Cognitive Radio

Krishna Sai Anirudh Katamreddy

Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 95112
E-mail: anirudh.ani.ani@gmail.com

Abstract

This report will document the design and implementation of wired and wireless communication between two embedded systems. Initial explanation talks on setting up the hardware and implementation of LISA to send and receive payload, Improving the process by implementing scrambling and de-scrambling. The main goal is to be able to implement a reliable cost effective and higher performance communication using LoRa 1276 module connected to the LPC1769, and improve the performance on changing the parameter like Spreading factor [SF], transmission power, frequency hopping etc, in real time on observing the performance .

1. Introduction

Hardware has to be setup by connecting the pins of the components as given in the pin diagram. Initially, establishing of hand shaking is necessary, to make sure the built receive and transmit systems are working. For sending and receiving some important data, we need to make sure that Ni and Nj are in synchronous, for that we are using LISA algorithm, through which we can retrieve the payload sent. The GPIO pins will be used from the LPC1769 so we can set and clear the pins to simulate the binary pattern being sent as the payload. Data has been sent using wired, wireless using RF 433Mhz module and LoRa 1276 state of art module. Observed the performance and builded a software which can improve its performance in its real time by itself.

2. Methodology

This report will consist of a detailed overview of the both the hardware design and software implementation of communication between two embedded systems This report will also give a comprehensive analysis of the data being tested through both wired and wireless communication. Improving the performance by implementing few techniques like increasing the bandwidth by increasing the spreading factor, power of the signal on observing the receiving data's delay in packet, CRC, RSSI, water marks etc.

2.1. Objectives and Technical Challenges

The main objectives of this project are listed as follows:

1. Building an Embedded system which can perform communication.
2. Implement LISA algorithm in C programming.
3. Establish a wired communication between two platforms (Ni and Nj)
4. Establish a wireless communication between two platforms (Ni and Nj) using RF module.
5. Establish a wireless communication between two platforms (Ni and Nj) using LoRa module.
6. Improve the performance of the communication and increase the received signal strength indication

One of the technical challenges we faced was to know how different parameters influence the performance of the communication and improving it in real time.

3. Implementation

The implementation of end to end communication can be mainly divided into two sections namely hardware designing and software implementation.

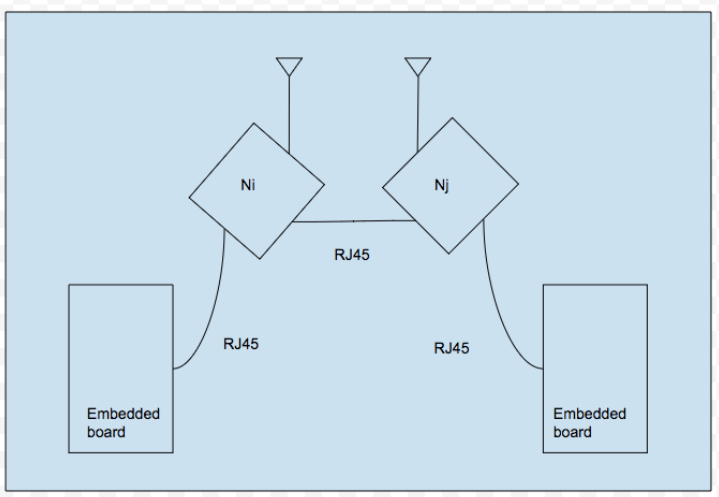
3.1. Hardware Design

The hardware platform consists of a pair of embedded system board and wireless module. Each pair is connected via 8 pin RJ45 cat5 Ethernet cable and there are two ways to communicate across the pair:

1. Wired data transfer(Landline connection):
We have used Wired – 8 pin RJ45 cat5 Ethernet cable connected to the input and output pins of LPC1769, between transmitter and receiver board.

2. Wireless data transfer (RF module):
We have used Wireless – 433Mhz ASK TX and RX to send and receive data wirelessly, and we have used a switch to establish landline to wireless connection.

System Block Diagram
Connecting two embedded boards to Node i and Node j, through which we can establish connection.

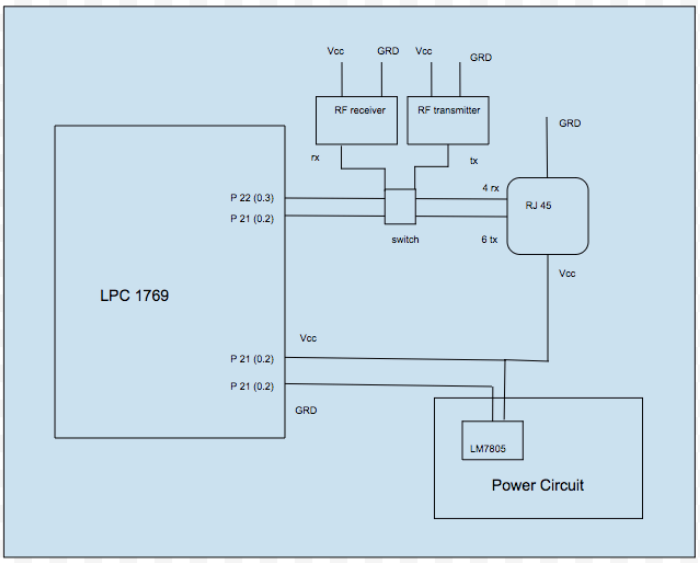


Bill of Material

Sr No.	Item	Quan-tity	Description
1	LPC1769	1	Micro con-troller
2	RF Trans-mitter & Re-ceiver	1 each	433Mhz , ASK <1Kbps speed
3	RJ45 connectors	2	8 pin Con-nectors
5	Ethernet cable	2	cat5 cable 8 pins
6	Power Adapter	1	9V, 1.65A rating wall adapter
7	LM7805	1	Power Regula-tor
8	Switch	1	RJ45 vs RF
11	USB cable	1	For con-nection with LPC1769
12	LoRa 1276	1 each	137 MHz to 1020 Mhz

Schematic Design

The diagram explains, how the circuit has been built between each of the embedded board and node i or node j RF board. A push button switch is used on each



of the board in order to switch between wired or wireless connection.

Pin Table:

Pin No	Description	Note
5	Power (3.3v - 5v)	External power
2 & 3	Ground	Common ground (J-21)
4	Data pin tx pin	P0.3 (J2-22)
5	Data pin rx pin	P0.2 (j2-21)

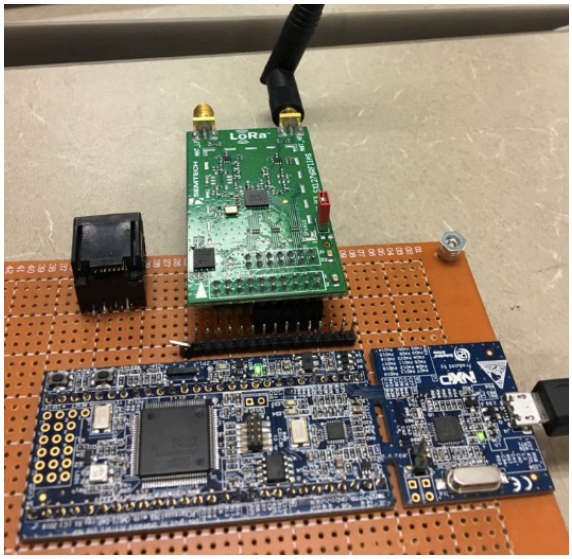
LoRa module (State of Art):

LoRa has to be connected with Serial Peripheral Interface, in order to establish connection, the connections are made using the following pin diagram

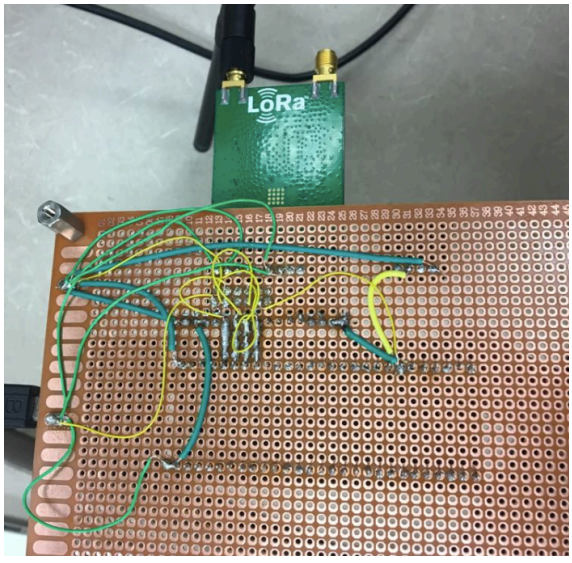
LoRa Hard-ware Image:

LoRa connected to LPC1769 board

Top view:



Bottom view:



PIN Connection Table:

LoRa SX 1276	LPC 1769	Note
P.1	J2.7	SCK
P.2	J2.28	Vcc 3.3V
P.3	J2.5	MOSI 1
P.4	J2.1	Ground
P.7	J2.8	SSEL 1
P.8	J2.6	MISO 1
P.10	J2.21	GPIO
P.22	J2.28	Vcc 3.3V
P.24	J2.54	Ground
P.32	J2.54	Ground
P.34	J2.28	Vcc 3.3V

3.2. Software Design

3.2.1 LISA implementation:
The software path for transmitting and receiving data is written in C, Implementation of the LISA (Linear Independent Synchronization Algorithm) is done for transfer and retrieval of payload, in a very constructive way.

While transmitting, initially few arbitrary bits are to be sent, followed by synchronous field and payload. Synchronous field is an important lock to be checked while receiving the data, to verify and confirm the length, location and start bit of the payload.

Creating data file and transmission:
We are sending a total of 1024 bits (128 bytes), in which 256 bits (32 bytes) are taken as our ‘Synchronous field’, then followed by one byte which specifies ‘payload length’ and ‘payload’ starts right after this byte. I have created a buffer, to store all the bits in the sequence I wanted to for transmission. Remaining bits we are sending some random arbitrary values, which can go up to a maximum size of [1024 - sync field bits - payload length]. After creating the data buffer, we are converting all the bytes into bits and storing then into a buffer and transmission is done through the port P0.3.

Corrupting the synchronous field:
We have written a function to corrupt the synchronous field ourselves depending on the corruption rate we

specify, which can help in checking the confidence level on receiving the data we sent.

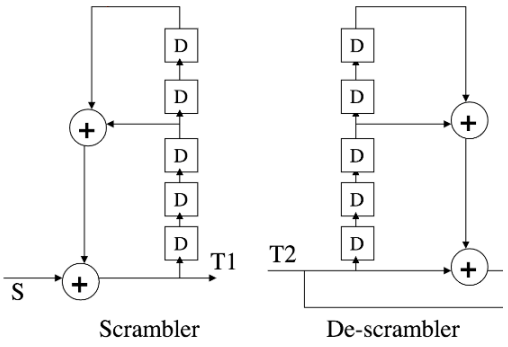
Receiving the data file and matching for confidence:
Initially, On receiving each bit from the transmission end, each bit is read and stored to the buffer. We have created a kernel of synchronous field bits, and start comparing all the bits on the kernel with the data in received buffer, and the kernel is shifted by one bit right, if there is no kernel match. On finding the kernel match with, we can assure that there is a match in the synchronous field and the byte next to it is the "Payload length" and read the payload from the receive buffer one by one until the payload ends. As we know the payload length, we can find where the payload ends.

Depending on the confidence level, we can check for the required number of bits to match, initially we have considered 100% confidence level, in which all the bits in the synchronous field have to match, on reducing confidence level, we can compute the number of bits to be matched and can extract payload even if there are certain amount of corruption bits.

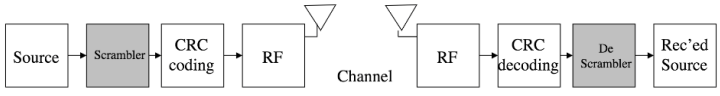
Scrambling and De-scrambling:

The purpose of scrambling and de-scrambling is to convert an input string into a seemingly random output string of the same length (e.g., by pseudo-randomly selecting bits to invert), thus avoiding long sequences of bits of the same value; in this context, a randomizer is also referred to as a scrambler.

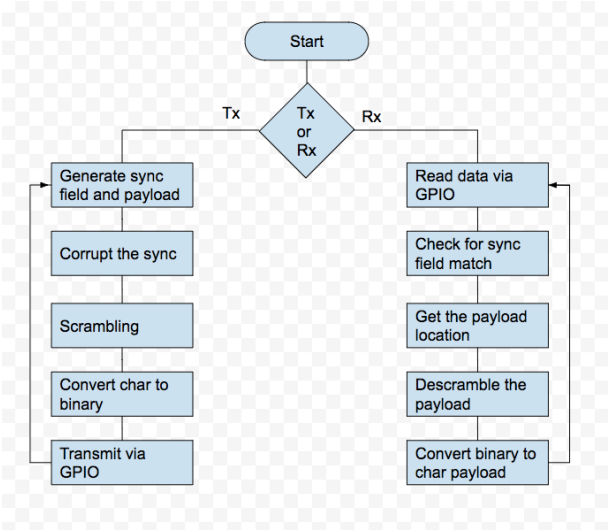
The image below is of order 5,



The data undergoes the below given process, on the transmission side as well as the receiving side, when scrambling and descrambling is done



1. **Flow chart:**



2. **Algorithm:**

The entire code can be divided into 8 modules.

I. Create data: 128 bytes of data is converted to 1024 bit stream. All characters are converted into binary, pattern bits are finally added to the data for synchronization. This is how the data is then transmitted.

II. Corrupting the synchronous field: 32 bytes of pre-defined sync field has to be converted to 256 bits on binary data, some of those bits can be corrupted depending on the corruption level.

III. Scrambling: The payload and the payload length are passed to a 5 order scrambler and the output bits are to be transmitted.

IV. Transmit Data: in this module, all the 1024 bits created in the section prior are sent by setting or clearing the LPC1769 0 port 3 pin.

V. Receive Data: in this module the data bit stream is received, the data is read through the LPC1769 0 port 2 pin.

VI. Synchronous field matching: From the received bits, all the bits are checked bitwise for the 256 bits sync field to match. The number of bits match can be varied depending on the confidence level.

VII. De-scrambling: The payload and the payload length which are scrambled are descrambled to get the original data.

VIII. Decode and get the payload: On finding the sync field as told in the previous step, the next bit is the length of the payload and reading the (payload length * 8) bits next to the payload length byte, and convert them into ASCII to get the character and store them to string can retrieve the payload.

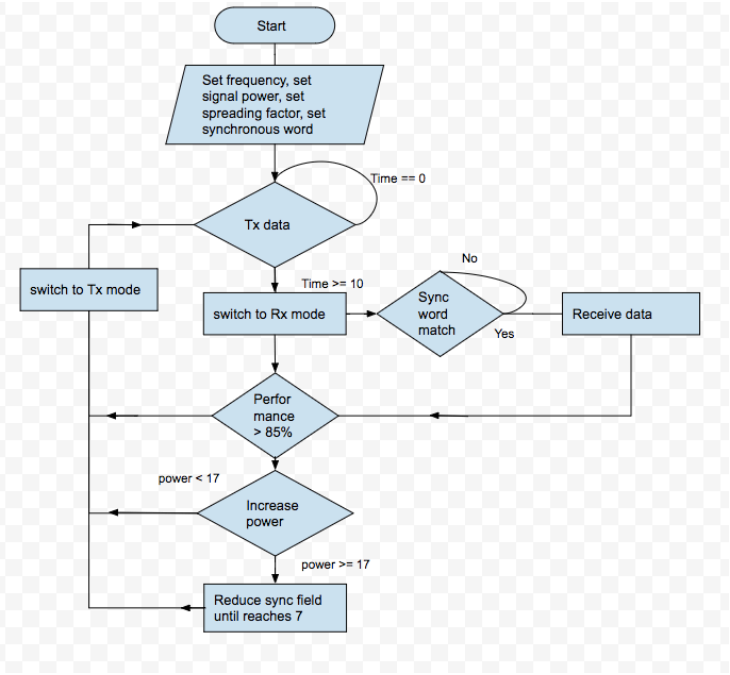
3.2.2 Implementing Cognitive Radio using LoRa module:

Interfacing the LoRa with SPI is the initial step, and following by performing basic handshaking is done with the given code. Further observations in the change in RSSI value, delay in the time between packet to packet while receiving and cyclic redundancy check which can notify if there is an misplace bit error while communication are the important observations we made in-order to improve the performance of the communication using LoRa modules.

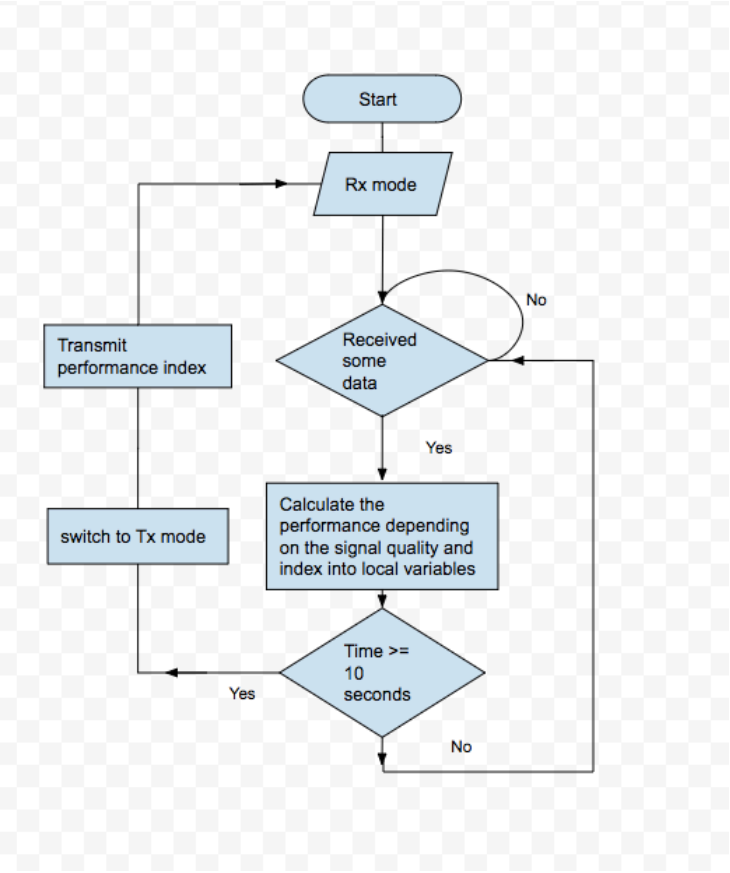
Initially we are transmitting the message from the transmitter to receiver, for every 10 seconds the receiver sends one ACK message to the transmitter which includes the performance of the message received, I have computed a formula myself depending on the RSSI (received signal strength indication), Delay between packets received, CRC (cyclic redundancy check), Water marks in between the original message, which can be checked for the sent value as the entropy at that bit is 100%, considering all these internal parameters I have computed a good formula to calculate a final performance which can be sent back to the transmitter, so that the transmitter can make some changes while transmitting the next message.

1. Flowchart:

Transmitter:



Receiver:



2. Algorithm:

The main transmission algorithm starts after once the handshaking is done. Initially, before transmission we need to set some default values, to make it cost efficient, we are setting the power to be lowest as possible, spreading factor to be high which consumes less bandwidth, we have set a default frequency of 869Mhz as to observe the performance in short ranges distances. As we are doing the address based communication, we have set the sync word and it check for the match which switching to receive mode, which can improve the security.

After setting up the default values, we start sending the data, after every 10 seconds the transmitter switches to receive mode, at which exactly the receiver sends the data which includes the performance of the received data for the past 10 seconds, on receiving the performance values, We have observed and sent a pattern to change the initially set input parameters and transmitter changes the default values to which it can improve the performance, and sends the next messages with the new updated parameters, and again for next 10 seconds of transmission, the receiver would observe the performance and send a new performance drop message, depending on the new statistics the transmitter would change the parameters to make the performance more better, after 20-30 seconds of continuous transmission and reception, the modules can understand the current surround situation and environment where the communication is happening and can improve the performance accordingly.

3. Pseudo code:

This section explains the step by step functionality of the program implemented.

Transmission:
createDataForTransmission();
convertCharToBinary();
corruptSyncBits()
sendDataViaPorts();

Receiving data:
readDataInputPin();
matchKernelSyncField;
extractPayload()
convert binaryToCharacter()

4. Testing and Verification

Following are the testing conditions:

1. Wired communication: transmitted data with 254 synchronous bits.
2. Wired communication: transmits data with corrupted synchronous bits, by setting up the corruption rate
3. Wireless communication: transmitted data with 254 synchronous bits.
4. Wireless communication: transmits data with corrupted synchronous bits, by setting up the corruption rate
5. Wired communication: Received data with proper sync bits but with confidence levels at 10 to 50.
6. Wired communication: Received data with corrupt sync bits and with confidence level at 10 to 50.

The data being transmitted in the wired and wireless communication is observed to be somewhat different in case of difference in distance. No failure observed in implementing LISA.

5. Conclusion

From the testing that was shown in the sections prior, you're able to see that the GPIO interrupt has the potential for corruption. In wireless communication, interrupts are often being called too often due to the concept of polling; therefore, noise is a factor that must be taken account for. In order to cope with, we're able to decrease the confidence level to a specific range. Doing this would cancel out much of the noise and allow us to extract data from the received bit stream properly. Therefore, LISA algorithm using this method will be able to detect and take out the data from the payload with much higher accuracy with this change in confidence level. Using LoRa has drastically increased performance as farther communication is being possible and more parameters can be manipulated or controlled in real time to increase the performance of the communication.

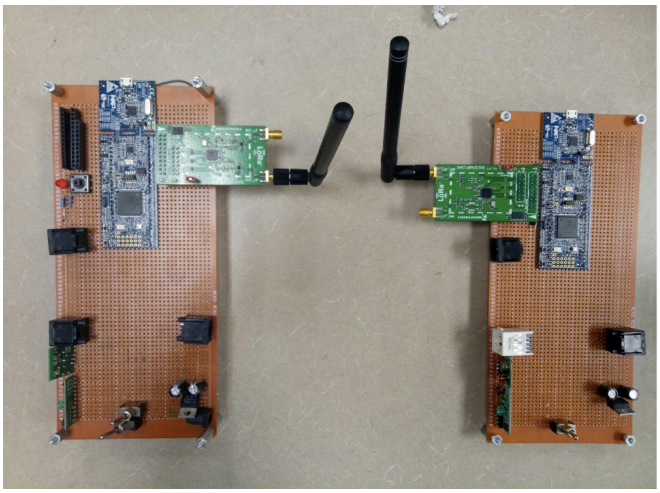
6. Acknowledgement.

I thank Prof. Hua Harry li and my fellow teammates, for guiding us to learn the implementation of cognitive radio, in real time using the state of art LoRa module. This experience helped us in gaining knowledge on wireless communication.

7. References

[1][online]https://standards.ieee.org/findstds/standard/802.11b-1999.html
[2][online]User Manual for LPC1769 http://www.nxp.com/documents/user_manual/UM10360.p df
[3][online]http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php
[4][online] http://www.nxp.com/products/software-and- tools/software-development-tools/software-tools/lpc-microcontroller-utilities/lpcxpresso-ide-v8.2.2:LPCXPRESSO

8. Appendix



8.1 LoRa connected to embedded modules:

8.2 Code Implemetation

LISA implementation:

```
/*
 * @file main.c

 * @brief The main application file;
 * 1) GPIO (I/O test) code under the macro: CMPE245_GPIO_TEST
 * @date Created on: 24-Sep-2017
 * @author(s): Krishna Sai Anirudh Katam-reddy
 */

#include "FreeRTOS.h"
#include "task.h"
#include "board.h"
#include "api.h"
#include "imp_LISA.h"
#include "scrambling_input.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//const int N = 11;

//#include "gpio_17xx_40xx.h"

/** CMPE 245 Assignments
 * @{ */
/** SW-LED Assignment: */
#define CMPE245_GPIO_TEST
/** @} */

typedef struct
{
    int port;
    int pin;
    int portLED;
    int pinLED;
}tSwitchInfo;

static void vSwitchListener(void* apv-Params)
{
    tSwitchInfo* pSWInfo = (tSwitchInfo*)apvParams;
    /** poll the input switch */
    do {
#if 0
```

```
        uint32_t* pGPIO_FIO0PIN0 = (uin-
t32_t*)(0x2009C014);
        uint8_t nBitVal = *pGPIO_FIO0PIN0 &
(0x01UL << 2);
        ///DEBUGOUT("nBitVal=%d pin=%d
%x\n", nBitVal, pSWInfo->pin, *pGPIO_-
FIO0PIN0);
#endif
        //DEBUGOUT("port=%d pin=%d\n",
pSWInfo->port, pSWInfo->pin);
        bool nBit;
        if(nBit =
Chip_GPIO_ReadPortBit(LPC_GPIO, pSWInfo-
>port, pSWInfo->pin))
        {
            DEBUGOUT("SW pressed\n");
            /** turn on the LED */
            unsigned int const CLEAR_P0_3 =
(1 << 3);
            Chip_GPIO_SetPinToggle(LPC_G-
PIO, pSWInfo->portLED, pSWInfo->pinLED);
        }
        else
        {
            //DEBUGOUT("SW not pressed");
        }
        /** delay 1s */
    } while(1);
}

static tSwitchInfo sw = {0, 2, 0, 3};

int main_app()
{
#ifdef CMPE245_GPIO_TEST
    /** set P0.2 (J2-21); Switch PIN to
Input */
    Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0,
2);
    Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 2,
IOCON_MODE_PULLDOWN);
    //LPC_IOCON_T* pIOCON =
(LPC_IOCON_T*)LPC_IOCON;
    //pIOCON->PINMODE[0] |= (3UL << 4);

#if 0
    uint32_t* pPINMODE0 = (uint32_t*)
(0x4002C040);
    *pPINMODE0 = *pPINMODE0 | (3UL << 4);
#endif

    /** set P0.3 (J2-22); LED PIN to Output
*/
    Chip_GPIO_SetPinDIROutput(LPC_GPIO, 0,
3);
    unsigned int const set_P0_3 = (1 << 3);
    Chip_GPIO_SetPortOutHigh(LPC_GPIO, 0,
set_P0_3);


    /** create the Switch Listener Thread
*/

    xTaskCreate(vSwitchListener,
(char*)"SW-Listener", (512 * 4), (void*)
(&sw), (tskIDLE_PRIORITY + 1UL), NULL);

#endif /**< CMPE245_GPIO_TEST */

    return 1;
}

static void vDataTransmitter(void* apv-
Params)
{
    /** set P0.3 (J2-22); LED PIN to Output
*/
    Chip_GPIO_SetPinDIROutput(LPC_GPIO, 0,
3);
    unsigned int const set_P0_3 = (1 << 3);
    Chip_GPIO_SetPortOutHigh(LPC_GPIO, 0,
set_P0_3);

    /** set P2.6 (J2-48); RF TX */
    Chip_GPIO_SetPinDIROutput(LPC_GPIO, 2,
6);
    unsigned int const CLEAR_P2_6 = (1 <<
6);
    //Chip_GPIO_SetPortOutHigh(LPC_GPIO2,
0, CLEAR_P2_6);

    char nOut = 0;
    while(1)
    {
        DEBUGOUT("sending %d\n", nOut);
        if(nOut)
        {
            /** active low LED */
            Chip_GPIO_SetPortOutLow(LPC_G-
PIO, 0, set_P0_3);
            Chip_GPIO_SetPortOutHigh(LPC_G-
PIO, 2, CLEAR_P2_6);
        }
        else
        {
            Chip_GPIO_SetPortOutHigh(LPC_G-
PIO, 0, set_P0_3);
```

```
            Chip_GPIO_SetPortOutLow(LPC_G-
PIO, 2, CLEAR_P2_6);
        }
        nOut = !nOut;
        vTaskDelay(5000);
    }

}

int main_tx_test()
{
    xTaskCreate(vDataTransmitter,
(char*)"DATA-Transmitter", (512 * 4), NULL,
(tskIDLE_PRIORITY + 1UL), NULL);
    return 1;
}

static void vDataReceiver(void* apvParams)
{
    /** set P0.2 (J2-21); Switch PIN to
Input */
    Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0,
2);
    Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 2,
IOCON_MODE_PULLDOWN);

    DEBUGOUT("PIN configured\n");

    do {
        //DEBUGOUT("port=%d pin=%d\n",
pSWInfo->port, pSWInfo->pin);
        bool nBit;
        if((nBit =
Chip_GPIO_ReadPortBit(LPC_GPIO, 0, 2)))
        {
            DEBUGOUT("SW pressed\n");
            /** turn on the LED */

        }
        else
        {
            //DEBUGOUT("SW not pressed\n");
        }
        /** delay 1s */
    } while(1);
}

int main_rx_test()
{
    xTaskCreate(vDataReceiver,
(char*)"DATA-Receiver", (512 * 4), NULL,
(tskIDLE_PRIORITY + 1UL), NULL);
    return 1;
}
```

```
#define portAPP_MSTOTICK(t) (t / (portTICK-
_RATE_MS))

void Chip_GPIO_SendBit(LPC_GPIO_T *pGPIO,
uint8_t port, uint32_t pin, uint8_t nBit)
{
    if(nBit) // if the current bit is one,
setting the pin high
    {
        pGPIO[port].SET = (1 << pin);
    }
    else
    {
        pGPIO[port].CLR = (1 << pin); // if
bit is zero, setting the pin to low
    }
}

uint8_t Chip_GPIO_ReadBit(LPC_GPIO_T *pG-
PIO, uint8_t port, uint32_t pin)
{
    return ((pGPIO[port].PIN >> pin) & 1);
}

void vLISADataTxRx(void* apvParams)
{
    tLISACtx* WData = (tLISACtx*)apvParams;
//setting the passed parameters about the
payload and the details on the corruption
    int i = 0;

    /** set P0.3 (J2-22); Tx PIN to Output
*/
    Chip_GPIO_SetPinDIROutput(LPC_GPIO, 0,
3);  //setting the GPIO pin as output
    //unsigned int const CLEAR_P0_3 = (1 <<
3);
    //Chip_GPIO_SetPortOutHigh(LPC_GPIO, 0,
CLEAR_P0_3);

    /** set P0.2 (J2-21); Rx to Input */
    Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0,
2);
    Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 2,
IOCON_MODE_PULLDOWN);  //pulling the p0.2
pin to active low initially after setting
the pin to input

    uint8_t payload[] = "sjsuID:4589";
    uint8_t scramload[1023];
```

```
    WData->pLISADataRaw->pPayload->pcPay-
load = payload;  //setting the payload to
be sent as the message
    WData->pLISADataRaw->pPayload->nLen =
1+strlen(WData->pLISADataRaw->pPayload-
>pcPayload);  //calculating the length of
the message and assigning the length to

#ifdef LISA_TX

    scramble_input_data(5,  WData->pLISA-
DataRaw->pPayload->pcPayload, WData->pLISA-
DataRaw->pPayload->nLen, scramload);

    WData->pLISADataRaw->pPayload->pcPay-
load = scramload;  //setting the payload to
be sent as the message
    WData->pLISADataRaw->pPayload->nLen =
1+strlen(WData->pLISADataRaw->pPayload-
>pcPayload);  //calculating the length of
the message and assigning the length

    /** generate payload in WData->pcBuffer
*/
    create_data_file(WData, WData->pLISA-
DataRaw); //creating the data file to be
transmitted
#endif
    while(1)
    {
        //int tick_count_1 = xTaskGetTick-
Count();
        if(i > 1023)   //if the bit cross
1024 bits then we can see a msg on the con-
sole
        {

vTaskDelay(portAPP_MSTOTICK(1000));
            DEBUGOUT("done\n");
            continue;
        }
        uint8_t nBit;
#ifdef LISA_TX
        /** test send  */
        Chip_GPIO_SendBit(LPC_GPIO, 0, 3,
(nBit = get_bit_value(WData->pcBuffer,
i)));  //getting a bit value from the buf-
fer one at a time and sending it through
p_0.3
        //Chip_GPIO_SetPortOutLow(LPC_GPIO,
0, 3);
        //DEBUGOUT("sending %d %d\n", nBit,
i);
#endif /**< LISA_TX */
#ifdef LISA_RX
        put_bit_value(WData->pcBuffer, i,
(nBit = Chip_GPIO_ReadPortBit(LPC_GPIO, 0,
2)));
        //DEBUGOUT("reading %d\n", nBit);
        uint8_t descramload[1023];
        if(i == 1023)
        {
            int end_bit = read_data_from_-
file(WData, WData->pLISADataRaw);

            descramble_output_data(5, WDa-
ta->pLISADataRaw->pPayload->pcPayload, WDa-
ta->pLISADataRaw->pPayload->nLen, descram-
load);

            WData->pLISADataRaw->pPayload-
>pcPayload = descramload;

            DEBUGOUT("payload_descrambled =
[%s]\n", descramload);

            printf("return %d", end_bit);
        }


#endif /**< LISA_RX */
        i++;
        /** Configuration for this board
says:
         * portTICK_RATE_MS = 1000 / 1000
         * means each TICK = 1 millisecond
         *
         * Let's say our datarate = 1kbps
         * 1000 bits in a second
         * So, each bit = 1ms
         *  */
        portTickType nTicks = portAPP_M-
STOTICK(10);
        //int tick_count_2 = xTaskGetTick-
Count();
        //DEBUGOUT("nTicks=%d\n", tick_-
count_2 - tick_count_1);
        vTaskDelay(nTicks);
    }

    DEBUGOUT("PIN configured\n");


}

int main_lisa()
{
```

```
    tLISACtx* WData = (tLISACtx*)calloc(1,
sizeof(tLISACtx));  //allocating memory for
the whole structure of the LISA, includes
the confidence details, buffer to store all
the message bits and tLISADataRaw pointer
    tLISADataRaw* sub_data =
(tLISADataRaw*)calloc(1, sizeof(tLISAData-
Raw) + sizeof(tLISAPayload)); //allocates
memory for the LISA sub str, includes pay-
load and corruption details.
    sub_data->nBufferSize =
BUFFER_SIZE;   //setting the buffer size,
to transmit and receive from and to the
buffer.
    sub_data->pPayload = (tLISAPayload*)
(sub_data + 1); //creating the payload

    WData->pLISADataRaw = sub_data;

    xTaskCreate(vLISADataTxRx, (signed
char*)"LISA-TxRx", (512 * 4), (void*)WData,
(tskIDLE_PRIORITY + 1UL), NULL); //creating
a thread to data transmit and receive
    return 1;
}
```

```
/**
 * @file imp_LISA.h

 *
 * @brief LISA API Header file
 *
 * @date created Sep 16 2017
 *
 * @author(s): Krishna Sai Anirudh Katam-
reddy
 *
 */

#ifndef _IMP_LISA_H_
#define _IMP_LISA_H_

#define DAT_FILE "lisa_test_data.dat"
#define BUFFER_SIZE (1024)
#define DEFAULT_PAYLOAD "CMPE-245"

#define LISA_SYNC_FIELD_LEN_BYTES (32)
#define LISA_SYNC_FIELD_LEN_BITS (32 * 8)
#define LISA_SYNC_FIELD_PREFIX1_4b (0x05)
#define LISA_SYNC_FIELD_PREFIX2_4b (0x0a)

#define LISA_PAYLOAD_LEN_FIELD_BITS (8)
```

```
typedef struct
{
    unsigned char nLen;   //sets the length
of the payload
    char* pcPayload;   //to create an array
of char's for the payload
}tLISAPayload;

typedef struct
{
    /** Percent of corruption in Sync Field
*/
    double fSFCorrruption;
    int    nBytesToCorrupt;
    int    nBufferSize;
    tLISAPayload* pPayload;
}tLISADataRaw;

typedef struct
{
    unsigned char pcBuffer[BUFFER_SIZE /
8];   //creating a buffer to store the sync
bits, payload and garbage if any
    double fSFConfidence;
    int nConfidence;

    tLISADataRaw* pLISADataRaw;
}tLISACtx;

static unsigned char const KERNEL_STRING[]
= {
    0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5,
0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xab, 0xac,
0xad, 0xae, 0xaf,
    0x50, 0x51, 0x52, 0x53, 0x54, 0x55,
0x56, 0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c,
0x5d, 0x5e, 0x5f
    };
```

```
/**
 * @brief bit manipulation functions
 * write/read a bit at a given bit position
in the buffer
 */
void put_bit_value(unsigned char* apcBuf-
fer, int anBitLoc, char anBitVal);
unsigned char get_bit_value(unsigned char*
apcBuffer, int anBitLoc);

/**
 * @brief write/read byte at a given bit
position in the buffer
```

```
 */
void put_byte(unsigned char* apcBuffer, int
anBitLoc, unsigned char anByte);
unsigned char get_byte(unsigned char* apc-
Buffer, int anBitLoc);

/**
 * @brief utility functions
 * print the buffer of length anLen in
hexadecimal
 */
void print_hex(unsigned char* apcBuffer,
int anLen);

int match_window_kernelbyte_at(unsigned
char* apcBuffer, int anBitLoc, unsigned
char anKernel);
int get_byte_match_pos(unsigned char* apc-
Buffer, int anStartLoc, int anLen, unsigned
char anByte);

int get_payload_start_pos(tLISACtx* apCtx,
unsigned char* apcBuffer, int* apnPayload-
Start, int anLen);

int create_data_file(tLISACtx* apCtx,
tLISADataRaw* apLISADataRaw);
int read_data_from_file(tLISACtx* apCtx,
tLISADataRaw* apLISADataRaw);

int match_window_kernel_at();

#endif /**< _IMP_LISA_H_ */



/**
 * @file imp_LISA.c

 *
 * @brief LISA Implementation file
 *
 * @date created Sep 16 2017
 *
 * @features:
 * 1) Creating a DAT file with 1024 bit
LISA protocol specific
 * data: - argv[1] == 0
 * a) Random GARBAGE
 * b) SYNC FIELD (with or without corrup-
tion) - use argv[2] to specify
 *
corruption amount (example = 1/32 = 0.03125
 *
will corrupt one of the 32 Sync Fields)
 * c) Payload (8-bit Length + the payload
given by user at argv[3])
 * 2) Reading the DAT file and printing the
payload
 * See the major function prototypes at
imp_LISA.h
 *
 * @author(s) Krishna Sai Anirudh Katamred-
dy
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

//#define DEBUG_E
#define VERBOSE_E
#include "debug_e.h"

#include "imp_LISA.h"

//#define TEST_GARBAGE_BYTE_ALIGNED

/**
 * @brief
 * NOTE: bit index starts at 0
 */
void put_bit_value(unsigned char* apcBuf-
fer, int anBitLoc, char anBitVal)
{
    unsigned char* pBuf = apcBuffer + (an-
BitLoc/8);
    LOGD("bit pos %d val=%d %x\n\n", anBit-
Loc, anBitVal, *pBuf);
    int nLeftOverBits = anBitLoc - ((anBit-
Loc/8)*8);             //calculating the
extra bits after all bytes
    if(nLeftOverBits)                 //if
not in byte boundary
    {
        /** write at the nLeftOverBits'th
position of pBuf */
        *pBuf &= (~(0x01 << (8 - nLeftOver-
Bits - 1)));           //clearing the bit
to zero
        *pBuf |= anBitVal << (8 - nLeft-
OverBits - 1);            //setting the bit
to input sent bit
```

```
    }
    else  //it will the first bit
    {
        /** exactly at byte boundary */
        /** clear that bit */
        *pBuf &= (~(0x01 << 7));    //
clearing the particular bit by making the
bit zero
        *pBuf |= (anBitVal << 7);   //in-
serting your bit value by using bit wise or
with the left shifted new bit value.
    }

}

void put_byte(unsigned char* apcBuffer, int
anBitLoc, unsigned char anByte)
{
    if(!apcBuffer)
        return;
    for(int i = 0; i < 8; i++)   //putting
all bits the byte one by one, by shifting
left and right to choose the position and
put the bit function
        put_bit_value(apcBuffer, anBitLoc +
i, ((anByte >> (8 - i - 1)) & 0x01));   //
to put one bit at a time into buffer
}

/**
 * @fn int create_data_file(tLISADataRaw*
apLISADataRaw)
 * @brief Function to create DAT_FILE with
RAW data
 * RAW data will have sync-field, payload
and other random
 * data
 * @param apLISADataRaw [IN] LISA network
packets to be written
 * into a file
 * @return SC_SUCCESS on successfully writ-
ing the file
 */
int create_data_file(tLISACtx* apCtx,
tLISADataRaw* apLISADataRaw)
{
    int nMaxGarbageLenBits = 0;
    int nRandGarbageLenBits = 0;
    int nIdx = 0; /**< current position in
pcBuffer */

    int nSFStart = 0;

    LOGD("DEBUGME\n");
    if(!apCtx || !apLISADataRaw)  //if the
LISA data or the lisa sizing is nullptr
return -1;
        return EC_FAILURE;
    LOGD("DEBUGME\n");

    /** 1) Generate random garbage
     *  a) Max length of garbage = BUFFER_-
SIZE - (sizeof(SF) + sizeof(payload))
     */
    #ifdef TEST_WITH_INIT_GARBAGE
    nMaxGarbageLenBits = BUFFER_SIZE -
(LISA_SYNC_FIELD_LEN_BITS + LISA_PAYLOAD-
_LEN_FIELD_BITS + (apLISADataRaw->pPayload-
>nLen * 8));
    if(nMaxGarbageLenBits < 0)
        return EC_FAILURE;

    nRandGarbageLenBits = ((rand() % nMax-
GarbageLenBits));

    #ifdef TEST_GARBAGE_BYTE_ALIGNED
    /** Test code; not compiled; forces the
garbage bits to be byte aligned */
    nRandGarbageLenBits = ((rand() % nMax-
GarbageLenBits)/8)*8;
    #endif /**< TEST_GARBAGE_BYTE_ALIGNED
*/

    for(nIdx = 0; nIdx < nRandGarbageLen-
Bits/8; nIdx++)
    {
        apCtx->pcBuffer[nIdx] = (unsigned
char)rand();
    }

    {
        int nLeftOverBits = nRandGarbage-
LenBits - ((nRandGarbageLenBits/8) * 8);
        if(nLeftOverBits)
        {
            unsigned char nKernel = (2 <<
(nLeftOverBits+1)) - 1; /**< 2^N - 1 is all
1's */
            nKernel = nKernel << (8 -
nLeftOverBits);
            apCtx->pcBuffer[nIdx++] = nKer-
nel & (unsigned char)rand();
        }
    }
    LOGD("nRandGarbageLenBits=%d\n", nRand-
GarbageLenBits);
    #endif
```

```
    /** 2) Sync field
     *  a) Corrupt the bits based on fSF-
Corrruption */
    nSFStart = nIdx = nRandGarbageLenBits;

    #if 0
    for(int i = 0; i < (LISA_SYNC_FIELD-
_LEN_BITS/8)/2; i++, nIdx += 8)
        put_byte((unsigned char*)apCtx-
>pcBuffer, nIdx, (0x05 << 4) | (i & 0x0f));
    for(int i = 0; i < (LISA_SYNC_FIELD-
_LEN_BITS/8)/2; i++, nIdx += 8)
        put_byte((unsigned char*)apCtx-
>pcBuffer, nIdx, (0x0a << 4) | (i & 0x0f));
    #endif

    for(int i = 0; i < LISA_SYNC_FIELD-
_LEN_BITS/8; i++, nIdx += 8)
    {
        put_byte((unsigned char*)apCtx-
>pcBuffer, nIdx, KERNEL_STRING[i]);   //
sending values in kernel string(char which
is one byte) one by one
    }

#ifdef TEST_WITH_CORRUPTION
    /** corrupt bits */
    //int nBytesToCorrupt = apLISADataRaw-
>fSFCorrruption * LISA_SYNC_FIELD-
_LEN_BYTES;
    LOGV("nBytesToCorrupt=%d
nSFStart=%d\n", apLISADataRaw->nBytesToCor-
rupt, nSFStart);
    for(int i = 0; i < apLISADataRaw-
>nBytesToCorrupt; i++)
    {
        int nRandBitPos = (nSFStart + (i*8)
+ (rand() % 7));
        LOGD("nRandBitPos=%d bot_val=%d
%x\n", nRandBitPos, get_bit_value(apCtx-
>pcBuffer, nRandBitPos), apCtx-
>pcBuffer[nRandBitPos/8]);
        put_bit_value(apCtx->pcBuffer,
nRandBitPos, (get_bit_value(apCtx->pcBuf-
fer, nRandBitPos) ? 0x00 : 0x01));
        LOGD("corrupted=%x\n", apCtx-
>pcBuffer[nRandBitPos/8]);
    }
#endif

    /** 3) Payload */
    /** a) write length */
    LOGD("len=%d payload=[%s]\n", apLISA-
DataRaw->pPayload->nLen, apLISADataRaw-
>pPayload->pcPayload);
    put_byte((unsigned char*)apCtx->pcBuf-
fer, nIdx, apLISADataRaw->pPayload->nLen);
//sending the data length which should be
specified before the payload
    nIdx += 8;

    //after the payload length send all the
payload bits one by one to the buffer.
    for(int i = 0; i < apLISADataRaw->pPay-
load->nLen; i++)
    {
        put_byte((unsigned char*)apCtx-
>pcBuffer, nIdx, apLISADataRaw->pPayload-
>pcPayload[i]);
        nIdx += 8;
    }

#ifdef WRITE_FILE
    FILE* fpDATA = fopen(DAT_FILE, "wb");
    fwrite(apCtx->pcBuffer, sizeof(unsigned
char), BUFFER_SIZE/8, fpDATA);
    fclose(fpDATA);
#endif

    return SC_SUCCESS;
}

unsigned char get_bit_value(unsigned char*
apcBuffer, int anBitLoc)
{
    unsigned char* pBuf = apcBuffer + (an-
BitLoc/8);
    int nLeftOverBits = anBitLoc - ((anBit-
Loc/8)*8);
    if(nLeftOverBits)
    {
        return ((*pBuf >> (8 - nLeftOver-
Bits - 1)) & 0x01);

    }
    else
    {
        /** exactly at byte boundary */
        return ((*pBuf >> 7) & 0x01);
    }
}

unsigned char get_byte(unsigned char* apc-
Buffer, int anBitLoc)
{
    unsigned char nByte = 0;
```

```
    for(int i = 0; i < 8; i++)
        nByte |= ((get_bit_value(apcBuffer,
anBitLoc+i) << (8 - i -1)));
    return nByte;
}

int match_window_kernel_at(unsigned char*
apcBuffer, int anBitLoc, unsigned char*
apcKernel, int anKernelSizeBytes)
{
    int nMatch = 0;
    /** match the kernel bits and return
score */
    for(int i = 0; i < anKernelSizeBytes;
i++)
    {
        LOGD("anBitLoc=%d i =%d %d %x\n",
anBitLoc + i*8, i, anKernelSizeBytes, apcK-
ernel[i]);
        int nM = match_window_kernel-
byte_at(apcBuffer, anBitLoc + i*8, apcKer-
nel[i]);
        nMatch += nM;
        if(nM != 8)
            break;


//        if(nM == 8)
//        {
//            nMatch += nM;
//
//        }
//        else if(nM != 8)
//        {
//            anKernelSizeBytes += 1;
//        }
//
//        if(anKernelSizeBytes >= 32)
//            break;
    }

    LOGD("nMatch=%d\n", nMatch);
    return nMatch;
}

int match_window_kernels_for_confi-
dence_level(tLISACtx* apCtx)
{
    int nPayloadStart = -1;
    /** return payload start bit location
if the confidence level match
     * else return -1 */
    if(!apCtx)
        return nPayloadStart;

    int nNumOfSFBytesToCheck = apCtx->nCon-
fidence;
    int nNumOfTotalChecks = 32 - apCtx-
>nConfidence + 1;
    int nMatch = 0;
    LOGV("nNumOfSFBytesToCheck=%d\n", apC-
tx->nConfidence);
    for(int i = 0; i < BUFFER_SIZE; i++) //
starting at ith bit in the buffer
    {
        nMatch = 0;
        LOGD("i=%d\n", i);
        for(int j = 0; j + nNumOfSFBytesTo-
Check <= 32; j++)
        {
            LOGD("j=%d bit=%d byte=%d\n",
j, i+(j*8), (i+(j*8))/8);
            print_hex(&apCtx->pcBuffer[(i+
(j*8))/8], BUFFER_SIZE - (i+(j*8))/8);
            int nM = match_window_ker-
nel_at((unsigned char*)apCtx->pcBuffer, i +
j*8, (unsigned char*)(&KERNEL_STRING[j]),
nNumOfSFBytesToCheck);
            nMatch += nM;    //do we also
have to plus the j by nM
            if(nM ==
nNumOfSFBytesToCheck*8)
            {
                LOGV("match perfect; itera-
tion=%d [%s]\n", j, &KERNEL_STRING[j]);
                nPayloadStart = i + 32*8;
                return nPayloadStart;
            }
        }
        #if 0
        if(nMatch/8 == nNumOfSFBytesTo-
Check)
        {
            nPayloadStart = i + 32*8;
            return nPayloadStart;
        }
        #endif
    }

    LOGD("nMatch=%d nMatchBytes=%d nNumOf-
SFBytesToCheck=%d\n", nMatch, nMatch/8,
nNumOfSFBytesToCheck);

    return nPayloadStart;
}
```

```
int match_window_kernelbyte_at(unsigned
char* apcBuffer, int anBitLoc, unsigned
char anKernel)
{
    int nMatch = 0;
    unsigned char nBitVal = 0;
    if(!apcBuffer)
        return nMatch;

    /**  */
    for(int i = 0; i < 8; i++)
    {
        nBitVal = get_bit_value(apcBuffer,
anBitLoc + i);
        LOGD("pos=%d nBitVal=%d %d\n", an-
BitLoc + i, nBitVal, ((anKernel >> (8 - i -
1)) & 0x01));
        if(nBitVal == ((anKernel >> (8 - i
- 1)) & 0x01))
        {
            nMatch++;
        }
    }

    return nMatch;
}

void print_hex(unsigned char* apcBuffer,
int anLen)
{
    #ifdef PRINT_HEXDUMP
    LOGD("apcBuffer=%p anLen=%d\n", apcBuf-
fer, anLen);
    for(int i = 0; i < anLen; i++)
    {
        printf("%x ", apcBuffer[i]);
        if(i % 16 == 0)
            printf("\n");
    }
    printf("\n");
    #endif /**< PRINT_HEXDUMP */
}

int get_byte_match_pos(unsigned char* apc-
Buffer, int anStartLoc, int anLen, unsigned
char anByte)
{
        for(int nIdx = anStartLoc; nIdx <
anLen; nIdx++)
        {
            int nMatch = match_window_ker-
nelbyte_at(apcBuffer, nIdx, anByte);
            LOGD("nMatch=%d\n", nMatch);
            if(nMatch == 8)
            {
                LOGD("full match %d
anByte=%x\n", nIdx, anByte);
                return nIdx;
            }
        }
        return -1;
}


int get_payload_start_pos(tLISACtx* apCtx,
unsigned char* apcBuffer, int* apnPayload-
Start, int anLen)
{
    int nConfidence = 0;
    int nIdx = 0;
    int nLastFullMatchIdx = 0;

    for(int i = 0; i < LISA_SYNC_FIELD-
_LEN_BYTES; i++)
    {
        unsigned char nKernel = (i <
(LISA_SYNC_FIELD_LEN_BYTES/2)) ? 0x50 | i :
0xa0 | (i - (LISA_SYNC_FIELD_LEN_BYTES/2));
        if((nIdx = get_byte_match_pos(apc-
Buffer, 0, anLen, nKernel)) != -1)
        {
            *apnPayloadStart = nIdx +
((LISA_SYNC_FIELD_LEN_BYTES-i)*8);
            nConfidence++;
            nLastFullMatchIdx = nIdx;
            if((((double)nConfidence) / 32)
>= apCtx->fSFConfidence)
            {
                LOGD("required confidence
achieved at %d\n", nIdx);
                break;
            }
        }
    }

    return nConfidence;
}

int read_data_from_file(tLISACtx* apCtx,
tLISADataRaw* apLISADataRaw)
{
    int nIdx = 0;
    int nPayloadStart = 0;
    if(!apCtx || !apLISADataRaw)
        return EC_FAILURE;

    /** read from file */
    FILE* fpDATA = fopen(DAT_FILE, "rb");
```

```
    fread(apCtx->pcBuffer, sizeof(unsigned
char), BUFFER_SIZE/8, fpDATA);
    fclose(fpDATA);

    print_hex(apCtx->pcBuffer, BUFFER_SIZE/
8);

    /** find sync field */
    #if 0
    for(nIdx = 0; nIdx < BUFFER_SIZE; nIdx+
+)
    {
        int nMatch = match_window_kernel-
byte_at(apCtx->pcBuffer, nIdx, 0x50);
        LOGD("nMatch=%d\n", nMatch);
        if(nMatch == 8)
        {
            LOGD("full match\n");
            break;
        }
    }
    #endif

    #if 0
    get_payload_start_pos(apCtx, apCtx-
>pcBuffer, &nPayloadStart, BUFFER_SIZE);
    #else
    nPayloadStart = match_window_kernels_-
for_confidence_level(apCtx);
    #endif
    if(nPayloadStart)
    {
        LOGD("payload start at %d bit\n",
nPayloadStart);
    }



    //nIdx += (LISA_SYNC_FIELD_LEN_BITS);
    nIdx = nPayloadStart;

    apLISADataRaw->pPayload->nLen =
get_byte(apCtx->pcBuffer, nIdx);
    nIdx += 8;

    apLISADataRaw->pPayload->pcPayload =
malloc(apLISADataRaw->pPayload->nLen);
    LOGV("payload len=%d nIdx=%d\n",
apLISADataRaw->pPayload->nLen, nIdx);

    for(int i = 0; i < apLISADataRaw->pPay-
load->nLen; i++)
    {
        apLISADataRaw->pPayload-
>pcPayload[i] = get_byte(apCtx->pcBuffer,
nIdx);
        nIdx += 8;
    }


    if(nPayloadStart == -1)
    {
        LOGV("payload unavailable; try bet-
ter confidence-percentage\n");
        return EC_FAILURE;
    }

    LOGV("payload_scrambled = [%s]\n",
apLISADataRaw->pPayload->pcPayload);

    return SC_SUCCESS;
}

#if 1
int xLISA(int argc, char* argv[])
{
    tLISACtx*     pLISACtx     = NULL;
    tLISADataRaw* pLISADataRaw = NULL;
    int nOption = 0;

    srand(time(NULL));
    /**
     * bin [0, 1] [data]
     * 0: generate DAT file [fSFCorrruption
payload_string]
     * 1: read from the DAT file and print
the payload []
     * fSFCorrruption: Sync Field corrup-
tion;
     * We have 32 * 8 bits in SF,
     * so corruption can be on any bit;
     * Lets model this in bytes.
     * 32 bytes; if corruption is 1/32 -
one of the SF byte is corrupted.
     */
    LOGD("argc=%d\n", argc);
    if(argc < 2)
    {
        return EC_FAILURE;
    }
    nOption = atoi(argv[1]);
    pLISACtx     = (tLISACtx*)calloc(1,
sizeof(tLISACtx));
    pLISADataRaw = (tLISADataRaw*)calloc(1,
sizeof(tLISADataRaw) +
sizeof(tLISAPayload));
    pLISADataRaw->fSFCorrruption = 0.0;
```

```
    pLISADataRaw->nBufferSize = BUFFER_-
SIZE;
    pLISADataRaw->pPayload =
(tLISAPayload*)(pLISADataRaw + 1);

    LOGD("nOption=%d\n", nOption);
    switch(nOption)
    {
        case 0:
            pLISADataRaw->fSFCorrruption =
(argv[2] && argc > 2) ? atof(argv[2]) :
0.0;
            pLISADataRaw->nBytesToCorrupt =
(int)lround((pLISADataRaw->fSFCorrruption /
100) * 32);
            pLISADataRaw->pPayload->pcPay-
load = (argv[3] && argc > 2) ? argv[3] :
DEFAULT_PAYLOAD;
            pLISADataRaw->pPayload->nLen =
strlen(pLISADataRaw->pPayload->pcPayload) +
1;
            create_data_file(pLISACtx,
pLISADataRaw);
        break;
        case 1:
            pLISACtx->fSFConfidence =
(argv[2] && argc > 2) ? atof(argv[2]) :
100.0;
            pLISACtx->nConfidence =
(int)lround((pLISACtx->fSFConfidence / 100)
* 32);
            if(pLISACtx->nConfidence < 0 ||
pLISACtx->nConfidence > 32)
                pLISACtx->nConfidence = 32;
            read_data_from_file(pLISACtx,
pLISADataRaw);
        break;
        default:
            return EC_FAILURE;
    }


    return 0;
}
#endif
```

LoRa Receive code:

```
/*
===============================================
=====================================
 Name        : RF_Handshaking.c
 Description : RF and PWM code for LPC1769

 CTI One Corporation released for Dr. Harry
Li for CMPE 245 Class use ONLY!
===============================================
=====================================
*/

//#include "FreeRTOS.h"
//#include "FreeRTOS.h"
#ifdef __USE_CMSIS
//#include "LPC17xx.h"
#endif

#include <cr_section_macros.h>

#include <stdio.h>
#include <stdbool.h>
#include "LoRa.h"
#include"common.h"
#include "timer.h"

#include <string.h>


#define RF_Receive 1
#define RF_Transmit 0
#define TransmittACk 0
#define ack_start_stop 0
int rfInit(void);

char receiveData=0;
```

```
int packetSize;


static int snP = DEF_POWER;
static int snSF = DEF_SF;

#define LOGD(...)
//printf(__VA_ARGS__)
#define LOGV(...) printf(__VA_ARGS__)


/
*****************************************
*****************************************
************
* @brief      wait for ms amount of mil-
liseconds
* @param      ms : Time to wait in mil-
liseconds
*****************************************
*****************************************
************/
static void delay_ms(unsigned int ms)
{
    unsigned int i,j;
    for(i=0;i<ms;i++)
        for(j=0;j<50000;j++);
}

/
*****************************************
*****************************************
************
* @brief      wait for delayInMs amount of
milliseconds
* @param      delayInMs : Time to wait in
milliseconds
*****************************************
*****************************************
************/
static void delay(uint32_t delayInMs)
{
       LPC_TIM0->TCR = 0x02;            /*
reset timer */
       LPC_TIM0->PR  = 0x00;            /*
set prescaler to zero */
       LPC_TIM0->MR0 = delayInMs * (9000000
/ 1000-1);
       LPC_TIM0->IR  = 0xff;            /*
reset all interrrupts */
       LPC_TIM0->MCR = 0x04;            /*
stop timer on match */
       LPC_TIM0->TCR = 0x01;            /*
start timer */
       /* wait until delay time has elapsed
*/
       while (LPC_TIM0->TCR & 0x01);
}


/
*****************************************
*****************************************
************
* main : Main program entry
*****************************************
*****************************************
************/
int main(void)
{
       LOGD("System clock is %d\n",System-
CoreClock);
       uint64_t nTimeElapsedBetweenPackets
= 0;
       uint64_t nTimeTickCount1 = 0;
       double fPerformanceIndex = 100.0;
       double fDelayPerformanceDrop = 0.0;
       double fCorruptionDrop = 0.0;
       // Working frequency range from 724
MHz to 1040 MHz.
       //LoRabegin(1040000000);
       //LoRabegin(1020000000);
       //LoRabegin(724000000);
       //LoRabegin(750000000);
       //LoRabegin(790000000);
       //LoRabegin(800000000);
       //LoRabegin(845000000);
       //LoRabegin(850000000);
       //LoRabegin(910000000);
       //LoRabegin(868000000);
       LoRabegin(869000000);

       //LoRabegin(915000000);
       int counter =0;
       int ind = 0;
       timer_initialise();
       uint64_t nCount = 0;
       uint64_t nRcvCount = 0;

       char en = '*';// = {'*', '*', '*',
'$\*', '%'};
       /* Start the tasks running. */
       //vTaskStartScheduler();

       /* If all is well we will never
reach here as the scheduler will now be
```

```
      running.  If we do reach here then
it is likely that there was insufficient
      heap available for the idle task to
be created. */
#if RF_Receive
      while(1)
      {

#if 1
          if((getTimeTickCount() - nCount)
>= 10)
          {
              //nTimeElapsedBetweenPackets
= nTimeElapsedBetweenPackets / nRcvCount;
              //change the config
              LOGV("changing SF=%d,
power=%d perf=%f RSSI=%d SNR=%f nTimeEl=%d
br=%d\n",
                      snSF, snP, fPerfor-
manceIndex, packetRssi(), packetSnr(),
                      nTimeElapsedBetween-
Packets,
                      getBitrate());
              nTimeElapsedBetweenPackets =
0;
              setTxPower(snP);
              setSpreadingFactor(snSF);

              nCount = getTimeTickCount();
              char buffer[1024];
              char Acknowledgement;
              int nSendSize =
snprintf(buffer, 1024, "%f", fPerformance-
Index);
              Acknowledgement = 'A';
              LOGD("Start Sending data
\n");
              //delay_ms(1000);
              LoRabeginPacket(0);
              //
writebyte(Acknowledgement);
              write(buffer, nSendSize);
              LoRaendPacket();
              LOGD("Data sent [%s]\n",
buffer);
          }
#endif
          rxModeCheck();
          //delay_ms(1000);

          //LOGD("parsePacket:\n");
             packetSize = parsePacket(0);
             LOGD("parsePacket size=%d
%llu\n", packetSize, getTimeTickCount());
             if (packetSize)
             {
                                double
entropy;
                 int count = 0;
                    counter = 0;
                    //
NVIC_EnableIRQ(TIMER0_IRQn);
                    //received a packet
                    //LOGD("Received packet
\n");
                    // read packet
                    if(nTimeTickCount1)
                    {
                        nTimeElapsedBe-
tweenPackets += (getTimeTickCount() -
nTimeTickCount1);
                    }

          counter = 1;
          char pcRecvBuffer[1024] = {0};
          int nIdxRcvBuffer = 0;
          int bGotSemi = 0;
          nTimeTickCount1 = getTimeTick-
Count();
                  while (available() &&
(++count <= packetSize))
                  {
                      pcRecvBuffer[nIdxR-
cvBuffer++] = receiveData = read();
                          //
LOGD("%c\n",receiveData);
                          //return 0;
                          // print RSSI of
packet
                          LOGD("Received
packet '");
                          LOGD("%c",re-
ceiveData);
                          LOGD("' with
RSSI ");
                          LOGD("%d; nTime-
ElapsedS=%d\n",packetRssi(), nTimeElapsed-
BetweenPackets);

                          if(receiveData
== ';')
                          {
                              bGotSemi =
counter;
                          }
                          if(bGotSemi)
                          {
                              continue;
```

```
                        }
                        if(counter%5 ==
0)
                        {
                            if(receive-
Data != en)
                            {
                                LOGD("no
match in water mark index : %d, count =
%d\n", ind, counter);
                            }
                            else
                            {

LOGD("****  Yes! match in water mark
index : %d, count = %d\n", ind, counter);
                                ind++;
                            }
                        }

                        counter++;

                    }
                    if(count == packetSize)
                    {
                        nRcvCount++;
                        /*Understand the
param config from TX and configure local
params
                         * ;P=%dSF=%d */
                        LOGV("buffer=[%s]
\n", pcRecvBuffer);
                        //split by semi
colon
                        //
blah1;blah2;blah3
                        char* token = str-
tok(pcRecvBuffer, ";");
                        while(token = str-
tok(NULL, ";"))
                        {
                            LOGV("data=[%s]
\n", token);
                            //this is
P=%dSF=%d
                            sscanf(token,
"P=%dSF=%d", &snP, &snSF);
                        }
                    }
                    entropy = ((packetSize
- ind)*1.0) /packetSize;

                    int nAsterixCountReq =
bGotSemi / 5;
                    if(nAsterixCountReq -
ind > 0)
                    {
                        LOGD("we did not
get %d *'s\n", nAsterixCountReq - ind);
                        int nAsterixWe-
Dropped = nAsterixCountReq - ind;
                        fCorruptionDrop =
(nAsterixWeDropped * 50.0) / nAsterixCount-
Req;
                    }


                    /** for each second
more of the time between packets,
                     * reduce 5% of the
performance index
                     *  */
                    if(nTimeElapsedBetween-
Packets > 3)
                    {
                        fDelayPerformance-
Drop += (5.0 *
(nTimeElapsedBetweenPackets /3));
                        if(fDelayPerfor-
manceDrop >= 50.0)
                        {
                            fDelayPerfor-
manceDrop = 50.0;
                        }
                    }
                    else
                    {
                        fDelayPerformance-
Drop = 0.0;
                    }

                    fPerformanceIndex =
100.0 - fDelayPerformanceDrop - fCorrup-
tionDrop;


                }

            }


#if TransmittACk
            const char buffer[] = "Data from
LPC1769";
            char Acknowledgement;
            Acknowledgement = 'A';
            while(1)
            {
```

```
            LOGD("Start Sending data
\n");
            delay_ms(1000);
            LoRabeginPacket(0);
            //writebyte(Acknowledgement);
            write(buffer,
sizeof(buffer));
            LoRaendPacket();
            LOGD("Data sent \n");

      }

#endif

#endif
#if RF_Transmit
    const char buffer[] = "Data from
LPC1769";
    char Acknowledgement;
    Acknowledgement = 'A';
    while(1)
    {
        LOGD("Start Sending data \n");
        delay_ms(1000);
        LOGD("delay done\n");
        LoRabeginPacket(0);
        //writebyte(Acknowledgement);
        LOGD("begin packet\n");
       size_t ret = write(buffer,
sizeof(buffer));
        LOGD("write done %d \n",ret);
        LoRaendPacket();
        LOGD("Data sent \n");

    }

#endif
}

void check_ack()
{
      LOGD("1 \n");

//
NVIC_EnableIRQ(TIMER0_IRQn);
            //
NVIC_DisableIRQ(TIMER0_IRQn);
}
```

```
/*
 * LoRa.h
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */

#ifndef LORA_H_
#define LORA_H_


/*
 * LoRa.h
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */


#include <stddef.h>
#include "ssp.h"


#define DEF_SF (12)
#define DEF_POWER (2)

//#define LORA_DEFAULT_SS_PIN    10
#define LORA_DEFAULT_RESET_PIN          2
#define LORA_DEFAULT_DIO0_PIN           3



  int LoRabegin(long frequency);
  void end();
  int LoRabeginPacket(int implicitHeader);
  int LoRaendPacket();

  int parsePacket(int size );
  void rxModeCheck();
```

```
  int packetRssi();
  float packetSnr();

  // from Print
  size_t writebyte(uint8_t byte);
  size_t write(const uint8_t *buffer,
size_t size);

  // from Stream
  int available();
  int read();
  int peek();
  void flush();

  void onReceive(void(*callback)(int));

  void receive(int size);
  void idle();
  void sleep();

  void setTxPower(int level);
  void setFrequency(long frequency);
  void setSpreadingFactor(int sf);
  void setSignalBandwidth(long sbw);
  void setCodingRate4(int denominator);
  void setPreambleLength(long length);
  void setSyncWord(int sw);
  void crc();
  void noCrc();

  uint8_t random();



 // void dumpRegisters(Stream& out);
uint8_t readRegister(uint8_t address);

  void explicitHeaderMode();
  void implicitHeaderMode();

  void handleDio0Rise();


  void writeRegister(uint8_t address, uin-
t8_t value);
  uint8_t singleTransfer(uint8_t address,
uint8_t value);

  static void onDio0Rise();
```

```
void digitalWrite(uint8_t pin, uint8_t val-
ue);

void gpioInit();

uint32_t getBitrate();




#endif /* LORA_H_ */


/*
 * LoRa.c
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */

/*
 * LoRa.cpp
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */


#include <stdio.h>
#include <stddef.h>
#include "LoRa.h"
#include "ssp.h"
#include "extint.h"


#define LOGD(...)
#define LOGV(...) printf(__VA_ARGS__)


//#include "FreeRTOS.h"
// registers
#define REG_FIFO                    0x00
#define REG_OP_MODE                 0x01
#define REG_FRF_MSB                 0x06
#define REG_FRF_MID                 0x07
#define REG_FRF_LSB                 0x08
#define REG_PA_CONFIG               0x09
#define REG_LNA                     0x0c
#define REG_FIFO_ADDR_PTR           0x0d
#define REG_FIFO_TX_BASE_ADDR       0x0e
#define REG_FIFO_RX_BASE_ADDR       0x0f
#define REG_FIFO_RX_CURRENT_ADDR 0x10
```

```
#define REG_IRQ_FLAGS               0x12
#define REG_RX_NB_BYTES             0x13
#define REG_PKT_RSSI_VALUE          0x1a

#define REG_PKT_SNR_VALUE           0x1b
//#define REG_PKT_SNR_VALUE          0x19

#define REG_MODEM_CONFIG_1          0x1d
#define REG_MODEM_CONFIG_2          0x1e
#define REG_PREAMBLE_MSB            0x20
#define REG_PREAMBLE_LSB            0x21
#define REG_PAYLOAD_LENGTH          0x22
#define REG_RSSI_WIDEBAND           0x2c
#define REG_DETECTION_OPTIMIZE      0x31
#define REG_NODE_ADDRESS             0x33 //
node address reg
#define REG_PACKET_CONFIG_1
0x30 //bts 1 and 2 are address filtering
#define REG_DETECTION_THRESHOLD     0x37
#define REG_SYNC_WORD               0x39
#define REG_DIO_MAPPING_1           0x40
#define REG_VERSION                 0x42

// modes
#define MODE_LONG_RANGE_MODE        0x80
#define MODE_SLEEP                  0x00
#define MODE_STDBY                  0x01
#define MODE_TX                     0x03
#define MODE_RX_CONTINUOUS          0x05
#define MODE_RX_SINGLE              0x06

//page 93; datasheet
#define REG_BITRATE_MSB             0x02
#define REG_BITRATE_LSB             0x03

//REG page 100; datasheet - BEACON (page 75
: definition of beacon )

// PA config
#define PA_BOOST                    0x80

// IRQ masks
#define IRQ_TX_DONE_MASK              0x08
#define IRQ_PAYLOAD_CRC_ERROR_MASK 0x20
#define IRQ_RX_DONE_MASK              0x40

#define MAX_PKT_LENGTH              255
#define       LOW    0
#define HIGH 1

#define function_not_required 0

int _packetIndex=0;
int _implicitHeaderMode=0;
int _frequency=0;
#if function_not_required
void (*_onReceive)(int);
#endif
char receiveLora=0;

#if function_not_required
void onReceivedata(int packetSize) {
  // received a packet
  LOGD("Received packet ");
  int i=0;
  // read packet
  for (i = 0; i < packetSize; i++) {
    LOGD("%c\n",(char)read());
  }

  // print RSSI of packet
  print("' with RSSI ",packetRssi());
}

void EINT3_IRQHandler(void)
{
       LPC_SC->EXTINT = EINT3;          /*
clear interrupt */

       LOGD("Interrupt triggered\n");

       // Toggle Led On
       //LPC_GPIO0->FIOPIN ^= (1<<2);

       handleDio0Rise();
       /*Clear interrupts */
       LPC_GPIOINT->IO0IntClr = 0xFFFFFFFF;

}
#endif

void gpioInit()
{
       // Select P0.2 as GPIO for RESET
       LPC_PINCON->PINSEL0 &= ~(3<<4);

       // P0.3 as GPIO
       LPC_PINCON->PINSEL0 &= ~(3<<6);

       // P0.2 as output For RESET
       LPC_GPIO0->FIODIR |= (1<<2);

       // P0.3 as input For DIO0
       //LPC_GPIO0->FIODIR &= ~(1<<3);

       //LPC_GPIOINT->IO0IntEnR |= (1<<3);

       //NVIC_EnableIRQ(EINT3_IRQn);
```

```
}

void digitalWrite(uint8_t pin, uint8_t val-
ue)
{
	LOGD("Pin : %d, value %d\n",pin,val-
ue);
	if(value == 1)
	{
		LPC_GPIO0->FIOPIN |=
(1<<pin);
	}
	else if(value == 0)
	{
		LPC_GPIO0->FIOPIN &=
~(1<<pin);
	}
}


int LoRabegin(long frequency)
{
  // setup pins
	uint8_t version =0;
	int i=0;
	gpioInit();
  // perform reset

  digitalWrite(LORA_DEFAULT_RESET_PIN,
LOW);
  for(i=0;i<100000;i++);
  digitalWrite(LORA_DEFAULT_RESET_PIN,
HIGH);
  for(i=0;i<10000000;i++);

  // set SS high
  CHIP_DESELECT();
  //digitalWrite(_ss, HIGH);
  SSP1Init();

  // start SPI
  //SPI.begin();

  // check version
  version = readRegister(REG_VERSION);
  LOGD("Version is %x\n",version);
  if (version != 0x12) {
    return 0;
  }

  // put in sleep mode
  sleep();

  // set frequency
  setFrequency(frequency);

  // set SF; Spreading Factor is essential-
ly chips per symbol
  // Register value SF ---> 2^SF chips per
symbol
  setSpreadingFactor(DEF_SF);

  // set base addresses
  writeRegister(REG_FIFO_TX_BASE_ADDR, 0);
  writeRegister(REG_FIFO_RX_BASE_ADDR, 0);

  // set LNA boost
  writeRegister(REG_LNA,
readRegister(REG_LNA) | 0x03);

  // set output power to 17 dBm
  setTxPower(DEF_POWER);

  //setting crc
  crc();

  //setting sync word
  setSyncWord(100);

  //setting node address
  //setNodeAddress();

  // put in standby mode
  idle();

  return 1;
}

void end()
{
  // put in sleep mode
  sleep();

  // stop SPI
  //SPI.end();
}

int LoRabeginPacket(int implicitHeader)
{
  // put in standby mode
  idle();

  if (implicitHeader)
  {
    implicitHeaderMode();
  }
  else {
```

```
    explicitHeaderMode();
  }

  // reset FIFO address and paload length
  writeRegister(REG_FIFO_ADDR_PTR, 0);
  writeRegister(REG_PAYLOAD_LENGTH, 0);

  return 1;
}

int LoRaendPacket()
{
  uint8_t rOut=0;

       // put in TX mode
  writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_TX);

  // wait for TX done
  uint8_t reg;
   while((reg = readRegister(REG_IRQ_FLAGS)
& IRQ_TX_DONE_MASK) == 0)
   {
       LOGD("reg =%x\n", reg);
   }

  // clear IRQ's writeRegister(REG_OP_MODE,
MODE_LONG_RANGE_MODE | MODE_TX);
  //readRegister(REG_IRQ_FLAGS) |
  writeRegister(REG_IRQ_FLAGS,  IRQ_TX_-
DONE_MASK);


  return 1;
}

void rxModeCheck()
{
    if (readRegister(REG_OP_MODE) != (MOD-
E_LONG_RANGE_MODE | MODE_RX_SINGLE))
    {
        // not currently in RX mode

        // reset FIFO address
        writeRegister(REG_FIFO_ADDR_PTR,
0);

        // put in single RX mode
        writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_SINGLE);
    }
}

int parsePacket(int size)
{
       int packetLength = 0;
       int irqFlags =
readRegister(REG_IRQ_FLAGS);
       //LOGD("irqFlags=%d\n", irqFlags);


       if (size > 0)
       {
              implicitHeaderMode();
              writeRegister(REG_PAYLOAD-
_LENGTH, size & 0xff);
       }
       else
       {
              explicitHeaderMode();
       }

       // clear IRQ's
       writeRegister(REG_IRQ_FLAGS, irq-
Flags);
       //writeRegister(REG_IRQ_FLAGS,
0xFF);

       if (   (irqFlags &
IRQ_RX_DONE_MASK))// && ((irqFlags & IRQ_-
PAYLOAD_CRC_ERROR_MASK) == 0))
       {
           if(irqFlags & IRQ_PAYLOAD_CR-
C_ERROR_MASK)
           {
               // CRC check failed!
               LOGD("CRC check failed\n");
               writeRegister(REG_IRQ_FLAGS,
readRegister(REG_IRQ_FLAGS) | (0x01 << 5));
           }
              // received a packet
              _packetIndex = 0;

              // read packet length
              if (_implicitHeaderMode)
              {
                     packetLength = readReg-
ister(REG_PAYLOAD_LENGTH);
              }
              else
              {
                     packetLength = readReg-
ister(REG_RX_NB_BYTES);
              }

              // set FIFO address to cur-
rent RX address
```

```
            writeRegister(REG_FIFO_AD-
DR_PTR, readRegister(REG_FIFO_RX_CURREN-
T_ADDR));

            // put in standby mode
            idle();
      }
      else if (readRegister(REG_OP_MODE) !
= (MODE_LONG_RANGE_MODE | MODE_RX_SINGLE))
      {
            // not currently in RX mode

            // reset FIFO address
            writeRegister(REG_FIFO_AD-
DR_PTR, 0);

            // put in single RX mode
            writeRegister(REG_OP_MODE,
MODE_LONG_RANGE_MODE | MODE_RX_SINGLE);
      }

      return packetLength;
}

int packetRssi()
{
      return (readRegister(REG_PKT_RSSI_-
VALUE) - (_frequency < 868E6 ? 164 : 157));
}

float packetSnr()
{
      return ((int8_t)readRegister(REG_PK-
T_SNR_VALUE)) * 0.25;
}

size_t writebyte(uint8_t byte)
{
      return write(&byte, sizeof(byte));
}

size_t write(const uint8_t *buffer, size_t
size)
{
      int currentLength =
readRegister(REG_PAYLOAD_LENGTH);
      size_t i=0;
      // check size
      if ((currentLength + size) > MAX_PK-
T_LENGTH)
      {
            size = MAX_PKT_LENGTH - cur-
rentLength;
      }

      // write data
      for (i = 0; i < size; i++)
      {
            writeRegister(REG_FIFO, buf-
fer[i]);
      }

      // update length
      writeRegister(REG_PAYLOAD_LENGTH,
currentLength + size);
      uint8_t reg = readRegister(REG_PAY-
LOAD_LENGTH);
      LOGD("reg val = %d == %d\n", reg,
currentLength + size);

      return size;
}

int available()
{
      return (readRegister(REG_RX_N-
B_BYTES) - _packetIndex);
}

int read()
{
      if (!available()) {
            return -1;
      }

      _packetIndex++;

      return readRegister(REG_FIFO);
}

int peek()
{
      if (!available()) {
            return -1;
      }

      // store current FIFO address
      int currentAddress =
readRegister(REG_FIFO_ADDR_PTR);

      // read
      uint8_t b = readRegister(REG_FIFO);

      // restore FIFO address
      writeRegister(REG_FIFO_ADDR_PTR,
currentAddress);

      return b;
```

```
}

void flush()
{
}

#if function_not_required
void onReceive(void(*callback)(int))
{
	_onReceive = callback;
	//writeRegister(REG_DIO_MAPPING_1,
0x00);
	if (callback)
	{
		writeRegister(REG_DIO_MAP-
PING_1, 0x00);

		//attachInterrupt(digitalPin-
ToInterrupt(_dio0), onDio0Rise, RISING);
	}
	else
	{
		//detachInterrupt(digitalPin-
ToInterrupt(_dio0));
	}
}

void receive(int size)
{
	if (size > 0)
	{
		implicitHeaderMode();
		writeRegister(REG_PAYLOAD-
_LENGTH, size & 0xff);
	}
	else
	{
		explicitHeaderMode();
	}

	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_CONTINUOUS);
}
#endif

void idle()
{
	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_STDBY);
}

void sleep()
{
	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_SLEEP);
}

void setTxPower(int level)
{
	if (level < 2)
	{
		level = 2;
	}
	else if (level > 17)
	{
		level = 17;
	}

	writeRegister(REG_PA_CONFIG, PA_-
BOOST | (level - 2));
}

void setFrequency(long frequency)
{
	_frequency = frequency;
	LOGD("frequency is %d,%d\n",frequen-
cy,_frequency);

	uint64_t frf = ((uint64_t)frequency
<< 19) / 32000000;

	writeRegister(REG_FRF_MSB, (uint8_t)
(frf >> 16));
	writeRegister(REG_FRF_MID, (uint8_t)
(frf >> 8));
	writeRegister(REG_FRF_LSB, (uint8_t)
(frf >> 0));
}

void setSpreadingFactor(int sf)
{
	if (sf < 6)
	{
		sf = 6;
	}
	else if (sf > 12)
	{
		sf = 12;
	}

	if (sf == 6)
	{
		writeRegister(REG_DETEC-
TION_OPTIMIZE, 0xc5);
		writeRegister(REG_DETEC-
TION_THRESHOLD, 0x0c);
	}
```

```
      else
      {
            writeRegister(REG_DETEC-
TION_OPTIMIZE, 0xc3);
            writeRegister(REG_DETEC-
TION_THRESHOLD, 0x0a);
      }

      writeRegister(REG_MODEM_CONFIG_2,
(readRegister(REG_MODEM_CONFIG_2) & 0x0f) |
((sf << 4) & 0xf0));
}

void setSignalBandwidth(long sbw)
{
  int bw;

  if (sbw <= 7.8E3) {
    bw = 0;
  } else if (sbw <= 10.4E3) {
    bw = 1;
  } else if (sbw <= 15.6E3) {
    bw = 2;
  } else if (sbw <= 20.8E3) {
    bw = 3;
  } else if (sbw <= 31.25E3) {
    bw = 4;
  } else if (sbw <= 41.7E3) {
    bw = 5;
  } else if (sbw <= 62.5E3) {
    bw = 6;
  } else if (sbw <= 125E3) {
    bw = 7;
  } else if (sbw <= 250E3) {
    bw = 8;
  } else /*if (sbw <= 250E3)*/ {
    bw = 9;
  }

  writeRegister(REG_MODEM_CONFIG_1, (read-
Register(REG_MODEM_CONFIG_1) & 0x0f) | (bw
<< 4));
}

void setCodingRate4(int denominator)
{
  if (denominator < 5) {
    denominator = 5;
  } else if (denominator > 8) {
    denominator = 8;
  }

  int cr = denominator - 4;

  writeRegister(REG_MODEM_CONFIG_1, (read-
Register(REG_MODEM_CONFIG_1) & 0xf1) | (cr
<< 1));
}

void setPreambleLength(long length)
{
  writeRegister(REG_PREAMBLE_MSB, (uint8_t)
(length >> 8));
  writeRegister(REG_PREAMBLE_LSB, (uint8_t)
(length >> 0));
}

void setSyncWord(int sw)
{
  writeRegister(REG_SYNC_WORD, sw);
}

void crc()
{
  writeRegister(REG_MODEM_CONFIG_2, read-
Register(REG_MODEM_CONFIG_2) | 0x04);
}

void noCrc()
{
  writeRegister(REG_MODEM_CONFIG_2, read-
Register(REG_MODEM_CONFIG_2) & 0xfb);
}

uint8_t random()
{
  return readRegister(REG_RSSI_WIDEBAND);
}
/*
void setPins(int ss, int reset, int dio0)
{
  _ss = ss;
  _reset = reset;
  _dio0 = dio0;
}

void dumpRegisters(Stream& out)
{
  for (int i = 0; i < 128; i++) {
    out.print("0x");
    out.print(i, HEX);
    out.print(": 0x");
    out.println(readRegister(i), HEX);
  }
}
*/
void explicitHeaderMode()
{
```

```
  _implicitHeaderMode = 0;

  uint8_t reg_modem_config_1 = readRegis-
ter(REG_MODEM_CONFIG_1);

  writeRegister(REG_MODEM_CONFIG_1, read-
Register(REG_MODEM_CONFIG_1) & 0xfe);
  //LOGD("reg_modem_config_1 was = %d; now
=%d should be=%d\n", reg_modem_config_1,
readRegister(REG_MODEM_CONFIG_1), reg_-
modem_config_1 & 0xfe);
}

void implicitHeaderMode()
{
  _implicitHeaderMode = 1;

  writeRegister(REG_MODEM_CONFIG_1, read-
Register(REG_MODEM_CONFIG_1) | 0x01);
}
#if function_not_required
void handleDio0Rise()
{
  int irqFlags =
readRegister(REG_IRQ_FLAGS);
  int i=0;
  // clear IRQ's
  writeRegister(REG_IRQ_FLAGS, irqFlags);

  if ((irqFlags & IRQ_PAYLOAD_CRC_ERROR_-
MASK) == 0) {
    // received a packet
    _packetIndex = 0;

    // read packet length
    int packetLength =
_implicitHeaderMode ? readRegister(REG_PAY-
LOAD_LENGTH) : readRegister(REG_RX_N-
B_BYTES);

    // set FIFO address to current RX ad-
dress
    writeRegister(REG_FIFO_ADDR_PTR, read-
Register(REG_FIFO_RX_CURRENT_ADDR));

//    if (_onReceive) {
//      _onReceive(packetLength);
//    }

    for (i = 0; i < packetLength; i++)
    {
       receiveLora = read();
       LOGD("Receive data is %c\n",receive-
Lora);
     }

    // reset FIFO address
    writeRegister(REG_FIFO_ADDR_PTR, 0);
  }
}
#endif

uint8_t readRegister(uint8_t address)
{
  return singleTransfer(address & 0x7f,
0x00);
}

void writeRegister(uint8_t address, uint8_t
value)
{

       singleTransfer(address | 0x80, val-
ue);
}

uint8_t singleTransfer(uint8_t address,
uint8_t value)
{
  uint8_t response=0;

  //digitalWrite(_ss, LOW);
  CHIP_SELECT();
  //SPI.beginTransaction(_spiSettings);
  //LOGD("areq %x\n", address);
  response = ssp1Transfer(address);
  //LOGD("response %x\n",response);
  //LOGD("vreq %x\n", value);
  response = ssp1Transfer(value);
  //LOGD("response %x\n",response);
  CHIP_DESELECT();

  //digitalWrite(_ss, HIGH);

  return response;
}

void setNodeAddress()
{

       writeRegister(REG_NODE_ADDRESS,
0x28);
}

void setInRxMode()
{
       if (readRegister(REG_OP_MODE) !=
(MODE_LONG_RANGE_MODE | MODE_RX_SINGLE))
```

```
                    {
                          // not currently
in RX mode

                          // reset FIFO
address

writeRegister(REG_FIFO_ADDR_PTR, 0);

                          // put in single
RX mode

writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_SINGLE);
                    }
}

uint32_t getBitrate()
{
    return readRegister(REG_BITRATE_MSB) <<
8 | readRegister(REG_BITRATE_LSB);
}

#if function_not_required
void onDio0Rise()
{
      handleDio0Rise();
}
#endif



/
*****************************************
*********************************
 *   $Id:: timer.h 5823 2010-12-07
19:01:00Z usb00423                      $
 *   Project: NXP LPC17xx Timer example
 *
 *   Description:
 *     This file contains Timer code header
definition.
 *

*****************************************
*********************************
 * Software that is described herein is for
illustrative purposes only
 * which provides customers with program-
ming information regarding the
 * products. This software is supplied "AS
IS" without any warranties.
 * NXP Semiconductors assumes no responsi-
bility or liability for the
 * use of the software, conveys no license
or title under any patent,
 * copyright, or mask work right to the
product. NXP Semiconductors
 * reserves the right to make changes in
the software without
 * notification. NXP Semiconductors also
make no representation or
 * warranty that such application will be
suitable for the specified
 * use without further testing or modifica-
tion.
*****************************************
*********************************/
#ifndef __TIMER_H
#define __TIMER_H
#define Stop 0
/* The test is either MAT_OUT or CAP_IN.
Default is MAT_OUT. */
/* If running DMA test, External match is
not needed to trigger DMA, but still
set timer as MATx instead of CAPx. */
#define TIMER_MATCH        1

/* TIME_INTERVALmS is a value to load the
timer match register with
   to get a 1 mS delay */
#define TIME_INTERVALmS    1000

#define TIME_INTERVAL      (9000000/100 -
1)

extern void delayMs(uint8_t timer_num, uin-
t32_t delayInMs);
extern uint32_t init_timer( uint8_t
timer_num, uint32_t timerInterval );
extern void enable_timer( uint8_t timer_num
);
extern void disable_timer( uint8_t
timer_num );
extern void reset_timer( uint8_t
timer_num );
#if Stop
extern void TIMER0_IRQHandler (void);
extern void TIMER1_IRQHandler (void);
#endif
extern void TIMER2_IRQHandler (void);
extern void TIMER3_IRQHandler (void);

#endif /* end __TIMER_H */
/
*****************************************
*********************************
**                            End Of File
```

```
*****************************************
*********************************/




#include "LPC17xx.h"
#include "common.h"
#define Stop 0
extern unsigned int SystemCoreClock;
unsigned int PreScaleMillli(uint8_t timerP-
clkBit);

uint64_t gnTimeTicks = 0;

uint64_t getTimeTickCount()
{
    return gnTimeTicks;
}
void timer_initialise(void)
{
       SystemInit();
       printf("timer init\n");
    /* Powe on Timer 1 and 0 */
    LPC_SC->PCONP |= (1<<LPC_TIMER0) |
(1<<LPC_TIMER1);

    LPC_TIM0->MCR  = (1<<SBIT_MR0I) |
(1<<SBIT_MR0R);
    LPC_TIM0->PR   = PreScaleMillli(PCLK_-
TIMER0);
    LPC_TIM0->MR0  = TIME_IN_MILLI;
    LPC_TIM0->TCR  = (1 <<TIMER_ENABLE);
    NVIC_EnableIRQ(TIMER0_IRQn);

/*
    LPC_TIM1->MCR  = (1<<SBIT_MR0I) |
(1<<SBIT_MR0R);
    LPC_TIM1->PR   = PreScaleMillli(PCLK_-
TIMER1);
    LPC_TIM1->MR0  = TIME_IN_MILLI;
    LPC_TIM1->TCR  = (1 <<TIMER_ENABLE);
    NVIC_EnableIRQ(TIMER1_IRQn);
*/

}
void TIMER0_IRQHandler(void)
{
    //printf("Timer 0 new code\n");
    unsigned int isrMask;
    isrMask = LPC_TIM0->IR;
    LPC_TIM0->IR = isrMask;         /*
Clear the Interrupt Bit */
    //LPC_TIM0->IR = 0xFF;
    gnTimeTicks++;
    //printf("hello\n");

#if Stop
    char temp;
    int temp1;
    temp1 = parsePacket(0);
       if(temp1)
       {
              temp = read();
              if(temp == 'X')
                     check_ack();
       }
       else
              printf("s/n");

#endif

}
void TIMER1_IRQHandler(void)
{
       //printf("Timer 1 new code");
    unsigned int isrMask;
    isrMask = LPC_TIM1->IR;
    LPC_TIM1->IR = isrMask;        /* Clear
the Interrupt Bit */
#if Stop

#endif
}
unsigned int PreScaleMillli(uint8_t timerP-
clkBit)
{
    unsigned int pclk,prescalarForUs;
    pclk = (LPC_SC->PCLKSEL0 >> timerPclk-
Bit) & 0x03;  /* get the pclk info for re-
quired timer */
    switch ( pclk )
/* Decode the bits to determine the pclk*/
    {
    case 0x00:
        pclk = SystemCoreClock/4;
        break;
    case 0x01:
        pclk = SystemCoreClock;
        break;
    case 0x02:
        pclk = SystemCoreClock/2;
        break;
    case 0x03:
        pclk = SystemCoreClock/8;
        break;
    default:
```

```
        pclk = SystemCoreClock/4;
        break;
    }
    /* Set theprescale for milli */
    prescalarForUs =pclk/1000 - 1;
    return prescalarForUs;

}
```

LoRa Transmit code:

```
/*
===============================================
=====================================
 Name        : RF_Handshaking.c
 Description : RF and PWM code for LPC1769

 CTI One Corporation released for Dr. Harry
Li for CMPE 245 Class use ONLY!
===============================================
=====================================
*/

//#include "FreeRTOS.h"
//#include "FreeRTOS.h"
#ifdef __USE_CMSIS
//#include "LPC17xx.h"
#endif

#include <cr_section_macros.h>

#include <stdio.h>

#include <stdbool.h>
#include "LoRa.h"
#include"common.h"
#include "timer.h"

#define RF_Receive 1
#define RF_Transmit 0
#define TransmittACk 0
#define ack_start_stop 0
int rfInit(void);

char receiveData=0;
int packetSize;


/
*****************************************
*****************************************
************
* @brief      wait for ms amount of mil-
liseconds
* @param      ms : Time to wait in mil-
liseconds
*****************************************
*****************************************
************/
static void delay_ms(unsigned int ms)
{
    unsigned int i,j;
    for(i=0;i<ms;i++)
        for(j=0;j<50000;j++);
}

/
*****************************************
*****************************************
************
* @brief      wait for delayInMs amount of
milliseconds
* @param      delayInMs : Time to wait in
milliseconds
*****************************************
*****************************************
************/
static void delay(uint32_t delayInMs)
{
       LPC_TIM0->TCR = 0x02;            /*
reset timer */
       LPC_TIM0->PR  = 0x00;            /*
set prescaler to zero */
       LPC_TIM0->MR0 = delayInMs * (9000000
/ 1000-1);
       LPC_TIM0->IR  = 0xff;            /*
reset all interrrupts */
       LPC_TIM0->MCR = 0x04;            /*
stop timer on match */
       LPC_TIM0->TCR = 0x01;            /*
start timer */
```

```
	/* wait until delay time has elapsed
*/
	while (LPC_TIM0->TCR & 0x01);
}

uint32_t gnSendPackCount = 0;

void sendData(int SFNew)
{
	char buffer[1024];

	int nSendPackSize = snprintf(buffer,
1024, "WIRE*LESS* 100*CMP%d*;P=%dSF=%d",
gnSendPackCount++ % 10,getTxPower(),SFNew);
	//char Acknowledgement;
	//Acknowledgement = 'A';
	printf("Start Sending data TXPow is
%d SF is %d \n", getTxPower(), SFNew);
	//delay_ms(1000);
	LoRabeginPacket(0);
	//writebyte(Acknowledgement);
	write(buffer, nSendPackSize);
	LoRaendPacket();
	printf("Data sent \n");
}

void improveTxPower()
{
	int TxPowerCurrent = getTxPower();
	//printf("TxPower was %d" ,TxPower-
Current);
	setTxPower(++TxPowerCurrent);
	//printf("TxPower new is %d" ,TxPow-
erCurrent);

}

int calculateNewSF()
{
	if(getTxPower() >= 17)
	{
		int newSF;
		if(getSpreadingFactor() >=8)
		{
			newSF = getSpreading-
Factor() -1;
			return newSF;
		}

	}
	else
	{
		return getSpreadingFactor();
	}
}


/
*****************************************
*****************************************
************
* main : Main program entry
*****************************************
*****************************************
************/
int main(void)
{
	printf("System clock is %d\n",Sys-
temCoreClock);
	uint64_t nTimeElapsedBetweenPackets
= 0;
	uint64_t nTimeTickCount1 = 0;
	double fPerformanceIndex = 100.0;
	double fDelayPerformanceDrop = 0.0;
	double fCorruptionDrop = 0.0;
	// Working frequency range from 724
MHz to 1040 MHz.
	//LoRabegin(1040000000);
	//LoRabegin(1020000000);
	//LoRabegin(724000000);
	//LoRabegin(750000000);
	//LoRabegin(790000000);
	//LoRabegin(800000000);
	//LoRabegin(845000000);
	//LoRabegin(850000000);
	//LoRabegin(910000000);
	//LoRabegin(868000000);
	LoRabegin(869000000);

	//LoRabegin(915000000);
	int counter =0;
	timer_initialise();
	int nCount = 0;
	float recdPerformance;
	int SFNew = getSpreadingFactor();
	/* Start the tasks running. */
	//vTaskStartScheduler();

	/* If all is well we will never
reach here as the scheduler will now be
	running.  If we do reach here then
it is likely that there was insufficient
	heap available for the idle task to
be created. */
#if RF_Receive
	while(1)
	{
```

```
            if(getTimeTickCount() - nCount <
10)
            {
                sendData(SFNew);
            }
            else
            {
               setSpreadingFactor(SFNew);
               while(1)
               {
                      rxModeCheck();
                      //delay_ms(1000);

                      //printf("parsePacket:
\n");
                      packetSize = parsePack-
et(0);
                      //printf("parsePacket
size=%d %llu\n", packetSize, getTimeTick-
Count());
                      if (packetSize)
                      {
                             int count = 0;
                             counter = 0;
                             //NVIC_Enable-
IRQ(TIMER0_IRQn);
                             //received a
packet
                             //printf("Re-
ceived packet \n");
                             // read packet
                             if(nTimeTick-
Count1)
                             {
                                    nTimeE-
lapsedBetweenPackets = getTimeTickCount() -
nTimeTickCount1;
                             }
                             nTimeTickCount1
= getTimeTickCount();

                             //buffer to
store the recd data and set perf as 0 again
                             char
recdbuffer[1024]=""; int index =0;
                             recdPerformance
=0;
                             while (avail-
able() && (++count <= packetSize))
                             {
                                    counter =
0;
                                    receive-
Data = read();
                                    //
printf("%c\n",receiveData);
                                    //return
0;
                                    // print
RSSI of packet

printf("Received packet '");

printf("%c",receiveData);

                                    //adding
to recd buffer
                                    recdbuf-
fer[index] = receiveData; index++;

                                    printf("'
with RSSI ");

printf("%d;
nTimeElapsedS=%d\n",packetRssi(), nTimeE-
lapsedBetweenPackets);
                                    /** for
each second more of the time between pack-
ets,
                                     * reduce
5% of the performance index
                                     *  */
                                    if(nTime-
ElapsedBetweenPackets > 3)
                                    {

fDelayPerformanceDrop += 5.0;
                                    }
                                    else
                                    {

fDelayPerformanceDrop = 0.0;
                                    }

                                    fPerfor-
manceIndex -= fDelayPerformanceDrop;
                             }
                             if(count ==
packetSize)
                             {

sscanf(recdbuffer,"%f",&recdPerformance);
                                    if(recd-
Performance < 85)
                                    {
                                           im-
proveTxPower(recdPerformance);
```

```
SFNew = calculateNewSF();
                                    }

                                    nCount =
getTimeTickCount();
                                    break;
                                }
                            }
                        }
                    }

        }


#if TransmittACk
        const char buffer[] = "Data from
LPC1769";
        char Acknowledgement;
        Acknowledgement = 'A';
        while(1)
        {
                printf("Start Sending data
\n");
                delay_ms(1000);
                LoRabeginPacket(0);
                //writebyte(Acknowledgement);
                write(buffer,
sizeof(buffer));
                LoRaendPacket();
                printf("Data sent \n");

        }

#endif

#endif
#if RF_Transmit
    const char buffer[] = "Data from
LPC1769";
    char Acknowledgement;
    Acknowledgement = 'A';
    while(1)
    {
        printf("Start Sending data \n");
        delay_ms(1000);
        printf("delay done\n");
        LoRabeginPacket(0);
        //writebyte(Acknowledgement);
        printf("begin packet\n");
       size_t ret = write(buffer,
sizeof(buffer));
        printf("write done %d \n",ret);
        LoRaendPacket();
                printf("Data sent \n");

    }

#endif
}

void check_ack()
{
       printf("1 \n");

//
NVIC_EnableIRQ(TIMER0_IRQn);
                //
NVIC_DisableIRQ(TIMER0_IRQn);
}
```

```
/*
 * LoRa.h
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */

#ifndef LORA_H_
#define LORA_H_

/*
 * LoRa.h
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */


#include <stddef.h>
#include "ssp.h"

//#define LORA_DEFAULT_SS_PIN    10
#define LORA_DEFAULT_RESET_PIN          2
#define LORA_DEFAULT_DIO0_PIN           3



  int LoRabegin(long frequency);
  void end();
```

```
  int LoRabeginPacket(int implicitHeader);
  int LoRaendPacket();

  int parsePacket(int size );
  void rxModeCheck();
  int packetRssi();
  float packetSnr();

  // from Print
  size_t writebyte(uint8_t byte);
  size_t write(const uint8_t *buffer,
size_t size);

  // from Stream
  int available();
  int read();
  int peek();
  void flush();

  void onReceive(void(*callback)(int));

  void receive(int size);
  void idle();
  void sleep();

  void setTxPower(int level);
  int getTxPower();
  void setFrequency(long frequency);
  void setSpreadingFactor(int sf);
  int getSpreadingFactor();
  void setSignalBandwidth(long sbw);
  void setCodingRate4(int denominator);
  void setPreambleLength(long length);
  void setSyncWord(int sw);
  void crc();
  void noCrc();

  uint8_t random();



 // void dumpRegisters(Stream& out);
uint8_t readRegister(uint8_t address);

  void explicitHeaderMode();
  void implicitHeaderMode();

  void handleDio0Rise();


  void writeRegister(uint8_t address, uin-
t8_t value);
  uint8_t singleTransfer(uint8_t address,
uint8_t value);
```

```
  static void onDio0Rise();



void digitalWrite(uint8_t pin, uint8_t val-
ue);

void gpioInit();



#endif /* LORA_H_ */
```

```
/*
 * LoRa.c
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */

/*
 * LoRa.cpp
 *
 *  Created on: Oct 29, 2017
 *  CTI One Corporation released for Dr.
Harry Li for CMPE 245 Class use ONLY!
 */


#include <stdio.h>
#include <stddef.h>
#include "LoRa.h"
#include "ssp.h"
#include "extint.h"
//#include "FreeRTOS.h"
// registers
#define REG_FIFO                 0x00
#define REG_OP_MODE              0x01
#define REG_FRF_MSB              0x06
#define REG_FRF_MID              0x07
```

```
#define REG_FRF_LSB              0x08
#define REG_PA_CONFIG            0x09
#define REG_LNA                  0x0c
#define REG_FIFO_ADDR_PTR        0x0d
#define REG_FIFO_TX_BASE_ADDR    0x0e
#define REG_FIFO_RX_BASE_ADDR    0x0f
#define REG_FIFO_RX_CURRENT_ADDR 0x10
#define REG_IRQ_FLAGS            0x12
#define REG_RX_NB_BYTES          0x13
#define REG_PKT_RSSI_VALUE       0x1a

#define REG_PKT_SNR_VALUE        0x1b
//#define REG_PKT_SNR_VALUE        0x19

#define REG_MODEM_CONFIG_1       0x1d
#define REG_MODEM_CONFIG_2       0x1e
#define REG_PREAMBLE_MSB         0x20
#define REG_PREAMBLE_LSB         0x21
#define REG_PAYLOAD_LENGTH       0x22
#define REG_RSSI_WIDEBAND        0x2c
#define REG_DETECTION_OPTIMIZE   0x31
#define REG_NODE_ADDRESS          0x33 //
node address reg
#define REG_PACKET_CONFIG_1
0x30 //bts 1 and 2 are address filtering
#define REG_DETECTION_THRESHOLD  0x37
#define REG_SYNC_WORD            0x39
#define REG_DIO_MAPPING_1        0x40
#define REG_VERSION              0x42

// modes
#define MODE_LONG_RANGE_MODE     0x80
#define MODE_SLEEP               0x00
#define MODE_STDBY               0x01
#define MODE_TX                  0x03
#define MODE_RX_CONTINUOUS       0x05
#define MODE_RX_SINGLE           0x06

// PA config
#define PA_BOOST                 0x80

// IRQ masks
#define IRQ_TX_DONE_MASK           0x08
#define IRQ_PAYLOAD_CRC_ERROR_MASK 0x20
#define IRQ_RX_DONE_MASK           0x40

#define MAX_PKT_LENGTH           255
#define      LOW    0
#define HIGH 1

#define function_not_required 0

int _packetIndex=0;
int _implicitHeaderMode=0;
int _frequency=0;
int _TxPower=0;
int _spreadingFactor =0;
#if function_not_required
void (*_onReceive)(int);
#endif
char receiveLora=0;

#if function_not_required
void onReceivedata(int packetSize) {
  // received a packet
  printf("Received packet ");
  int i=0;
  // read packet
  for (i = 0; i < packetSize; i++) {
    printf("%c\n",(char)read());
  }

  // print RSSI of packet
  print("' with RSSI ",packetRssi());
}

void EINT3_IRQHandler(void)
{
       LPC_SC->EXTINT = EINT3;          /*
clear interrupt */

       printf("Interrupt triggered\n");

       // Toggle Led On
       //LPC_GPIO0->FIOPIN ^= (1<<2);

       handleDio0Rise();
       /*Clear interrupts */
       LPC_GPIOINT->IO0IntClr = 0xFFFFFFFF;

}
#endif

void gpioInit()
{
       // Select P0.2 as GPIO for RESET
       LPC_PINCON->PINSEL0 &= ~(3<<4);

       // P0.3 as GPIO
       LPC_PINCON->PINSEL0 &= ~(3<<6);

       // P0.2 as output For RESET
       LPC_GPIO0->FIODIR |= (1<<2);

       // P0.3 as input For DIO0
       //LPC_GPIO0->FIODIR &= ~(1<<3);

       //LPC_GPIOINT->IO0IntEnR |= (1<<3);
```

```
	//NVIC_EnableIRQ(EINT3_IRQn);
}

void digitalWrite(uint8_t pin, uint8_t val-
ue)
{
	printf("Pin : %d, value
%d\n",pin,value);
	if(value == 1)
	{
		LPC_GPIO0->FIOPIN |=
(1<<pin);
	}
	else if(value == 0)
	{
		LPC_GPIO0->FIOPIN &=
~(1<<pin);
	}
}


int LoRabegin(long frequency)
{
  // setup pins
	uint8_t version =0;
	int i=0;
	gpioInit();
  // perform reset

  digitalWrite(LORA_DEFAULT_RESET_PIN,
LOW);
  for(i=0;i<100000;i++);
  digitalWrite(LORA_DEFAULT_RESET_PIN,
HIGH);
  for(i=0;i<10000000;i++);

  // set SS high
  CHIP_DESELECT();
  //digitalWrite(_ss, HIGH);
  SSP1Init();

  // start SPI
  //SPI.begin();

  // check version
  version = readRegister(REG_VERSION);
  printf("Version is %x\n",version);
  if (version != 0x12) {
    return 0;
  }

  // put in sleep mode
  sleep();

  // set frequency
  setFrequency(frequency);

  // set SF; Spreading Factor is essential-
ly chips per symbol
  // Register value SF ---> 2^SF chips per
symbol
  setSpreadingFactor(12);

  // set base addresses
  writeRegister(REG_FIFO_TX_BASE_ADDR, 0);
  writeRegister(REG_FIFO_RX_BASE_ADDR, 0);

  // set LNA boost
  writeRegister(REG_LNA,
readRegister(REG_LNA) | 0x03);

  // set output power to 17 dBm
  setTxPower(2);

  //setting crc
  crc();

  //setting sync word
  setSyncWord(100);

  //setting node address
  //setNodeAddress();

  // put in standby mode
  idle();

  return 1;
}

void end()
{
  // put in sleep mode
  sleep();

  // stop SPI
  //SPI.end();
}

int LoRabeginPacket(int implicitHeader)
{
  // put in standby mode
  idle();

  if (implicitHeader)
  {
    implicitHeaderMode();
  }
```

```
  else {

    explicitHeaderMode();
  }

  // reset FIFO address and paload length
  writeRegister(REG_FIFO_ADDR_PTR, 0);
  writeRegister(REG_PAYLOAD_LENGTH, 0);

  return 1;
}

int LoRaendPacket()
{
  uint8_t rOut=0;

       // put in TX mode
  writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_TX);

  // wait for TX done
  uint8_t reg;
   while((reg = readRegister(REG_IRQ_FLAGS)
& IRQ_TX_DONE_MASK) == 0)
   {
       //printf("reg =%x\n", reg);
   }

  // clear IRQ's writeRegister(REG_OP_MODE,
MODE_LONG_RANGE_MODE | MODE_TX);
   //readRegister(REG_IRQ_FLAGS) |
  writeRegister(REG_IRQ_FLAGS,   IRQ_TX_-
DONE_MASK);


  return 1;
}

void rxModeCheck()
{
    if (readRegister(REG_OP_MODE) != (MOD-
E_LONG_RANGE_MODE | MODE_RX_SINGLE))
    {
        // not currently in RX mode

        // reset FIFO address
        writeRegister(REG_FIFO_ADDR_PTR,
0);

        // put in single RX mode
        writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_SINGLE);
    }
}
```

```
int parsePacket(int size)
{
       int packetLength = 0;
       int irqFlags =
readRegister(REG_IRQ_FLAGS);
       //printf("irqFlags=%d\n", irqFlags);


       if (size > 0)
       {
              implicitHeaderMode();
              writeRegister(REG_PAYLOAD-
_LENGTH, size & 0xff);
       }
       else
       {
              explicitHeaderMode();
       }

       // clear IRQ's
       writeRegister(REG_IRQ_FLAGS, irq-
Flags);
       //writeRegister(REG_IRQ_FLAGS,
0xFF);

       if (   (irqFlags &
IRQ_RX_DONE_MASK))// && ((irqFlags & IRQ_-
PAYLOAD_CRC_ERROR_MASK) == 0))
       {
           if(irqFlags & IRQ_PAYLOAD_CR-
C_ERROR_MASK)
           {
               // CRC check failed!
               printf("CRC check
failed\n");
               writeRegister(REG_IRQ_FLAGS,
readRegister(REG_IRQ_FLAGS) | (0x01 << 5));
           }
              // received a packet
              _packetIndex = 0;

              // read packet length
              if (_implicitHeaderMode)
              {
                     packetLength = readReg-
ister(REG_PAYLOAD_LENGTH);
              }
              else
              {
                     packetLength = readReg-
ister(REG_RX_NB_BYTES);
              }
```

```
            // set FIFO address to cur-
rent RX address
            writeRegister(REG_FIFO_AD-
DR_PTR, readRegister(REG_FIFO_RX_CURREN-
T_ADDR));

            // put in standby mode
            idle();
      }
      else if (readRegister(REG_OP_MODE) !
= (MODE_LONG_RANGE_MODE | MODE_RX_SINGLE))
      {
            // not currently in RX mode

            // reset FIFO address
            writeRegister(REG_FIFO_AD-
DR_PTR, 0);

            // put in single RX mode
            writeRegister(REG_OP_MODE,
MODE_LONG_RANGE_MODE | MODE_RX_SINGLE);
      }

      return packetLength;
}

int packetRssi()
{
      return (readRegister(REG_PKT_RSSI_-
VALUE) - (_frequency < 868E6 ? 164 : 157));
}

float packetSnr()
{
      return ((int8_t)readRegister(REG_PK-
T_SNR_VALUE)) * 0.25;
}

size_t writebyte(uint8_t byte)
{
      return write(&byte, sizeof(byte));
}

size_t write(const uint8_t *buffer, size_t
size)
{
      int currentLength =
readRegister(REG_PAYLOAD_LENGTH);
      size_t i=0;
      // check size
      if ((currentLength + size) > MAX_PK-
T_LENGTH)
      {
            size = MAX_PKT_LENGTH - cur-
rentLength;
      }

      // write data
      for (i = 0; i < size; i++)
      {
            writeRegister(REG_FIFO, buf-
fer[i]);
      }

      // update length
      writeRegister(REG_PAYLOAD_LENGTH,
currentLength + size);
      uint8_t reg = readRegister(REG_PAY-
LOAD_LENGTH);
      //printf("reg val = %d == %d\n",
reg, currentLength + size);

      return size;
}

int available()
{
      return (readRegister(REG_RX_N-
B_BYTES) - _packetIndex);
}

int read()
{
      if (!available()) {
            return -1;
      }

      _packetIndex++;

      return readRegister(REG_FIFO);
}

int peek()
{
      if (!available()) {
            return -1;
      }

      // store current FIFO address
      int currentAddress =
readRegister(REG_FIFO_ADDR_PTR);

      // read
      uint8_t b = readRegister(REG_FIFO);

      // restore FIFO address
```

```
	writeRegister(REG_FIFO_ADDR_PTR,
currentAddress);

	return b;
}

void flush()
{
}

#if function_not_required
void onReceive(void(*callback)(int))
{
	_onReceive = callback;
	//writeRegister(REG_DIO_MAPPING_1,
0x00);
	if (callback)
	{
		writeRegister(REG_DIO_MAP-
PING_1, 0x00);

		//attachInterrupt(digitalPin-
ToInterrupt(_dio0), onDio0Rise, RISING);
	}
	else
	{
		//detachInterrupt(digitalPin-
ToInterrupt(_dio0));
	}
}

void receive(int size)
{
	if (size > 0)
	{
		implicitHeaderMode();
		writeRegister(REG_PAYLOAD-
_LENGTH, size & 0xff);
	}
	else
	{
		explicitHeaderMode();
	}

	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_CONTINUOUS);
}
#endif

void idle()
{
	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_STDBY);
}

void sleep()
{
	writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_SLEEP);
}

void setTxPower(int level)
{
	if (level < 2)
	{
		level = 2;
	}
	else if (level > 17)
	{
		level = 17;
	}

	writeRegister(REG_PA_CONFIG, PA_-
BOOST | (level - 2));
	_TxPower = level;
}

int getTxPower()
{
	return _TxPower;
}
void setFrequency(long frequency)
{
	_frequency = frequency;
	printf("frequency is %d,%d\n",fre-
quency,_frequency);

	uint64_t frf = ((uint64_t)frequency
<< 19) / 32000000;

	writeRegister(REG_FRF_MSB, (uint8_t)
(frf >> 16));
	writeRegister(REG_FRF_MID, (uint8_t)
(frf >> 8));
	writeRegister(REG_FRF_LSB, (uint8_t)
(frf >> 0));
}

void setSpreadingFactor(int sf)
{
	if (sf < 6)
	{
		sf = 6;
	}
	else if (sf > 12)
	{
		sf = 12;
	}
```

```
	if (sf == 6)
	{
		writeRegister(REG_DETEC-
TION_OPTIMIZE, 0xc5);
		writeRegister(REG_DETEC-
TION_THRESHOLD, 0x0c);
	}
	else
	{
		writeRegister(REG_DETEC-
TION_OPTIMIZE, 0xc3);
		writeRegister(REG_DETEC-
TION_THRESHOLD, 0x0a);
	}

	writeRegister(REG_MODEM_CONFIG_2,
(readRegister(REG_MODEM_CONFIG_2) & 0x0f) |
((sf << 4) & 0xf0));
	_spreadingFactor = sf;
}

int getSpreadingFactor()
{
	return _spreadingFactor;
}

void setSignalBandwidth(long sbw)
{
  int bw;

  if (sbw <= 7.8E3) {
    bw = 0;
  } else if (sbw <= 10.4E3) {
    bw = 1;
  } else if (sbw <= 15.6E3) {
    bw = 2;
  } else if (sbw <= 20.8E3) {
    bw = 3;
  } else if (sbw <= 31.25E3) {
    bw = 4;
  } else if (sbw <= 41.7E3) {
    bw = 5;
  } else if (sbw <= 62.5E3) {
    bw = 6;
  } else if (sbw <= 125E3) {
    bw = 7;
  } else if (sbw <= 250E3) {
    bw = 8;
  } else /*if (sbw <= 250E3)*/ {
    bw = 9;
  }
  writeRegister(REG_MODEM_CONFIG_1, (read-
Register(REG_MODEM_CONFIG_1) & 0x0f) | (bw
<< 4));
}

void setCodingRate4(int denominator)
{
  if (denominator < 5) {
    denominator = 5;
  } else if (denominator > 8) {
    denominator = 8;
  }

  int cr = denominator - 4;

  writeRegister(REG_MODEM_CONFIG_1, (read-
Register(REG_MODEM_CONFIG_1) & 0xf1) | (cr
<< 1));
}

void setPreambleLength(long length)
{
  writeRegister(REG_PREAMBLE_MSB, (uint8_t)
(length >> 8));
  writeRegister(REG_PREAMBLE_LSB, (uint8_t)
(length >> 0));
}

void setSyncWord(int sw)
{
  writeRegister(REG_SYNC_WORD, sw);
}

void crc()
{
  writeRegister(REG_MODEM_CONFIG_2, read-
Register(REG_MODEM_CONFIG_2) | 0x04);
}

void noCrc()
{
  writeRegister(REG_MODEM_CONFIG_2, read-
Register(REG_MODEM_CONFIG_2) & 0xfb);
}

uint8_t random()
{
  return readRegister(REG_RSSI_WIDEBAND);
}
/*
void setPins(int ss, int reset, int dio0)
{
  _ss = ss;
  _reset = reset;
```

```
  _dio0 = dio0;
}

void dumpRegisters(Stream& out)
{
  for (int i = 0; i < 128; i++) {
    out.print("0x");
    out.print(i, HEX);
    out.print(": 0x");
    out.println(readRegister(i), HEX);
  }
}
*/
void explicitHeaderMode()
{
  _implicitHeaderMode = 0;

  uint8_t reg_modem_config_1 = readRegis-
ter(REG_MODEM_CONFIG_1);

  writeRegister(REG_MODEM_CONFIG_1, read-
Register(REG_MODEM_CONFIG_1) & 0xfe);
  //printf("reg_modem_config_1 was = %d;
now =%d should be=%d\n", reg_modem_con-
fig_1, readRegister(REG_MODEM_CONFIG_1),
reg_modem_config_1 & 0xfe);
}

void implicitHeaderMode()
{
  _implicitHeaderMode = 1;

  writeRegister(REG_MODEM_CONFIG_1, read-
Register(REG_MODEM_CONFIG_1) | 0x01);
}
#if function_not_required
void handleDio0Rise()
{
  int irqFlags =
readRegister(REG_IRQ_FLAGS);
  int i=0;
  // clear IRQ's
  writeRegister(REG_IRQ_FLAGS, irqFlags);

  if ((irqFlags & IRQ_PAYLOAD_CRC_ERROR_-
MASK) == 0) {
    // received a packet
    _packetIndex = 0;

    // read packet length
    int packetLength =
_implicitHeaderMode ? readRegister(REG_PAY-
LOAD_LENGTH) : readRegister(REG_RX_N-
B_BYTES);

    // set FIFO address to current RX ad-
dress
    writeRegister(REG_FIFO_ADDR_PTR, read-
Register(REG_FIFO_RX_CURRENT_ADDR));

//    if (_onReceive) {
//      _onReceive(packetLength);
//    }

    for (i = 0; i < packetLength; i++)
    {
       receiveLora = read();
       printf("Receive data is %c\n",re-
ceiveLora);
     }

    // reset FIFO address
    writeRegister(REG_FIFO_ADDR_PTR, 0);
  }
}
#endif

uint8_t readRegister(uint8_t address)
{
  return singleTransfer(address & 0x7f,
0x00);
}

void writeRegister(uint8_t address, uint8_t
value)
{

       singleTransfer(address | 0x80, val-
ue);
}

uint8_t singleTransfer(uint8_t address,
uint8_t value)
{
  uint8_t response=0;

  //digitalWrite(_ss, LOW);
  CHIP_SELECT();
  //SPI.beginTransaction(_spiSettings);
  //printf("areq %x\n", address);
  response = ssp1Transfer(address);
  //printf("response %x\n",response);
  //printf("vreq %x\n", value);
  response = ssp1Transfer(value);
  //printf("response %x\n",response);
  CHIP_DESELECT();

  //digitalWrite(_ss, HIGH);
```

```
  return response;
}

void setNodeAddress()
{

       writeRegister(REG_NODE_ADDRESS,
0x28);
}

void setInRxMode()
{
       if (readRegister(REG_OP_MODE) !=
(MODE_LONG_RANGE_MODE | MODE_RX_SINGLE))
                     {
                            // not currently
in RX mode

                            // reset FIFO
address

writeRegister(REG_FIFO_ADDR_PTR, 0);

                            // put in single
RX mode

writeRegister(REG_OP_MODE, MODE_-
LONG_RANGE_MODE | MODE_RX_SINGLE);
                     }
}

#if function_not_required
void onDio0Rise()
{
       handleDio0Rise();
}
#endif
```