Introduction to Microcontrollers

Experiment 5:

(KEY FOR ECE AND CYENG STUDENTS)

Encrypted Communication with LoRa

Developed 2020 by S. Rowland and M. Eckels

Contents

1 Objectives

To utilize LoRa wireless radio modules for setting up end-to-end encrypted communication.

2 Background

2.1 Encryption

2.1.1 Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data. AES is a subset of the Rijndael block cipher – a deterministic algorithm with an unchanging transformation that is set by a symmetric key. This encryption standard was designed for efficient software and hardware encryption allowing for widespread usage; even on low-end hardware.

AES works on a design principle known as a Substitution-Permutation Network (SPN) which is a series of linked mathematical operations; as its name suggests the network consists of substitution boxes and permutation boxes that transform blocks of input bits into output bits. Substitution boxes transform the input bits into output bits and permutation boxes permute and rearrange bits across the substitution box inputs.

The key size used for an AES cipher specifies the number of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of rounds is 10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

Each round consists of several processing steps, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

2.2 What is a LoRa radio module?

The LoRa Radio Module is a type of long-range low data rate data radio modem based on Sx1276 from Semtech. It is a low-cost sub-1 GHz transceiver module designed for operations in the unlicensed ISM (Industrial Scientific Medical) and LPRD bands. Frequency spectrum modulation/demodulation, multi-channel operation, high bandwidth efficiency and anti-blocking performance make LoRa modules easy to realize the robust and reliable wireless link.

Also, these packet radios are simpler than WiFi or BLE, you don’t have to associate, pair, scan, or worry about connections. All you do is send data whenever you like, and any other modules tuned to that same frequency (and, with the same encryption key) will receive. The receiver can then send a reply. The modules do packetization, error correction and can also automatically retransmit so it’s not like you have worry about everything but less power is wasted on maintaining a link or pairing.

2.2.1 Long Range Wireless Radio

LoRa, essentially, is a clever way to get very good receiver sensitivity and low bit error rate (BER) from inexpensive chips. That means low-data rate applications can get much longer range using LoRa rather than using other comparably priced radio technologies.

2.2.2 LoRaWAN

LoRaWAN is different. It is a media access control (MAC)-layer protocol built on top of LoRa, built using Semtech's LoRa modulation scheme. LoRaWAN, however, is rarely used for industrial (private network) applications. It is a better fit for public wide-area networks because all the channels are tuned to the same frequencies; for single-area use, it's better to have only one network operating in order to avoid collision problems.

LoRaWAN will not be used in this lab, but it follows the same protocol and ease-of-use.

2.3 Arduino Open Source Libraries

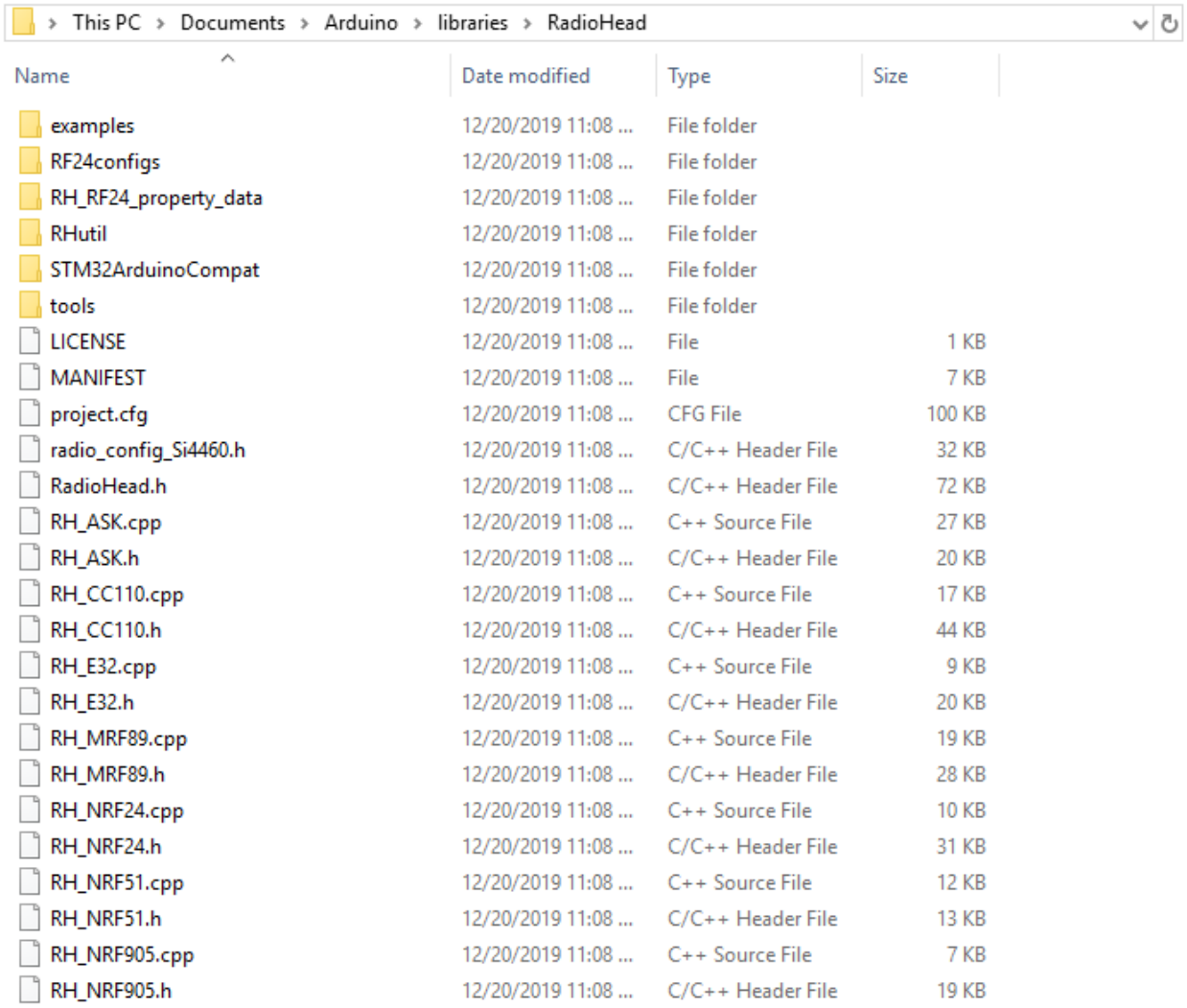
2.3.1 Arduino Libraries

We will need to install secondary Arduino libraries to allow the communication and handling of the LoRa module. The library that we will be using – RadioHead – is not included in a global repository to allow downloading from the library manager within the Arduino IDE. We will need to install it separately.

2.3.2 Installing Open Source Libraries

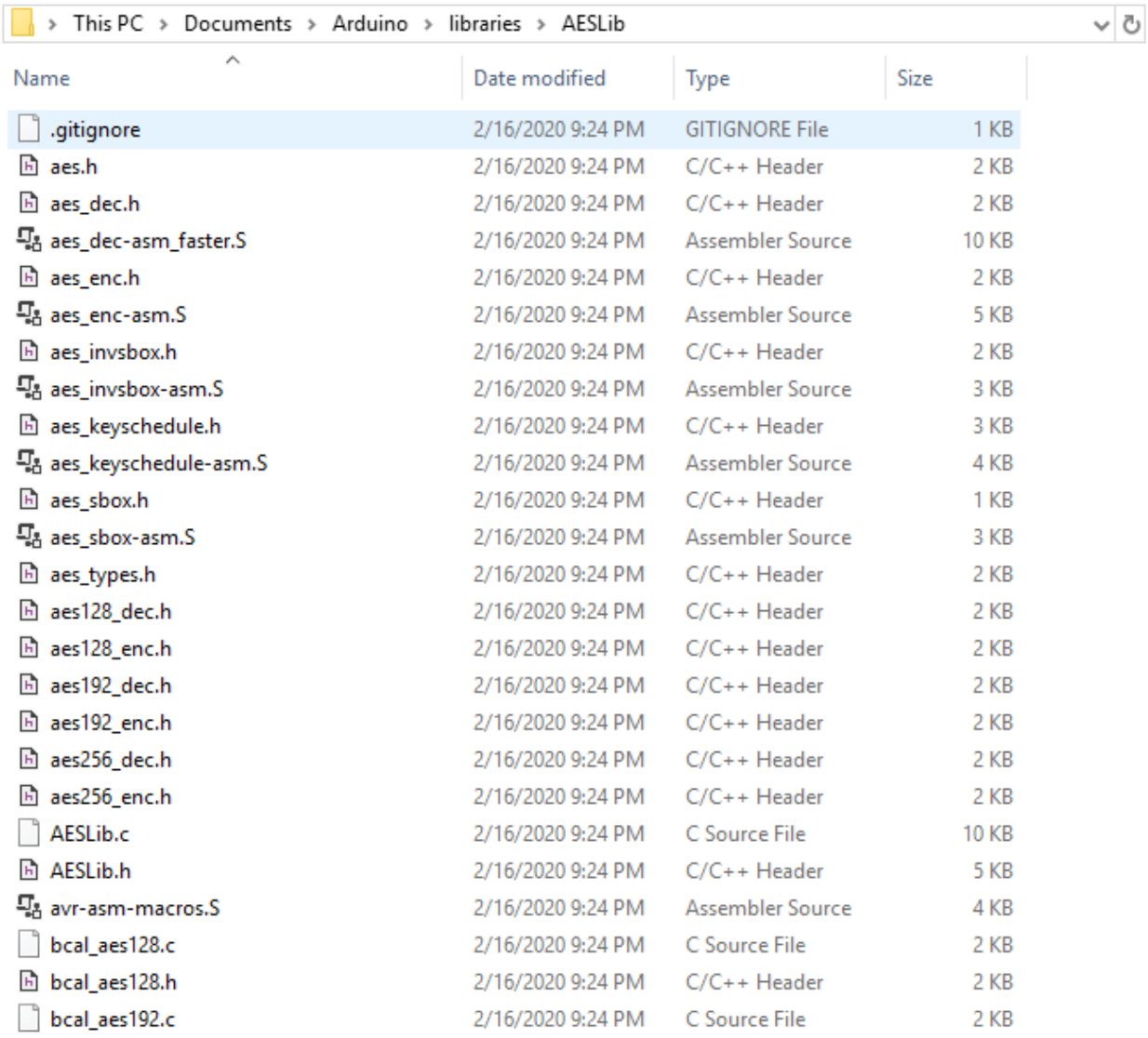
To install an external library that is not included in a repository, we will need to start with finding and downloading the library. The RadioHead library can be found at github.com/adafruit/RadioHead.

On GitHub, click the Clone or download drop down and download the Zip file. Extract the zip file to the libraries folder under the Arduino folder in the Documents folder in your user account (C:\Users\xxxxxxxx\Documents\Arduino\libraries\RadioHead). Verify that the RadioHead folder at this location contains something like the following:



Repeat the same steps for the AESLib installation which can be found at: github.com/DavyLandman/AESLib.

Verify the installation by checking that your folder structure is like the following:



2.4 SPI Communication

The LoRa radio modules will communicate with the Arduino with the SPI protocol as opposed to the previous lab which utilized the I2C protocol.

2.4.1 SPI Protocol

SPI devices communicate in full duplex mode using a master-slave architecture (alternate terminology being main and secondary) with a single master. The master device originates the frame for reading and writing. Multiple slave-devices are supported through selection with individual slave select (SS), sometimes called chip select (CS), lines.

The SPI bus specifies four logic signals:

- SCLK: Serial Clock (output from master)
- MOSI: Master Output Slave Input, or Master Out Slave In (data output from master)
- MISO: Master Input Slave Output, or Master In Slave Out (data output from slave)
- SS: Slave Select (often active low, output from master)

2.4.2 Advantages of SPI Communication

- Full duplex communication in the default version of this protocol
- Push-pull drivers (as opposed to open drain) provide good signal integrity and high speed
- Higher throughput than I²C or SMBus. Not limited to any maximum clock speed, enabling potentially high speed
- Complete protocol flexibility for the bits transferred
- Not limited to 8-bit words
- Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
- Typically, lower power requirements than I²C or SMBus due to less circuitry (including pull up resistors)
- Slaves use the master's clock and do not need precision oscillators
- Slaves do not need a unique address – unlike I²C or GPIB or SCSI
- Uses only four pins on IC packages, and wires in board layouts or connectors, much fewer than parallel interfaces
- At most one unique bus signal per device (chip select); all others are shared
- Signals are unidirectional allowing for easy galvanic isolation
- Simple software implementation

2.4.3 Disadvantages of SPI Communication

- Requires more pins on IC packages than I²C, even in the three-wire variant
- No in-band addressing; out-of-band chip select signals are required on shared buses
- No hardware flow control by the slave (but the master can delay the next clock edge to slow the transfer rate)
- No hardware slave acknowledgment (the master could be transmitting to nowhere and not know it)
- Typically supports only one master device (depends on device's hardware implementation)
- No error-checking protocol is defined
- Without a formal standard, validating conformance is not possible
- SPI does not support hot swapping (dynamically adding nodes).

3 Procedures

3.1 Encryption on the Arduino Mega

3.1.1 Encrypting and Decrypting on the Arduino

With the AESLib library for Arduino, encryption and decryption with 128-bit and 256-bit AES is very simple. Here is an example of encrypting and decrypting a string of 16 characters with a specified key.

```
#include <AESLib.h>

void setup() {
  Serial.begin(9600);
}

uint8_t key[] =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28
,29,30,31};
char data[] = "0123456789012345";

void loop() {
  aes256_enc_single(key, data);
  Serial.print("encrypted:");
  Serial.println(data);
  aes256_dec_single(key, data);
  Serial.print("decrypted:");
  Serial.println(data);

  delay(1000);
}
```

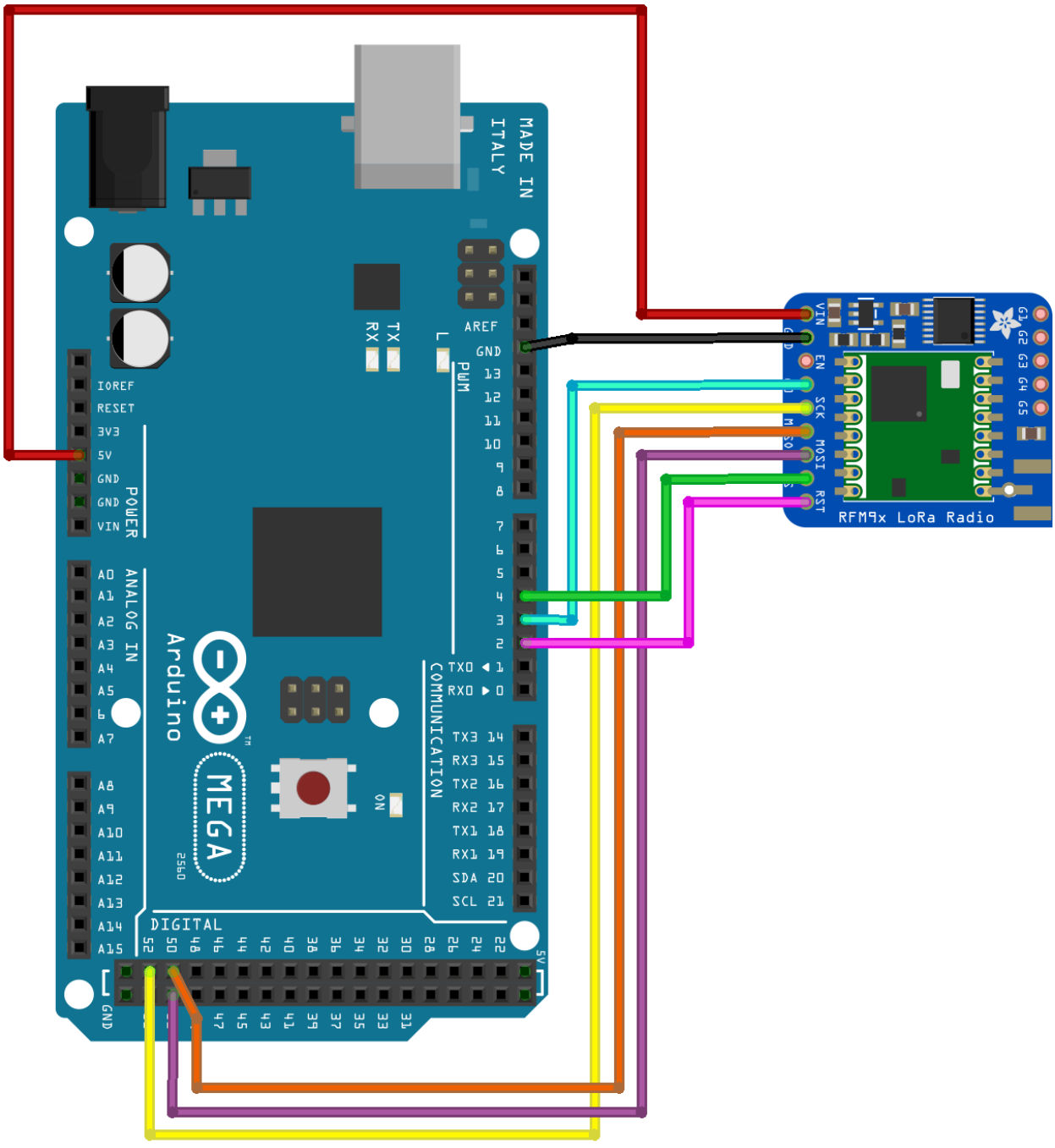
3.2 Setup and Programming

Two identical setups will be used in this lab to create encrypted end-to-end communication. Using the sample AES encryption code from 3.1.1, the server code (3.2.2), and the client code (3.2.3) create a messaging service for duplex messaging between the Arduinos with up to 64-character long strings.

Messages should be able to be received and sent on both the client and server side of the Arduino. A message must be able to be sent and received within 5 seconds. Messages must be encrypted – they cannot be read in plaintext across the LoRa transmission. Messages should not be able to be decrypted with the standard key – any array of exactly 32 bytes will work for 256-bit encryption. Messages with strings less than 64 characters long will still use 64-character packets, fill the empty space with # symbols. Messages should be entered to the Arduinos from the serial port.

3.2.1 Wiring the LoRa Module

Wire the two LoRa modules to the two Arduino Mega boards as follows; there will be two identical setups for this lab.



3.2.2 Programming Example Part 1: Server

The following example code gives what you need to send a message to a client and listen for a response. Coordinate with the class to guarantee that your definition of MY_ADDRESS does not overlap with other students.

```
#include <SPI.h>
#include <RH_RF95.h>
#include <RHReliableDatagram.h>

#define RFM95_CS 4
#define RFM95_RST 2
#define RFM95_INT 3

#define RF95_FREQ 915.0

// who am i? (server address)
#define MY_ADDRESS     2

// Singleton instance of the radio driver
RH_RF95 rf95(RFM95_CS, RFM95_INT);

// Class to manage message delivery and receipt, using the driver declared
above
RHReliableDatagram rf95_manager(rf95, MY_ADDRESS);

#define LED 13

void setup()
{
  pinMode(LED, OUTPUT);
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  while (!Serial);
  Serial.begin(9600);
  delay(100);

  Serial.println("Arduino LoRa RX Test!");

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  while (!rf95_manager.init()) {
    Serial.println("LoRa radio init failed");
    while (1);
  }
  Serial.println("LoRa radio init OK!");

  if (!rf95.setFrequency(RF95_FREQ)) {
```

```
    Serial.println("setFrequency failed");
    while (1);
  }
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);

  rf95.setTxPower(13, false);
}

uint8_t data[] = "And hello back to you";
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];

void loop()
{
  if (rf95_manager.available())
  {
    // Wait for a message addressed to us from the client
    uint8_t len = sizeof(buf);
    uint8_t from;
    if (rf95_manager.recvfromAck(buf, &len, &from)) {
      buf[len] = 0; // zero out remaining string

      Serial.print("Got packet from #"); Serial.print(from);
      Serial.print(" [RSSI :");
      Serial.print(rf95.lastRssi());
      Serial.print("] : ");
      Serial.println((char*)buf);

      // Send a reply back to the originator client
      if (!rf95_manager.sendtoWait(data, sizeof(data), from))
        Serial.println("Sending failed (no ack)");
    }
  }
}
```

3.2.3 Programming Example Part 2: Client

The following example code gives what you need to receive a message from a server and then send a response. Coordinate with the class to guarantee that your definition of MY_ADDRESS and DEST_ADDRESS does not overlap with other students. The DEST_ADDRESS (destination address) must be the same as your MY_ADDRESS definition in section 3.2.2.

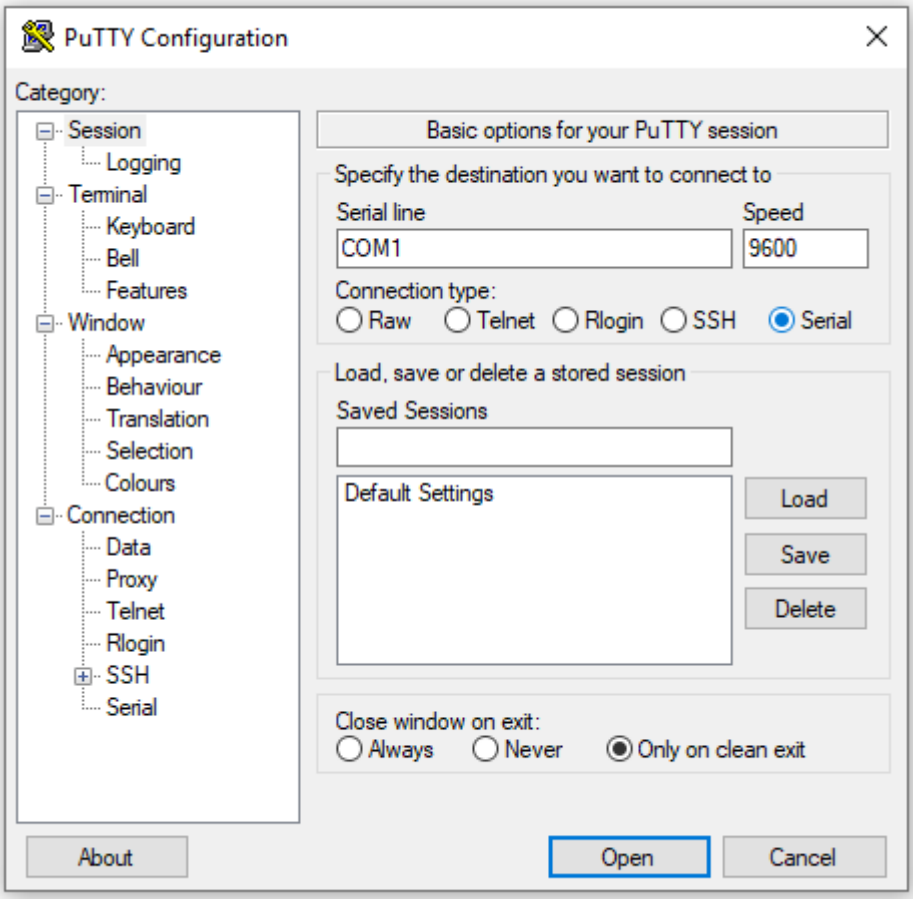
```
#include <SPI.h>
#include <RH_RF95.h>
#include <RHReliableDatagram.h>

#define RFM95_CS 4
#define RFM95_RST 2
#define RFM95_INT 3

#define RF95_FREQ 915.0

#define MY_ADDRESS     1
// Where to send packets to!
#define DEST_ADDRESS   2

RH_RF95 rf95(RFM95_CS, RFM95_INT);
RHReliableDatagram rf95_manager(rf95, MY_ADDRESS);

void setup()
{
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  while (!Serial);
  Serial.begin(9600);
  delay(100);

  Serial.println("Arduino LoRa TX Test!");

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  while (!rf95_manager.init()) {
    Serial.println("LoRa radio init failed");
    while (1);
  }
  Serial.println("LoRa radio init OK!");

  if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
    while (1);
  }
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);
```

```
  rf95.setTxPower(13, false);
}

uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t data[] = "  OK";
int16_t packetnum = 0;  // packet counter, we increment per xmission

void loop()
{
  char radiopacket[20] = "Hello World #      ";
  itoa(packetnum++, radiopacket+13, 10);
  Serial.print("Sending "); Serial.println(radiopacket);

  // Send a message to the DESTINATION!
  if (rf95_manager.sendtoWait((uint8_t *)radiopacket, strlen(radiopacket),
DEST_ADDRESS)) {
    // Now wait for a reply from the server
    uint8_t len = sizeof(buf);
    uint8_t from;
    if (rf95_manager.recvfromAckTimeout(buf, &len, 2000, &from)) {
      buf[len] = 0; // zero out remaining string

      Serial.print("Got reply from #"); Serial.print(from);
      Serial.print(" [RSSI :");
      Serial.print(rf95.lastRssi());
      Serial.print("] : ");
      Serial.println((char*)buf);

    } else {
      Serial.println("No reply, is anyone listening?");
    }
  } else {
    Serial.println("Sending failed (no ack)");
  }

  delay(1000);  // Wait 1 second between transmits, could also 'sleep' here!
}
```

3.2.4 Connecting to the Client and the Server

To start the messaging service, upload the server code that was modified to the first Arduino setup and the client code that was modified to the second Arduino setup.

Open both COM ports on the two Arduino setups simultaneously; open two instances of Putty, change the connection type to Serial, set the Baudrate being used in the Arduino code, and use the respective COM ports for the two separate Arduinos.



Once the information is entered in both Putty configuration windows, choose the Open option at the bottom of the window in both windows.

3.2.5 Listening In

Time permitting, capture LoRa transmission packets from other students in your class by listening in on their server or client address. But do not attempt to decrypt the packet. Given that the packet has 256-bit encryption or 2^{255} possible keys, it would take the Summit (OLCF-4) – the world's most powerful supercomputer, clocking in at 143.5 petaFLOPS – approximately $1.2776e^{51}$ years or 9.32^{e40} ages of the universe to break the encryption. This would give them the key that you have set now – if you changed your key, they would need to rerun the computation to calculate your new key again.

4 Study Questions

1. Provide a comprehensive report that demonstrates your completion of this laboratory assignment. Key sections to include in your report are "**Introduction and Background**", "**Methodologies**", and "**Results and Conclusions**" with inclusion of **figures**, **tables**, **codes**, **flowcharts**, and so forth in different sections.
2. What are the benefits of using end-to-end encryption?
3. Should you always use the highest security encryption available?
4. Should you always encrypt your transmissions when working with IoT devices?

Instructor: Dr. Shayan (Sean) Taheri.

Note – Cheating and Plagiarism: Cheating and plagiarism are not permitted in any form and cause certain penalties. The instructor reserves the right to fail culprits.

Deliverable: All your responses to the assignment questions should be included in a single compressed file to be uploaded in the Gannon University (GU) – Blackboard Learn environment.

5 Equipment

Name	Quantity
Arduino Mega Microcontroller	2
USB-A to USB-B Cable	2
Male-to-Female Dupont Jumpers	12
Breadboard	1 or 2
LoRa Radio Modules	2

LoRa Radio Modules – References:

1. LoRa Radio Module - 868MHz - DFRobot

2. Simple Arduino LoRa Communciation (8km) - Arduino Project Hub

3. The Arduino Guide to LoRa® and LoRaWAN® | Arduino Documentation | Arduino Documentation

4. How to use LoRa with Arduino | LoRa Tutorial with Circuit Code and AT commands | Reyax RYLR896 - YouTube

5. How to use LoRa with Arduino | LoRa Tutorial with Circuit Code and AT commands | Reyax RYLR896 - YouTube