Arduino labs

Version 24.0 Build 4

Electrical and Information Technology

April 8, 2025

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Table of Contents

Part 1 Introduction to the reference manual **1**

Part 2 Protocol and communication specifications **2**

1 Interfaces between layers . . . **3**

1.1 Interface between physical and data link layer . . . 4

1.2 Interface between data link and application layer . . . 4

2 Physical layer . . . **4**

2.1 Collision avoidance . . . 5

2.2 Preamble . . . 5

2.3 Start frame delimiter (SFD) . . . 5

3 Data link layer . . . **6**

3.1 The frame structure . . . 6

3.2 Addressing . . . 6

3.3 Type of message: DATA and ACK frames . . . 7

3.4 Sequence numbers . . . 7

3.5 Parity bits and cyclic-redundancy-check (CRC) . . . 7

3.6 Automatic Repeat Request (ARQ) . . . 8

4 Application layer . . . **9**

Part 3 Arduino hardware and software **11**

1 Arduino hardware . . . **11**

2 Arduino software . . . **12**

2.1 Data types . . . 13

2.1.1 Arrays . . . 13

2.2 Bitwise operations . . . 14

2.2.1 How to read and write bits from bytes . . . 14

2.3 Logical operators . . . 15

2.4 The ternary (conditional) operator . . . 15

2.5 Read the on board clock: millis() . . . 15

3 Shield board . . . **16**

3.1 Communication circuits . . . 17

3.2 Application circuits . . . 18

3.3 Debug and service LEDs . . . 18

4 Debugging tools . . . **19**

4.1 Serial monitor . . . 19

4.2 Debug LEDs . . . 19

Part 4 The Nodes **20**

1 *Development Node* . . . **20**

1.1 Sketch skeleton . . . 21

1.2 Variable declarations . . . 21

1.3 The `setup` function . . . 21
1.4 The `loop` function . . . 22
1.5 Area for functions . . . 22
2 *Master Node* . . . **22**
2.1 States in the state machine . . . 22
2.2 Controlling the *Master Node* . . . 26
3 *Access Point* . . . **26**
Part 5 Libraries **29**
1 Predefined values and global constants . . . **29**
2 The `Shield` class . . . **31**
2.1 Shield's constructor . . . 31
2.2 Shield's public variables . . . 31
2.3 Shield's public methods . . . 32
3 The `Frame` class . . . **34**
3.1 Frame's constructor . . . 34
3.2 Frame's public variables . . . 34
3.3 Frame's public methods . . . 34
4 The `Transmit` class . . . **34**
4.1 Transmit's constructor . . . 34
4.2 Transmit's public variables . . . 35
4.3 Transmit's public methods . . . 35
5 The `Receive` class . . . **35**
5.1 Receive's constructor . . . 35
5.2 Receive's public variables . . . 35
5.3 Receive's public methods . . . 36
Part A Physical Lab Environment **37**
1 Physical layer labs . . . **37**
2 Data link layer labs . . . **38**
Part B Skeleton.ino **39**

List of Figures

1 The lab three layered reference model. 2
2 The link frame structure as seen on the physical and the data link layer. . . 3
3 The preamble as a signal. 4
4 A CRC decoder 8
5 Example communication scenario *(Not to proportion)* 9
1 Arduino UNO board 11
2 Shield layout 16
3 Radiation characteristics for the IR LED, the transmitter. 17
4 Radiation characteristics for Photo diod, the receiver. 17
5 Communication circuitry 18
6 Application circuitry 18
7 Service and debug circuitry 19
1 A simplified transition graph for the complete *Development Node*. 20
2 *Master Node* states 23
3 Flow chart of the physical layer receive state. 24
4 Link layer frame receive state 25
5 *Access Point* flow diagram. 27
1 The setup of the network. 38

List of Tables

1 Train of symbols on the physical layer 5
2 Data link layer *frame* structure. 6
3 Application layer *message* structure. 10
1 Arduino hardware specifications 12
2 Ardunio data types 13
3 Pin assignments 16
1 Global variables defined in the skeleton 21
2 *Master Node*DIP switch functions 26
3 *Access Point* CRC validation 28
1 Constants addressing the shield's pins. 29
2 Predefined states constants 30
3 Definition of constant values and variables for the *physical* layer. 30
4 Definition of constant values and variables for the *data link* layer. 31
5 Definition of constant values and variables for the *application* layer. 31
6 Frame class' public variables 34
7 Transmit's public variables 35
8 Receive's public variables 35

Part 1
Introduction to the reference manual

This reference manual is to be used for the Arduino labs, built on Arduino boards and a specially designed shield. The objective of the reference manual is to support the lab manuals with the theoretical and practical background needed to successfully complete the labs.

The reference manual consists of five parts and three appendices. The parts and appendices are presented shortly below.

- **Part 1** is the introduction, which you are reading now.
- **Part 2** defines the protocol and communication specifications with related background theory. The layers used in these labs and the interfaces used to communicate between the layers are described.
- **Part 3** presents the Arduino hardware and software. The needed programming syntax is outlined in the software section. The shield board used in the labs as well as the debugging tools available are also described.
- **Part 4** presents the three nodes: the *Development Node*, the *Master Node* and the *Access Point*. While the first one is for you to implement, the workings of the two latter are described here. The sketch skeleton for you to use is also outlined.
- **Part 5** describes the library provided for you. This includes the predefined values and global constants as well as descriptions of the different classes with related constructors, public variables and public methods.
- **Part A** describes the physical lab environment.
- **??** describes the files and folder structures.
- **Part B** is a printing of the skeleton used in the labs.

Part 2

Protocol and communication specifications

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

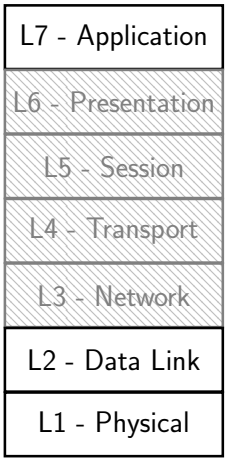


Figure 1: The lab three layered reference model.

The layers as well as the interfaces between the layers are described in the following sections. The naming convention follows [9] regarding encapsulation:

- A *message* is passed from and to the application layer.
- On the Data Link Layer (L2) layer a *frame* is defined.

The link between the physical layer (L1) pulses and the data link layer (L2) frame are as follows. The Physical Layer (L1) symbol train, as seen in Figure 2, consists of three parts: the preamble, the Start Frame Delimiter (SFD) and payload. The payload contains the L2 frame with its header, payload and tail, corresponding 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.

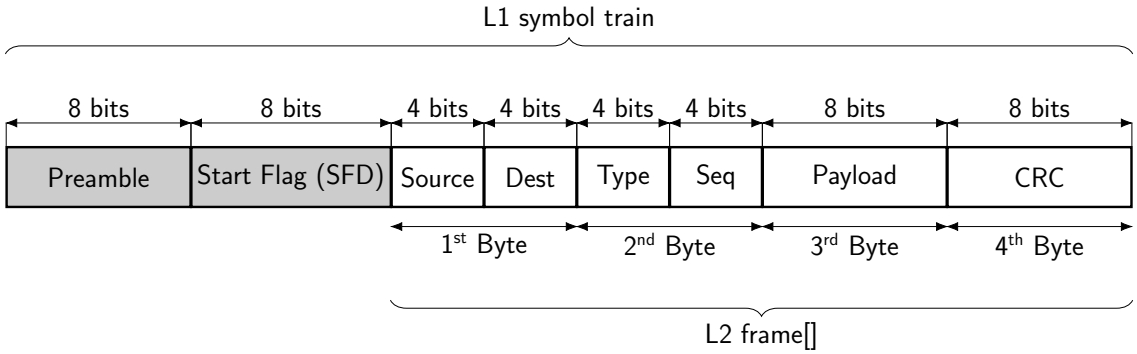


Figure 2: The link frame structure as seen on the physical and the data link layer.

On L1, information is transmitted as pulses of light or the absence of light. Since On-Off keying is used, the pulses are all of the same duration, that is the sample time T_s, which in our case is 100 ms. The bytes and integers forming the L2 frame has to be converted to something bit oriented, that is the L1 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 to send all the symbols as pulses. In our case the symbols are the same as the L2 frame bit values. Once the symbols corresponding to the preamble, the SFD and the L2 frame are at hand, we can send the symbol train by reading out pulse representations of the preamble, the SFD and the data transmit buffer (i.e. the L2 frame), symbol by symbol, with an interval of T_s, and let them control the sending device, in our case the Infra-red (IR) transmitter. Remember that the preamble and the SFD belongs to the Physical Layer (L1), while the data *frame* belongs to the Data Link Layer (L2). The *frame* is also defined as the interface between the two layers.

The receiving of signals or pulses works more or less the same way, but backwards. Once the preamble has triggered the receiver, the receiver can synchronise the sampling of pulses to the received pulse train. This is the objective of the preamble, namely synchronisation. Once the receiver is synchronised, light pulses can be sampled each T_s. The sampled values, the symbols, corresponds to and will be represented by a 1 or a 0, which will be stored in a receive buffer. Once the SFD has been identified, the beginning of the symbols corresponding to the L2 frame is located. The sampled symbols can now be stored directly in the data receive buffer, i.e. the interface between L1 and L2, which is sent to L2 where the decomposition to the L2 frame variables can take place.

1 Interfaces between layers

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

1.1 Interface between physical and data link layer

The interface between the layers L2 and L1 is a buffer containing the bits forming the L2 frame. The bit buffer is 32 bits long. It is suggested that this interface is implemented as a 32 bit variable, an `unsigned long`.

1.2 Interface between data link and application layer

The interface between the application and the Data Link Layer (L2) equals the message format, see Table 3. The address field should contain the destination's address when sending the message from the application to L2, and the source's address when the application receives a message from L2. The payload field should contain the selected user Light Emitting Diode (LED) when sending the message from the application to L2, and a user LED to turn on when the application receives a message from L2. It is suggested to implement this interface as an array.

2 Physical layer

The communication link is physically achieved by a half duplex IR ($\lambda = 900nm$) 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 L1, one symbol 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 3.

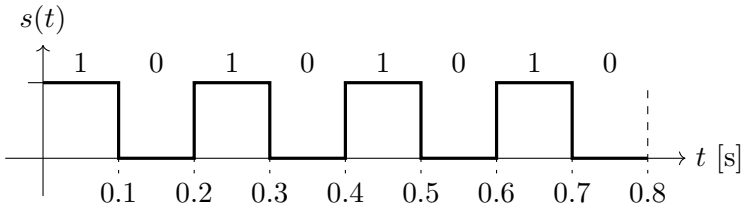


Figure 3: The preamble as a signal.

The symbols, that corresponds to the L2 frame bits, must be preceded by a preamble of eight symbols followed by an SFD of eight symbols. The full L1 symbol train can be seen in Table 1

Table 1: Train of symbols on the physical layer

Field	Length (bits)	Description
Preamble	8	0b10101010, for synchronisation of receiver to transmitted pulses
SFD	8	0b01111110, marks start of a frame
Data	32	The Data Link Layer (L2) frame

2.1 Collision avoidance

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

2.2 Preamble

The first part of the L1 symbol train is the eight bit long 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 from no light to light, that is a high pulse, is this positive flank. The other objective of the preamble is to allow the receiver's sampler to synchronise to the transmitter's clocking. If the transmitter shifts between light and no light in a well-known fashion, the receiver can adjust the timing of the sampler so that it synchronises with the transmitter's clock. The preamble has the bit pattern `10101010`, which solves both objectives. In our case, we only use the leading positive flank of the preamble to trigger the receiver. The synchronisation of the clocks can be omitted, because the micro-controllers' clocks are stable enough when compared to the pulse time T_s and the frame length.

2.3 Start frame delimiter (SFD)

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 received symbols with the known 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. Following symbols can be translated to bits and stored in the received data interface buffer.

3 Data link layer

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

3.1 The frame structure

The L2 frame format is shown in Table 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 2: Data link layer *frame* structure.

Field	**Length (bits)**	**Description**
From	4	Source address
To	4	Destination address
Type	4	Type of message [ACK ($=0001_2$) | DATA ($=0010_2$)]
SEQ	4	Sequence number
Payload	8	Data, that is application message payload
CRC	8	CRC of frame

See Table 6 for corresponding variable names.

3.2 Addressing

Each node has a four-bit address, that is an address space of 16. A node should only process received messages addressed to itself. In a Peer-to-Peer setup as in the first two labs, addressing is not used, and these fields should be set to zero. The address of a node is set differently depending on node type:

- The *Development Node*'s address should be coded in the *Development Node*'s sketch. The *Development Node*'s Dual In-line Package (DIP) switches is used for setting the destination of the frames, that is in most cases the address of the *Master Node*.
- Only two of the *Master Node*'s four DIP switches are used for addressing, see Section 2.2 for details.
- The *Access Point* does not have an address. The DIP switches are only used for ARQ control, see Section 3.

Note that addressing is not used in the first two labs.

3.3 Type of message: DATA and ACK frames

A DATA frame carries application data, which is stored in the 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 received DATA frame. It is only sent once, and carries an empty payload. The SEQ number field contains the sequence number of the acknowledged DATA frame. An ACK frame is denoted by a 0001_2 in the Type field.

3.4 Sequence numbers

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

3.5 Parity bits and cyclic-redundancy-check (CRC)

The objective of adding parity bits to a frame is to make sure that the bits of a frame are, to a certain possibility, correctly received. There are several methods defined for this. In this lab, we will use CRC-8 Bluetooth [3] using a 9-bit divisor which is defined by the CRC generator polynomial $x^8 + x^7 + x^5 + x^2 + x^1 + 1$ and by the hexadecimal number 0xA7.

The generation of the parity bits, the CRC value, is calculated over the L2 frame, and the value is added to the CRC field at the end of the frame before transmission.

The CRC validation on received frames should in this lab be done after the L1_RECEIVE state has finished, meaning when an L2 frame is defined. The code should mimic a shift register with feedback loops, see Figure 4.

1. Create a register with number of bits equal to the grade of the generator. Set the register to 0.

2. For each received bit, shift the register left one step and shift in the received bit as the register's Least Significant Bit (LSB).

3. If the bit shifted out of the register, i.e. the register's Most Significant Bit (MSB) before the left shift, is =1, XOR the bit representation of the generator with the register. If the shifted out bit is 0, do nothing. Note that both the generator's MSB and the bit shifted out are equal to one and thus can be omitted in the XOR.

4. Repeat until all bits of the frame has been received. If the register is equal to 0, the validation was successful and the frame was received correctly.

The generation of CRC is done in a corresponding way with the difference that what is left after step 4 is the CRC to be added last in the frame.

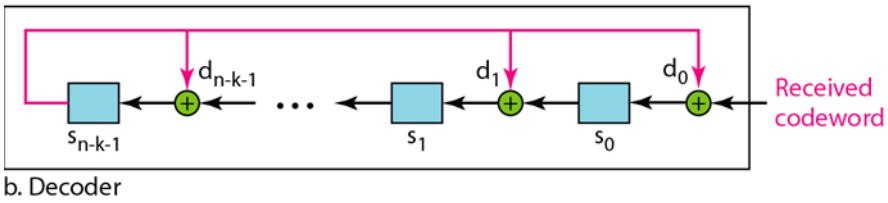
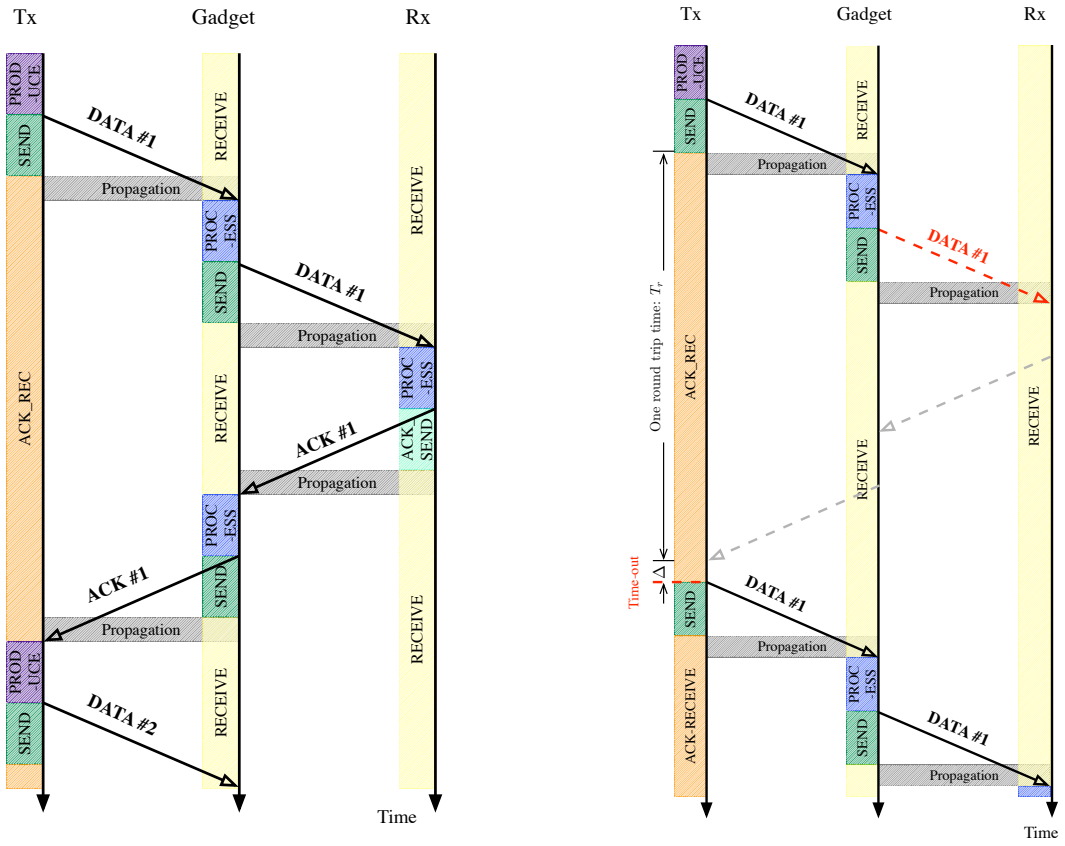


Figure 4: A CRC decoder

3.6 Automatic Repeat Request (ARQ)

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 or collision due to simultaneous transmission.

Given these circumstances, to achieve a rudimentary degree of reliability, the nodes employ a Stop-and-wait ARQ scheme. The sender of a DATA frame should retransmit that frame, persisting the sequence number, if it does not receive an ACK frame with the same sequence number from the recipient within a certain time. The number of re-transmissions must of course be limited. Similarly, if retrieved successfully and correctly addressed, the recipient of a DATA frame should transmit an ACK to the sender pertaining the same sequence number, see Figure 5



(a) Normal procedure

(b) Data retransmission

Figure 5: Example communication scenario *(Not to proportion)*

As previously said, the ARQ method used is Stop-and-Wait. Only DATA frames are to be acknowledged. If no acknowledgement is received within 20 seconds, the DATA frame should be retransmitted. If an acknowledgement with source address equal to the DATA frame destination address is received but contains an invalid sequence number the DATA frame should be retransmitted. If a DATA frame is retransmitted more than three (3) times, the corresponding message should be considered undeliverable.

4 Application layer

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 3. For simplicity, the two application message fields are implemented as integer arrays in the library.

Table 3: Application layer *message* structure.

Field	**Length (bits)**	**Description**
Address	4	Destination or source address
Payload	8	Message content

Part 3
Arduino hardware and software

This part contains reference information on the Arduino software and hardware. The special *shield* adapted for the lab is also documented here as well as the available tools for debugging.

1 Arduino hardware

The *Master Node*, the *Development Node* and the *Access Point*are all constructed using an Arduino board and micro-controller [8], 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 [5]. Both the Arduino board and the development environment are open source. In this lab you will not modify the hardware but focus on implementing the desired functionality in software.

The Arduino micro-controller is fitted onto a small board (UNO) with a set of digital and analogue Input/Output (I/O) pins, see Table 1. 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 1

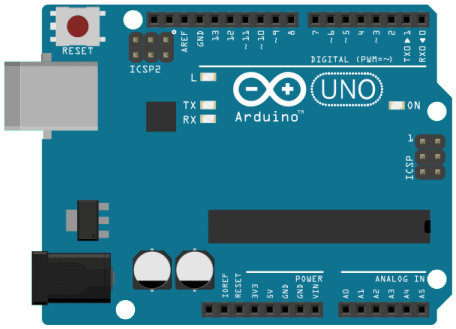


Figure 1: Arduino UNO board

Table 1: Arduino hardware 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

2 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) [6].

You write your program in a sketch, which has two primary functions, `setup()` and `loop()` [5]. The `setup()` function is where you declare how you want the I/O to behave and initialise your global variables, see Listing 3.1. The code contained inside the `loop()` is looped during runtime. You can declare your own functions, variables, and constants outside of these two functions. Please consult the Arduino beginners guide [7] (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 3.1: Sample Arduino code to Transmit (Tx) and Receive (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 transmitter
  delay(100);                 // wait for 100 ms
  digitalWrite(PIN_TX, LOW);  // turn off the IR transmitter

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

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

2.1 Data types

Table 2 lists some data types that might be useful in this lab. Using proper types for different variables helps to save memory and accelerate the process. For example, we claim variable `SFD` as type `byte` since the SFD contains 8 bits.

Table 2: 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

2.1.1 Arrays

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]`.[1] 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.

[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`.

So be aware of your index pointers. Apply the modulus operator `%` with an appropriate constant to the index pointer.

The C++ library string.h contains some useful functions for handling arrays.

2.2 Bitwise 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. The bitwise operators works independently on the bits in two variables.

Bitwise AND
The AND operator is `&`.

Bitwise OR
The OR operator is `|`.

Bitwise XOR
The XOR operator is `^`. XOR of two bits returns 0 if they are the same, otherwise 1 is returned. XOR of two bytes returns 0 if they are the same, otherwise a non-zero value is returned.

2.2.1 How to read and write bits from bytes

Vital bit operations are the setting and resetting as well as reading of individual bits in a variable. The Arduino programming language has not defined any native bit operations. Instead, bit operations are functions which allows you to address individual bits. These functions are slow. Instead you can use logical AND (`&`), logical OR (|) and shift operations to set or read individual bits.

Left shift `<<` and right shift `>>` are used to move bits a defined number of steps left or right in a variable. 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 3.2 shows an example to read the third bit from the right-hand side of a byte.

Listing 3.2: 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 LSB position, and `& 0x01` zeros all the bits other than the LSB.

To write bits, e.g. to the lower part of a byte, use `<<` to move the bits to the right position and then use logical OR (`|`) to add the bits in. See the example below, Listing 3.3, to save `parameter_1` to the higher 4 bits of the `frame` and `parameter_2` to the lower 4 bits.

Listing 3.3: 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.

2.3 Logical operators

Logical operators are used in Boolean algebra.

Logical AND
The logical AND operator is `&&`. The result of a logical AND is true if both boolean variables are true.

Logical OR
The logical OR operator is `||`. The result of a logical OR is true if one of the boolean variables are true.

Logical NOT The logical NOT operator is `!`.

2.4 The ternary (conditional) operator

`condition ? <if true> : <if false>` : The ternary, or conditional, operator returns different values depending on if the condition is true or false. Instead of writing if-else statements, you can assign a value to a variable depending on a condition. Compared to the bit functions, the ternary operator is ten times faster. In the following example the variable `largest` is assigned the greater value of a and b: `largest = (a>b) ? a : b;`.

2.5 Read the on board clock: millis()

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

3 Shield board

The physical so called shield attaches to the board and supplies the communication, interaction, and service/debugging functionality. The UNO board's pins have been assigned according to Table 3. The shield is laid out as Figure 2.

Table 3: Pin assignments

Assignment	**Pin number**	**Type**
Receive diode	0	Analogue
Transmit LED	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
Blue user LED	10	Digital
Green user LED	11	Digital
Red user LED	12	Digital

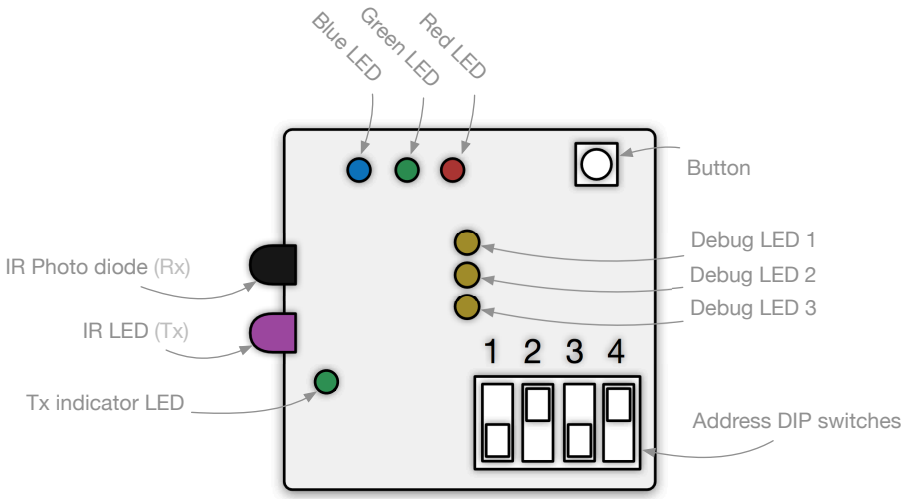


Figure 2: Shield layout

3.1 Communication circuits

The communication circuit provides the board with one IR receive photo diode and one IR transmit LED. The transmit LED is complimented with a red LED to provide visual feedback whether the node is transmitting. The radiation characteristics for the IR transmitter is found in Figure 3 and for the photo diode in Figure 4.

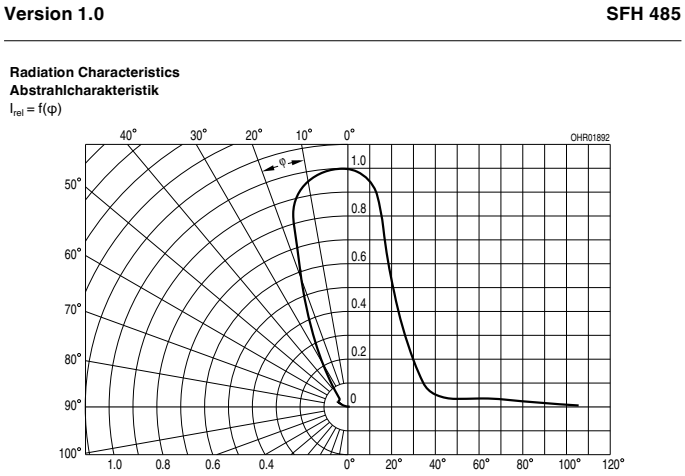


Figure 3: Radiation characteristics for the IR LED, the transmitter.

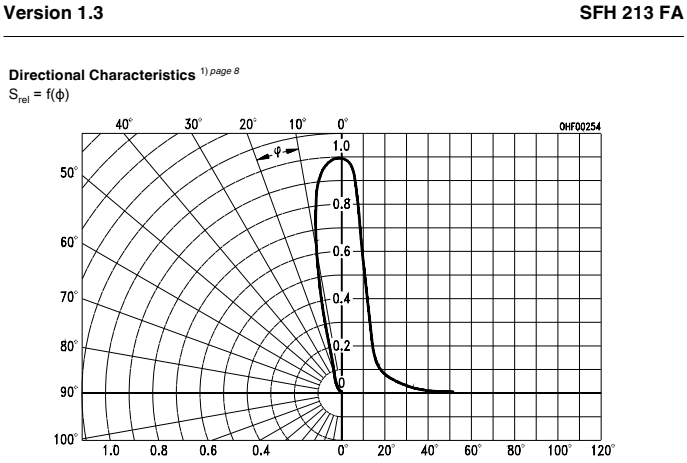


Figure 4: Radiation characteristics for Photo diod, the receiver.

To be able to assign the node an address, the communication circuit is also equipped with a four-toggle dip-switch, see Figure 5. The most significant bit is set using the left-hand-side switch, DIP Switch 1 which is connected to PIN 6.

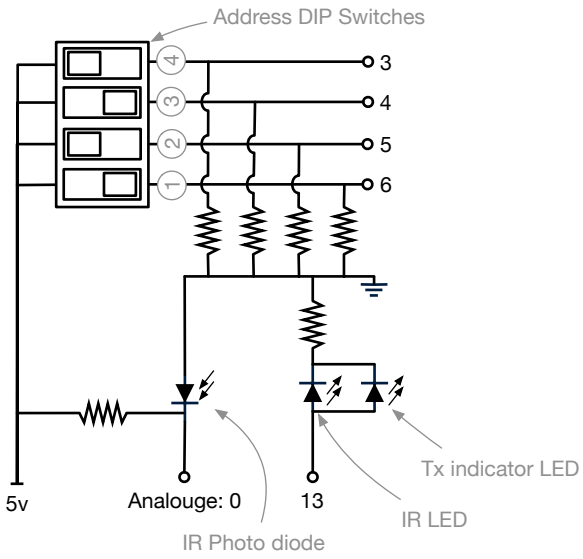


Figure 5: Communication circuitry

3.2 Application circuits

The application circuits consists of three differently coloured user LEDs and a button, see Figure 6.

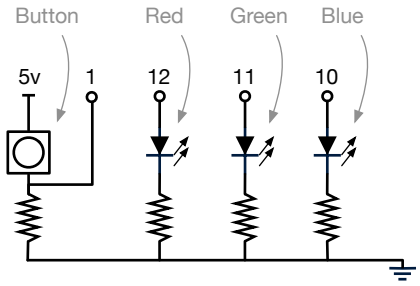


Figure 6: Application circuitry

3.3 Debug and service LEDs

In addition to the debug messages sent to the Arduino IDE *Serial Monitor*, the shield has been equipped with the three user customisable LEDs, the so called debug LEDs, accessible on pins $7, 8, 9$, labelled D3, D4 and D5 on the circuit board. The transmit indicator LED will light when the IR transmit LED is activated.

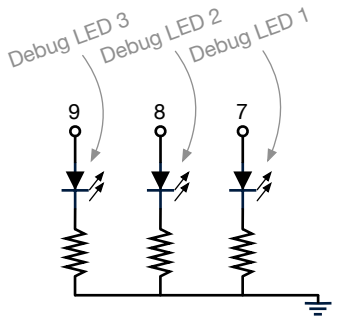


Figure 7: Service and debug circuitry

4 Debugging tools

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

4.1 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 these two labs 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.

4.2 Debug LEDs

There are three debug LEDs on the shield to your disposal. The pins associated with each LED are found in Table 3. Writing a HIGH to a pin turns the corresponding LED on; turn it of by writing a LOW.

Part 4
The Nodes

In this part, the three nodes - the *Development Node*, the *Master Node* and the *Access Point* - are described. During all labs, you will have access to one *Master Node* and one *Development Node* and for the third and forth lab one additional *Access Point*.

1 *Development Node*

The *Development Node*'s hardware is identical to the *Master Node*'s and the *Access Point*'s, but the *Development Node* will not come with a complete software sketch. It is your task to implement the *Development Node* according to the lab manuals. A state machine is provided in the `void loop()` function, which is yours to develop. When developing the node you can seek help from the *Master Node* specifications in Section 2 and the supplied software skeleton, as later described.

After the final task of the last lab, the *Development Node* should be able to handle addressing, ARQ - in this case Stop-And-Wait - and CRC functionality. A simplified state transition graph of the fully functional *Development Node* is shown in Figure 1. Note that the box marked `ERROR` includes e.g. when the maximum number of retransmission is achieved. Also, the SEND state in the figure includes the sketch states L2_DATA_SEND and L1_SEND, while the ACK REC state includes the sketch states L1_RECEIVE, L2_FRAME_REC and L2_ACK_REC.

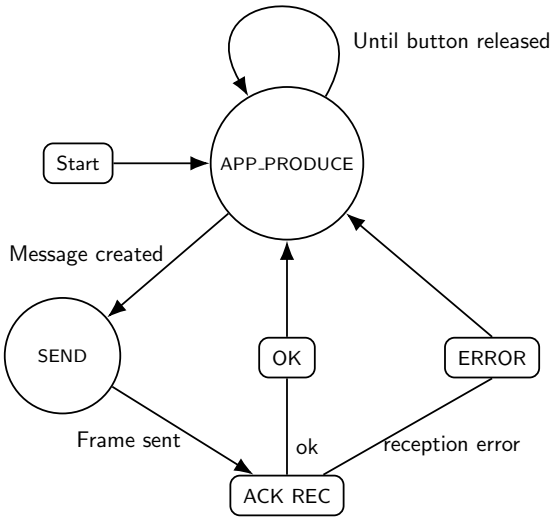


Figure 1: A simplified transition graph for the complete *Development Node*.

1.1 Sketch skeleton

For your help, a sketch skeleton and a supporting library has been devised. For the latter, see Part 5. The skeleton - see Part B - is the base for your sketch. It defines the Arduino sketch functions `setup()` and `loop()`. The state machine is the major part of the `loop()` function, and it is here that you add your code. At the end of the code there is a placeholder for your functions. Also, a few constants and variables are already defined. The skeleton begins with the library include statement: `#include <datacommlib.h>`. The skeleton is further divided into four major areas that will be briefly described here. These areas are:

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

1.2 Variable declarations

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

Table 1: Global variables defined in the skeleton

Type	**Name, declaration**	**Description**
int	`state = NONE`	state machine control variable
Shield	`sh`	Declaration of object, instance of class Shield
Transmit	`tx`	Declaration of object, instance of class Transmit
Receive	`rx`	Declaration of object, instance of class Receive

Note that the *constructors* of the classes, `Shield`, `Transmit` and `Receive` are called without parenthesis. This is true for constructors that takes no arguments, which is the case here.

1.3 The `setup` function

Initialisation of the hardware at hand and the software is done in the sketch's `setup` function. The Shield class' method `begin()`, which performs this task, is thus called from here.

1.4 The `loop` function

The sketch's `loop` function holds the main part of the sketch, which is the state machine. Each state has its own code area, and must be finished of with a `break` statement. See the skeleton code listing in Appendix B for more details.

1.5 Area for functions

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.

2 *Master Node*

The *Master Node* consists of an Arduino and a shield board. Its hardware 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.

2.1 States in the state machine

The *Master Node* has been implemented with the states as 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 can not work in parallel for both receiving data and transmitting data. In Figure 2 the behaviour for receiving data and replying with an ACK is shown.

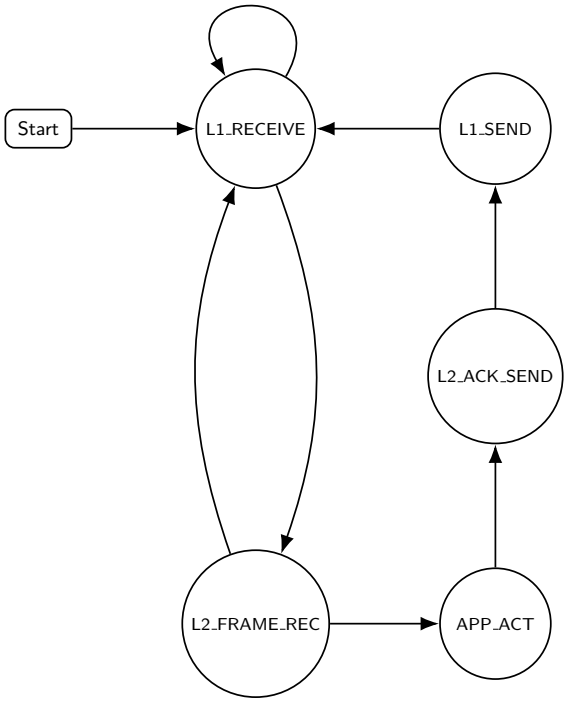


Figure 2: *Master Node* states

L1_RECEIVE

The L1 receive state, as depicted in Figure 3, starts with continuously reading the input source. This continues until the preamble has been detected or the process has timed out. During this state, the debug LEDs 1 and 3 are lit. When the preamble has been found, these two debug LEDs are unlit and sampling of the received symbols starts. The sampled symbols are first stored in a byte buffer to detect the SFD. Once the SFD has been found, the sampled symbols are stored in a receive buffer, in our case this is the same as the L2 frame, that is `Receive::frame[]`, 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()` function gets control.

Because execution is sequential, sampling and SFD detection is done repeatedly. For each sample the SFD detection mechanism is performed. To keep the symbols synchronised a delay of $T_s - \hat{T}_c$ is added between each sample. $\hat{T}_c$ is the processing time for the SFD detection that is performed for each sample.

As soon as the receiver detects signals the debug LED 1 starts to flash, following the received symbols. 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 received symbols. Once all symbols of the frame are received debug LED 2 will go to a fixed light.

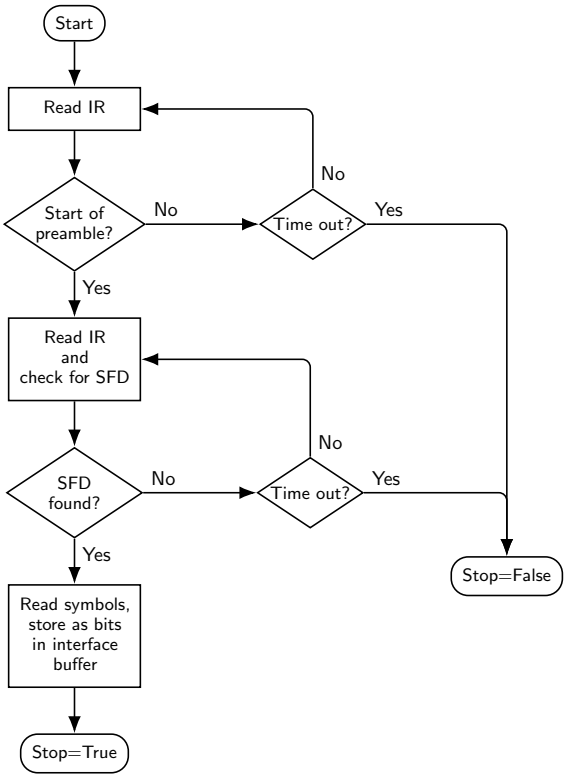


Figure 3: Flow chart of the physical layer receive state.

L2_FRAME_REC

The L2_FRAME_REC state is depicted in Figure 4. The received frame in the `Receive::frame[]` buffer 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, such as checking the address and follow the type of frame.

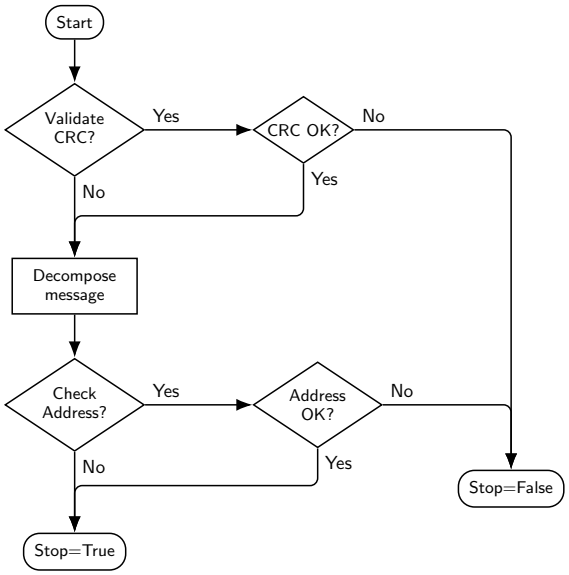


Figure 4: Link layer frame receive state

APP_ACT
The message that was decoded in the L2_FRAME_REC state is acted upon in this state. If, for example, the received message instructed the node to turn on the blue LED this state will carry out that action.

L2_ACK_SEND
In this state, all the L2 field variables are set to correct values:

- `Transmit::frame_to = Receive::frame_from`
- `Transmit::frame_from = sh.my_address`
- `Transmit::frame_type = FRAME_TYPE_ACK`
- `Transmit::frame_seqnum = Receive::frame_seqnum`
- `Transmit::frame_payload = 0`

If CRC is active, `Transmit::frame_crc` is calculated and stored. If not, this field is set to zero. The `Transmit::frame[]` is then filled, and the process continues with the L1_SEND state.

L1_SEND
This state sends the content of the preamble, SFD and `Transmit::frame[]`. The state is set to `L1_RECEIVE` and the major loop continues.

2.2 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 2.

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, the addresses of the frames 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, that is 1, 2 or 3.

DIP switch 3 controls the sequence number of the returned ACK. If set to off, the *Master Node* ACKs with the sequence number of the received 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 functionality.

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

Table 2: *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 = received sequence number - 1
4	off	CRC inactive
4	on	CRC active

3 *Access Point*

The *Development Node* and the *Master Node* can only communicate with each other if the nodes are facing each other. If this is not the case they have to communicate via the *Access Point*. The documentation below briefly describes how the *Access Point*'s functionality has been implemented and how you can expect it to behave.

The *Access Point* works on the Data Link Layer (L2), in a Store-and-Forward fashion, which means that it receives a full frame before it repeats the frame on the link again. The *Access Point* works in promiscous mode. This means that it has no address and accepts all frames independent of a frame's destination address.

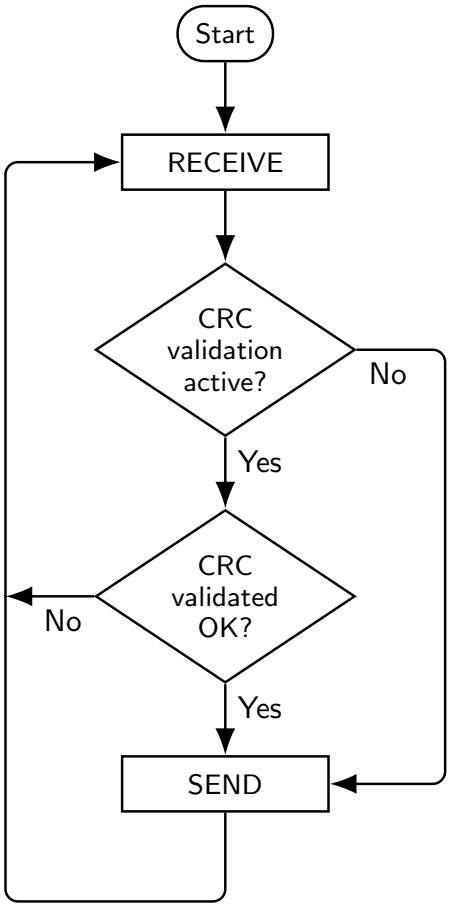


Figure 5: *Access Point* flow diagram.

The *Access Point* starts up in the L1_RECEIVE state, see Section 3. This state is indicated by the debug LEDs 1 and 3 which then are lit. Once a preamble is found, the debug LED 1 repeats the received symbol. When an SFD has been detected in the train of sampled symbols, debug LED 2 is lit. Once a frame has been fully received, the L2 frame is identified and can be de-composed. If the CRC validation is inactive, the state is changed to L1_SEND and the frame is transmitted. If the CRC validation is active and the validation is successful, debug LED 3 is lit and the state is changed to L1_SEND and the frame is transmitted. Otherwise the received frame is silently dropped.

If the *Access Point* should perform CRC validation or not is controlled by the DIP switch 4, see Table 3.

Table 3: *Access Point* CRC validation

DIP switch 4	**CRC validation**
off	not active
on	active

Part 5
Libraries

The library define global constants, arrays, variables and classes that you can use in your code. The classes define private – in some cases public – variables and methods. The global constants are separate from the classes and can thus be used once the library is included in the main sketch.

To include a library in your sketch, an `\#include <datacommlib.h>` statement should be set at the beginning of the sketch. The global constants and the classes `Shield`, `Transmit` and `Receive` are described in the following sections. The class `Frame` is inherited by the classes Transmit and Receive.

1 Predefined values and global constants

The shield's pins are given corresponding values.

Table 1: Constants addressing the shield's pins.

Name	Value	Description
`LED_B`	10	Blue user LED pin
`LED_R`	11	Red user LED pin
`LED_G`	12	Green user LED pin
`DEB_1`	7	Debug LED 1
`DEB_2`	8	Debug LED 2
`DEB_3`	9	Debug LED 3
`PIN_RX`	0	Receive diode pin
`PIN_TX`	13	Transmit LED pin
`BUTTON`	2	Button pin

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

Table 2: Predefined states constants

Name	Value	Description
NONE	-1	No state
L1_RECEIVE	0	Receive frame
L1_SEND	1	Transmit frame
L2_FRAME_REC	10	Process received frame on the Data Link Layer (L2)
L2_DATA_SEND	11	Process the Data Link Layer (L2) frame to be sent
L2_ACK_REC	12	Process reception of an ACK frame
L2_ACK_SEND	13	Process sending of an ACK
L2_RETRANSMIT	14	Control of ARQ
APP_PRODUCE	20	Produce content/message to send
APP_ACT	21	Act on received payload
WAIT	-2	Wait, see note 1
DEBUG	-3	Print all system proporties, see note 1
HALT	-4	"Halt" the system, meaning an infinite loop

Note 1): The WAIT and DEBUG states are defined in the library but not used in the skeleton's state machine.

The global constants are presented per layer or function.

Table 3: Definition of constant values and variables for the *physical* layer.

Type	Name, declaration	Value	Description
value	T_S	100 ms	T_s, symbol length
value	AD_TH	900	A/D converter threshold
value	MAX_TX_ATTEMPTS	3	Max transmission attempts (should be limited)
value	LEN_PREAMBLE	8	Preamble size
byte	PREAMBLE_SEQ	0b10101010	Preamble bit sequence
value	LEN_SFD	8	SFD size
byte	SFD_SEQ	0b01111110	SFD bit sequence

Table 4: Definition of constant values and variables for the *data link* layer.

Type	Name, declaration	Value	Description
value	FRAME_TYPE_ACK	1	ACK message type
value	FRAME_TYPE_DATA	2	DATA message type
value	LEN_FRAME_PAYLOAD	8	L2 frame payload size
value	LEN_FRAME_TYPE	4	L2 frame message type size
value	LEN_FRAME_SEQNUM	4	L2 frame sequence number size
value	LEN_FRAME_ADDR	4	L2 frame address size
value	LEN_FRAME_CRC	8	L2 frame CRC size
value	LEN_FRAME	32	L2 frame length, see note 2
unsigned long	testframe	see note 3	A complete frame aimed for testing

Note 2: The LEN_FRAME is 32 as defined by the protocol. It is equal to the sum of LEN_FRAME_ADDR*2 + LEN_FRAME_TYPE + LEN_FRAME_SEQNUM + LEN_FRAME_PAYLOAD + LEN_FRAME_CRC, that is the L2 frame header, payload and tail.
Note 3: The testframe contains a full L2 frame, excluding preamble and SFD.

Table 5: Definition of constant values and variables for the *application* layer.

Type	Name, declaration	Value	Description
value	LEN_MESSAGE	2	Application layer message size
value	MESSAGE_ADDRESS	0	Pointer to the message address field
value	MESSAGE_PAYLOAD	1	Pointer to the message payload field

2 The Shield class

The Shield class contains variables and methods that are related to hardware and the sketch. The Shield has methods for initialization of the board and shield, reading the select button, turning on and off user and debug LEDs and reading the output of the IR receiver.

2.1 Shield's constructor

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

2.2 Shield's public variables

This class has no public variables.

2.3 Shield's public methods

Initialize sketch and shield class: Shield::begin()
`void begin()` : Initialization of sketch and shield. Must be called at the beginning of the sketch's `setup` function. See Part B for an example.

- Input: None
- Returns: Nothing

Read the address switches: Shield::get_address()
`int get_address()` : Reads the address DIP switches and returns the values as one integer.

- Input: None
- Returns: Address according to the DIP switch settings

Set the nodes own address: Shield::setMyAddress(address)
`void void Shield::setMyAddress(int value)` : Set the private variable myAddress.

- Input: Address
- Returns: Nothing

Get the nodes own address: Shield::getMyAddress()
`int getMyAddress()` : Get value of the private variable myAddress.

- Input: None
- Returns: The nodes own address

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

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

Sample the receiver channel: Shield::sampleRecCh(int pin)
`int sampleRecCh(int pin)`: Samples the receiver channel (i.e. the receive pin) and returns a binary value, 0 or 1, depending on the A/D converting threshold value set by the Shield::setAdThreshold() method, see Crefsec:setadthr and Section 2.3.

- Input: Receiver channel pin number

- Returns: The sampled binary value of the receiver

Note! The simulated library method reads a binary value. In the simulated environment] the A/D threshold has no function. See also **??**.

Set the A/D threshold value used by Shield::sampleRecCh(), see Section 2.3: Shield::setAdThreshold(int value)
`void Shield::setAdThreshold(int value)`: Set the A/D threshold.

- Input: A/D threshold
- Returns: None

The A/D threshold value is used by Shield::sampleRecCh(), see Section 2.3.

Read the current A/D threshold: Shield::getAdThreshold(int value)
`int Shield::getAdThreshold()`: Get curren A/D threshold.

- Input: None
- Returns: Current A/D threshold

The A/D threshold is used by Shield::sampleRecCh(), see Section 2.3.

Turn on all user LEDs: Shield::allLedsOn()
`void allLedsOn()` : Turn all user LEDs on.

- Input: None
- Returns: Nothing

Turn off all user LEDs: Shield::allLedsOff()
`void allLedsOff()` : Turn all user LEDs off.

- Input: None
- Returns: Nothing

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

- Input: None
- Returns: Nothing

Halt the execution: Shield::halt()
`void halt()` : Infinity empty loop, which effectively finishes execution of a sketch.

- Input: None
- Returns: Nothing

3 The `Frame` class

The `Frame` class defines the L2 frame variabel and the frame fields as individual public integer variables. These variables corresponds to the individual fields of the L2 frame. A public method for print out of the frame contents on the *Serial Monitor*is defined.

3.1 Frame's constructor

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

3.2 Frame's public variables

Table 6: Frame class' public variables

Type	**Name, declaration**	**Description**
unsigned long	`frame`	L2 layer frame
int	`frame_from`	Source address
int	`frame_to`	Destination address
int	`frame_type`	Message/frame type
int	`frame_seqnum`	Sequence number
int	`frame_payload`	Payload
int	`frame_crc`	CRC

3.3 Frame's public methods

Print frame content: Frame::print_frame()
`void print_frame()` : Hex print out of the `frame` on the *Serial Monitor*.

- Input: None
- Returns: Nothing

4 The `Transmit` class

The `Transmit` class inherits the `Frame` class. Thus, all the public variables and methods defined in class Frame is also defined in class Transmit. It contains methods and variables for creating an L2 frame in the form of a 32 bit unsigned integer.

4.1 Transmit's constructor

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

4.2 Transmit's public variables

Table 7: Transmit's public variables

Type	Name, declaration	Description
int	`message[LEN_MESSAGE]`	Application layer message
int	`tx_attempts`	Transmission attempts counter

4.3 Transmit's public methods

Generate a frame from the frame field variables: Transmit::frame_generation()
`void frame_generation()` : Fills the L2 frame buffer `frame` with bits according to the content of the L2 frame variables, see Table 6.

Add CRC bits to a frame: Transmit::add_crc()
`void add_crc(int crc)` : Fill the CRC part of the class' variable `frame` with the value of the input parameter. Note that no CRC calculation is performed.

- Input: The CRC value calculated over the `Transmit::frame`.
- Returns: None

5 The `Receive` class

The `Receive` class inherits the `Frame` class. Thus, all the public variables and methods defined in class Frame is also defined in class Receive. It contains a method and variables for decomposing a received L2 frame into individual integers corresponding to the L2 frame fields.

5.1 Receive's constructor

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

5.2 Receive's public variables

Table 8: Receive's public variables

Type	Name, declaration	Description
int	`message[LEN_MESSAGE]`	Application layer message

5.3 Receive's public methods

Decompose a frame into field variables: Receive::frame_decompose()
`void frame_decompose()` : Decomposes the bits in `Receive::frame` into individual integers representing each L2 frame field, see Table 6.

References

[1] `https://www.arduino.cc/en/reference/millis`. Last visited 2023-01-09.

[2] Alohanet, pure aloha. `https://en.wikipedia.org/wiki/ALOHAnet#Pure_ALOHA`. Last visited 2023-01-09.

[3] Mathematics of cyclic redundancy checks. `https://en.wikipedia.org/wiki/Mathematics_of_cyclic_redundancy_checks`. Last visited 2023-01-09.

[4] Arduino language reference. `https://www.arduino.cc/en/Reference/HomePage`, 2015. Last visited 2023-01-09.

[5] Arduino software. `https://www.arduino.cc/en/Main/Software`, 2015. Last visited 2023-01-09.

[6] Arduino software (ide). `https://www.arduino.cc/en/Guide/Environment`, 2015. Last visited 2023-01-09.

[7] Getting started with arduino. `https://www.arduino.cc/en/Guide/HomePage`, 2015. Last visited 2023-01-09.

[8] Introduction to the arduino board. `https://www.arduino.cc/en/Reference/Board`, 2015. Last visited 2023-01-09.

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

Part A
Physical Lab Environment

During the you will have access to a lab computer equipped with the Arduino IDE [6] and a USB power supply. You will also have access to one *Master Node* and one *Development Node*, and for lab three and four, one additional node, the *Access Point*.

Use the following account to log in to the computers: User name: `etsf15_stud` and password: `25etsf15lab`.

An Arduino sketch has to be stored in a folder with the same name as the sketch file. For example, if the sketch is called `MySketch.ino`, it has to be stored in a folder called `MySketch`. The lab's skeleton sketch is thus stored in a folder with the same name—omitting the extension—as the skeleton sketch itself.

Download the code skeleton from Canvas to your computer. Use a USB stick to move it to the lab computer. Open the folder on the computer called `Arduino` and store the skeleton's `.ino` file in a folder with the same name as the sketch. You can open the sketch in the Arduino IDE by double-clicking the `.ino` file in the folder. You can also open the Arduino IDE by opening the `Terminal` and type `/usr/arduino-1.8.18/arduino`. You need to include the library for the labs in the sketch file, i.e. write `#include <datacommlib.h>`

When you are done, take a USB stick, and move your code from the lab computer to your own computer (or your own USB stick) and remember to bring it the next time you are going to work on the labs. Delete your file on the lab computer before you logout.

To set up the communication with the Arduino board ensure that, under Tools, "Uno" is chosen and that the Port where the Arduino is connected is chosen. Try to upload the skeleton sketch 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 the three LEDs D3 to D5 on the shield (pins 7, 8 and 9) and the `Serial` functions for printing data to the Arduino IDE's *Serial Monitor* (See the Debugging tools in Section 4 for details).

If you are interested, you can download and install the Arduino IDE on your own computer. In the development environment you can find typical examples that are included in the distribution. Go to `File–Examples` to have a look. There are also many examples on the Internet. e.g. visit [4] for a reference on the Arduino language.

1 Physical layer labs

For the first two labs, the two Arduino boards should be on your working bench, one connected to the computer, that is the *Development Node*, marked with a T, and one connected to a USB power outlet, that is the *Master Node* marked with R (if you connect it to the computer, you need to make sure to not write code to the wrong Arduino). The

Master Node, as fully described in the Section 2, will loop through the states[2] and you will not have access to manipulate or view its software sketch. 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. The *Master Node* is a fully functioning remote host that can be used both for all labd. It can receive and decode a frame, and perform the remote host ARQ functionality. Optionally it can validate CRC parity bits of incoming frames as well create CRC parity bits for outgoing frames. For the first two labs, all four DIP switches must be set to 0.

2 Data link layer labs

In the lab environment for lab three and four there should be three Arduinos on your working bench. The *Development Node* should be connected to the computer, and the *Master Node* and *Access Point* should be connected to a USB power outlet. It is important that the nodes are aligned so that both the *Development Node* and the *Master Node* can send to and receive from the *Access Point*, while not being able to send to each other. Place the *Development Node* and the *Master Node* side by side, both facing the *Access Point*, somewhat like you see in Figure 1.

The *Access Point* works in store-and-forward mode. Once the *Access Point* discovers a frame transmission on the link it starts receiving that frame, decodes it and transmits it on the link again. Thus the original transmitter will receive a copy of the frame it just sent. This explains why addressing is needed even with only two operational nodes accessing the link.

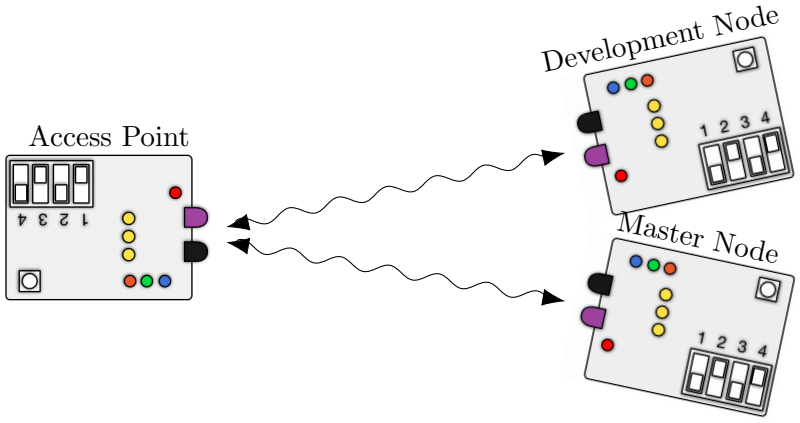


Figure 1: The setup of the network.

[2]The *Master Node* state machine is depicted in figure Figure 2.

Part B
Skeleton.ino

```
1   ////////////////////////////
2   //
3   // V21.1 Skeleton.ino
4   //
5   // Adapted to the physical and simulated environments
6   //
7   // 2022-12-17 Jens Andersson
8   //
9   ////////////////////////////
10
11  //
12  // Select library
13  //
14  // uncomment for the physical environment
15  //#include <datacommlib.h>
16  //
17  // uncomment for the simulated environment
18  //#include <datacommsimlib.h>
19
20  //
21  // Prototypes
22  //
23
24  //predefined
25  void l1_send(unsigned long l2frame, int framelen);
26  boolean l1_receive(int timeout);
27  // your own
28
29  //
30  // Runtime
31  //
32
33  ////////////////////////////////////////////////////////
34  //
35  // Add your global constant and variabel declaretions here
36  //
37
38  int state = NONE;
39  Shield sh; // note! no () since constructor takes no arguments
40  Transmit tx;
41  Receive rx;
42
43  ////////////////////////////////////////////////////////
44
45  //
46  // Code
47  //
48  void setup() {
49      sh.begin();
50
51      ////////////////////////////////////////////////////////
52      //
53      // Add init code here
54      //
55
56      state = NONE;
57
58      // Set your development node's address here
59
60      ////////////////////////////////////////////////////////
61  }
62
63  void loop() {
64
65      ////////////////////////////////////////////////////////
66      //
67      // State machine
68      // Add code for the different states here
69      //
```

```
70
71      switch(state){
72
73          case L1_SEND:
74              Serial.println("[State] L1_SEND");
75              // +++ add code here and to the predefined function void l1_send(unsigned
                long l2frame, int framelen) below
76
77              // ---
78              break;
79
80          case L1_RECEIVE:
81              Serial.println("[State] L1_RECEIVE");
82              // +++ add code here and to the predifend function boolean l1_receive(int
                timeout) below
83
84              // ---
85              break;
86
87          case L2_DATA_SEND:
88              Serial.println("[State] L2_DATA_SEND");
89              // +++ add code here
90
91              // ---
92              break;
93
94          case L2_RETRANSMIT:
95              Serial.println("[State] L2_RETRANSMIT");
96              // +++ add code here
97
98              // ---
99              break;
100
101         case L2_FRAME_REC:
102             Serial.println("[State] L2_FRAME_REC");
103             // +++ add code here
104
105             // ---
106             break;
107
108         case L2_ACK_SEND:
109             Serial.println("[State] L2_ACK_SEND");
110             // +++ add code here
111
112             // ---
113             break;
114
115         case L2_ACK_REC:
116             Serial.println("[State] L2_ACK_REC");
117             // +++ add code here
118
119             // ---
120             break;
121
122         case APP_PRODUCE:
123             Serial.println("[State] APP_PRODUCE");
124             // +++ add code here
125
126             // ---
127             break;
128
129         case APP_ACT:
130             Serial.println("[State] APP_ACT");
131             // +++ add code here
132
133             // ---
134             break;
135
136         case HALT:
```

```
            Serial.println("[State] HALT");
            sh.halt();
            break;

        default:
            Serial.println("UNDEFINED STATE");
            break;
    }

    ////////////////////////////////////////////////////////

}
////////////////////////////////////////////////////////
//
// Add code to the predfined functions
//
void l1_send(unsigned long frame, int framelen) {

}

boolean l1_receive(int timeout) {

    return true;
}

////////////////////////////////////////////////////////
//
// Add your functions here
//

```