EITF45 - Computer Communication
Lab 1 - Specifications
Point to Point Communication

Manual Version 4.1.2

Electrical and Information Technology

November 20, 2019

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Table of Contents

Part I
Protocol and Communication Specifications

I.1 A layered model

The base for the communication protocol in this and the following lab is a layered reference model. The model has three layers. As can be seen in Figure I.1, the transport and network layers are not defined in the model.

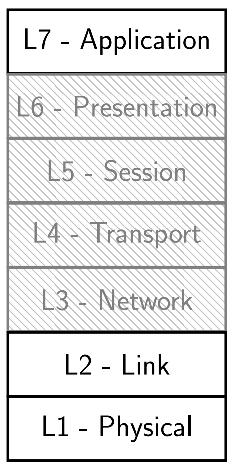

Figure I.1: The lab three layered reference model.

The naming convention follows [7] regarding encapsulation. A *message* is passed from and to the application layer. On the Layer 2 (L2) layer a *frame* is defined.

The layers as well as the interfaces between the layers are described in the following sections.

I.2 Application

The *Development Node* and the *Master Node* have different objectives on the application layer. The *Development Node* supports the operator's control of the application, while the *Master Node* reacts on the data sent to it from the *Development Node*. The *message* structure is similar in both use cases as can be seen i Table III.10. For simplicity, the two application message fields are implemented as integer arrays in the library.

Table I.1: Application layer *message* structure.

Field	**Length (bits)**	**Description**
Payload	8	Message content

I.3 The Link Layer

This layer defines the frame that is passed between the nodes. Addressing, reliable transmission, i.e. Automatic Repeat Request (ARQ), and fault detection is also defined here.

I.3.1 Frame structure

The L2 frame format is seen i Table I.2. The frame size is fixed. Each frame has two address fields, the destination and the source. Each address field is four bits long. There are two types of frames defined, DATA and ACK.

Each frame has a 4-bit sequence number.

The payload is allocated 8-bits and used for the application layer message payload.

Each frame can carry 8 parity bits using the CRC-8 Bluetooth generator `0xA7` calculated over the frame. If Cyclic Redundancy Check (CRC) is not used, this field should be set to zero.

Table I.2: L2 *frame* structure. See **??** for corresponding variable names.

Field	**Length (bits)**	**Description**
From	4	Source address
To	4	Destination address
Type	4	Type of message [ACK | DATA]
SEQ	4	Sequence number
Payload	8	Data, i.e. application message payload
CRC	8	CRC of frame

I.3.2 Addressing (Not used in this lab)

Each node has a four-bit address, i.e. an address space of 16. The *Development Node*'s address should be code in the *Development Node*'s sketch. The destination's address is set using the four-toggle dip-switch located on the board. A node shall only process Receiverd messages address to it. If addressing is not used, these fields should be set to zero.

I.3.3 DATA and ACK frames

A DATA frame carries application data, which is stored in 8 bits Payload field. A DATA frame is denoted by a 0010_2 in the Type field.

An ACK frame is the answer to a correctly Receiverd DATA frame. It is only sent once, and carries an empty payload. The SEQ number field contains the sequence number of the acknowledge DATA frame. An ACK frame is denoted by a 0001_2 in the Type field.

I.3.4 Sequence numbers (Not used in this lab)

The sequence number must be incremented for each new DATA frame. In an ACK frame it is used to identify the successfully Receiverd DATA frame that is acknowledged. If sequence numbers are not used, this field should be set to zero.

I.4 Reliable transmission (Not studied in this lab)

The nodes employ a Stop-and-wait ARQ scheme. The sender of a DATA frame shall reTransmitter that frame, persisting the sequence number, if it does not Receiver an ACK frame with the same sequence number from the recipient within a certain time. The number of re-transmissions must be limited. Similarly, if retrieved successfully and is correctly addressed, the recipient of a DATA frame shall Transmitter an ACK to the sender pertaining the same sequence number, see Figure I.2

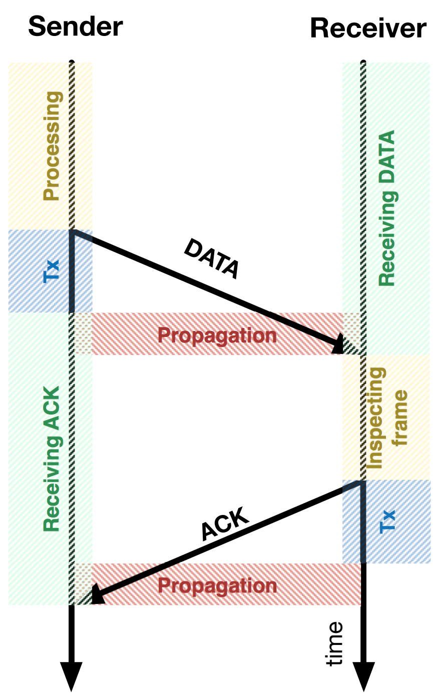

Figure I.2: Example communication scenario *(The time lines are not proportional)*

I.5 The Physical layer

The communication link is physically achieved by using a pair of Infra-red (IR) Light Emitting Diodes (LEDs) ($\lambda = 900nm$) over a half duplex channel. Communication on the link is coded and propagated using On-Off keying; the symbol 0 is coded as no light and and the symbol 1 is coded as light. On Layer 1 (L1), symbols corresponds to one bit on L2. A pulse length, corresponding to one symbol, is defined as $T_s = 100ms$. The node's respective clocks are not synchronised.

As an example of a signal, the preamble is shown in Figure I.3.

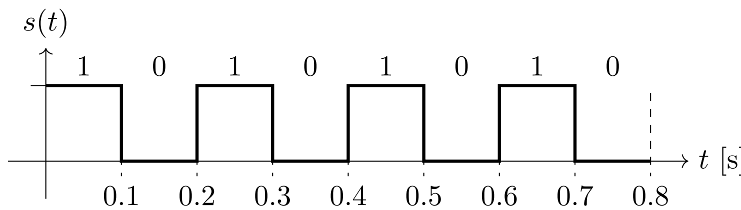

Figure I.3: The preamble as a signal.

The symbols, that corresponds to the L2 frame bits, must be preceded by a preamble of eight symbols followed by an Start Frame Delimiter (SFD) of eight symbols. The full L1 symbol train can be seen in Table I.3

Table I.3: Train of symbols on the L1 layer

Field	**Length (bits)**	**Description**
Preamble	8	0b10101010, for synchronisation of Receiver to Transmitterted pulses
SFD	8	0b01111110, marks start of a frame
Data	32	The L2 layer frame

I.5.1 Collision avoidance

Accessing the channel is done without checking if it is available or not. Thus, the channel mimics Pure ALOHA. [2]

I.6 Interfaces

Between each two layers in the reference model, an *interface* is defined.

I.6.1 Application – L2 interface

The interface between the application and the Layer 2 (L2) equals the message format, see Table III.10. The address field should contain the destination's address when sending the message from the application to the L2 layer, and the source's address when the application Receivers a message from the L2 layer. The payload field should contain the selected LED when sending the message from the application to the L2 layer, and the LED to turn on when the application Receivers a message from the L2 layer.

I.6.2 L2 – L1 interface

The interface between the L2 and the L1 layers is a buffer containing the bits forming the L2 frame. The bit buffer is 32 bits long. As discussed above, it is suggested that this buffer is implemented as an array of bytes, each byte contains exactly one bit.

Part II
Some Background Theory

II.7 L1 Pulses and L2 Frames

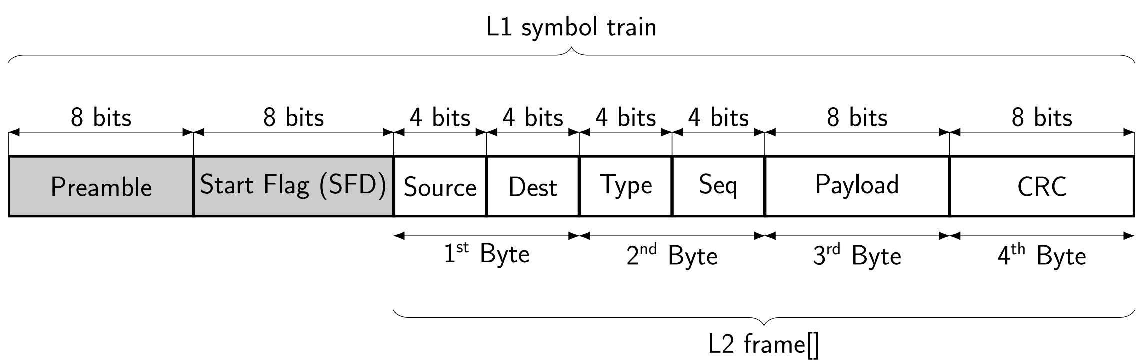

Figure II.4: The L2 frame structure as seen on L1 and L2.

The L1 symbol train, as seen in Figure II.4, consists of three parts, the preamble, the SFD and payload. The payload contains the L2 frame with its header, payload and tail, correspond to individual integer variables. The SFD is stored as a byte, but is actually bit oriented. Since the frame has a fixed length, there is no need for a stop flag.

On L1, information is transferred as pulses of light or the absence of light. Since On-Off keying is used, the pulses are all of the same duration, i.e. the sample time T_s, which in our case is 100 ms. The bytes and integers forming the frame has to be converted to something bit oriented, i.e. the symbols, before we can transmit a frame. Let the binary value 1 represent a light pulse and the binary value 0 represent a pulse with no light. Our task is now send all the symbols as pulses. In our case the symbols are the same as L2 frame bit values. Once the symbols corresponding to the preamble, the SFD and the L2 are at hand, we can send the symbol train by reading out pulse representations from the arrays `PREAMBLE[]`, `SFD[]` and the `tx_frame`, one symbol by one symbol, with an interval of T_s, and let them control the sending device, in our case the IR diode. Note that the preamble and the SFD belongs to the physical layer, L1, while the `tx_frame` and `rx_frame` belongs to the link layer, L2, but is also defined as the interface between the two layers.

The reception of signals or pulses works more or less the same way, but backwards. Light pulses are sampled each T_s once the preamble has trigged the Receiver. The sampled values will be represented by a 1 or a 0, which will be stored in a Receiver buffer. Ones the SFD has been identified, the data part of the frame is located. The bits can now be converted and stored directly in the `RECEIVED::frame`, i.e. the interface between L1 and L2, which is sent to the L2 where the decomposition to L2 frame variables can take place.

II.8 Preamble and SFD

Each frame is preceded by an eight symbol preamble. The objective of the preamble is twofold. The first objective is to trigger the Receiver to start sampling. This is solved by allowing the idle Receiver to be triggered by a positive flank. Since no light is used when the link is idle, a transition to light, i.e. a high pulse, is this positive flank. The other objective of the preamble is to allow the Receiver's sampler to synchronise to the Receiver. If the Transmitter shifts between light and no light in a well-known fashion, the Transmitter can adjust the timing of the sampler so that it synchronises with the Transmitter. The preamble has the bit pattern 10101010, which solves both objectives.

In our case, we only use the leading positive flank of the preamble for to trigger the Receiver. The synchronisation of the clocks can be skipped, because the micro-controllers' clocks are stable enough when compared to the pulse time T_s and the frame length.

Once the Receiver has started to sample pulses in a synchronised fashion, the Receiver has to detect the start of the L2 frame. This is done by comparing a consecutive number, equal to the length of the SFD, of Receiverd symbols with the SFD. The SFD is a byte in our case, so each time a new incoming symbol has been decoded it, and the seven symbols preceding it, is compared against the SFD. One way of doing this is to left shift the incoming symbols into a buffer of the same length as the SFD and simply perform bitwise XOR with the SFD. Once this operation results in a zero value the SFD has been found, and the buffer can be omitted, and symbols can be translated to bits and stored in the interface buffer `Receiver::frame_buffer[]`.

Part III
Arduino, Code and Shield

This chapter contains reference information on the Arduino software and hardware. Documentation of the skeleton and library, i.e. the basis for your coding, is found here.

III.9 Arduino Software

An Arduino micro-controller is programmed using a language which has many similarities with C/C++. You preferably develop your code in the Arduino Integrated Development Environment (IDE) [4].

A sketch has two primary functions, `setup()` and `loop()`. [3] The `setup()` function is where you declare how you want the Input/Output (I/O) to behave and initialise your global variables, see Listing III.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 [5] (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` sketch. This is the equivalent to the "Hello World" program.

Listing III.1: Sample Arduino code, Transmit (Tx) and Receive (Rx)

```
// Assign pin num
const int PIN_RX = 0;          // Receiver pin \#
const int PIN_TX = 13;         // Transmitter pin \#

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

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

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

}

III.9.1 Arrays in Arduino (C/C++)

In Arduino, as in C/C++, a vector is represented by an *array*, typically initiated with a declaration 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 array without any complaints. If so, you are writing and reading other memory elements then intended, which will typically cause strange errors. So be aware of your index pointers. Apply the modulus operator `%` with an appropriate constant on the index pointer. [1]

The C++ library string.h contains some useful functions for handling arrays. On is `memmove()`, see Section III.9.5.2, which you can use directly in your sketch.

III.9.2 Data types in Arduino (C/C++)

Table III.4 lists some data types that might be useful in this lab. Using proper types for different variables helps to save the memory and accelerate the process. For example, we claim variable `SFD` as type `byte` since the SFD contains 8 bits.

Table III.4: Ardunio data types

Datatype	**RAM usage**	**Range**
`boolean`	1 byte	logical
`byte`	1 byte	0 ∼ 255
`int`	2 bytes	-32,768 ∼ 32,767
`unsigned int`	2 bytes	0 ∼ 65,636
`long`	4 bytes	-2,147,483,648 ∼ -2,147,483,647
`unsigned long`	4 bytes	0 ∼ 4,294,967,295

III.9.3 Bit operations

To read, write or manipulate individual bits in variables, bit operations are needed. Bit operations can be performed on any type of signed and unsigned integer variables, `byte`, `integer`, `word` or `long`. In the following, operations on bytes are used as example.

[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[0]+i*4`.

III.9.3.1 Read or write a specific bit from a byte

Vital bit operations are the setting and resetting as well as reading of individual bits in a variable. The Arduino programming language has a set of bit operations which allows you to address individual bits. These operations are considered slow. Instead you can use logical AND (`&`), logical OR (|) and shift operations.

Left shift `<<` and right shift `>>` are used to move bit(s) a defined number of steps left or right. When shifting left, the most significant bits are shifted out and 0s are shifted in from the right. Similarly, shifting right means that the least significant bits are shifted out of the byte and 0s are shifted in from the left.

Logical AND (`&`) can be used to mask out not valid bits or to set bits to 0. If you want to set a specific bit to 1 you use logical OR (|).

Listing III.2 shows an example to read the third bit from the right-hand side of a byte.

Listing III.2: Sample Arduino code, read a bit

```
byte  my_byte, third_bit;
third_bit = (my_byte >> 2) & 0x01;
```

In this example, `>> 2` moves the content of `my_byte` two steps to the right. Thus the third bit is moved to the Least Significant Bit (LSB) position, and `& 0x01` zeros all the bits other than the LSB.

To write bits e.g. the lower part of a byte, use `<<` to move the bits to the right position and the use logical or (`|`) to add the bits in. See the example below, Listing III.3 to save `parameter_1` to the higher 4 bits of the `frame` and `parameter_2` to the lower 4 bits.

Listing III.3: Sample Arduino code, write bits

```
byte parameter_1 = 0x07; // parameter_1 = [0 1 1 1]
byte parameter_2 = 0x0A; // parameter_2 = [1 0 1 0]
byte frame;
frame = (parameter_1 << 4) | parameter_2;
```

Note that the bit-shift operator will not change the variable itself. The result have to be assigned a variable with the `=` operator.

III.9.3.2 XOR

The XOR operator, `^`, in Arduino works bitwise. XOR of two bits returns 0 if they are the same, otherwise returns 1. XOR of two bytes returns 0 if they are the same, otherwise returns a non-zero value.

III.9.4 Ternary (Conditional) operator

`condition ? <if true> : <if false>`

The ternary, or conditional, operator can be very hand. Instead of writing if-else statements, you can assign a variable a value depending on a condition. In the following example the variable `largest` is assigned the greater value of a and b: `largest = (a>b) ? a : b;`.

III.9.5 Useful Arduino function(s)

III.9.5.1 millis()

`unsigned long millis()` : Returns the number of milliseconds since the start of the Arduino bord. See [1].

III.9.5.2 memmove()

`void * memmove ( void * destination, const void * source, size_t num )` : Copies `t_num` bytes from the memory position pointed out by `source` to the memory position pointed out by `destination`. The pointers `source` and `destination` can be pointing to the start of arrays. To copy the L2 frame in `rx.frame[]` to `tx.frame[]`, issue the commmand `memmove(rx.frame, tx.frame, LEN_FRAME);`.

III.10 Debugging tools

There are two debugging tools available: the *Serial Monitor* and LEDs on the shield.

III.10.1 Serial Monitor

For debugging purpose, you can let the sketch send text messages to the *Serial Monitor* by using the `Serial` functions in your code. Open the *Serial Monitor* by clicking the magnifying glass in the upper right corner of the Arduino IDE.

The functions that are most used are `Serial.print()` and `Serial.println()`. Both take a string or a variable as input parameter, and both accept formatting strings. But for this lab it is enough to know that `Serial.println()` prints the content of the input parameter, and finishes with a carriage return and a line feed, while `Serial.print()` only prints the content.

III.10.2 Debug LEDs

There are three debug LEDs on the shield for your disposal. The pins associated with each LED are found in Table III.12. Writing a HIGH to a pin turns the corresponding LED on; turn it of by writing a LOW.

III.11 The Development Node

During the lab you will have access to one *Master Node* and a *Development Node*. The *Development Node*'s Hardware (HW) is identical to the *Master Node* but the *Development*

Node will not come with a complete Software (SW) stack. It will be your task to achieve the goals outlined in **??** 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.

III.12 The Skeleton and the Library

For your help, a skeleton and a supporting library has been devised.

The skeleton is the base for your code. For one thing, it defines the Arduino sketch functions `setup()` and `loop()`. The state machine is the major part of the `loop()` function, and it is here you add your code. At the end of the code there is a place-holder for your functions. Also, a few constants and variables are defined. The skeleton is described in Section III.12.1.

The library defines global constants, arrays and variables, and classes you can use in your code. As with all classes, they have to be instantiated into objects before you can use them. The library is described in Section III.12.2

III.12.1 Skeleton Details

The skeleton begins with the library include statement: `#include <etsf15lib.h>`. The skeleton is then divided into four major areas:

- Variable declarations
- The `setup` function
- The `loop` function
- Area for optional functions

III.12.1.1 Variable Declarations

In this area of the skeleton a number of variables are declared, that can be used in the state machine and your functions.

Note that the *constructors* of the classes are called without parenthesis. This is because these constructors takes no arguments when they are initiated. You may need to re-assign the arguments to some of them in you implementation if needed.

III.12.1.2 The `setup` function

Initialisation of the hardware at hand and the software is performed in the `setup` function. The Shield class has a `begin()` method which is called from here.

Table III.5: Global variables

Type	Name, declaration	Description
int	`state = NONE`	state machine control variable
int	`selected_led`	for temporary storage
int	`src`	source address
int	`dst`	destination address
int	`type`	message type
Shield	`sh`	Declaration of object, instance of class Shield
Transmitter	`tx`	Declaration of object, instance of class Transmitter
Receiver	`rx`	Declaration of object, instance of class Receiver
Payload	`Payload`	Declaration of object, instance of class Payload
RECEIVED	`received`	Received message struct
Frame	`tx_frame`	Transmit frame
Frame	`rx_frame`	Receive frame

III.12.1.3 The `loop` function

The `loop` function hold the main part of the sketch, i.e. the state machine. Each state has its own code area, and is finished of with a `break` statement. See the skeleton code for more details.

III.12.1.4 Your functions area

This area is found at the end of the skeleton. If you are to construct your own functions, this is the recommended area for them.

III.12.2 Library Details

The library `ETSF15lib` contains constants and methods that supports the construction of the lab sketch. As with all Arduino libaries, the `ETSF15lib` is built on classes that defines variables and methods. The global constants are not defined in classes and can thus be used once the library is included in the main sketch. To include the library an \#include <etsf15lib.h> statement has to be deployed in the beginning of the sketch. The global constants and the classes `Shield`, `Transmitter`, `Receiver`, `Frame`, `Payload` and `RECEIVED` are described in the following sections.

III.12.2.1 Global Constants

Table III.6: Definition of the library ETSF15lib constants.

Type	Name, declaration	Value	Description
int	T_S	100 ms	T_s, symbol length
int	AD_TH	900	A/D converter threshold
int	MAX_TX_ATTEMPTS	3, see note 1)	Max transmission attempts
int	LEN_PREAMBLE	8	preamble size
byte	PREAMBLE_BYTE	0b10101010	preamble
byte	PREAMBLE[]	{1,0,1,0,1,0,1,0}	byte array version of preamble
byte	SFD_BYTE	0b01111110	Start Frame Delimiter (SFD)
int	LEN_SFD	8	SFD size
byte	SFD[]	{0,1,1,1,1,1,1,0}	byte array version of SFD
int	LEN_BUFFER	see note 2)	L1 buffer size
int	LEN_FRAME_PAYLOAD	8	L2 frame payload size
int	LEN_FRAME_TYPE	4	L2 frame message type size
int	LEN_FRAME_SEQNUM	4	L2 frame sequence number size
int	LEN_FRAME_ADDR	4	L2 frame address size
int	LEN_FRAME_CRC	8	L2 frame CRC size
int	LEN_FRAME	see note 3)	L2 frame length
int	FRAME_TYPE_ACK	1	ACK message type
int	FRAME_TYPE_DATA	2	DATA message type
byte	test_frame[]	see note 4)	a full test frame
int	LED_B	10	Blue LED pin
int	LED_R	11	Red LED pin
int	LED_G	12	Green LED pin
int	DEB_1	9	Debug LED #1
int	DEB_2	8	Debug LED #2
int	DEB_3	7	Debug LED #3
int	PIN_RX	0	Rx diode pin
int	PIN_TX	13	Tx LED pin
int	BUTTON	2	Button pin

Note 1): The number of transmission tries for each unique message is limited. Note 2): The LEN_BUFFER is equal to the sum of LEN_PREAMBLE + LEN_SFD + LEN_FRAME_ADDR*2 + LEN_FRAME_TYPE + LEN_FRAME_SEQNUM + LEN_FRAME_PAYLOAD + LEN_FRAME_CRC.
Note 3): The LEN_FRAME is equal to the sum of LEN_FRAME_ADDR*2 + LEN_FRAME_TYPE + LEN_FRAME_SEQNUM + LEN_FRAME_PAYLOAD + LEN_FRAME_CRC, i.e. the L2 frame header, payload and tail.

Note 4): The `test_frame[]` contains a full L2 frame excluding preamble and SFD. It is not a `Frame` type object but a simple array, you can send it bit by bit by reading all the elements in the array. See Listing III.2 to know how to send this `test_frame[]` array.

For the state machine, a number predefined states, have been defined as constants.

Table III.7: Predefined states (constants)

Type	Name	Value	Description
int	`NONE`	-1	No state
int	`L1_PHY_Receiver`	0	Rx: Receiver frame
int	`L1_PHY_SEND`	1	Tx: Transmitter frame
int	`L2_LINK_FRAME_DECOMPOSE`	10	Process Receiverd payload on layer L2
int	`L2_LINK_FRAME_COMPOSE`	11	Process the L2 payload to be sent
int	`L2_LINK_ACK_REC`	12	Process reception of an ACK frame
int	`L2_LINK_ACK_SEND`	13	Process sending of an ACK
int	`L2_LINK_RETransmitter`	14	Control of ARQ
int	`L7_APP_PRODUCE`	20	Produce content/message to send
int	`L7_APP_ACT`	21	Act on Receiverd payload
int	`WAIT`	-2	Wait
int	`DEBUG`	-3	Print all system proporties
int	`HALT`	-4	"halt" the system, i.e. an infinite loop

Note that the WAIT and DEBUG states are not included in the skeleton's state machine.

III.12.3 The `Shield` class

The `Shield` class contains variables and methods that are related to hardware and the sketch. Two of the methods are defined `static`, thus they must be called with the class name, not the object name.

III.12.3.1 Shield()

The class constructor `Shield()` is empty and takes no arguments.

III.12.3.2 Shield's public variable(s)

The public variable of this class is an integer to hold the node's own address.

Table III.8: Shield()'s public variable(s)

Type	Name, declaration	Description
int	`my_address`	node's own address

III.12.3.3 Shield::begin()

`void begin()` : Initialisation of sketch and shield. Must be called in the beginning of the sketch's `setup` function.

- Input: None
- Returns: Nothing

III.12.3.4 Shield::get_address()

`int get_address()` : Reads the adress Dual In-line Package (DIP) switches and returns the values as one integer.

- Inpute: None
- Returns: Address accordning to DIP switch settings

III.12.3.5 Shield::select_led()

`int select_led()` : Returns the pin number of the selected LEDs on the shield. When called, all three LEDs are lit. You can now press the button. As long as you hold down the button the LEDs will be turned on in a round robin fashion. Release the button when the LED of your choice is lit.

- Input: none
- Returns: pin number of selected LED

III.12.3.6 Shield::adConv()

`int adConv(int value)` : A very simple A/D converter. Takes the integer `value`, which is the sampled IR detector value, and returns a binary value, 1 or 0. The constant AD_TD contains the threshold value used in the conversion.

III.12.3.7 Shield::allLedsOn()

`void allLedsOn()` : Turn all application LEDs on.

III.12.3.8 Shield::allLedsOff()

`void allLedsOff()` : Turn all application LEDs off.

III.12.3.9 Shield::allDebsOff()

`void allDebsOff()` : Turn all debug LEDs off.

III.12.3.10 Shield::halt()

`void halt()` : Infinity empty loop, that effectively finishes execution of a sketch.

- Input: None
- Returns: Nothing

III.12.3.11 Shield::int_to_binarray()

`static void int_to_binarray(int in, int len, byte bin_array[], int start)` : Converts decimal value in `in` to byte array `bin_array` of `len` bit values from `start` cell in the array.

- Input: `in` integer to convert, `len` number of bits to convert to, `bin_array[]` destination array, `start` where in `bin_array[]` to start filling in the bits
- Returns: nothing

III.12.3.12 Shield::binarray_to_int()

`static int binarray_to_int(byte bin_array[], int start, int len)`: Converts a binary value, stored as a `len` bytes in an array from cell `start`, to an integer.

- Input: `bin_array[]` array contaning the binary value, `start` start in `bin_array[]` of binary value, `len` number of cells of binary value
- Returns: integer containing the decimal value

III.12.4 The `Transmitter` class

The `Transmitter` class contains methods and variables for creating a byte array version of an L2 frame. The variables corresponds to the individual fields of the L2 frame.

III.12.4.1 Transmitter()

The class constructor `Transmitter()` is empty and takes no arguments.

III.12.4.2 Transmitter::transmit_frame()

Sending a frame on the physical layer. Takes a Frame as argument.

- Input: `Frame`
- Return: Nothing

III.12.5 The `Receiver` class

The `Receiver` class contains a method and variables for decomposing a Receiverd L2 frame into individual integers corresponding to the L2 frame fields.

III.12.5.1 Receiver()

The class constructor `Receiver()` is empty and takes no arguments.

III.12.5.2 Receiver::receive_frame()

Receive a frame on the physical layer. Takes the time out as argument:

- Input: `int time_out` (ms), the maximum waiting time to receive an `ACK` after a frame is transmitted.
- Return: `RECEIVED` message struct.

III.12.6 The `Frame` class

The `Frame` class private variable:

Table III.9: `Frame` struct private variables

Name	type	Description
`from_address`	`int`	Source address
`to_address`	`int`	Destination address
`type`	`int`	Type of message [ACK | DATA]
`seqnum`	`int`	Sequence number
`payload`	`int`	Data, i.e. application message payload
`crc`	`int`	CRC of frame

III.12.6.1 Frame()

The class constructor `Frame()`. Several constructions can be used.

- `Frame()`, empty construction, takes no arguments, the variables are empty with this construct.
- `Frame(byte frame_array[])`, construct the frame with an array of bits, the bits will be converted to the `int` variables.
- `Frame(Payload payload,int src,int dst,int type,int seqnum,int crc)`, construct the frame with all the arguments and assign the to the variables. The `payload` is an `Payload` object but will be converted to an `int` to assign the `Frame::payload` variable.

- `Frame(Payload payload, int src, int dst, int frame_type)`, construct with only payload, source address, destination address and the frame type, the other variables remain 0 in this construction.

If a `Frame` object needs to be reconstructed, `operator=` is provided in the library for copy assignment, example usage: '

```
// Empty construction
Frame tx_frame;
//Reconstruct (copy assignment)
tx_frame = Frame(payload,src,dst,type,seqnum,crc);
}
```

III.12.6.2 Frame::generate()

Convert all the `int` variables into bits and form an array with all the information, may return error if some variables are empty.

- Input: Nothing.
- Return: An array of `byte[]`

III.12.6.3 Frame::decompose()

Outputs and prints all the `int` frame information with human readable format.

- Input: Nothing.
- : Return: Nothing.

III.12.6.4 Frame::print()

Prints all the `int` frame in the format of an array with 0 and 1.

- Input: Nothing.
- Return: Nothing.

III.12.6.5 Frame::get_dst()

Get the destination address of the frame.

- Input: Nothing.
- Return: `int`, the destination address.

III.12.6.6 Frame::get_payload()

Get the payload message of the frame.

- Input: Nothing.
- Return: `int`, the payload message.

III.12.7 The `Payload` class

Table III.10: *Payload* struct private variables

Variable	type	Description
`led`	`int`	Message content

III.12.7.1 Payload()

The class constructor `Payload()`. Several constructions can be used.

- `Payload()`, empty construction, takes no arguments, the variables are empty with this construct.
- `Payload(int data)`, construct with the application message, assigned to the `Payload::led` variable.

Reconstruction method is similar to `Frame` if needed.

III.12.7.2 Payload::get_payload()

Get the payload message.

- Input: Nothing.
- Return: `int`, the payload message.

III.13 The `RECEIVED` struct

This struct is returned by `Receiver::receive_frame()` method, it consist s of two variables:

- `Frame frame`, the franme converted from the received bits by the `Receiver`, empty if time_out.
- `boolean time_out`, indicating if the receiving process was time out or not.

III.14 Arduino Hardware

Both the *Master Node* and the *Development Node* are constructed using an Arduino board and micro-controller [6], 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 [3]. 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 III.15 you will get an overview of the Arduino board, followed by an introduction to the shield in Section III.16. A brief introduction to the software is given in Section III.9.

III.15 Board

The Arduino micro-controller is fitted onto a small board with a set of digital and analogue I/O pins, see Table III.11. These pins can easily be manipulated and read from 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 III.5

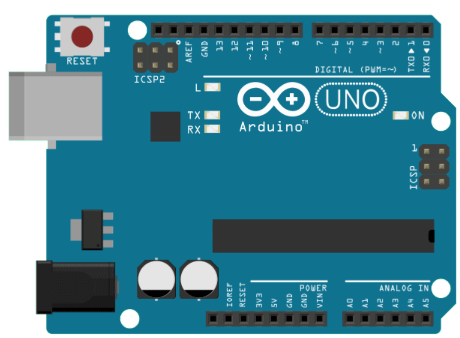

Figure III.5: Arduino UNO board

Table III.11: 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

III.16 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 III.12. The shield is laid out as Figure III.6.

Table III.12: Pin assignments

Assignment	Pin number	Type
Rx diode	0	Analogue
TX diode	13	Digital
Button	2	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

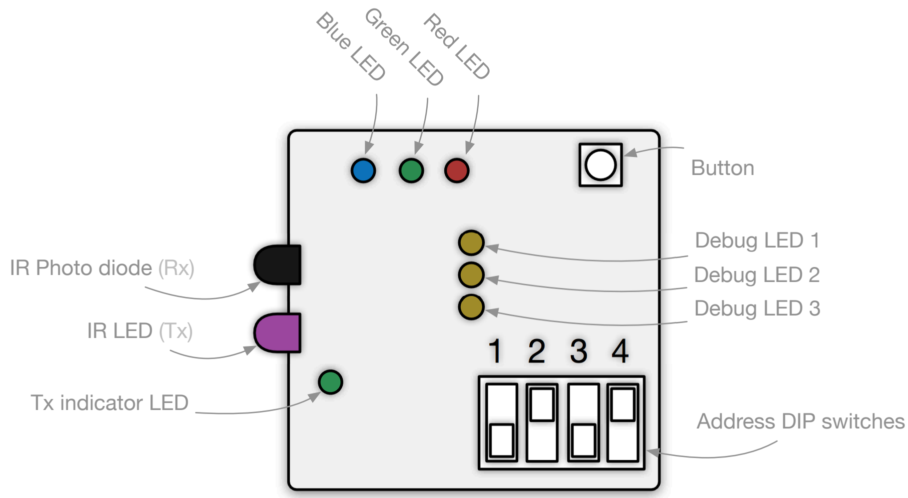

Figure III.6: Shield layout

III.16.1 Communication

The communication circuit provides the board with a set of Rx and Tx IR diodes. The Tx diode is complimented with a red LED to provide visual feedback whether the node is Transmitterting.

To be able to assign the node an address, the communication circuit is also equipped with a four-toggle dip-switch, see Figure III.7. The most significant bit is set using the left-hand-side switch, DIP Switch 1 which is connected to PIN 6.

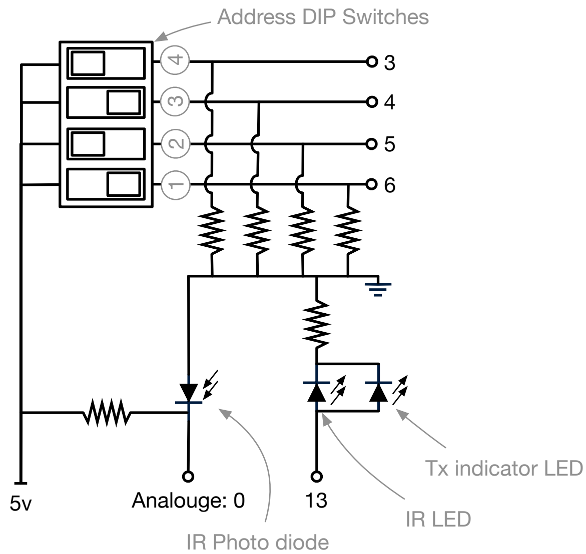

Figure III.7: Communication circuitry

III.16.2 Application

The application circuit consists of three differently coloured LEDs and a button, see Figure III.8.

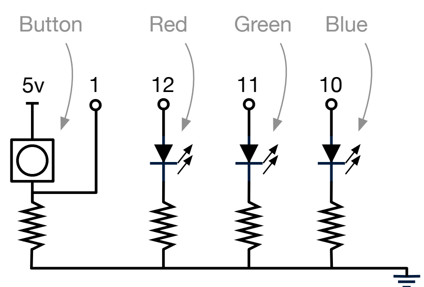

Figure III.8: Application circuitry

III.16.3 Service and debug

In addition to the debug messages sent to the Arduino IDE *Serial Monitor*, the shield has been equipped with the three user customisable LEDs accessible on pins $7, 8, 9$, labelled D3, D4 and D5 on the circuit board. Additionally, as previously mentioned, the Tx LED will light when the Tx diode is activated.

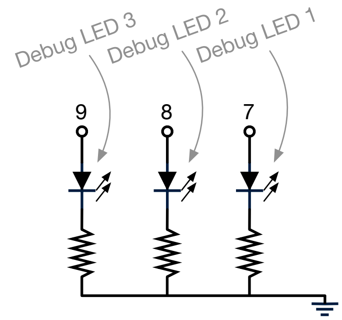

Figure III.9: Service and debug circuitry

Part IV

The 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 for receiving data and acting upon that data. The documentation below details how the node's functionality has been implemented and how you can expect it to behave.

IV.16.4 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 it cannot work in parallel for both receiving data and Transmitterting data. In Figure IV.10 the behaviour for receiving data and replying with an ACK is shown.

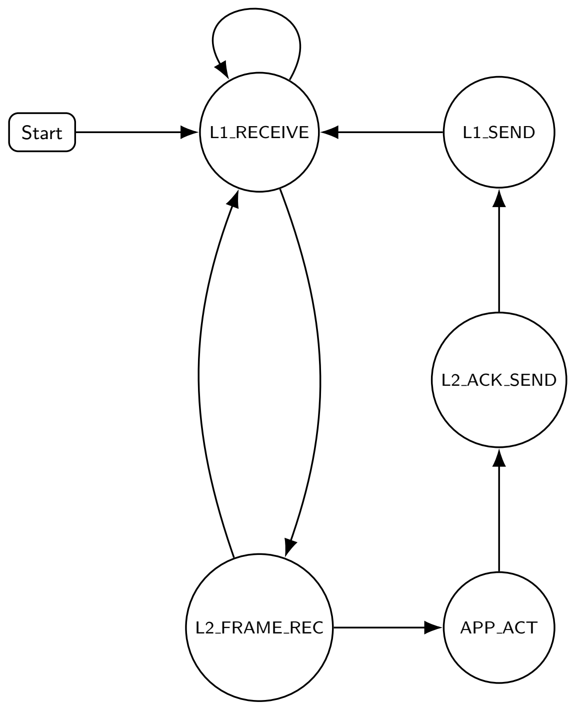

Figure IV.10: *Master Node* states

IV.16.5 L1_PHY_RECEIVE

The L1 Receiver state, depicted in Figure IV.11, starts with continuously reading the input source. This continues until the preamble has been detected or the process has timed out. When the preamble has been found, sampling of the Receiverd symbols starts. These symbols are first stored in a byte buffer to detect the SFD. Once the SFD has been found, the sampled symbols can be stored in the Receiver buffer, which is here used as the interface between L1 and L2.

If no preamble or SFD has been detected within a time-out, the execution exits this state and the sketch's `loop()` funtion gets control.

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

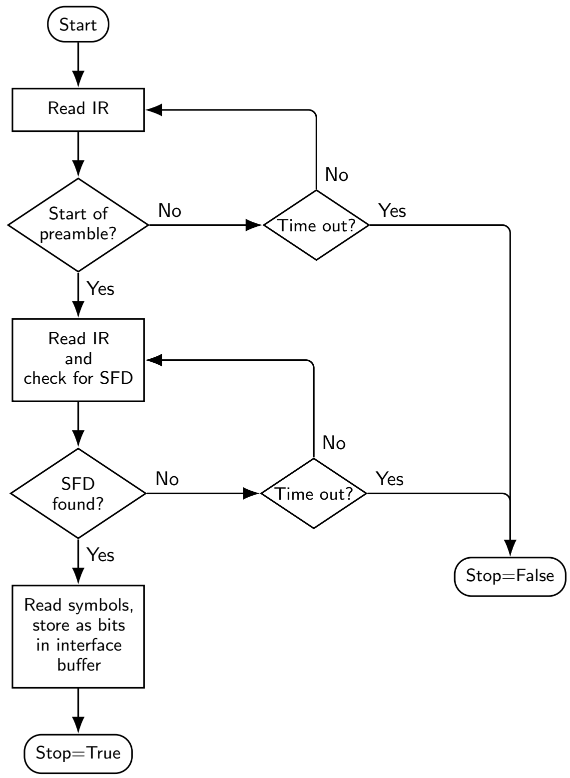

Figure IV.11: Flow chart of the L1 Receiver state.

As soon as the Receiver detects signals the debug LED #1 starts to flash, following the Receiverd symbol. When the *Master Node* detects an SFD it will show a fixed light on debug LED #1, and the debug LED #2 will flash following the Receiverd symbol. Once all symbols of the frame are Receiverd Debug LED #2 will go to fixed light.

IV.16.6 L2_LINK_FRAME_DECOMPOSE

The L2_LINK_FRAME_DECOMPOSE state is depicted in Figure IV.12. The Receiverd frame is decomposed into the frame field integers. A successful outcome of a conditional CRC validation will light debug LED #3 on the *Master Node*'s shield. Now other conditions can be applied to for example check the correct recipient and follow the type of frame.

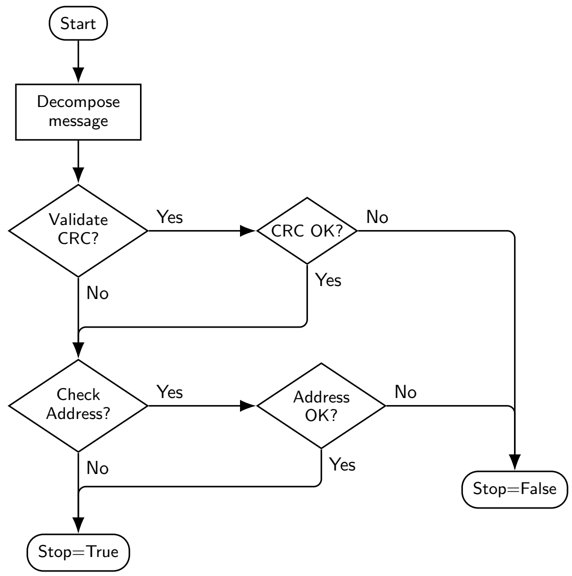

Figure IV.12: L2 frame Receiver state

IV.16.7 L7_APP_ACT

The message that was decoded in the L2_LINK_FRAME_DECOMPOSE state is acted upon in this state. If for example, the Receiverd message instructed the node to turn on the blue LED this state will carry out that action.

IV.16.8 L1_PHY_SEND

This state sends the content of the `Transmitter::frame[]`, preceeded by a preamble and SFD. The state is set to `L1_PHY_Receiver` and the major loop continues.

IV.16.9 Controlling the *Master Node*

The four DIP switches have an extended functionality on the *Master Node*. The objective has been changed from mere addressing to control of different functions, see Table IV.13.

The two DIP switches #1 and #2 are used to both activating addressing and to set the address of the *Master Node*. If both switches are set to off, frame addresses are not relevant. If one of the switches is set to on, addressing is active and the address of the *Master Node* is determined by the switches, i.e. 1, 2 or 3.

DIP switch #3 controls the sequence number in the returned ACK. If set to off, the *Master Node* ACKs with the sequence number of the Receiverd frame. If set to on, the sequence number is decremented by one before stored in the ACK frame. This allows for test of the *Development Node*'s ARQ.

DIP switch #4 controls whether CRC functionality should be active (DIP switch set to on) or inactive (DIP switch set to off).

Table IV.13: *Master Node*DIP switch functions

DIP switch	**State**	**Function**
1 & 2	off, off	Addressing not active
1 & 2	off, on	Addressing active, address = 1
1 & 2	on, off	Addressing active, address = 2
1 & 2	on, on	Addressing active, address = 3
3	off	normal sequence number handling
3	on	ACKed sequence number = Receiverd sequence number - 1
4	off	CRC inactive
4	on	CRC active

References

[1] https://www.arduino.cc/en/reference/millis.

[2] Alohanet, pure aloha. https://en.wikipedia.org/wiki/ALOHAnet#Pure_ALOHA.

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

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

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

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

[7] James Kurose and Keith Ross. *Computer Netwroking, A Top Down Approach*. Pearson, 7th edition, 2017.