University of Verona

A.A 2018/2019

Laboratory of Networked Embedded Systems

Lesson 3

Networked Embedded Systems Design

Enrico Fraccaroli
Sohail Mushtaq
Davide Quaglia

May 14, 2019

Contents

Chapter 1

Introduction

This lesson aims to introduce networked embedded system programming by using Arduino platform. It starts by introducing the development process and environment. Then basic examples are presented to train readers on hardware and software aspects.

Then three network-based scenarios are considered:

1. The **first** one concerns the use of two Arduino boards to transmit packets by means of a Radio-Frequency (RF) board, namely the `NRF24L01+`.

2. The **second** one concerns the sniffing of packets sent from a node via `nRF24L01+` wireless module. In this case we are going to change the configuration of the first scenario.

3. The **third** one concerns the analysis of the power consumption of the wireless module as a function of payload size and transmission power.

In the following there is a brief description of the software and components required for the lesson.

1.1 Arduino Software

In order to use the Arduino board, download the Arduino IDE from

`https://www.arduino.cc/en/Main/Software`

How to run the IDE

In order to execute the IDE, use the file named `arduino` inside the downloaded package.

How to install libraries

In order to install a new library, you just need to click on

Sketch $\longrightarrow$ Include Library $\longrightarrow$ Manage Libraries

Libraries

In order to do this laboratory some external libraries are needed:

- **RF24** by TMRh20
- **OneWire** by Jim Studt
- **DallasTemperature** by Miles Burton
- **Adafruit INA219** by Adafruit

1.2 Light Emitting Diode (LED)

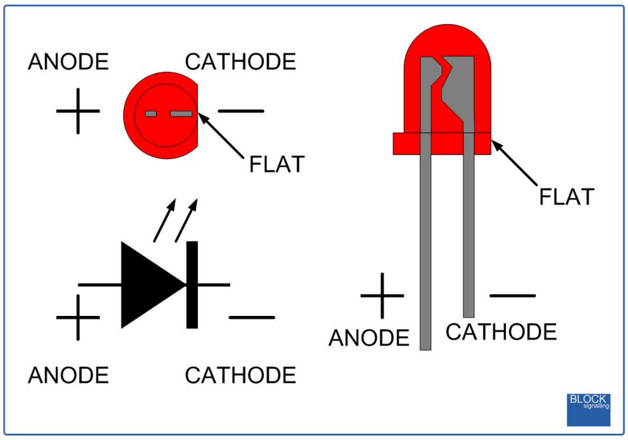

Figure 1.1: Led structure.

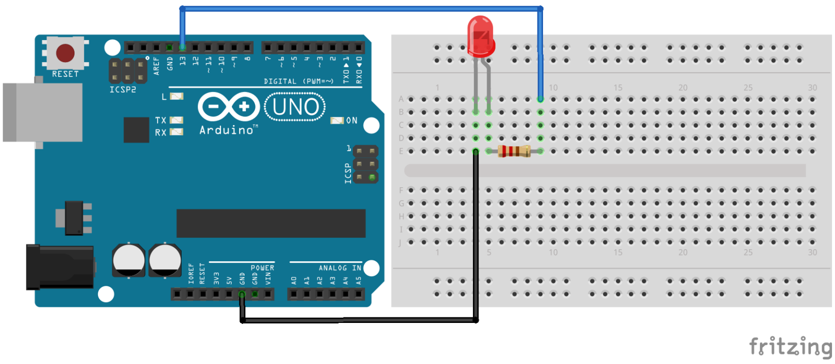

Figure 1.2: Led setup.

As shown in Figure 1.1, a Light Emitting Diode (LED) consists of an **anode** and a **cathode**. The former is connected to a voltage source (*i.e.*, PIN 13 of Arduino) while the latter (*i.e.*, the negative pole) is connected to **Ground**. The setup shown in Figure 1.2 allows to operate the LED with an Arduino. Please, note how to distinguish anode and cathode since they cannot be switched. Please, also note the aspect of the resistor; its colors encode the resistance value and you have to use the right one. The 220Ω resistor is used for the LED. Please, check a color table on the Internet to calculate values.

Below you can see the software code to turn a LED on and off. Please, note that every Arduino source program (*i.e.*, sketch) consists of a `setup()`

function and a `loop()` function. During the setup phase you have to set the pin number to which the LED is connected and its type (*i.e.*, INPUT or OUTPUT). During the loop phase, to change the state of the LED, use the `digitalWrite` command and pass the `pin_number` and `pin_state`. Using **HIGH** will turn the LED on while **LOW** will turn it off.

```
#define led0 13
int status = 0;
void setup() {
    pinMode(     led0, OUTPUT );
    digitalWrite(led0, status);
}
void loop() {
    if ( status == LOW ) status = HIGH;
    else                 status = LOW;
    digitalWrite( led0, status);
    delay(200);
}
```

1.3 Button

The previous example shows how to deal with an **output functionality**. Now we add an **input functionality**.

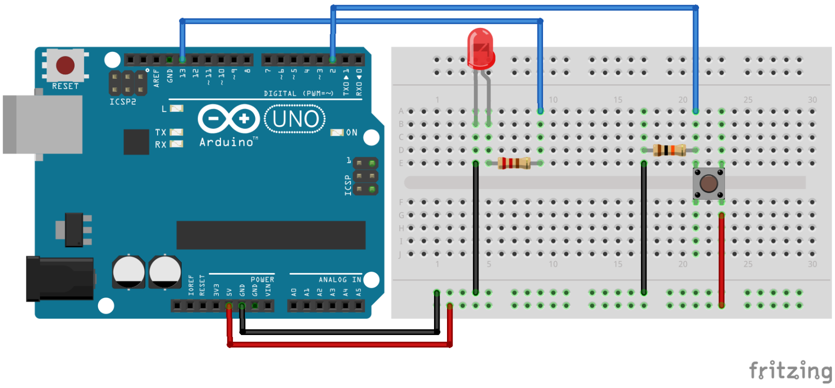

Figure 1.3: Schema of a button which controls a LED via software.

Figure 1.3 shows the button setup. First, it must be powered and thus it must be connected to ground (*i.e.*, `GND`) and to a potential source (*i.e.*, `VCC` of Arduino). Even though it has 4 pins, they are actually only two since the pins on the same side are connected in series. A cable must be used to send the signal from the button to the PIN 4 of the Arduino.

Please, note that resistors are different and cannot be switched. The 220Ω resistor is used for the LED while the $10k\Omega$ resistor is used for the button. Please, check a color table on the Internet to calculate values.

The button is an input device, during the setup phase you have to initialize the pin by setting it as INPUT pin. During the loop phase, the code has to verify if the button is pressed by means of the `digitalRead` function. This function indicates value 1 if the button is pressed and value 0 if the button is not pressed. Please, note that by pressing the button for a short time, the LED stays on for a longer time. You can play with such time since **software control** is in the middle between button and LED. This is your first **embedded system**.

```
#define pinButton  2
#define pinLed     13
void setup() {
    pinMode(pinButton, INPUT);
    pinMode(pinLed,     OUTPUT);
}
void loop() {
    if (digitalRead(pinButton) == 1) {
        digitalWrite(pinLed, HIGH);
        delay(2000); //milliseconds
```

```
        digitalWrite(pinLed, LOW);
    }
}
```

1.4 Buzzer

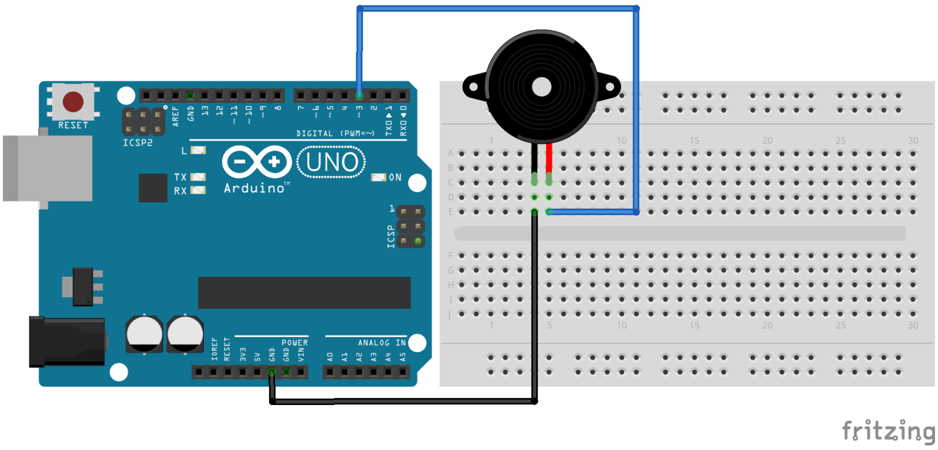

Figure 1.4: Buzzer setup.

As shown in Figure 1.4, a buzzer must be connected to a power supply (or an Arduino digital pin) and ground. During the Setup the buzzer is initialize like a LED, that is with the `pinMode(#buzzer, OUTPUT)` command. However, inside the loop function it must be used by means of the command `tone(#buzzer, 1000, 200)`, where `#buzzer` is the pin number to which the buzzer is connected, `1000` is the frequency of the tone in Hertz and `200` is its duration in milliseconds.

Please, note that some buzzers have non-switchable pins. In that case such pins have different length and the shorter one should be connected to ground (GND).

```
#define PIN_BUZZER  3
void setup(){
    pinMode(PIN_BUZZER, OUTPUT);
}
void loop(){
    tone(PIN_BUZZER, 1000, 200);
    delay(1000);
    tone(PIN_BUZZER,  750, 300);
    delay(1000);
    tone(PIN_BUZZER,  500, 400);
    delay(1000);
}
```

1.5 Temperature Sensor

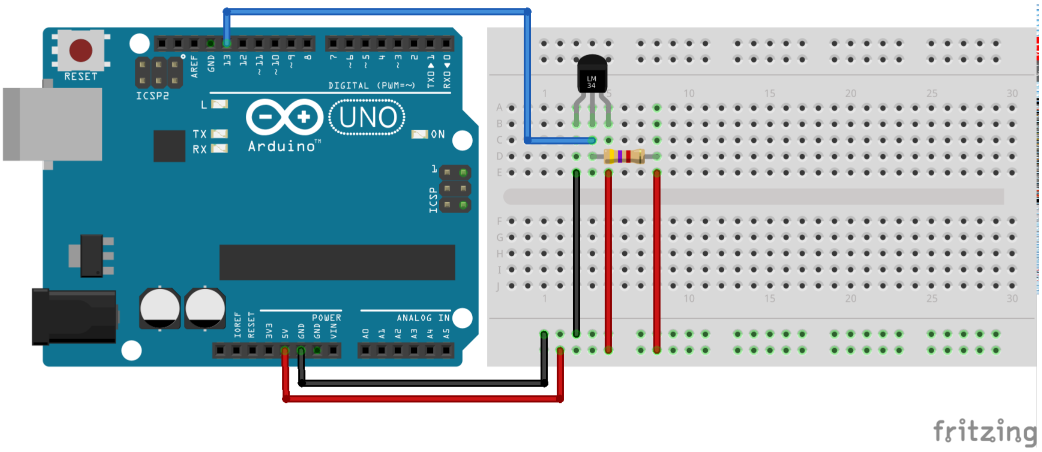

Figure 1.5: DALLAS DS18B20 sensor setup.

The `DS18B20` is a smart temperature sensor which includes analog-to-digital conversion and provides a binary value as a one-wire transmission on the data pin.

Considering the flat face as reference:

- LEFT pin: Connected to `GND`.
- CENTRAL pin: Connected to digital port of Arduino, and also connected to `VDD` through a $4.7k\Omega$ resistor.
- RIGHT pin: Connected to `VDD`.

It can be powered with a voltage between `3.0V` and `5.5V`, thus you can simply connect the **right** pin to `VDD`, the `5V` Arduino pin, as shown in Figure 1.5. However, the DS18B20 can also extract its power from the data pin, which means it only needs two wires for the connection. *This makes it ideal for use as an external sensor.*

The **left** pin must be connected to ground of the Arduino. While the data line, the **central** pin, must be connected to a digital pin of the Arduino (in this case pin 13). Furthermore, the data line, **central** pin, must be connected through a $4.7k\Omega$ resistor to `VDD`.

Please, also note the aspect of the resistor; its colors encode the resistance value and you have to use the right one. The $4.7k\Omega$ resistor is used for the DS18B20. Please, check a color table on the Internet to calculate values.

The following libraries are required:

- DallasTemperature
- OneWire

The `OneWire` library handles one-wire data transmission and is used for many sensor types as the DS18B20.

The code required to acquire the temperature is the following:

```
#include <DallasTemperature.h>
#include <OneWire.h>
#define ONE_WIRE_BUS 13
OneWire           oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
void setup(void)
{
    Serial.begin(9600);
    sensors.begin();
}
void loop(void){
    sensors.requestTemperatures();
    Serial.println(sensors.getTempCByIndex(0));
}
```

The line `OneWire oneWire(ONE_WIRE_BUS)` sets the oneWire communication mechanism, while `DallasTemperature sensors(&oneWire)` instantiate the sensing library. Finally, by using the `Sensors.begin()`, the sensor begins to acquire the temperature.

Inside the loop, two basic commands are used:

- `Sensors.requestTemperatures()`
  - Sends a temperature request to the sensor.
- `Sensors.getTempCByIndex(0)`
  - Acquires the temperature in Celsius.

Please, note the use of the static class `Serial` to print messages on the host PC through the serial USB connection. To read such messages in the software development environment you just need to click on

Tools $\longrightarrow$ Serial Monitor

The symbol rate (baud rate) of the Serial Monitor window should match with the rate set in the `begin()` method.

1.6 NRF24L01 Module

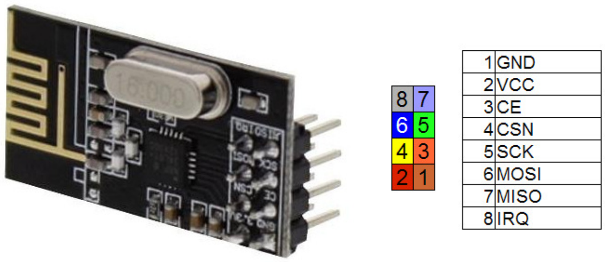

Figure 1.6: Interface of NRF24L01.

The NRF24L01 module is powered by a voltage between `1.9V` and `3.6V` while `5V` damages it. Figure 1.6 shows the configuration of the NRF24L01 pins as follows:

- `GND` The ground pin.
- `VCC` The power supply.
- `CE` The **Chip Enable**. Determines whether the module should be placed in receiving or transmitting state.
- `CSN` SPI **Chip Select**.
- `SCK` SPI **Clock**. A clock signal used to synchronize the data transfer through the serial bus.
- `MOSI` SPI **Master Out Slave In**. A line exiting the master and entering the slaves.
- `MISO` SPI **Master In Slave Out**. An input line exiting the slave and entering the master.
- `IRQ` Interrupt.

1.6.1 Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by micro-controllers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two micro-controllers. With an SPI connection there is always one master device (usually a micro-controller) which controls the peripheral devices.

Schema

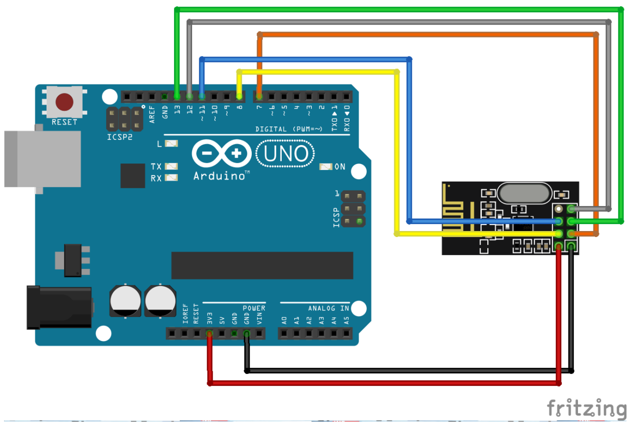

Figure 1.7: NRF24L01-Arduino connections.

The `GND` and `VCC` pin of the module should be connected respectively to the `GND` and `3.3V` pins of the Arduino.

Be Careful
Do not connect the module to the 5V voltage supply, it is too high.

The `CE` pin and the `CSN` pin can be connected to any digital pin as they are configured via software, in this case the digital pins 7 and 8. The `SCK`, `MOSI` and `MISO` pins are default as they are used for **SPI Communication**: pin 11 (MOSI), pin 12 (MISO), pin 13 (SCK). These support pins are used by the SPI library, used by the RF module. `IRQ` is not used in this scenario.

1.6.2 Main Methods

Constructor

```
RF24 (uint8_t cePin, uint8_t csPin)
```

Instantiate the Radio Frequency (RF) module manager.

- `cePin`: Pin connected to the RF Module Chip Enable;
- `csPin`: Pin connected to Chip Select.

Setup

```
RF24::begin(void);
```

Configures the module created with the constructor. This command is placed inside the setup phase and must be called before any other command of the module.

Writing pipe

```
RF24::openWritingPipe (uint64_t address)
```

Opens the pipe in write mode. The pipe is specified by the 40-bit hexadecimal address address.

Reading pipe

```
RF24::openReadingPipe (uint8_t number, uint64_t address)
```

Opens the pipe in read mode.

- `number`: the pipes number, ranging from 0 to 5, the pipes from 1 to 5 share the first 32 bits, the pipe 0 is usually used as writing pipe.
- `address`: The 40 bit address of the pipe to open.

Start listening

```
RF24::startListening (void)
```

Starts listening to open pipes in read mode. Make sure that the `openReadingPipe()` function has been previously called. You can not call the write function if you have not called the `stopListening()` function before.

Stop listening

```
RF24::stopListening (void)
```

Stops listening for incoming messages. This must be called before a write operation.

Write data

```
RF24::write (const void * buf, uint8_t len)
```

Writes on the opened writing pipe. First, make sure that the openWritingPipe() function has been previously called.

- `Buf`: pointer to the data to send
- `Len`: number of bytes to send

It returns `True` if the data has been sent, `False` otherwise.

Check for incoming data

```
RF24::available ()
```

Checks if there are bytes available to be read. It returns `True` if there is an incoming stream of data, `False` otherwise.

Read incoming data

```
RF24::read (void * buf, uint8_t len)
```

Reads incoming data and returns the last received data.

- `Buf`: buffer pointer where the data was written
- `Len`: maximum number of bytes to read in the buffer

It returns `True` if the data was successfully delivered, `False` otherwise.

1.6.3 Configuration methods

The methods listed below can be used to set different chip configurations.

Change communication channel

```
RF24::setChannel (uint8_t channel)
```

It sets the RF communication channel.

- `channel`: Which RF channel is used to communicate. It allows to select a value from 0 to 127.

Change payload size

```
RF24::setPayloadSize (uint8_t size)
```

This deployment usually utilizes a premium payload size for all transmissions. If the method is not called, the device transmits to the maximum payload size, or 32 bytes.

Change power amplifier level

```
RF24::setPALevel (rf24_pa_dbm_e level)
```

It sets the power amplifier level in one of four levels:

`RF24_PA_MIN`	-18 dBm
`RF24_PA_LOW`	-12 dBm
`RF24_PA_MED`	-6 dBm
`RF24_PA_HIGH`	0 dBm

Change transmission speed

```
RF24::setDataRate (rf24_datate_e speed)
```

It sets the transmission speed at one of three speeds:

`RF24_250KBPS`	250 Kbs
`RF24_1MBPS`	1 Mbps
`RF24_2MBPS`	2 Mbps

Warning

All the communications must have the same:

- Transmission Speed;
- Channel;
- Payload Size.

Chapter 2

Networked embedded systems for temperature monitoring

This scenario concerns the use of two Arduino boards to transmit packets through the use of the NRF24L01 Radio-Frequency (RF) board.

The project creates a temperature measurement system which generates an acoustic alarm if the temperature level crosses an upper-bound value. Two LEDs (*i.e.*, Red and Green) allow the user to know if the temperature is getting near the upper-bound value.

Thus, the following information are provided to the user:

GREEN LED	the temperature is low.
RED LED	the temperature is rising.
SOUND	the temperature has crossed the upper-bound value.

2.1 Required material

- 2 x Arduino Uno
- 2 x Prototyping Board
- 2 x NRF24L01 Module
- 1 x Red LED
- 1 x Green LED
- 1 x Button
- 1 x Buzzer
- 1 x Temperature Sensor (DALLAS1820)
- 2 x Resistors 220Ω

- 1 x Resistors $4.7k\Omega$
- 1 x Resistors $10k\Omega$

2.2 Structure

The structure of the project is depicted in Figure 2.1 and consists of two parts:

1. The **first** one is made up of an Arduino, a NRF24L01 module, two LEDs and a button. This system has the role of **Master**.

2. The **second** one is made up of an Arduino, a NRF24L01 module, a DALLAS18B20 temperature sensor, and a buzzer. This system has the role of **Slave**.

The node to which the button is connected is considered as a master. Whenever the user **presses** the master's button, a wireless message is sent to the slave to request the **temperature** measured at the place where the slave is deployed. Once the temperature is received by the master, it is displayed on the serial monitor of the Arduino and used to switch LEDs. The time elapsed from the request to the response is shown on the serial monitor along with the temperature.

However, if the time elapsed exceeds a threshold (*i.e.*, it takes too long to be received), the value is discarded since it cannot be considered as valid and an error message is shown.

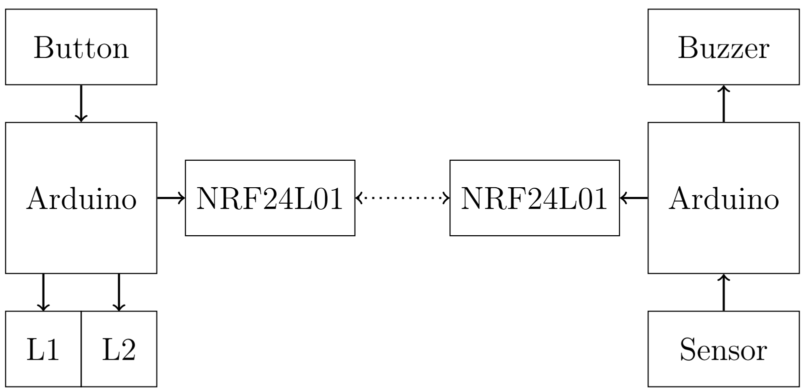

Figure 2.1: Temperature monitoring distributed architecture.

2.3 Development instructions

2.3.1 Master

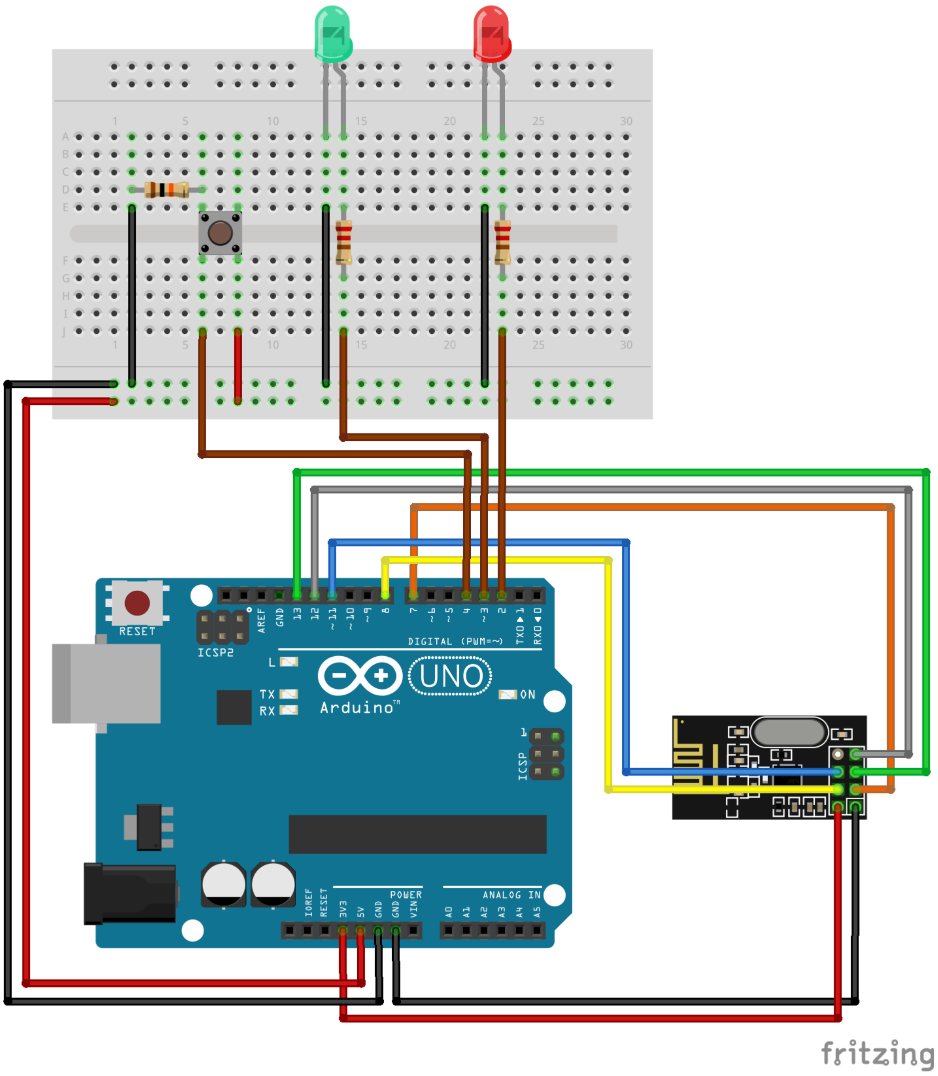

Figure 2.2: Master node configuration.

Whenever the button is pressed, the master node sends a request to the slave node for the temperature value. Based on the temperature value supplied by the slave, the LEDs turn on for about 2 seconds. On each transmission, along with the request/response, the time at which the packet is send is provided. Thus, a structure with both time and temperature values is created and used during the communication. The current time value is acquired by means of the `micros()` function, which returns the number of microseconds since the Arduino started running the program. Such value is written into the request message. Once the slave has received the request and the time-stamp, it updates the acquired temperature value inside the packet and then sends it back to the master. The master receives the packet and evaluates the elapsed time by using the stored value inside the packet and the current time value got by the aforementioned function. If the elapsed time is not above a pre-determined threshold, the packet is displayed on the serial monitor.

2.3.2 Slave

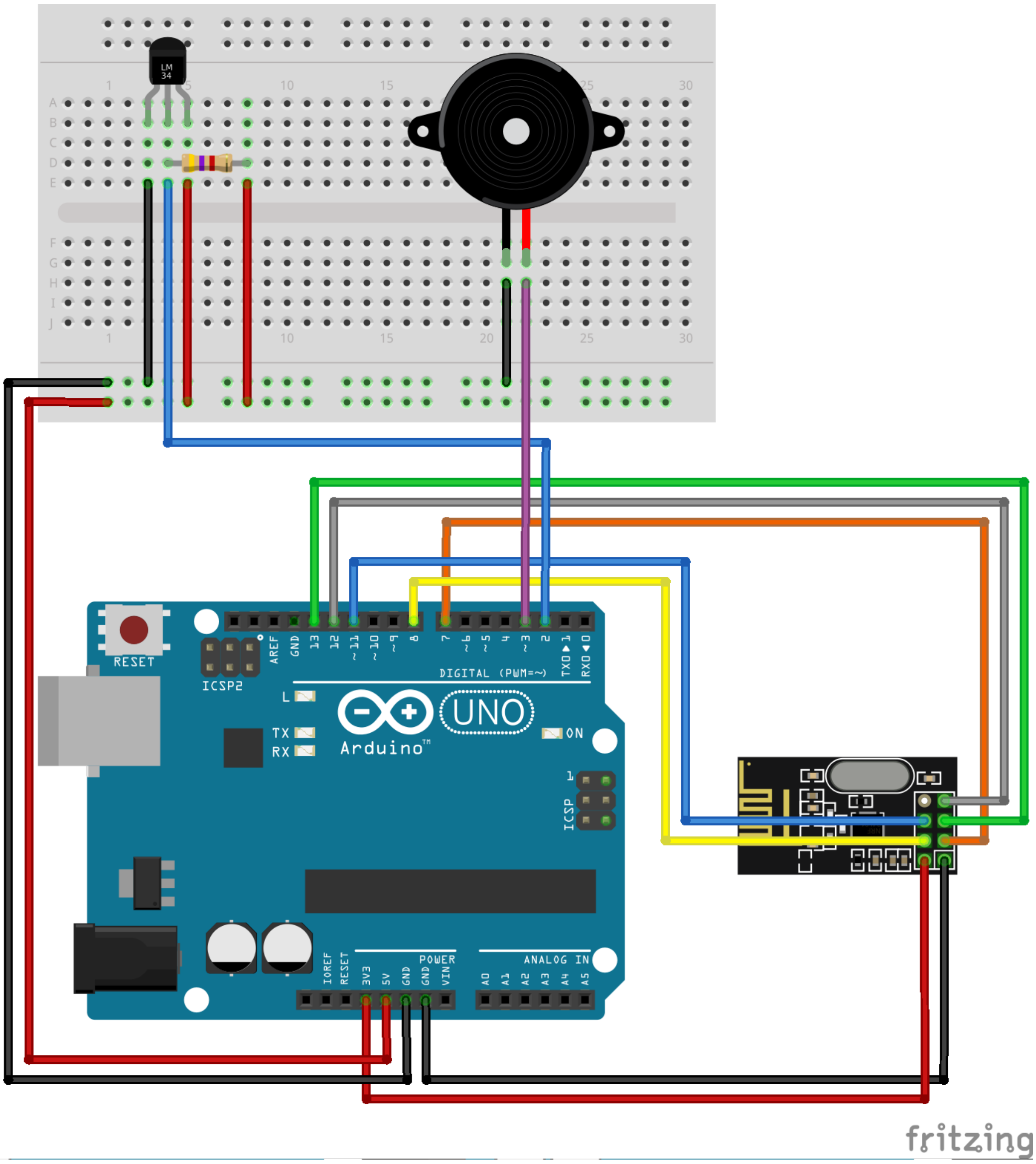

Figure 2.3: Slave node configuration.

The slave node waits for the requests from the master. Furthermore, if the temperature is too high, the local buzzer is activated every 2 seconds, until the temperature becomes normal.

2.3.3 Exercise

The code of the Slave is incomplete. The missing parts are denoted by $FILL$ and, for each of them, a comment shows what is missing. Complete the code in order to allow the Slave node to receive a request from the Master and reply with the temperature.

Chapter 3

Sniffing wireless communications

The objective of this scenario is to intercept the packets sent by the temperature board.

In the previous scenario, the temperature board detects and transmits the temperature on an explicit request of the master. In this scenario we do not have the Master node to re-cycle the Arduino board to implement a Sniffer (Figure 3.1). First, we will give you the necessary background for performing packet sniffing, and then you will be given the task of modifying the previous scenario for performing network sniffing. The Sniffer will attempt to intercept the packets transmitted by the temperature board. However, the Sniffer needs to know some configuration parameters of the node, such as the transmission channel, the data rate and the base address of the communication pipe.

3.1 Required material

- 2 x Arduino Uno
- 1 x Prototyping Board
- 2 x NRF24L01 Module
- 1 x Buzzer
- 1 x Temperature Sensor (DALLAS1820)
- 1 x Resistors $4.7k\Omega$

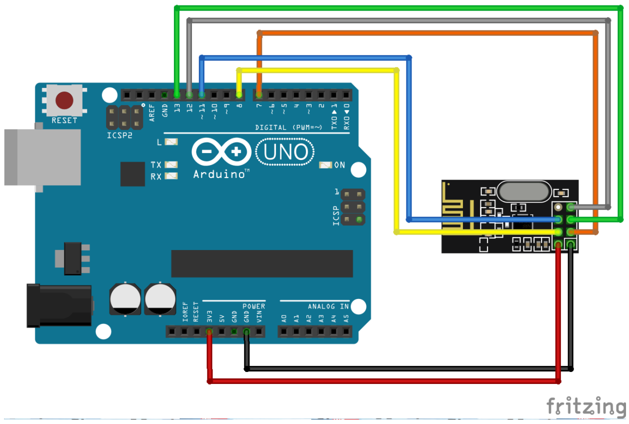

Figure 3.1: Sniffer schema.

3.2 Background

The Nordic Semiconductor `nRF24L01+` card operates at `3.3V` voltage and transmits on channels at a frequency of `2.4GHz`, which is used by common WiFi radios. Packets transmission can take place in two ways:

- **Regular Packet**: The payload size is defined a priori inside the code;
- **Enhanced Shockburst**: It has the ability to use dynamic payloads, identify ACK packets and Auto-ACK functionality.

This chip does not support promiscuous mode, which would normally allow you to detect all kind of network traffic. However, the difference between the two supported modes can be used to our advantage, albeit with some limitations.

3.2.1 Packet structure

The first part of the header of the packet transmitted by both the aforementioned modes are identical. In particular it has the following structure:

- A 1-byte preamble is used to identify nRF24 packets;
- Address, length from 3 to 5 bytes, to specify the destination address.

In **Regular Mode**, this header is immediately followed by a payload, of length to be defined in the code, between 0 and 32 bytes. In both modes, a section dedicated to the CRC is provided at the end of the packet. This CRC is used to check the integrity of the data and it has a fixed length which varies between 0 and 2 bytes.

Preamble	Address	Payload	CRC
1 byte	3-5 byte	0-32 byte	0-2 byte

The **Enhanced Shockburst** mode provides the ability to send payloads with dynamic length. Its header contains three other fields which are not byte-aligned, defined as follows:

- *Payload length*: 6 bits used to specify the payload length, which will always be limited to 0 to 32 bytes;
- *2-bit PID (Packet Identifier)*: Used to number packets, useful for detecting retransmissions;
- *NO_ACK*: 1 bit flag which is set to 1 if the packet does not provide a acknowledge in response.

Preamble	Address	Payload Length	PID	NO_ACK	Payload	CRC
1 byte	3-5 byte	6 bit	2 bit	1 bit	0-32 byte	0-2 byte

3.2.2 How to simulate promiscuous mode

The slave node use the **Enhanced Shockburst** mode. Because the `nRF24L01+` chip does not support promiscuous mode, you need to find a way to simulate it by using the two aforementioned modes. The **Enhanced Shockburst** mode differs from the regular one only for some additional fields in the header. You can set the sniffer to detect **Enhanced Shockburst packets** as if they were regular packets, by "embedding" the three additional sections and the variable part of the address (which will be interpreted as a specific node address) inside the payload.

Preamble	Base Address	Node Address	Payload Length	PID	NO_ACK	Payload	CRC
1 byte	4 byte	1 byte	6 bit	2 bit	1 bit	0-32 byte	0-2 byte
		↓	↓	↓	↓	↓	↓

Preamble	Address	Payload
1 byte	4 byte	0-32 byte

In order to do this, you need to change the sniffer code:

- Specify the length of the address field to limit it to the base address length (the initial part of the address, shared between all nodes);
- Disable Enhanced Shockburst mode;
- Disable CRC parity check;
- Set a fixed length payload.

The first three points are required to "mask" and therefore to handle the packets transmitted by means of the **Enhanced Shockburst**. The last point is required in order to specify the size of the data we need to read, since the sniffer works in **Regular Mode**. The CRC control needs to be disabled. The sanity check will always fail since the payload size of all packets is not known a priori. The address of the two communicating nodes must maintain a format compatible with this mode. Thus, the 4 leftmost bytes are dedicated to the base address while the rightmost ones is reserved to the identification of the node.

As you can see, the first limitation appears to be the payload length. In **Regular Mode**, however, the payload is limited to a maximum size of 32 bytes, and it must be able to contain (besides the actual payload) the additional fields included in it (node address, payload length, PID, NO_ACK and CRC), which then limits the available space for the actual payload.

3.2.3 Packet sniffing mechanism

The `nRF24L01+` must be configured with the appropriate parameters which are known a priori, namely: the base address, its length (in addition to that of the node address), the channel, the data rate, the maximum payload length. Furthermore, the CRC control must be deactivated. This is done by the function `activateConf()`, which also sets the interrupt handler, `handleNrfIrq()`.

The wireless module receives a packet and sends an interrupt request to the dedicated digital pin, which in our configuration is connected to the Arduino to handle this request. Whenever the interrupt handler is called, it has to wait for packets corresponding to the configuration (via the `radio.available()`). When it receives a packet (through the function `Radio.read ()`), it inserts the packet inside a circular buffer used to store the received data. At each primary loop iteration, the data inside the buffer is provided to the packet dissector and the buffer is emptied. When the buffer is full, any received packet is considered as lost and a counter is incremented.

3.2.4 Packet dissection mechanism

The packet contained in the buffer is actually an array of byte. Each sniffed packet is passed to a `dissect()` function which identify its fields, align them to the byte, rearrange and decode them to obtain legible data. For what concerns the node address packet sections, payload length, PID, NO_ACK, and CRC, an alignment and subsequent printing are performed. For what concerns the Payload, it is stored inside the buffer as Little-Endian, thus it must be first converted to Big-Endian. Then, the raw data is splitted and converted into the actual data which wre originally contained inside the packet. Once the alignment and conversion are completed, the various sections of the package are printed, both in binary and human-readable format.

3.3 Exercise

Please, modify the Slave code of the previous scenario to send temperature periodically (*i.e.*, without waiting for any request). Then use the Sniffer to intercept the packets and show their content on the Serial Monitor of the host PC (code is provided).

Chapter 4

Profiling power consumption of communicating devices

The objective of this scenario is to analyze the power consumption of the wireless module as a function of the payload size, transmission power and transmission speed.

4.1 Required material

- 2 x Arduino Uno
- 1 x Prototyping Board
- 1 x NRF24L01 Module
- 1 x Buzzer
- 1 x Temperature Sensor (DALLAS1820)
- 1 x Current sensor Adafruit INA219
- 1 x Resistors $4.7k\Omega$

4.2 Structure

This scenario make use of the temperature sensor node of the previous scenario, appropriately modified so that it periodically transmits data. We aim to measure the current absorbed by its wireless module.

As shown in Figure 4.1, the current sensor of the other Arduino is connected in serial with the power supply of the wireless module, measures the current value, converts it into milliampere values and prints it on the Serial Plot of the host PC. To show the plot in the software development environment you just need to click on

$$\text{Tools} \longrightarrow \text{Serial Plotter}$$

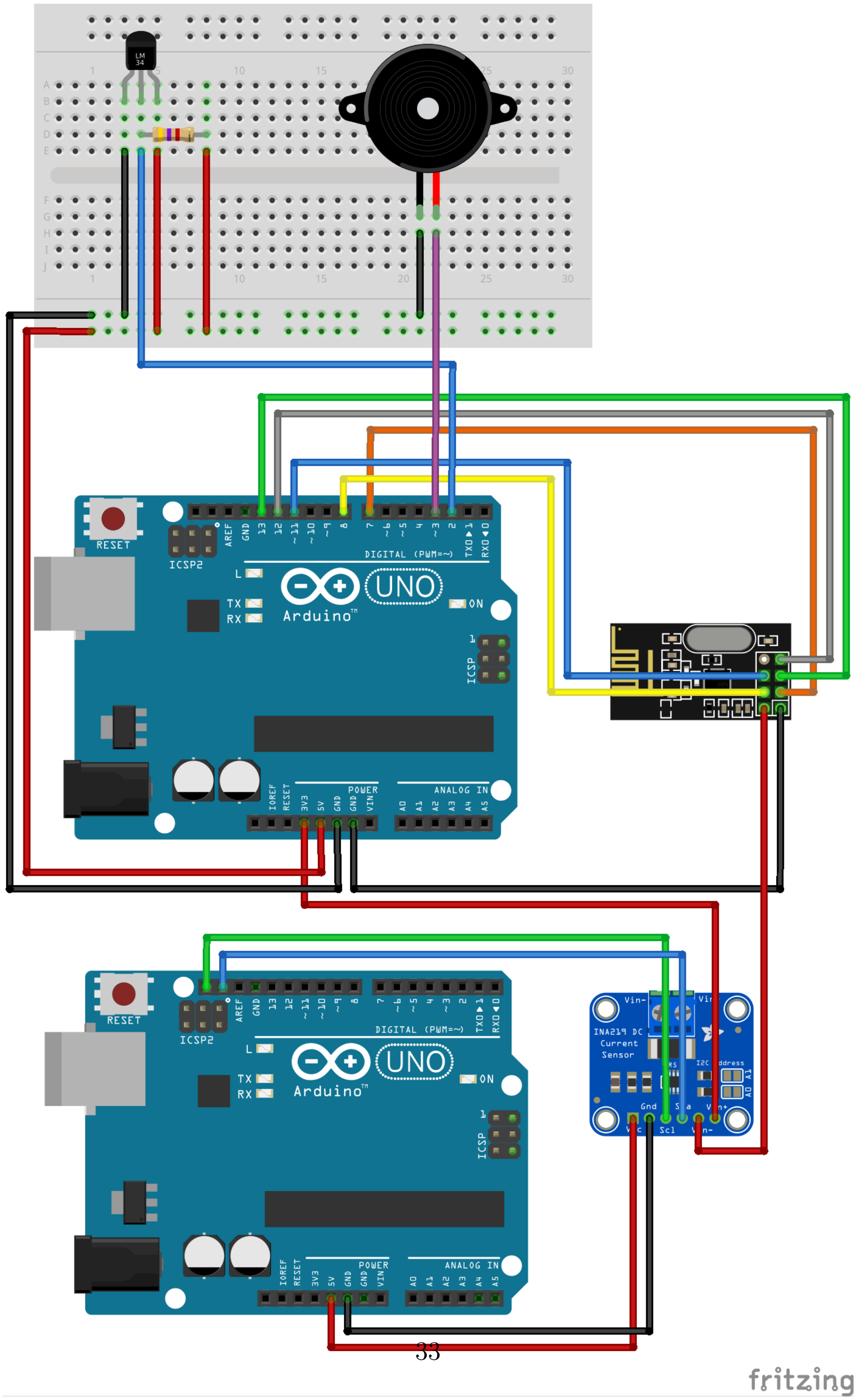

Figure 4.1: Current sensor node configuration.

4.3 Exercise 1

In this scenario, the temperature node has to:

- Listen for n times and between each listen there is a delay.
- Wait for 2 seconds.
- Send temperature for m times.

The code of the temperature node is incomplete. The missing parts are denoted by $FILL$ and, for each of them, a comment indicates what is missing. Add the missing code and analyze plot. How much current is needed for transmission? And for reception?

4.4 Exercise 2

Modify transmission power and analyze the plot.

4.5 Exercise 3

Modify transmission speed and analyze the plot.

That's all folks