ETSF15 - Point to point communication

Lab 1 - Manual v.1

February 1, 2016

LUNDS UNIVERSITET

Lunds Tekniska Högskola

Contents

Part I Introduction **2**

1 Application . . . **2**

Part II Communication specifications **3**

2 Physical . . . **3**

2.1 Flags . . . 3

2.2 Communication parameters . . . 3

3 Medium access . . . **3**

3.1 Frame structure . . . 3

3.1.1 DATA . . . 4

3.1.2 ACK . . . 4

3.2 Collision avoidance (Not in this lab) . . . 5

3.3 Addressing (Not in this lab) . . . 5

4 Reliable transmission . . . **5**

Part III Arduino **7**

5 Board . . . **7**

6 Shield . . . **8**

6.1 Communication . . . 9

6.2 Application . . . 10

6.3 Service and debug . . . 10

7 Software . . . **11**

Part IV Master Node **13**

8 States . . . **13**

8.1 SEND . . . 14

8.2 RECEIVE . . . 14

8.3 PROCESS . . . 15

8.4 ACK SEND . . . 16

8.5 ACK RECEIVE . . . 17

8.6 ACT . . . 17

9 Parameters . . . **17**

Part V Development Node **19**

10 Software skeleton . . . **19**

Part VI Instructions **21**

11 Preparations . . . **21**

12 Lab set-up . . . **21**

13 Tasks . . . **22**

Part A Skeleton.ino **26**

Part I
Introduction

This text supplies a background to the lab, an introduction to the lab hardware and software, and a manual to the first lab. The Point to point communications consists of two labs. In this first lab you will develop the necessary mechanisms to enable two units to communicate over an Infra-red (IR) link. One unit, the *Master Node* will be provided for you and is fully functional according to the specifications in Parts II and IV. The *Development Node*, with which you will be working, comes with a complete set of Hardware (HW) and a Software (SW) skeleton downloadable from the website. The aim of the lab is to implement the necessary functionality to succeed in making the *Development Node* communicate with the *Master Node*. To your aid is a description of the *Master Node*, the communication standard, the HW for the *Development Node*, and a SW skeleton.

The nodes have limited memory, computational, and Input/Output (I/O) resources. This imposes constraints on the speed of communication, redundancy, and complexity of the application. The nodes are for example single threaded. It is therefore, non-trivial to run concurrent processes such as simultaneous transmission and reception on the device. These constraints have to be accommodated for and dealt with in your implementation.

The remainder of this document is structured as follows. In Part II an overview is given of the singular practical objective of the lab, to get two primitives nodes to distributively perform a simple task. Then details the of communication standard used in the lab is outlined. Part III gives an overview of the HW you will be working with. Part IV provides a specification for the *Master Node*. Part V covers what is known about the *Development Node*. Finally, Part VI details the lab and provides you with a lab manual.

1 Application

The reason why these two nodes even need to communicate with each other is to be able to remotely set which light/Light Emitting Diode (LED) to illuminate on the other node. One node acts as the agent and the other as the actuator. In this lab the *Master Node* will act as the actuator and the *Development Node* will act as the agent. The designated agent node has three different-coloured LEDs and a button. The LEDs on the *Master Node* light randomly at a fixed rate. When the button is pressed, the ID of the lit LED shall be transmitted to the actuator node, where the same coloured LED shall be illuminated until the next instruction. Upon successful transmission the agent node shall return to waiting for the next input. Similarly, after addressing the instructions in the ingress frame, the actuator node returns to waiting for the next instruction.

Part II
Communication specifications

2 Physical

The communication link is physically achieved by using a pair IR LEDs ($\lambda = 900nm$) over a simplex channel. Communication on the link is coded and propagated using On-Off keying, meaning no light is a zero and light is a one. The nodes clocks are asynchronous.

2.1 Flags

As the device is only capable of intermittent reception, a start and a stop frame is used to delimit a frame and synchronise the nodes. The flags are 11-bit Baker codes.

- **Start flag**
  11100010010
- **Stop flag**
  00011101101

2.2 Communication parameters

The system codes the transmission as On-off keying (OOK) with symbol length T_s and oversampling factor N_s, see Table 1. A zero is transmitted as a high signal (light), and a one as no signal (no light).

Table 1: Communication parameters

Parameter	**Value**	**Description**
T_s	$100ms$	Symbol length
N_s	2	Oversampling factor
T_b	$500ms$	Backoff time when sensing the channel

The oversampling means here thet for each bit transmitted this is the number of signals sent. The bit time is N_sT_s, which in our case is 200 ms. Hence, the starting flag 11100010010 will be signalled as shown in Figure 1.

3 Medium access

3.1 Frame structure

The frame size is fixed and de-marked at either end with a start and a stop flag, see Table 2 and Figure 2. The flags are as specified in Section 2.1. There are two types of

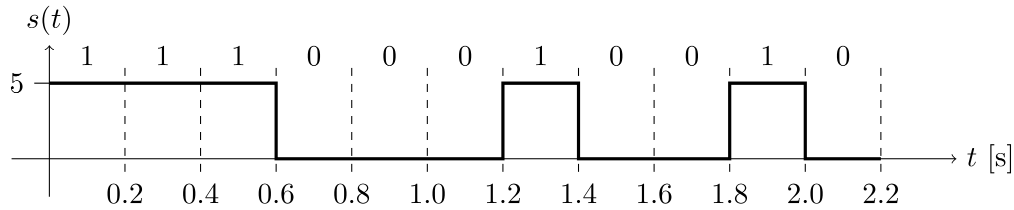

Figure 1: Signal for the Start Flag.

frames, DATA and ACK. Each frame has a 2-bit sequence number. The sequence number is incremented for each DATA frame and is used by the ACK frame to acknowledge a successfully received DATA frame. Each frame type carries a checksum of its content. The payload is only allocated 4-bits and used by the application. In this lab addressing is not used. The important parts at this point are Start and Stop flags, Payload (i.e. LED number) and Type.

Table 2: Frame decomposition

Field	**Length (bits)**	**Description**
Start flag	11	De-marks start of transmission
From	4	Sender address
To	4	Receiver address
Type	2	Type of message [ACK — DATA]
Seq	2	Sequence number
Payload	4	Message content
Checksum	4	Checksum of message excluding flags
Stop flag	11	De-marks end of transmission

3.1.1 DATA

A DATA frame carries information from the application and is allocated the entire Payload section of 4-bits. The Seq number is incremented for each new DATA frame. A DATA frame is denoted by a 10_2 in the type field.

3.1.2 ACK

The ACK or acknowledgement frame trails a received DATA frame to signal that is has been received correctly. It is only sent once, and carries an empty payload. An ACK frame is denoted by a 01_2 in the type field.

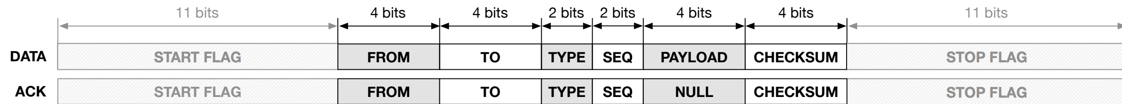

Figure 2: Frame structure.

3.2 Collision avoidance (Not in this lab)

Nodes on the link employ collision avoidance by sensing the link before transmission. When transmitting, if highs are detected on the link for T_c ms, the nodes proceeds with the transmission. On the other hand, if a high is detected the backs-off for T_b ms before sensing the channel again.

3.3 Addressing (Not in this lab)

Each node has a four-bit address, i.e. an address space of 16. The address is set using the four-toggle dip-switch located on the board. A node shall only process received messages address to it. The node reads and subsequently updates its address for every state transition, more on this in Section 8.

4 Reliable transmission

There are a number of reasons why a frame might not have been received correctly:

- Recipient out of range
- Recipient not receiving
- Sender and receiver out of sync
- Partial reception of the frame
- Collision due to simultaneous transmission

Given these circumstances, to achieve a rudimentary degree of reliability, the nodes employ a Stop-and-wait Automatic Repeat Request (ARQ) scheme. The sender of a DATA frame shall retransmit that frame, persisting the Seq number, if it does not receive an ACK frame with the same Seq number from the recipient in T_t. Similarly, if retrieved successfully and is correctly addressed, the recipient of a DATA frame shall transmit an ACK to the sender pertaining the same Seq number, see Figure 3

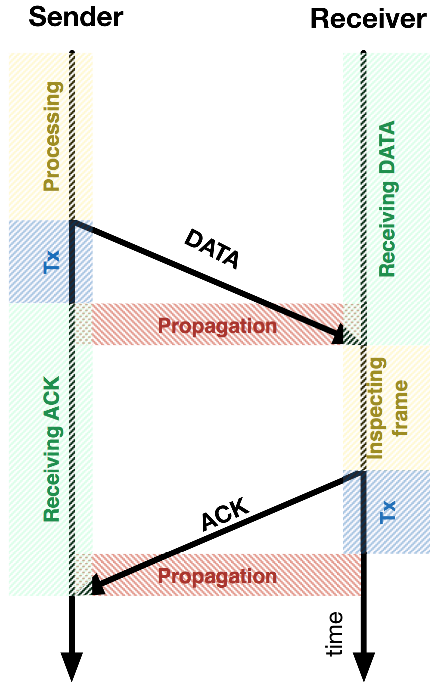

Figure 3: Example communication scenario *(No to proportion)*

Part III
Arduino

Both the *Master Node* and the *Development Node* are constructed using an Arduino board and micro-controller [4], complimented by a custom made shield attached to the board. The micro-controller is single threaded and is programmed using a language called Processing. The programming environment used in the lab is the default Arduino software, that can be downloaded from [1]. Both the Arduino board and the development environment are open source. In this lab you will not modify the HW but focus on implementing the desired functionality in SW. In Section 5 you will get an overview of the Arduino board, followed by a introduction to the shield in Section 6. A brief introduction to the software is given in Section 7.

5 Board

The Arduino micro-controller is fitted onto a small board with a set of digital and analogue I/O pins, see Table 3. These pins can easily be manipulated and read form the programmable micro-controller. The RISC micro-controller is 8-bit and is clocked to 16 MHz. You communicate with the board over USB, see Figure 4

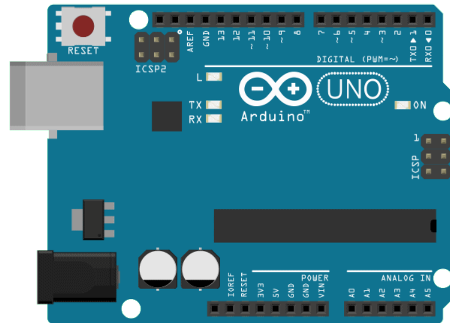

Figure 4: Arduino UNO board

Table 4: Pin assignments

Assignment	**Pin number**	**Type**
Receive (Rx) diode	0	Analogue
TX diode	13	Digital
Button	1	Digital
Address DIP 1	6	Digital
Address DIP 2	5	Digital
Address DIP 3	4	Digital
Address DIP 4	3	Digital
Debug LED #1	7	Digital
Debug LED #2	8	Digital
Debug LED #3	9	Digital
Application Blue LED	10	Digital
Application Green LED	11	Digital
Application Red LED	12	Digital

Table 3: Arduino specifications

Component	**Property**
Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB
Flash Memory for Bootloader	0.5 KB
SRAM	2 KB
EEPROM	1 KB
Clock Speed	16 MHz

6 Shield

The shield attaches to the board and supplies the communication, interaction, and service/debugging functionality. The boards pins have been assigned according to Table 4. The shield is laid out as Figure 5.

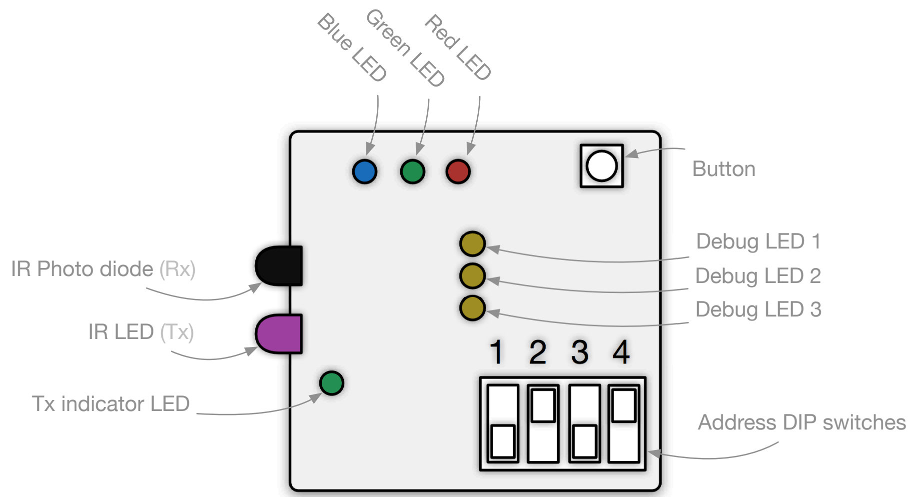

Figure 5: Shield layout

6.1 Communication

The communication circuit provides the board with a set of Rx and Transmit (Tx) IR diodes. The Tx diode is complimented with a red LED to provide visual feedback whether the node is transmitting. To be able to assign the node an address, the communication circuit is also equipped with a four-toggle dip-switch, see Figure 6. The most significant bit is set using the left-hand-side switch, DIP Switch 1 which is connected to PIN 6.

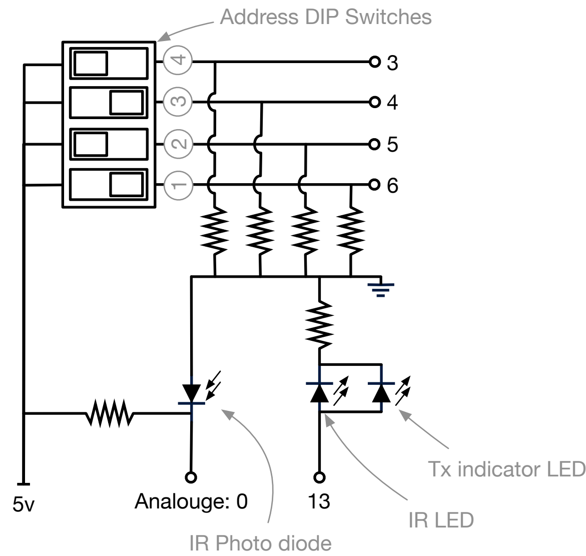

Figure 6: Communication circuitry

6.2 Application

The application circuit is constructed to enable the application specified in Section 1. It therefore consists of three differently coloured LEDs and a button, see Figure 7.

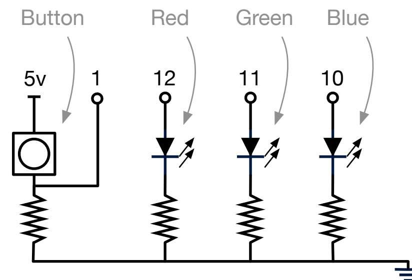

Figure 7: Application circuitry

6.3 Service and debug

In addition to the debug messages outputted by the Arduino, accessible though the Arduino IDE, the shield has been equipped with the three user customisable LEDs accessible on pins $7, 8, 9$, labeled D3, D4 and D5 on the circuitboard. Additionally, as previously mentioned, the Tx LED will light when the Tx diode is activated.

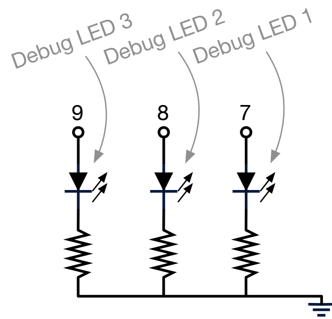

Figure 8: Service and debug circuitry

7 Software

An Arduino micro-controller is programmed using a language called Processing [5] which has many similarities with C/C++. You preferably develop your code in the Arduino Integrated Development Environment (IDE) [2].

Arduino programs are called sketches [1]. A sketch has of two primary functions, `setup()` and `loop()`. The `setup()` function is where you declare how you want the I/O to behave and initialise your global variables, see Code Snippet 1. The code contained inside the `loop()` is looped in runtime. You can declare your own functions, variables, and constants outside of the these two functions. Please consult the Arduino beginners guide [3] (https://www.arduino.cc/en/Guide/HomePage) before you begin the lab. There are numerous code example to be found by a quick web search. Have a look at the typical `Blink.ino` program. This is the counterpart of the "Hello World" program.

Listing 1: Sample Arduino code, Tx and Rx

```
// Assign pin num
const int PIN_RX = 0;          // Receive pin \#
const int PIN_TX = 13;         // Transmit pin \#

void setup() {
  Serial.begin(9600);          // Configure serial port
  pinMode(PIN_TX, OUTPUT);     // Configure output pin
}
void loop() {
  // Transmit
  digitalWrite(PIN_TX, HIGH);  // turn on the IR LED
  delay(100);                  // wait for a 100ms
  digitalWrite(PIN_TX, LOW);   // turn off the IR LED
```

```
  // Receive
  rx_bit = analogRead(PIN_RX);// read input pin
  Serial.println(rx_bit);      // print input

  // Delay until next cycle
  delay(1000);                 // wait for a 1s
}
```

Part IV
Master Node

The *Master Node* consists of an Arduino and the lab-shield. Its HW is identical to the *Development Node*. The *Master Node* is a fully functioning node and should be seen as a reference. The documentation below details now the node's functionality has been implemented and how you can expect it to behave.

8 States

The *Master Node* has been implemented with the states detailed below. The state transitions can be configured in any manner to achieve different functionalities and behaviours. As the Arduino node is single-threaded and asynchronous to the other nodes, it traverses the states sequentially. The systems time-outs and delays have thus to be configured accordingly. For example, the intended recipient of a message might not be in the RECEIVE state at the time you want to send it a message. The time-outs on the sender node therefore need to reflect the time it might take for the receiving node to re-enter the RECEIVE state. Furthermore, in this lab the node is configured to behave as depicted in Figure 9.

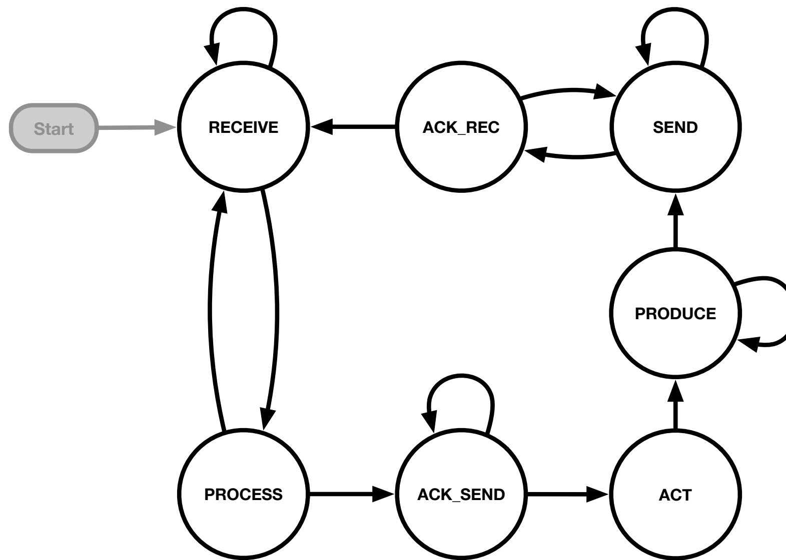

Figure 9: *Master Node* states

8.1 SEND

The send state depicted in Figure 10, transmits the message stored in a Tx buffer according to the Tx parameters in Table 5. Before it constructs and transmits the frame, it senses the channel and wait until a time-out for it to be free.

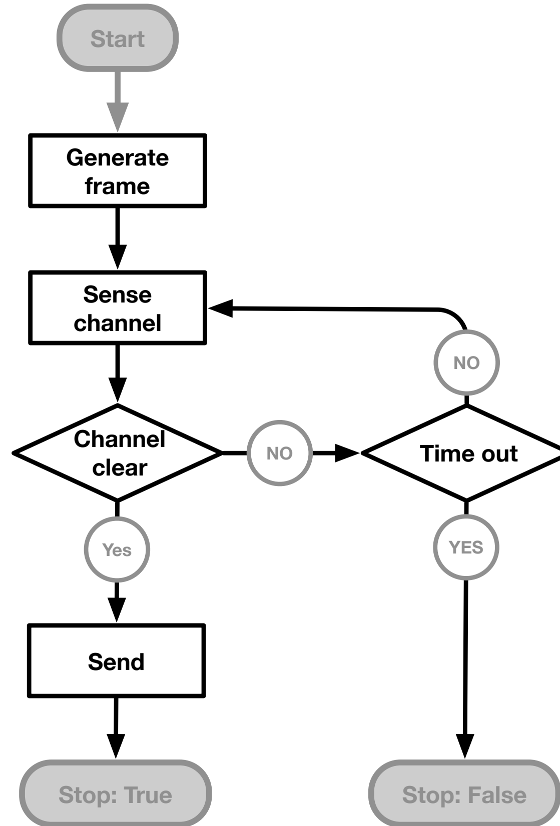

Figure 10: Send state

8.2 RECEIVE

The receive state depicted in Figure 11, continuously reads the input source and saved to an Rx buffer. The Rx buffer is the same size as a frame. After every received bit the Rx buffer is correlated with either the start or the stop flag masks depending on which internal state it is in. The loop is exited once both flags have been found or the processes has timed out.

Because execution is sequential, sampling and correlating is done in sequence. To try to keep the symbols synchronised a delay of $T_s - \hat{T}_c$, where $\hat{T}_c$ is the estimated correlation time, is added between samplings.

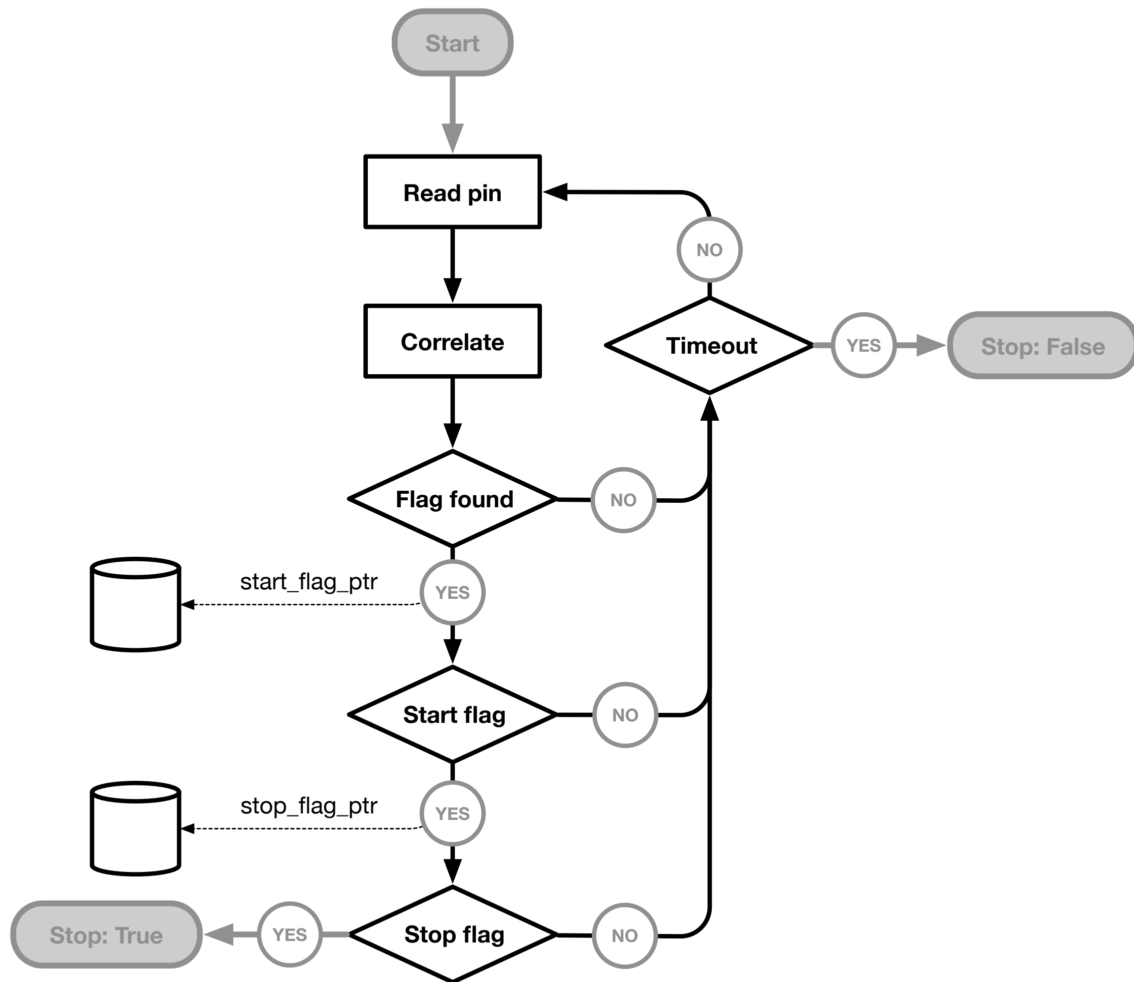

Figure 11: Receive state

When the *Master Node* detects a start flag it will light Debug LED #1 on its shield. If the Rx buffer is overflown Debug LED #1 is turned of again. However, if a end flag is found, Debug LED #3 is illuminated to convey that a frame has successfully been received. Furthermore, is the PROCESS state, Debug LED #2 is illuminated if the frame is intact and its payload successfully retrieved.

8.3 PROCESS

The process state depicted in Figure 12, down samples, decodes, and decomposed the message in the Rx buffer. Conditions can be applied to for example the type of message, and recipient. A successfully received message will light Debug LED #2 on the *Master Node*'s shield.

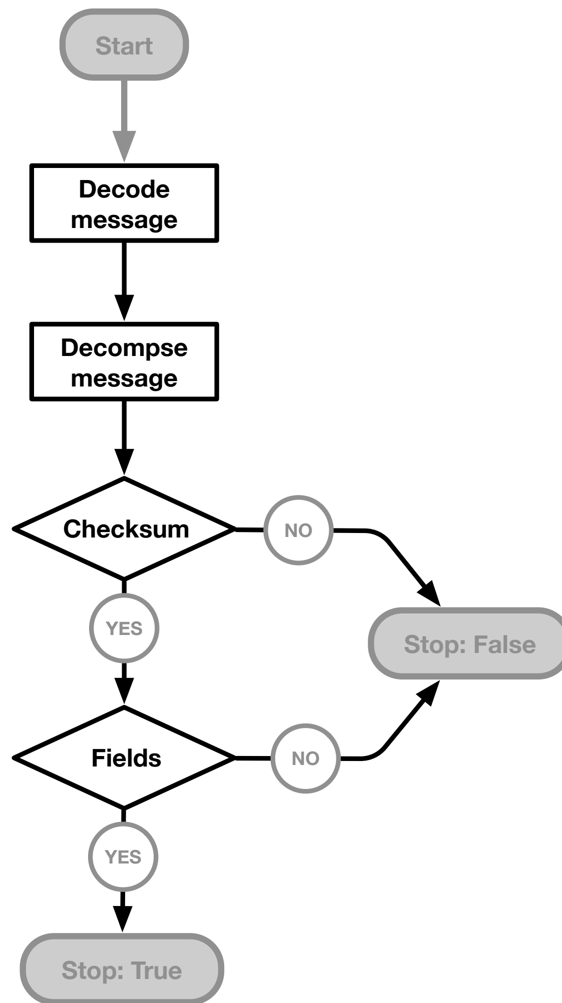

Figure 12: Process state

8.4 ACK SEND

The acknowledgement send state depicted in Figure 13, sends a ACK frame and the proceeds to the following state.

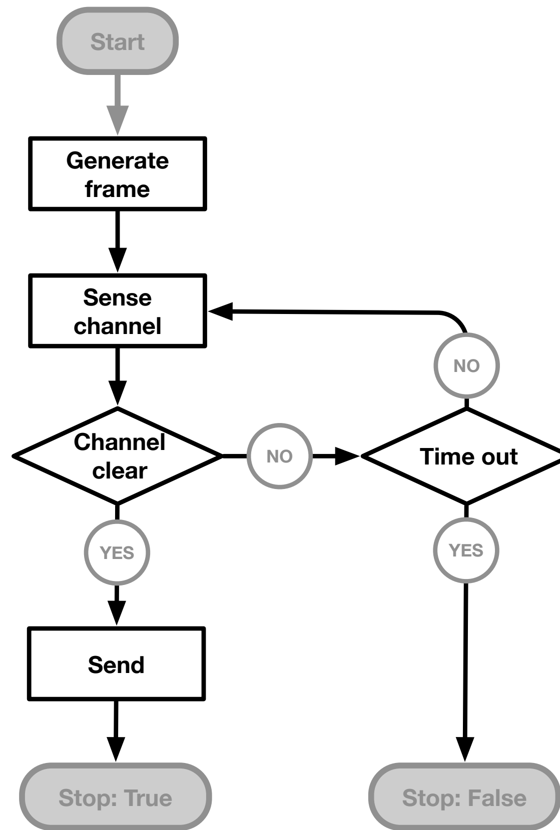

Figure 13: ACK send state

8.5 ACK RECEIVE

In acknowledgement receive state depicted in Figure 13, the sender of a DATA frame awaits an ACK message from the recipient with the same Seq number. If the expected ACK frame is not received within $T_{ACK,t}$ the node transitions to the SEND state.

8.6 ACT

The message that was decoded in the PROCESS state is in this state acted open. If for example, the received message instructed the node to turn on the blue LED this state will carry out that action.

9 Parameters

Table 5 presents the fundamental parameters that define the behaviour of the *Master Node*.

Table 5: Arduino specifications

Parameter	**Value**	**Description**
T_s	100 ms	Symbol time
N_o	2	Oversampling factor

Part V

Development Node

During the lab you will have access to one *Master Node* and a *Development Node*. The *Development Node*'s HW is identical to the *Master Node* but the *Development Node* will not come with a complete SW stack. It will be your task to achieve the goals outlined in Part VI by programming the *Development Node* accordingly. When developing the node you can seek help from *Master Node* specifications in Part IV and the supplied SW skeleton. A state machine is provided in the `void loop()` function.

Th program skeleton given for the lab is also printed in Appendix A, at the end of this manual. In the code some parts are marked with A, B, C and D. These parts will be referred to in the Instructions part, Part VI.

10 Software skeleton

The software skeleton contains a set of basic support functions, constants, and variables. The support functions and the constants they are governed by are described below, see Table 7.

Table 6: Software constants

Constant	**Parameter**	**Value**	**Description**
`T_S`	T_s	100 ms	Symbol time
`OVERSAMPLING`	N_o	2	Oversampling factor
`START_FLAG`	-	`{1,1,1,0,0,0,1,0,0,1,0}`	Oversampled start flag
`STOP_FLAG`	-	`{0,0,0,1,1,1,0,1,1,0,1}`	Oversampled stop flag
`LED_B`	-	10	Blue LED pin
`LED_R`	-	11	Red LED pin
`LED_G`	-	12	Green LED pin
`DEB_1`	-	9	Debug LED #1
`DEB_2`	-	8	Debug LED #2
`DEB_3`	-	7	Debug LED #3
`PIN_RX`	-	0	Rx diode pin
`PIN_TX`	-	13	Tx LED pin
`MSG_TYPE_ACK`	-	1	ACK frame type identifier
`MSG_TYPE_DATA`	-	2	DATA frame type identifier
`CORRELATION_THLD`	-	10	Correlation threshold

Table 7: Software constants

Constant	Type	Description
`rx_buffer`	int array	Circular Rx buffer
`rx_buffer_ptr`	int	RX buffer pointer
`address_bin`	int array	Binary representation of the address
`address_dec`	int	Decimal representation of the address

Correlate

`int correlate(const int mask[])` : This function takes a mask input and correlates it with the `rx_buffer` starting at `rx_buffer_ptr` for the length of the mask.

Decimal to Binary conversion

`void dec_to_bin(int number, int size_of_bin, int dest[], int start_point)` : This function converts decimal values to binary.

Binary to decimal conversion

`int bin_to_dec(int data[], int start_point, int stop_point)`: This function converts binary values to decimal.

Get address

`void get_address()` : Retrieves the address by reading the address pins and stores values in `address_bin` converts it to decimal and saves it in `address_dec`.

Part VI
Instructions

11 Preparations

In order to be able to complete the lab in the allotted time you need to understand how to develop and deploy an Arduino sketch and what its limitations are, prior to the lab session. Please refer to the Arduino getting started guide [3]. Study the behaviour of the *Master Node*, the SW skeleton, and the tasks in Section 13. Before you begin the lab you will be asked to present a basic outline of how you intend to implement the *Development Node*.

12 Lab set-up

During the lab you will have access to a lab computer equipped with the Arduino IDE and a USB power supply. You will also have access to one *Master Node* and one *Development Node*. The *Master Node* will loop through the states depicted in Figure 9 and you will not have access to manipulate or view its SW stack/sketch. By default, the *Master Node* will be powered by the USB power supply. You can however connect it to the lab computer to view its debug output.

To log in the computers in the lab room use the following credentials

- User name: `comnat`
- Password: `Kanejbytas123`

After logging in you should create a personal directory under U:\. This is where you will store your program project. Download the *Development Node* program skeleton from the course home page **?????** and save it in your directory. Then start the Arduino software located in the Windows Start menue, and copy-paste the code into the text editor and save the project in your directory.

To set up the communication with the Arduino board ensure that

- `Tools`–`Board` is set to "Arduino/Genuino Uno".
- `Tools`–`Port` is set to "COM3 (Arduino/Genuino Uno)". It can be some other number on the COM but the text should be here.

In the development environment you can also find typical examples that comes by default. Go to `File`–`Examples` to have a look. There are also many examples on the Internet. e.g. visit `https://www.arduino.cc/en/Reference` for a reference on the Arduino language.

There should be two Arduinos on your working bench, one connected to the computer, i.e. the *Development Node*, marked with a D, and one connected to a USB power outlet,

i.e. the *Master Node* marked with M. Try to upload the program to the *Development Node*. This can be done using the icons in the upper left corner of the text editor.

For debugging purposes you can use either the three LEDs D3 to D5 on the shield (pins 7, 8 and 9) or use the Serial functions for printing data to the computer. Then open the Serial Monitor by clicking the magnifying glass in the upper right corner.

13 Tasks

The primary practical objectives of this lab is to:

- Implement the necessary functionality in the *Development Node* so that it is able to communicate with the static and predicable *Master Node*
- Convey an action from the agent node (*Development Node*) to the actuator node (*Master Node*). The actuator shall then actuate the action.

These two objectives have been broken down in to tasks which you should complete during this lab:

1. Develop the PRODUCE state In the Skeleton code the implementation of the state transition graph is made by a `switch` statement that you find marked by a B in Appendix A. The code is started in state PRODUCE, so this where your first code goes. Here the parts of the frame should be produced, which you find marked with a C in the code. The transmitted frame is made up from the `tx`-variables and the received frame the `rx`-variables.

   (a) Start by turning off all the colour LEDs. In the Skeleton.ino code there are some constants defined, you find them marked with A in the appendix list. These constants give the pin numbers for the LEDs used by the program. In the function `void setup()` they are defined as inputs or outputs. After you turned them off insert a delay so you can see what is happening.

   (b) Choose one of the colour LEDs and light it. It can e.g. be chosen at random (see `random()`) or sequentially looping through them for a random number of iterations. This is the colour that eventually will be send to the *Master Node*.

   (c) Update the `tx`-variables that will form the frame (mark C in the appendix). This is the information that will go into the transmitted frame.

   (d) After you are done with the PRODUCE state the next time the program comes to the `switch` statement it should choose the SEND state. This is accomplished by updating the variable `state`.

   Verify the outcome by outputting the result over the serial connection to the computer. e.g. `Serial.print` or `Serial.println`.

2. Implement the SEND state. The first thing to do is to convert all the parts (as decimal numbers) to binary vectors and store serialised in `tx_msg`.

   (a) Since the signalling is binary, the frame must be represented as a binary vector. In Arduino, as in C/C++, a vector is represented by an array, typically initiated with something like `int Values[10];`. Then an array of length 10 is allocated. The values are accessed by indexing starting at 0, so the values are `Values[0]`, `Values[1]`, ..., `Values[9]`. As in C/C++ there is no runtime check of the indexing, so you can continue to write and read outside the vector without any complaints. But then you are writing and reading on other memory elements which will typically cause strange errors. So be aware of your index pointer and use the module `%` operator.[1] Use the function `dec_to_bin` given in the skeleton code. Test it first and see the answer in the Serial Monitor.

   (b) The addressing, sequence number and checksum is not used in this lab and therefore not checked by the receiver. These can all be set to zeros. There are two types of frames, DATA and ACK, and this should be set as DATA.

   (c) Convert the decimal values in a frame buffer to binary and store the frame in `tx_msg`. Use `Serial.print` and `Serial.println` to see that it is correct. The frame structure is depicted in Figure 2. There is a predefined array for storing the binary frame, called `tx_msg`, see mark C in the appendix code. This is allocated as an array with length `LEN_FRAME`, which is the length of the frame, excluding the starting and ending flags.

   (d) Transmit start flag, frame and end flag in a sequence using the transmit IR LED, defined as `PIN_TX`. Notice that $T_s = 100$ ms is the sample time and that there are $N_s = 2$ samples per signal pulse. When transmitting the signal the *Master Node* respond by lighting the debugging LEDs. The first is lit when the start flag is detected and the second when the frame is completed. The third is lit after the checksum is accepted, but since this is not used here it comes directly after the second LED. If the frame was successfully received by the *Master Node* it will turn on the LED specified in the payload. If it doesn't, there is something wrong in the signalling. Use the debugging LEDs and the Serial.print to find out what.

   Now the first part of the lab is completed and you can send data from the *Development Node* to the *Master Node*. When the *Master Node* receives the data it sends back an Acknowledge frame, and ACK. This frame looks the same, but the type is set to ACK. So the next step is to receive a frame and check that the *Master Node*

[1] initialisation of the array allocates space for 10 integers in this case. The variable Value is a pointer to the first value in the memory, and the index is used to increment the pointer a number of positions in the memory. An integer uses 4 bytes so the value of `Value[i]` is read by pointing to the memory at position `Value+i*4`.

has received your data. This is typically done in the RECEIVE state. See to that the program ends up here after transmitting the frame.

3. Implement the RECEIVE state. In the Skeleton code a read buffer `rx_buffer` is defined. The idea of receiving is to listen to channel continuously, with sample time T_s, and correlate against the starting flag. Reading the channel means reading the analogue value of `PIN_RX` and storing the sequence in the buffer. Then the buffer needs to be implemented circularly, in this case with cycle length `LEN_BUFFER`. The buffer pointer `rx_buffer_ptr` is incremented using modulo `LEN_BUFFER`. If this buffer and buffer pointer are used, the correlation can be performed using the function `int correlate` defined at the end of the Skeleton code in the appendix. As input you give the sequence to correlate against, i.e. start or end flag.

   (a) Read the Rx pin and deposit value in `rx_buffer` and increment `rx_buffer_ptr`. Correlate the signal to determine if you encountered a flag. Remember to use `analogRead()` when reading the Rx diode.

   (b) Wait until the next sample time. The time spent in correlation is not negligible, meaning that just waiting time T_s to the next sample might get your program out of sync. The sample time is the time between two consecutive sampling starts. Use the Arduino function `millis()` which gives the time since start in milliseconds.

   (c) Exit when a full message has been received. Do not forget about the oversampling. Use two consecutive samples to estimate if it was a `0` or a `1` that was sent.

   (d) Decode the message and determine if it was a DATA or ACK frame and then proceed to the correct subsequent state. If it is ACK the *Master Node* has received your data, and the program can start from the beginning.

4. What happens if you send a frame to the *Master Node* and it is not received? How can you solve the problem?

5. What happens if you send a frame to the *Master Node* and it sends back an ACK frame, but you don't receive it? How can you solve the problem?

References

[1] Arduino software. `https://www.arduino.cc/en/Main/Software`, 2015.

[2] Arduino software (ide). `https://www.arduino.cc/en/Guide/Environment`, 2015.

[3] Getting started with arduino. `https://www.arduino.cc/en/Guide/HomePage`, 2015.

[4] Introduction to the arduino board. `https://www.arduino.cc/en/Reference/Board`, 2015.

[5] Ben Fry, Casey Reas, et al. Processing. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2007.

Part A
Skeleton.ino

In the following pages the code for Skeleton.ino is given.

D

```
//
// Communication parameters
//
  // Tx/Rx
  const int T_S = 100;
  const int TX_DELAY = 5000;
  const int OVERSAMPLING = 2;

  // Messageing
  const int LEN_MSG = 4;
  const int LEN_MSG_TYPE = 2;
  const int LEN_SEQ_NBR = 2;
  const int LEN_ADDR = 4;
  const int LEN_CHECKSUM = 4;

  // Flags
  const int LEN_FLAG = 11;
  const int START_FLAG[] = {1,1,1,0,0,0,1,0,0,1,0}; // Shorter and
flags design for teh IR medium would be better
  const int START_FLAG_MASK[]  =
{1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,1,1,1,-1,-1,-1,-1,1,1,1,-1,-1}; //
Oversampling = 2, TO-DO - Generate dynamically
  const int STOP_FLAG[] = {0,0,0,1,1,1,0,1,1,0,1}; // Shorter and
flags design for teh IR medium would be better
  const int STOP_FLAG_MASK[]  =
{-1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,-1,1,1,1,1,-1,-1,1,1}; //
Oversampling = 2, TO-DO - Generate dynamically
  const int CORRELATION_THLD = 10;

  // A/D Converter
  const int AD_TH = 900;

  // Buffers
  const int LEN_BUFFER = LEN_FLAG*OVERSAMPLING*2 +
LEN_ADDR*OVERSAMPLING*2 + LEN_MSG_TYPE*OVERSAMPLING +
LEN_SEQ_NBR*OVERSAMPLING + LEN_MSG*OVERSAMPLING +
LEN_CHECKSUM*OVERSAMPLING;
```

A

```
//
// Hardware
//
  // LEDs
  const int LED_B   = 10;
  const int LED_G   = 11;
  const int LED_R   = 12;
  const int DEB_1   = 7;
  const int DEB_2   = 8;
```

A

```
  const int DEB_3   = 9;
  const int PIN_RX  = 0;
  const int PIN_TX  = 13;

  // Address
  const int PIN_ADDR[] = {3,4,5,6};
```

C

```
//
// Messaging
//
  const int MSG_TYPE_ACK   = 1;
  const int MSG_TYPE_DATA   = 2;

  const int LEN_FRAME = LEN_ADDR*2 + LEN_MSG_TYPE + LEN_SEQ_NBR +
LEN_MSG + LEN_CHECKSUM;

  // Rx message
  int rx_msg[LEN_FRAME];
  int rx_msg_ptr           = 0;
  int rx_msg_len           = 0;
  int rx_msg_from          = -1;
  int rx_msg_to            = -1;
  int rx_msg_type          = -1;
  int rx_msg_payload       = -1;
  int rx_msg_seq_nbr       = -1;
  boolean rx_msg_checksum  = -1;

  // Tx message
  int tx_msg[LEN_FRAME];
  int tx_msg_ptr           = 0;
  int tx_msg_ctr           = 0;
  int tx_msg_from          = -1;
  int tx_msg_to            = -1;
  int tx_msg_type          = -1;
  int tx_msg_payload       = -1;
  int tx_msg_seq_nbr       = -1;

//
// Runtime
//
  // States
  const int NONE        = -1; // No state
  const int RECEIVE     = 0; // Rx: Receive message with a timeout.
  const int SEND        = 1; // Tx: Transmitt what is in the
  const int PROCESS     = 2; // Process payload
  const int ACK_SEND    = 3; // Handle ACK
  const int ACK_REC     = 4; // Handle ACK
  const int PRODUCE     = 5; // Produce content/messahe to send
```

```
  const int ACT         = 6; // Act on payload
  const int WAIT        = 7; // Wait
  const int DEBUG       = 8; // Print all system proporties

  // Rx buffer
  const int START_FLAG_LEN_BUFFER = 11;
  int rx_buffer[LEN_BUFFER]; // Seize of 2 flags and max message
  int start_flag_ptr   = -1;
  int stop_flag_ptr    = -1;
  int rx_buffer_ptr    = 0;

  // Address
  int address_bin[LEN_ADDR]; // Address is dynamically assigned using
DIP switches.
  int address_dec = -1;

  // Runtime variables
  int i, j, result, corr, mean, sensor_value, start_point, index; //
Not very readable, sorry :)
  boolean outcome;

  // Channel state
  boolean ch_available = 0;
  int ch_state[] = {-1,-1,-1,-1};

  // Timekeeping
  unsigned long timer, time, t_s_delay_temp;

  // State
  int state = NONE;

//
// Device functionality
//
int current_led = -1;

//
// Code
//
void setup() {
  Serial.begin(9600);
  pinMode(LED_B, OUTPUT);
  pinMode(LED_G, OUTPUT);
  pinMode(LED_R, OUTPUT);
  pinMode(DEB_1, OUTPUT);
  pinMode(DEB_2, OUTPUT);
  pinMode(DEB_3, OUTPUT);
  pinMode(PIN_TX, OUTPUT);
```

```
  // Address pins
  for (i=0; i<LEN_ADDR; i++) {
    pinMode(PIN_ADDR[i], INPUT);
  }

  // Inital state
  //state = RECEIVE;
  state = PRODUCE;
}
```

B

```
void loop() {
  get_address();
  // State machine
  switch(state){

    case SEND:
      break;

    case RECEIVE:
      break;

    case PROCESS:
      break;

    case ACK_SEND:
      break;

    case ACK_REC:
      break;

    case PRODUCE:
      break;

    case ACT:
      break;

    default:
      break;
  }
}

// Retreive address from DIP switch
void get_address(){
  for (i=0; i<LEN_ADDR; i++) {
      address_bin[i] = digitalRead(PIN_ADDR[i]);
  }
```

```
  address_dec = bin_to_dec(address_bin, 0, LEN_ADDR-1);

  Serial.print("[F] Address is: ");
  Serial.println(address_dec);
}

//
// Frame/Message processing.
//

// Binary to decimal
int bin_to_dec(int data[], int start_point, int stop_point){
  Serial.print("\t [F] Binary to decimal: ");
  result = 0.0;
  j = 0;
  for(i=stop_point; i>=start_point; i--){
    Serial.print(data[i]);
    result += data[i]*pow(2.0, j) + data[i]*0.01; // +0.01 for float
to int rouding error
    j++;
  }
  Serial.print(" -> ");
  Serial.println((int)result);
  return result;
}

// Decimal to binary with destination array with pointers
void dec_to_bin(int number, int size_of_bin, int dest[], int
start_point){
  Serial.print("\t [F] Dedcimal to Binary: Number ");
  Serial.println(number);
  Serial.print("\t");
  for(i=0; i<size_of_bin; i++){
    dest[start_point + size_of_bin - 1 - i] = number & 1 ? 1: 0;
    Serial.print(start_point + size_of_bin - 1 - i);
    Serial.print(":");
    Serial.print(number & 1 ? 1: 0);
    Serial.print(" ");
    number = number/2;
  }
  Serial.println(" ");
}

// A/D Converter
int adConv(int value){
  return sensor_value > AD_TH ? 0 : 1;
}
```

```
// Correlate signal with flag
int correlate(const int mask[]){
  corr = 0;

  start_point = (LEN_BUFFER + rx_buffer_ptr - LEN_FLAG * OVERSAMPLING)
% LEN_BUFFER;
  for (i=0; i< LEN_FLAG * OVERSAMPLING; i++){
    corr += mask[i] * rx_buffer[(start_point + i) % LEN_BUFFER];
  }

  if (corr>=CORRELATION_THLD){ // Set automatically based on
oversampling rate, flag, and mask.
    Serial.println("[Rx] Flag found");
    return rx_buffer_ptr;
  }
  return -1;
}
```