energy and saves energy in a supercapacitor for continuous mission for multiple days without solar charging.

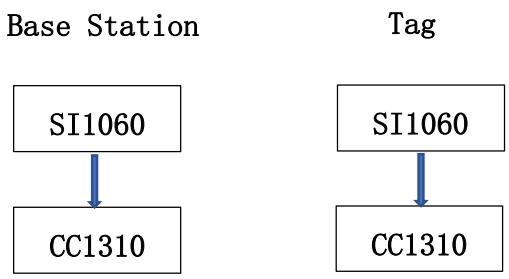

Fig 1. Inherit and Update Project

As the research team switches the microcontroller from Si1060 to CC1310, the main task to upgrade the communication system is to implement the program in the new MCU with both previous functions and newly designed features such as state switching or reading data from light sensor. Since the languages and libraries for Si 1060 and CC1310 are totally different, the project is from the ground up.

The team selected CC1310 MCU in order to save more energy while working, which with a RF core as the built-in chip for the bird tag to interact with the base station. We will develop the communication protocol, improve the code for cc1310 and experiment on finished tags and base station for transmission efficiency under very limited power supply.

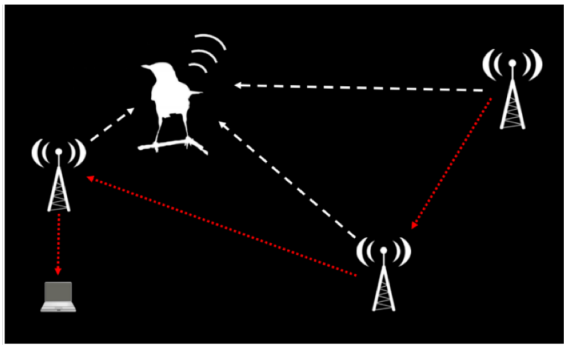

Fig 2. Interaction Between Bird Tag and Base Station

2. Project Specifications

2.1 Project Problems

As for our engineering part, we aim to build such a small and efficient bird tag. To make the bird tag portable and low energy consuming, the chip of cc1310 is selected as the main controller of the tag, which has a small package size of 7 mm × 7 mm and low standby current of 0.7 µA. Also, to make the frequency usable with a free license in the world-wide range, the transmission band is set around 434 MHz.

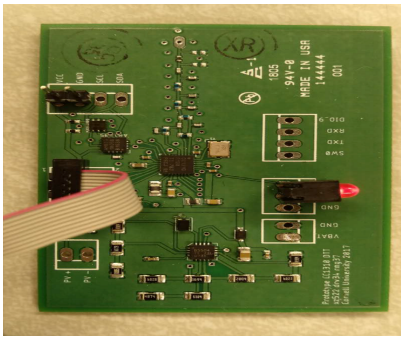

Fig 3. Bird Tag Prototype

The tag could transmit simple signals when working in SI1060 which chose before, however, after our team changing the chip, the software part is supposed to be rebuilt. As for the new added functions, this tag needs to transmit exact data reading from light sensor in packets with base station and change among different states now.

2.2 Project Requirements

Our basic object for this project is to update previous system with new chip and add more functions such as reading data from light sensor and sending bird IDs as well as sensor data to the base station if there is one in neighborhood. We also expect the bird tags have a life time in the field for as long as 1 year.

There are several main issues to be concerned when designing the communication protocol.

- The first one is to minimize the energy consumption of bird tags, as once they are attached to birds they can only get recharged by solar. To deal with that, we set the constant mode of the bird tag into deep sleep, only interrupted when it listens to the signal from the base station or reads the data from light sensor.
- The second one is to allow tag to change from one state to another state according to different signal. To solve this problem, we plan to set predefined time interval using Real Time Clock (RTC) or create hardware interrupt by receiving packets from base station to realize state changing.

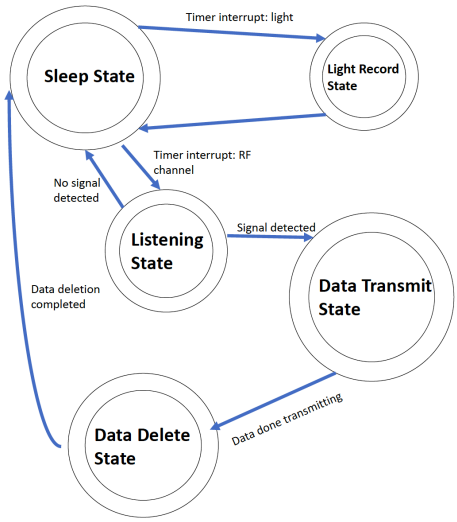

Fig 4. Tag State Diagram

- The third concern is to read data from light sensor. As the bird may not fly within the range of base station for a long time, the data should be saved in the chip steadily. Also, we are supposed to implement data checking to ensure the correctness of data after transmission.
- The fourth concern is the robustness of code, as the tag will not be debugged once put into use on birds, we need to guarantee that the program has a correct timer and reliable performance. Therefore, we would choose simple program routine without considering the time zone changing or twilight - normal light classifying during the

birds' voyages. Also, we plan to do 10 hours and 24 hours testing to observe its performance as well as accuracy.

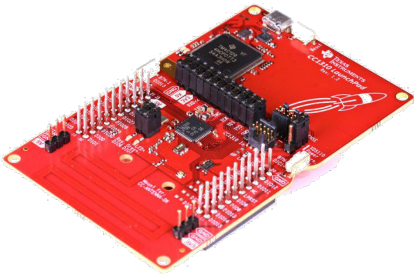

Fig 5. TI CC1350 MCU

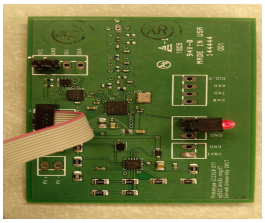

Fig 6. Bird Tag Prototype

3. Range of solutions

3.1 Selection of RF Microcontroller

The main controller in this project is supposed to contain RF transmitting modules and interface with the external RTC and light sensor, as well as the power system. We choose from 2 types of chips, si1060 of Silicon Labs and cc1310 of Texas Instrument (TI), which can both serve the functions of RF communication and integrating with the bird tag system. The sizes of the two chips are below 7 x 7 mm^2, so they are both applicable on a light, small bird tag. The frequency band we choose is 433+ MHz, which is an ultra- high frequency band in which license free communication devices are allowed to operate in a lot of regions all over the world range. These two chips would also meet these requirements with their frequency settings.

In terms of energy saving, we compared the current specifications relative to different states in our state machine, to find that among the 5 states all through one cycle, the RF transmitting (TX) and Receiving(TX) periods consume the major part of the power. Part of the reason why the previous choice of MCU by our team is si1060, is that they consume the least power in standby mode, which makes account for the energy taking at most time. However, due to the minute absolute current variance in standby mode between the two chips, the advantage of si1060 in standby current has no obvious effect. The lower RX and TX currents of cc1310 gains its superiority in this selection.

The statistics from the datasheet of cc1310 and si1060 is listed in the bellowing table. Comparing the specifications of these two chips, the chip of cc13x0 has the lower transmission current both in RX (5.4mA) and TX(13.4mA). As for power consumption from the MCU core in active mode, the TI cc1310 is 51 uA/MHz and the si1060 is 160uA/MHz. The standby current which the tag would spend most of the time at, is less than 1 uA for both two types, so the influence to the total consumption hardly matters and thus is not considered. Therefore, TI cc1310 matches better for our project.

Chip	Tx Current	Rx Current	Standby Current	CPU active current
cc1310	13.4 mA (10 dbm)	5.4 mA	700nA	0.4mA
si1060	18 mA (10 dbm)	10/13mA	50nA	0.16mA

Form 1. cc1310 VS si1060 in Power Consumption

3.2 Selection of Operating System

In programming on the MCU of cc1310 we choose, we would choose the operating system of Real Time (RTOS) or Non Real Time (NON-RTOS) to interface with our program and the TI drivers.

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application to process data as it comes in, typically without buffering delays. TI RTOS is a library of services that are commonly used in the TI devices to manage scheduling, inter-thread communications, power and peripheral drivers, so the system can have a deterministic latency and meet sure deadlines, as well as with other properties such as thread-safety, and standardized low-level programming. The kernel of RTOS is lightweight and event-driven, which means even if there are no tasks or threads running in the system, the kernel would not take 100% of the CPU, as it is not running until it is called in the application programs.

As users of RTOS, in developing our project of bird tag based on cc1310, we can focus on implementing each state in our designed state machine independently, as an individual thread, and then register each thread as an object, with their thread types and priorities, into RTOS in the main program. Those objects are self-contained and changing one of them does not affect others. Once we call the kernel of RTOS, the scheduler of RTOS will schedule all the threads for us with preemptive scheduling mechanism. Differing from the “time-sliced” operating system, this preemptive method always chooses the thread with highest

priority, until that thread finishes or is blocked. It is the users' work to design how and when each thread to be blocked or unblocked so they can run in the sequence as is expected.

If we do not choose TI RTOS and decide to develop the program routine on our own, we need to finish the thread context switching manually to convert from one thread to another. In each switching of the thread context, we need to store the state of this thread, so that it can be restored and execution resumed from the same point later. The specific settings on registers will vary depending on devices. Therefore, the more threads there are, the more complicatedly our program will expand, the harder it is to extend with new threads or modified sequences, or to transport the program to other type of chips. Especially in our project, there are two periodical interrupts to trigger different threads. One is to read the data from the light sensor, the other is to listen and detect the signals from the base station, which run at varying frequencies. Therefore if we implement the scheduling and set the priorities manually in the code, we need to deal with the potential nesting situation. These might lead to errors or lack of robustness of the program, as the period of each cycle might be changed further by users of our product.

Besides the concern of scheduling, the TI RTOS also has its advantage in power management. For the sake of energy saving, the idle mode in our system is designed to have as the lowest current as possible. In NON-RTOS, we can achieve such states by configuring certain registers in cc1310, which works, except that the code will not be portable to other devices. In TI RTOS, the system will manage the power consumption to the lowest automatically. When the task is blocked for more than 0.2 ms, RTOS will set the values of the related registers in cc1310 in the low-level code, packaged inside the library functions, and there is no need to add anything to our code while still the lowest-current state is reached in idle mode.

Based on the mentioned reasons, we choose TI RTOS as the environment of our program. This also brings challenges for us in learning RTOS and making use of its functions and mechanisms. As the former work of the bird tag was developed in NON-RTOS system on a

different type chip (MPS430), we would need to rewrite the whole project in RTOS and aim to implement the state diagrams in our communication design.

4. Project Design

4.1 Communication Protocol

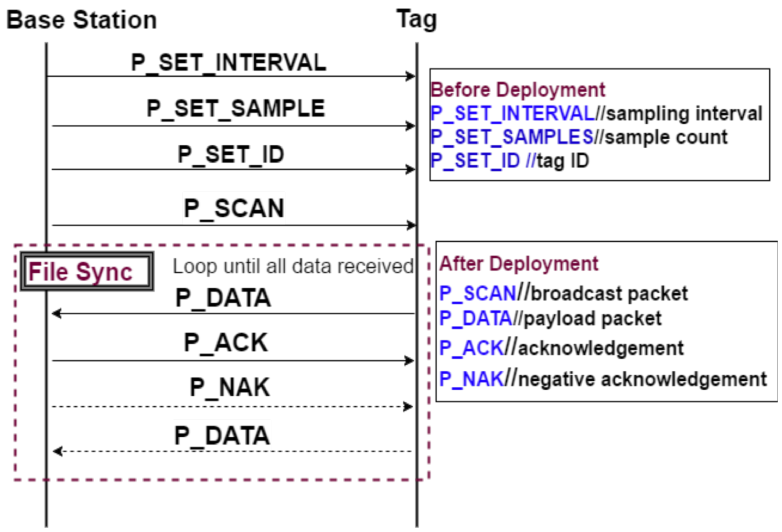

Figure 7. Overview of Initial Communication Protocol Design

As is indicated by Figure.1, The upper 3 packets are used for initializing the ID and sampling parameters for each tag at the begin point of their work, and then the lower 5 packets for data transmission are used when a bird tag get within the range of a base station, and start listening mode and then build communication with the receiver.

To be more specific on the process of data transmission, each tag is programed wake up from sleep mode, on a regular basis, to listen for a wakeup broadcast, P_SCAN, from the base station. Once the bird tag radio receives P_SCAN, it will send the first page of payload in the P_DATA packet. If the P_DATA packet is incomplete, the base-station should reply a P_NAK message to the tag so that the tag would send the same packet again. If the P_DATA packet is complete, the transmission is successful and the base-station should reply a P_ACK message. The tag would read the next page of payload from its flash and send it in a P_DATA packet. Both sides will stop the data transmission when end of payload is reached.

4.2 Tag State Diagram

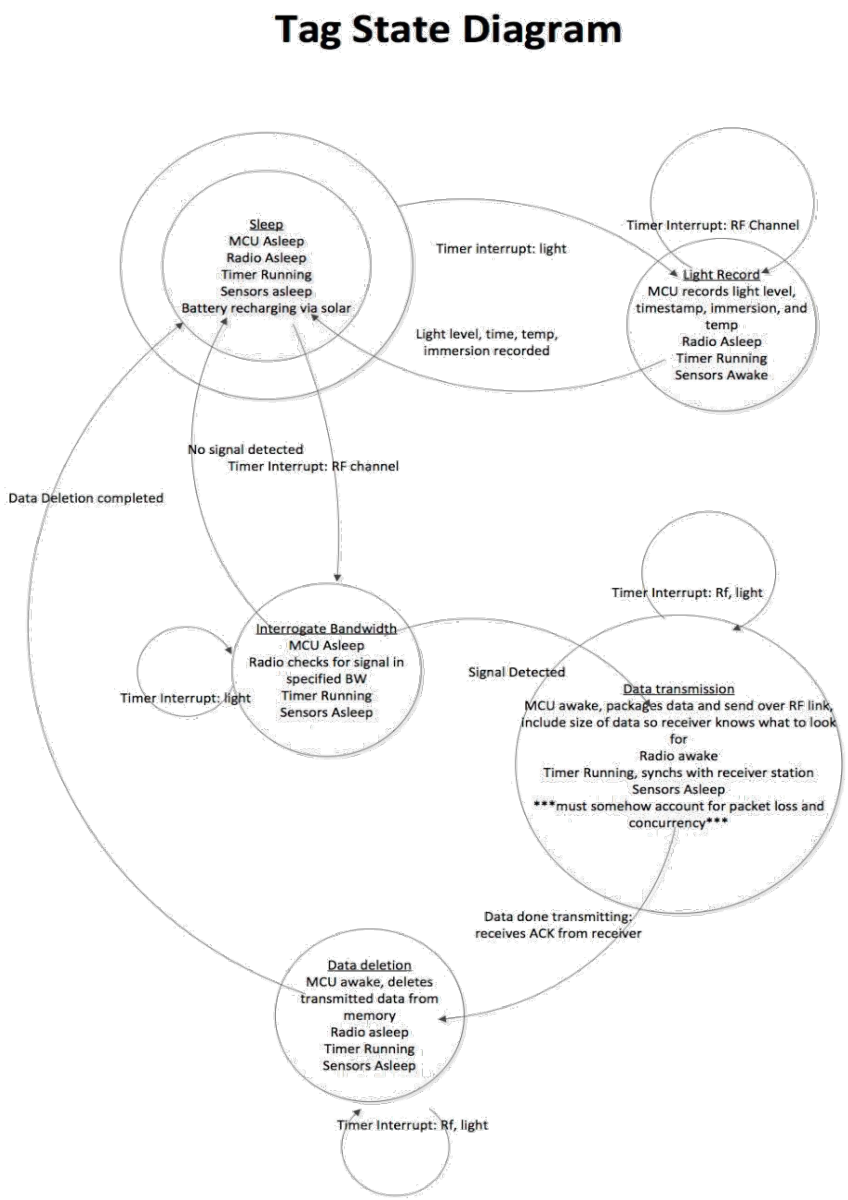

Figure 8. Overview of tag state switching

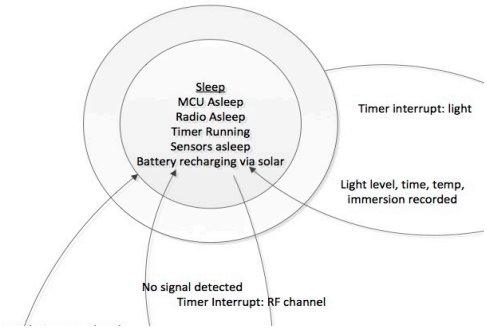

Fig 9. Sleep State

State 1:

The bird tag stays in sleep state at the most of time, which consumes less energy than other states in order to save energy. In this state, except for the timer, all the other parts are asleep. The timer is running to record the time. Two separate time intervals have been set, including the timer interrupt 'light' to wake the tag to read data detected by sensor, as well as the timer interrupt 'RF channel' to wake the tag to start the listening mode. It reminds the tag after the predetermined time interval.

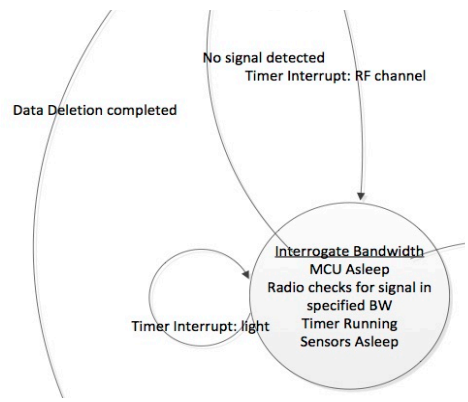

Fig 10. Listening State

State 2:

The bird tag will listen to the base station after predetermined time interval, that is, the tag opens RF channel and detects that whenever there is a packet sent by base station. If no signal detected, the tag will go back to sleep state, otherwise, it switches to data transmission state,

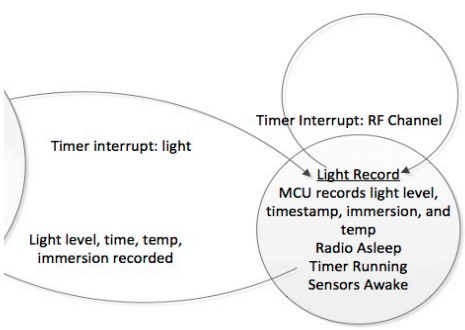

Fig 11. Light Record State

State 3:

The sensor is activated to detect light level and temperature, and the MCU is also activated to record these data. In this case, the sensor is active and measures the light intensity with a set sampling amount for each active process. The data recorded by the sensor is written into the on-board flash.

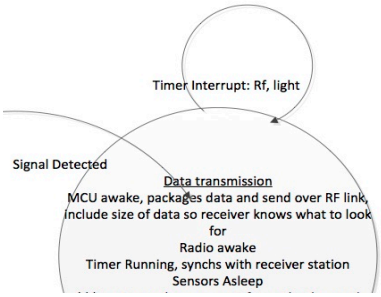

Fig 12. Data Transmission State

State 4:

Transmit data packets to base station. In this case, MCU is powered up to transmit package data, including light intensity, time stamp and bird ID.

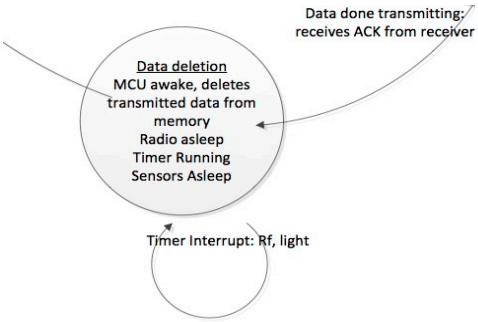

Fig 13. Data Delete State

State 5:

Once the tag has sent data completely, there is no need to keep old data. In this case, tag will delete transmitted data from memory and then return to the sleep state.

Figure 8 is the tag state diagram of our program. The main mode of cc1310 is 'sleep', and timer interrupts will switch the tag to another state with a set interval. This logic flow is simple and clean, which avoids many potential failures of code; also, the bird tag listens to the base station first before it sends any packet to base station. As we have mentioned, we just set the Tag to record the light intensity information with a set sampling period, without considering the time zone changing and difference between twilight signals and normal light signals, thus avoiding potential failure of code.

5. Implementing the design

To implement the plan, we use cc1310 as the micro controller of the bird tag. It is light, small and has ultra-low transmitting power. After configuring the parameters of Radio Frequency Module on cc1310, we run programs to call the RF transmission tasks by hardware interrupts from real-time Clock. The system will remain in sleep mode unless the real-time clock wakes it up to begin transmission or reading data from light sensor on a regular basis. We arrange those modes of reading, transmitting and sleeping as multiple threads and schedule them in the TI Real time operating system. The packet transmitted includes the birdID and data from the light sensor, which we integrate with cc1350 by I2C. In order to test the robustness of the system, we are doing the long-term testing over different distances and get the final result. In this part we will focus on implementing, and next section we will describe more on testing.

5.1 Ultra-Low power RF transmitting Configuration

We send RT commands to the RF module of the chip 1310 by setting the transmission properties, including the frequency band, rate, output power with the software SmartRF (a software to quickly configure RF modules in series of products of TI), which can automatically generate the set - up commands for RF modules. Then we imported these commands into CCS (Code Composer Studio, the IDE of CC1310) with modified parameters. By this we realized the packet sending and receiving between the 2 chips. Below shows the parameters we set and how the inter-chip transmitting works.

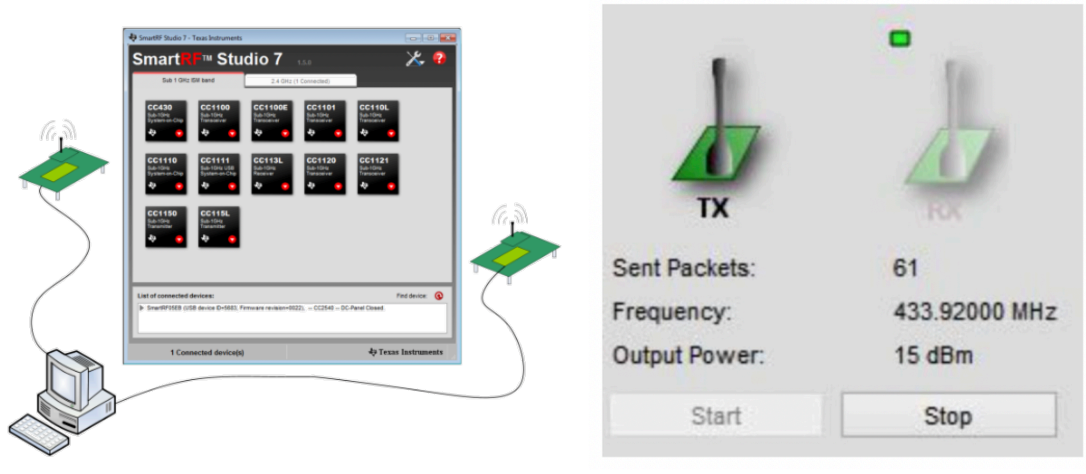

Fig14. Set RF Parameters with SmartRF Studio

5.2 Thread Scheduling and Hardware Interrupts via TI RTOS

We can create functions in the system, and then register them into a specific thread type. When they are given priorities and their stacks, they will be running as a thread. When we start from main, and launch the TI-RTOS kernel by 'BIOS.start()', the program will begin to execute the thread with the highest priority registered in the main program, until this thread is finished, blocked or get preempted by another thread with higher priority in the queue. If no threads are registered, the system goes to idle mode, which is also taken as a thread, but with lowest priority in TI RTOS. In such a preemptive scheduling mechanism, how can we pause the running thread after it finishes one cycle and restart that thread's new cycle when it is needed? We can suspend the thread by blocking the thread.

In TI RTOS, there are two common ways to block a thread, which is task_sleep() function and Semophore_pend() function. By blocking a thread, those threads with lower priorities can be executed, even if the blocked one is not finished, and differing from "killing" that thread, a block always comes with an unblock. For task_sleep() function, it sets the time to block the thread, and after the timeout the thread will be unblocked by TI RTOS automatically. For Semophore_pend() function, it waits for a particular semaphore to continue with the following steps, and it stays in the blocking mode if still waiting, is only unblocked when that semaphore it waits for is released by another threads in the system.

By implementing Semaphore, we can set those tasks of different states to be only set off when the specific semaphore is posted, so it can be set off only by hardware interrupt. The detailed scheduling implemented by our program is listed below, which contains 2 hardware interrupts (written as 'Hw1' below) and 2 tasks. Those are all threads in TI RTOS, except that only tasks could be blocked, while hardware interrupts must finish its interrupt service routine (ISR) once triggered by some external PIN changes from RTC. Therefore, in order to control the blocking of the 2 tasks with hardware interrupts from RTC, we let the Hw1 posted a semaphore in its ISR and let the task pend that semaphore with Semophore_pend() at the beginning of each iteration in its loop.

Threads	Priority	Function
Hw1	HIGHEST	Post Semophore A when there is an time interrupt from RTC
Hw2	HIGHEST	Post Semophore B when there is another time interrupt from RTC
Task1	1	Waits for Semophore A posted by RTC, then calls function of reading data from light sensor.
Task 2	2	Waits for Semophore B posted by RTC, and calls the function of listening to signals from the base stations.
Idle	0	The Standby (Sleep) mode of the bird tag.

Form2. Threads and Corresponding Functions

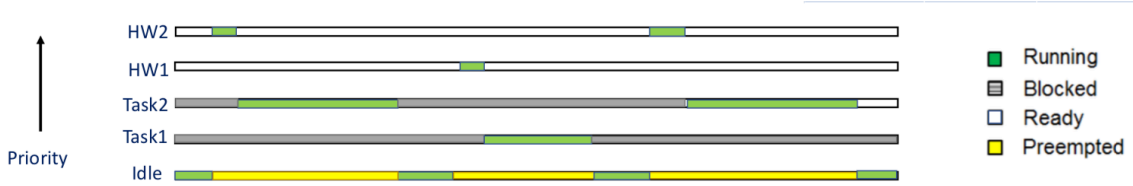

Fig15. Thread Switching diagram

As the table indicates, by TI RTOS scheduling we can create those task functions independently, such as to read lux data from the light sensor, or to start listening and transmitting process. Then we registered those tasks as threads in the main program and create their relative instance and putting them into RTOS. This makes our program expandable and portable, also avoids many potential failures in scheduling and thread synchronization.

5.3 I2C Implementation

We use i2c to connect the ambient light sensor MAX44009 with our cc1310 launchpad. We finished the program to initialize I2C driver, for the master (cc1350) to read the data from the light sensor (MAX44009) and for the cc1350 to display the data it reads to PC using UART, which was assembled in the library display.h.

Through the MAX44009 data sheet, we found the slave address of MAX44009. In the writing process, If A0 pin is connected to VCC, its address is "10010110”, or if the A0 pin is connected to the ground, its address is "10010100".

For MAX44009, every time before we read data of one byte, the register pointer must firstly be set through a write operation. Therefore, we send the slave address with the R/W set to 0, at the beginning, followed by the address of the register that needs to be read. After a Repeated START condition, send the slave address with the R/W bit set to 1 to initiate a read operation.

However, with this solution the program fail to function well in i2c in transferring. In order to figure out if the part of MCU in this i2c connection is working or not, we connected the SDA and SCL pin with Oscilloscope and plot the data we get.

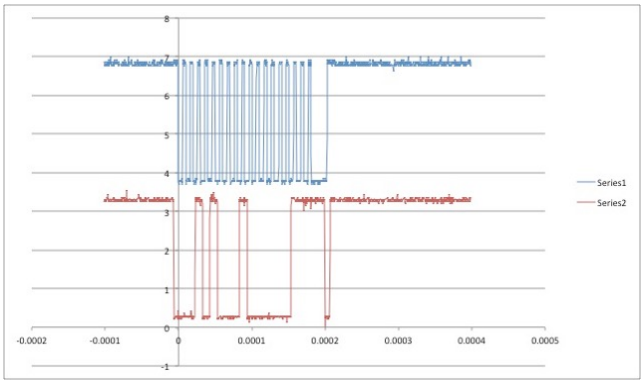

Fig16. SDL and SDA signals detected by Oscilloscope

This graph, with blue showing the SCL voltage and the red one showing the SDA voltage, indicates that the i2c signal triggered by cc1310 is functioning. Our failure type is usually caused by slave device not responding so you should check if your Slave device is well connected and functional good. After some retrying, we find one tricky problem is the address of slave on cc1350 i2c implementation. As we set the write/read operation in our code already, there is no need to include the last bit of slave address to indicate the 'R/W'. Therefore the slave address should be seven bit in our program, not the same as it is in MAX44009 datasheet.

Thus we succeed in reading one byte data of 'luxHigh' registers in MAX44009, we just goes on read the neighboring register of 'luxLow'. However, we do not set the register pointer to luxLow after we finish reading, but set the reading command to read 2 bytes of data from the register pointer we set at the first time. Then we can compute the final result of illumination data as the output of the light sensor from 'luxLow' and 'luxHigh'. If we read 'lucLow' by ending this transmission and restart another, then the data from 'luxLow' and 'luxHigh' we read may not be synchronized and thus causes error to the computed outcome. That is how we implemented i2c and read the data from the light sensor via i2c.

6. Testing and Results

In order to test the robustness of our system, we implemented CRC checking, which is an error-detecting code to detect accidental changes to raw data, into transmitting packets between two development boards of cc1350, we calculated the error rates of tests among different distances and time.

The platform we used is smartRF which can show the number of successful packets as well as failed packets automatically and calculate error rate for us directly.

We treated 0.1 second as one second to shorten the testing time and got the following result.

- Sending a packet per second, lasting for 10 hours

Distance (cm)	Error Rates
10	0%
15	1.1%
20	90.6 %

Testing time: 10 hours (1 hour in practice)

Number of transmitted packets: 7200

State: without antenna

Form3. Distance and Error Rates in 10-hour Test

- Sending a packet per second, lasting for 24 hours

Distance (cm)	Error Rates
10	0.2%
15	21.4%
20	92.5 %

Testing time: 24 hours (2.4 hour in practice)

Number of transmitted packets: 17280

State: without antenna

Form4. Distance and Error Rates in 24-hour Test

According to the result, we found that the distance matters a lot. Longer distance will cause higher error rate. However, in practice, the tag is supposed to attach an antenna which will increase distance definitely.

7. Conclusions

7.1 Achievements

1. We concluded the communication pattern of birdtag geolocating system needed by Taber team, and redesigned the state-changing implementation in the context of TI RTOS.

2. We learned the functions and scheduling mechanism of TI RTOS on the chip cc1310, and implemented the program via TI RTOS environment.

3. On the cc1310 Launchpad, we write programs for 2 tasks and register them in RTOS to be controlled by the hardware interrupts. We substitute the Real Time Clock with buttons on the Launchpad to simulate the hardware interrupts. Once button1 is pressed, task 1 of receiving RF packet will be running; once button 0 is pressed, task 2 of reading the data from light sensor MAX44009 will be running. After tasks finish, the system goes back to standby mode.

4. We programmed to realize i2c initialization and i2c data transferring between cc1310 MCU and the light sensor.

5. We went on with long-term testing on the RF transmitting accuracy of cc1310 with different distance.

7.2 Acknowledgement

We would like to thank Professor Joseph Skovira, Professor David Ward Winkler, Professor Rich Gabrielson and other TABER members, for their instruction and advice on our project. Their insights always keep our project in the right direction and help us make great progresses. We really appreciate their help to our project.

8. References

- CC1310 SimpleLink™ Ultra-Low-Power Sub-1 GHz Wireless MCU Manual
- CC13x0, CC26x0 SimpleLink™ Wireless MCU--Technical Reference Manual
- SYS/BIOS On-Line Training

9. Appendix

9.1 System Installation

- Install smartRF
- Install Code Composer Studio

9.2 Code

```
/*
 * This program is modified from pinShutdown_CC1350_LAUNCHXL_tirtos_ccs in
Simplelink Examples of cc1350.
 *
 * This is the main program of our object, which will trigger the listening phase by
pressing button 1 (DIO14 on the development
 * board of cc1350) and trigger the reading phase by pressing button 0 (DIO13).
 *
 * The LEDs on develop board are initialized to be off, when button 0 is pressed,
the i2c transfer begins and the
 * red LED toggles for 20 times to read the light data and store it in an array
variable of lux[20]. When button 1 is
 * pressed, the green LED is on to indicates that it is waiting for a packet. Once
the packet is received, the green LED
 * will be off and the red LED toggles to indicate a successful receiving and
toggles again to show transmitting a packet.
 * After that it will go back to sleep mode.
 *
 * When buttons are pressed, the BIOS system will release a semaphore (listenSem or
readSem), corresponding to the specific thread
```

```
 * it calls, and the thread that is waiting for the semaphore will catch it so as to
move on to the following steps.
 * To avoid overlapping of listening state and reading data state, during the cycle
of each thread, the Hardware interrupt is
 * disabled by PIN_setConfig in buttonCb. After finishing one cycle, the thread
(listenThread or readThread) will call PIN_setConfig
 * to enable the Hardware Interrupt on each other, so then main program catches the
semaphore and enable hardware interrupts again.
 */

int main(void)
{
    Power_init();
    PIN_init(BoardGpioInitTable);
    Board_initGeneral();

    hButtons = PIN_open(&buttonState, ButtonTable);  // let every registers in the
table configured as the table said
    PIN_registerIntCb(hButtons, buttonCb);  // when the button becomes the same state
as in the 4th column of the ButtonTable
    // hardware interrupt happened and buttonCb is called.

    Clock_Params clockParams;
    Clock_Params_init(&clockParams);
    clockParams.arg = (UArg)hButtons;  //   hButtons are trigger of the clock
    Clock_construct(&buttonClock, buttonClockCb, 0, &clockParams);
    hButtonClock = Clock_handle(&buttonClock);

    /* The listening task initialization*/
```

```
Task_Params listenTaskParams;
Task_Params_init(&listenTaskParams);
listenTaskParams.stack = listenTaskStack;
listenTaskParams.priority = 1;
listenTaskParams.stackSize = sizeof(listenTaskStack);
Task_construct(&listenTask, (Task_FuncPtr)listenThread, &listenTaskParams, NULL);

Semaphore_Params semParams;
Semaphore_Params_init(&semParams);
semParams.mode = Semaphore_Mode_BINARY;
Semaphore_construct(&listenSem, 0, &semParams);




/* The Reading light sensor task initialization*/

Task_Params readTaskParams;
Task_Params_init(&readTaskParams);
readTaskParams.stack = readTaskStack;
readTaskParams.priority = 2;
readTaskParams.stackSize = sizeof(readTaskStack);
Task_construct(&readTask, (Task_FuncPtr)readThread, &readTaskParams, NULL);

Semaphore_Params semParam2;
Semaphore_Params_init(&semParam2);
semParam2.mode = Semaphore_Mode_BINARY;
Semaphore_construct(&readSem, 0, &semParam2);



BIOS_start();
```

```
}



/*
 * This program is used to initialize I2C driver, for the master (cc1350) to read
the data from the light sensor ( MAX44009)
 * and for the cc1350 to display the data it reads to PC using UART, which was
assembled in the library display.h.
 *
 * The slave address of MAX44009:
 * |A0      SLAVE ADDRESS FOR Writing    SLAVE ADDRESS FOR Reading  |
 * |GND           10010100                    10010101            |
 * |VCC           10010110                    10010111            |
 *
 * As the last bit of address is to indicate R/W, we set MAX_ADDR as 8-bit number
"10010100" with the write digit set.
 * If A0 is connected to VCC, change it to "10010110". The #define is implemented in
Board.h
 *
 * We connect A0 to ground and set the slave address to be 0x4a (only contains 7 bit
regardless of the last bit)
 *
 * For MAX44009, we read a byte of data, the register pointer must at first be set
through a write operation
 * Send the slave address with the R/W set to 0, followed by the address of the
register that needs to be read.
 * After a Repeated START condition, send the slave address with the R/W bit set to
1, to initiate a read operation
```

```
 *
 * The register address of MAX44009
 * luxHigh: 0x03
 * luxLow:  0x04
 *
 * The following code includes Display.h for sake of debugging. In debugging,
connect the launchpad cc1350 to PC with CCS
 * software, and clicked "view" on top of CCS Edit mode to open the serial port.
click the right blue icon on the terminal,
 * choose terminal as "serial terminal" and choose the serial port as "" (which may
vary from different devices), then the
 * display can be seen through the terminal output.
 *
 * In actual use of this code, comment out all 'display' statements and shorten the
sleep time.
 */

void *readThread(void *arg0)
{
    unsigned int    i;
    uint16_t        lightLevel;
    uint8_t         txBuffer[1];
    uint8_t         rxBuffer[2];
    I2C_Handle      i2c;
    I2C_Params      i2cParams;
    I2C_Transaction i2cTransaction;

    /* Call driver init functions */
    Display_init();
    GPIO_init();
```

```
I2C_init();

/* Configure the LED pin */
GPIO_setConfig(Board_GPIO_LED0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);

/* Open the HOST display for output */
display = Display_open(Display_Type_UART, NULL);
if (display == NULL) {
    while (1);
}

/* Turn on user LED */
GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
Display_printf(display, 0, 0, "Starting the MAX44009 example\n");

/* Create I2C for usage */
I2C_Params_init(&i2cParams);
i2cParams.bitRate = I2C_100kHz;
i2c = I2C_open(Board_I2C_TMP, &i2cParams);
if (i2c == NULL) {
    Display_printf(display, 0, 0, "Error Initializing I2C\n");
    while (1);
}
else {
    Display_printf(display, 0, 0, "I2C Initialized!\n");
}




/* Point to the T ambient register and read its 2 bytes */
```

```
    txBuffer[0] = MAX_Lux_High;
    i2cTransaction.slaveAddress = Board_MAX_ADDR; // Board_MAX_ADDR is identified in
Board.h
    i2cTransaction.writeBuf = txBuffer;
    i2cTransaction.writeCount = 1;
    i2cTransaction.readBuf = rxBuffer;
    i2cTransaction.readCount = 2;

/* only for checking if i2c is fuctioning */
//    if (I2C_transfer(i2c, &i2cTransaction)){
//        Display_printf(display, 0, 0, "Config read!\n");
//        Display_printf(display, 0, 0, "Conf %u:  0x%x 0x%x\n", rxBuffer[0],
rxBuffer[1]);
//    } else {
//        Display_printf(display, 0, 0, "Config read fail!\n");
//    }



    while (1) {

        Semaphore_pend(Semaphore_handle(&readSem), BIOS_WAIT_FOREVER);

        /* Take 20 samples and print them out onto the console */
        for (i = 0; i < 20; i++) {
            if (I2C_transfer(i2c, &i2cTransaction)) {
                /* Extract LIGHT LEVEL from the received data; see MAX44009 datasheet
*/
                int exponent = (rxBuffer[0] & 0xf0) >> 4;
                int mant = (rxBuffer[0] & 0x0f) << 4 | rxBuffer[1];
```

```
                lightLevel = (float)(((0x00000001 << exponent) * (float)mant) *
0.045);
                lux[i] = lightLevel;

               Display_printf(display, 0, 0, "Sample %u: %d (C)\n", i, lux[i]);

    /* A different computing method:  https://github.com/herrfz/openwsn-
fw/blob/master/bsp/chips/max44009/max44009.c   */

    //            exponent = (( max44009_data[0] >> 4 )  & 0x0F);
    //            mantissa = (( max44009_data[0] & 0x0F ) << 4) | ((max44009_data[1] &
0x0F));
    //
    //            result = ( (uint16_t) exponent << 8 ) | ( (uint16_t) mantissa << 0);
    //
    //            return result;

            }
            else {
                Display_printf(display, 0, 0, "I2C Bus fault\n");
            }

            /* Sleep for 1 second */
            GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
            sleep(0.5);
            GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
            sleep(0.5);
        }
    }
```

```
    /* Deinitialized I2C */
    I2C_close(i2c);
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_OFF);
    Display_printf(display, 0, 0, "I2C closed!\n");

    return (NULL);
}



/*
 * This program is transformed from rfEchoRx_CC1350_LAUNCHXL_tirtos_ccs
 *
/

  while(1)
    {
        /* Wait for a packet
         * - When the first of the two chained commands (RX) completes, the
         * RF_EventCmdDone and RF_EventRxEntryDone events are raised on a
         * successful packet reception, and then the next command in the chain
         * (TX) is run
         * - If the RF core runs into an issue after receiving the packet
         * incorrectly onlt the RF_EventCmdDone event is raised; this is an
         * error condition
         * - If the RF core successfully echos the received packet the RF core
         * should raise the RF_EventLastCmdDone event
         */
        Semaphore_pend(Semaphore_handle(&listenSem), BIOS_WAIT_FOREVER);
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED1, 1);
```

```
RF_EventMask terminationReason =
        RF_runCmd(rfHandle, (RF_Op*)&RF_cmdPropRx, RF_PriorityNormal,
                echoCallback, (RF_EventRxEntryDone |
                RF_EventLastCmdDone));

switch(terminationReason)
{
case RF_EventCmdDone:
    // A radio operation command in a chain finished
    break;
case RF_EventLastCmdDone:
    // A stand-alone radio operation command or the last radio
    // operation command in a chain finished.
    break;
case RF_EventCmdCancelled:
    // Command cancelled before it was started; it can be caused
    // by RF_cancelCmd() or RF_flushCmd().
    break;
case RF_EventCmdAborted:
    // Abrupt command termination caused by RF_cancelCmd() or
    // RF_flushCmd().
    break;
case RF_EventCmdStopped:
    // Graceful command termination caused by RF_cancelCmd() or
    // RF_flushCmd().
    break;
default:
    // Uncaught error event
    while(1);
}
```

```
uint32_t cmdStatus = ((volatile RF_Op*)&RF_cmdPropRx)->status;
switch(cmdStatus)
{
case PROP_DONE_OK:
    // Packet received with CRC OK
    break;
case PROP_DONE_RXERR:
    // Packet received with CRC error
    break;
case PROP_DONE_RXTIMEOUT:
    // Observed end trigger while in sync search
    break;
case PROP_DONE_BREAK:
    // Observed end trigger while receiving packet when the command is
    // configured with endType set to 1
    break;
case PROP_DONE_ENDED:
    // Received packet after having observed the end trigger; if the
    // command is configured with endType set to 0, the end trigger
    // will not terminate an ongoing reception
    break;
case PROP_DONE_STOPPED:
    // received CMD_STOP after command started and, if sync found,
    // packet is received
    break;
case PROP_DONE_ABORT:
    // Received CMD_ABORT after command started
    break;
case PROP_ERROR_RXBUF:
```

```
            // No RX buffer large enough for the received data available at
            // the start of a packet
            break;
        case PROP_ERROR_RXFULL:
            // Out of RX buffer space during reception in a partial read
            break;
        case PROP_ERROR_PAR:
            // Observed illegal parameter
            break;
        case PROP_ERROR_NO_SETUP:
            // Command sent without setting up the radio in a supported
            // mode using CMD_PROP_RADIO_SETUP or CMD_RADIO_SETUP
            break;
        case PROP_ERROR_NO_FS:
            // Command sent without the synthesizer being programmed
            break;
        case PROP_ERROR_RXOVF:
            // RX overflow observed during operation
            break;
        default:
            // Uncaught error event - these could come from the
            // pool of states defined in rf_mailbox.h
            while(1);
        }
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED1, 0);
    }
}

static void echoCallback(RF_Handle h, RF_CmdHandle ch, RF_EventMask e)
{
```

```
#ifdef LOG_RADIO_EVENTS
    eventLog[evIndex++ & 0x1F] = e;
#endif// LOG_RADIO_EVENTS

    if (e & RF_EventRxEntryDone)
    {
        /* Successful RX */
        /* Toggle LED2, clear LED1 to indicate RX */
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED1, 0);
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED2,
                           !PIN_getOutputValue(Board_PIN_LED2));

        /* Get current unhandled data entry */
        currentDataEntry = RFQueue_getDataEntry();

        /* Handle the packet data, located at &currentDataEntry->data:
         * - Length is the first byte with the current configuration
         * - Data starts from the second byte */
        packetLength      = *(uint8_t *)(&(currentDataEntry->data));
        packetDataPointer = (uint8_t *)(&(currentDataEntry->data) + 1);

        /* Copy the payload + status byte to the rxPacket variable, and then
         * over to the txPacket
         */
        memcpy(txPacket, packetDataPointer, packetLength);

        RFQueue_nextEntry();
    }
    else if (e & RF_EventLastCmdDone)
    {
```

```
        /* Successful Echo (TX)*/
        /* Toggle LED2, clear LED1 to indicate RX */
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED1, 0);
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED2,
                           !PIN_getOutputValue(Board_PIN_LED2));

    }
    else // any uncaught event
    {
        /* Error Condition: set LED1, clear LED2 */
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED1, 1);
        PIN_setOutputValue(ledPinHandle, Board_PIN_LED2, 0);
    }
}
```