# TITLE

**Subtitle**

Project Paper

SW503E15

Aalborg Universitet
Department of Computer Science
Selma lagerlöfs vej 300
9220 Aalborg Øst

Here you can write something about which tools and software you have used for typesetting the document, running simulations and creating figures. If you do not know what to write, either leave this page blank or have a look at the colophon in some of your books.

AALBORG UNIVERSITY
STUDENT REPORT

**Department of Computer Science**
Selma lagerlöfs vej 300
9220 Aalborg Øst
http://cs.aau.dk

**Title:**
TITLE

**Theme:**
Embedded Systems

**Project Period:**
Fall 2015

**Project Group:**
SW503E15

**Members:**
Claus Worm Wiingreen
Marc Tom Thorgersen
Mathias Sass Michno
Morten Pedersen
Troels Beck Krøgh
Søren Hvidberg Frandsen

**Supervisor:**
Jiří Srba

**Copies:** 8

**Page Numbers:** ??

**Date of Completion:**
19th November 2015

**Abstract:**

Abstract here

# Contents

# Todo list

# Preface

Here is the preface. You should put your signatures at the end of the preface.

Sig tak for hjÃęlp til i.e. Kim Larsen, Jiri etc.

Aalborg University, 19th November 2015

Claus Worm Wiingreen
<cwiing13@student.aau.dk>

Marc Tom Thorgersen
<mthorg13@student.aau.dk>

Mathias Sass Michno
<mmichn13@student.aau.dk>

Morten Pedersen
<morped13@student.aau.dk>

Troels Beck Krøgh
<tkragh13@student.aau.dk>

Søren Hvidberg Frandsen
<sfrand12@student.aau.dk>

# Chapter 1

# Introduction

Wired communication is considered the fastest and most reliably mean of digital communication. Wireless communication allows for a wider field of use cases, this is a trade-off between reliability and speed on the one side and flexibility on the other. [**wirelessTradeoffs**]

Devices which utilise wireless communication have heavily influenced the world in the last couple of years, with smart phones becoming more and more common to most people in the western world. [**2013-SmartPhoneUse**] However, there exists many more uses for wireless technology than just being used in mobile phones, tablets and laptops, it can also be implemented in smaller embedded systems[1] in every day use.

An example of these smaller embedded systems could be modern fire alarms for larger buildings. Currently a connected fire alarm system such as those found in office buildings, require wire to run between the individual units. This could be a problem for older buildings where places to hide wires for this system was not though into the design of the building.

That is where a system of wireless devices could be useful. However such a system is in competition with its predecessor, and the wired technology has the edge of being very reliable and fast once it is set up.

The main problem of a wired network is the setup of the network and later addition of new devices. New wires have to be installed into the building and the network must be updated in order to know about the new unit(s). For devices such as fire alarms this is less of a problem but for other uses, such as home automation, where new devices might enter and leave the network at any time this can pose as a hurdle.

This could be solved by using Wi-Fi networks. Wi-Fi provides decades of research in communications which allows them to transfer lots of data reasonably fast. For embedded systems this might be excessive depending on the system in question. The fire alarm for instance would only need to send a few bits of data, so a Wi-Fi soltion would require more power but still only use a fraction of the features available.

## 1.1 Problem Statement

In this project the focus is going to be on a solution which can be modified for a number of usecases, while being just enough to solve the problem. In other words; a cost efficient solution to the problem, and we think that there might be enough power in radio frequencies modules

[1]In this project ‘embedded systems’ is used to describe a category of computational units, which have limited resources and are relatively small in size.

that even when using only one frequency a communications network can be established and maintained.

This gives the problem that will be worked on in this report.

*How can a network of devices with radio transceivers of a single frequency communicate, such that any devices can send messages to any other device in the network, in a reliable and time-critical way?*

The following part of the paper will describe, analyse and evaluate the initiating problem, which concerns the difficulties of having multiple embedded systems communicate on the same radio frequency. Furthermore, some of the possible solutions to the problem including hardware, software and techniques will be introduced and discussed. It could make sense to use more frequencies to communicate with so that multiple channels could be communicating at the same time. However, a problem arises when there is many devices, because we will run out of frequencies at some point, so therefore the problem of solving how multiple devices communicate on the same frequency is still an important issue to solve in this project. Finally the part will end in a problem statement in which will be used in the design phase of this project.

sec:problemStatement

# Part I

# Analysis

# Chapter 2

# Time Division Multiple Access

Tilpas sÃě det passer som kapitel 2

**Time Division Multiple Access (TDMA)** is a method of sharing the same radio frequency between two or more devices.

This method was first used in practice in 1991 for use in the cellular telephones, to increase the number of concurrent users on the telephone network.[**networkencyclopedia2013time**]

### 2.0.1 How TDMA works

If multiple radios transmit on the same frequency at the same time while in range of eachother, then interference occurs as can be seen on Figure 2.1. On the figure the square-device is in range of both circle-devices, so if they broadcast on the same frequency at the same time then the square-device will most likely not be able to decode either of the signals.[**networkencyclopedia2013time**, **networkencyclopedia2013advanced**]

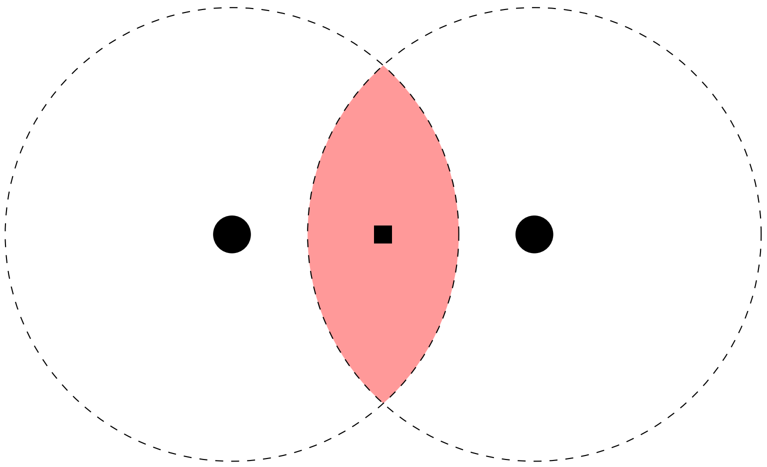
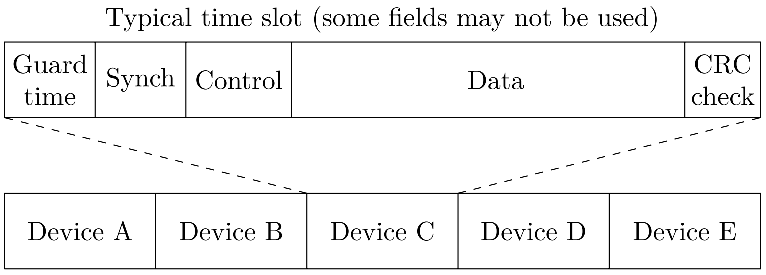


**Figure 2.1:** The two devices (circles) communicate on the same frequency making the receiver (square) unable to read either signal.

This is solved by having the transmitters taking turns transmitting their message in smaller time slots as seen on figure Figure 2.2. The figure shows five devices which share the same frequency, and to do this they get a timeslot each in the frame. A frame is the complete cycle of these timeslots.



(a) A frame with five time slots

**Figure 2.2:** An example of a time slot allocation for five devices on a single frequency by using TDMA

Just like frames are divided into time slots; the following five parts are what the individual time slots consists of:

**Guard time**
A period where no important data is sent. This is here to make sure that devices not properly synced to the loop is less likely to corrupt the signal of other senders.

**Synch**
A possible additional delay before the signal depending on the distance to the receiver.

**Control**
One or more control values which determines the nature of the signal.

**Data**
The main data of the signal.

**CRC check**
The **Cyclic Redundancy Check (CRC)** is an error detecting code. It is there to make sure that the correct message was received.

The aim of this division is to ensure that the signals does not intersect, the message being sent is isolated, and that the data is received as intended.

What is the purpose of the code below? - Troels

```
void decodeStatus(uint8_t *status, uint8_t *ok, uint16_t *sync) {
                             // Bit masks
 *ok   = status[0]           & 0x80;
 *sync = *(uint16_t*)status & 0x7FFF;
}
```

**Listing 2.1:** Test

# Chapter 3

# Hardware

In this section, the hardware aspect of the project will be explored and the best suited devices and components for our purposes will be presented.

Synes der er meget her som ikke helt behÃÿves, altsÃě hvor meget giver det til rapporten?

## 3.1 Development Platform Selection

To be able to develop an embedded system a development platform is needed. There are some strict requirements for this development platform ("need to have") and some which are advantageous but not critical ("nice to have"). The strict requirements must be upheld for a development platform to be usable in this project and are:

*a*) able to communicate using a single radio frequency, or have a way of using additional hardware to do so;

*b*) the ability to debug and develop programs in a fast and reliable way;

*c*) a relatively low cost, within the constrains of the budget given by the institution; and

*d*) an ability to implement a real-time aspect.

Furthermore the following parameters would be advantageous for this project and e.g. the real world application hereof:

*e*) low power consumption, and possibility for running on batteries;

*f*) small form factor of the device itself;

*g*) ability to minimise and customise into a production like product; and

*h*) widespread adaptation for fuller documentation.

After exploring the market the Arduino ATmega328P based development boards seems like an ideal choice. Specifically the Arduino Uno (see [**ArduinoUNO**]), which is the most known model seems like a perfect fit.

Another notable candidate is the Raspberry Pi (see [**RaspberryPI**], which features a much faster CPU, HDMI output, two USB ports etc., however it falls short on the real-time aspect in conjunction with the ability to debug them. There does exist a version of the real-time operating system RTOS (Open Source Real Time Operating System) for the Raspberry Pi. However each Raspberry Pi costs above 200 DKK each, this would only allow the project group to buy two devices, and this would severely limit the possibilities of the project. An Arduino compatible board can be bought from many manufacturers for the low cost of 38 DKK, which means many Arduinoes can be obtained for the project.

Other notable candidates are Teensy, a Arduino compilable device in which size has been minimised. This features all the same in almost all categories, there is neither a clear advantage of choosing this over the Arduino. Therefore it is decided that the Arduino will be the development platform for this project since it uphold the aforementioned requirements, while still being the cheapest platform. The Arduino platform will be more thoroughly investigated in the follow section.

## 3.2 Arduino

Arduino is a company which designs and produces open source hardware and software. This means that anyone which would like to can copy their hardware design and make their own improvements or simply a copy. The software used on Arduinos will be explained in 3.5.

### Arduino Uno

The Arduino Uno is a ATmega328P based micro-controller board. It has female pins in which wires can be attached, most notably there are fourteen digital input/output pins (labelled pin 0 - 13), six analog pins (labeled A0 - A5), some of these pins have special abilities which will be explained later see 3.2. Code can be uploaded through a USB connection, which can also power the Arduino, and provide a debugging interface for serial communication. The total size of an Arduino Uno is roughly five times seven centimetres.

The processor of the Ardunio is the ATmega328P. It has 32 KB of flash memory, 2 KB of SRAM, and 1 KB of EEPROM, and clock speed of 16 MHz.

EEPROM is a small memory section which functions like a small harddrive in the sense that data are saved thereon even when the devices are turned off, more on this can be found on [**EEPROM**].

### Resources

When using the Arduinoes and similar microcontrollers there are some rather serious constraints on memory and speed. One of the most popular Arduino boards the “Arduino

The Birth of the Arduino [**birthofarduino**]

In 2005 a group of students, at the Interaction Design Institute Ivrea in Italy, developed the Arduino board as a project, so there would be a cheaper alternative to the “BASIC Stamp” (another microcontroller-based development kit), used by the institute at the time, which roughly cost $100.

Uno" has 32 KB of flash memory and a mere 2 KB of SRAM, and the frequency of the microcontroller is only 16 MHz.

These constraints are in great contrast to the majority of today's programming practises where e.g. memory is seen more or less as in abundance.

### Shields and modules

A core concept of the Arduino and the environment around it, is the ability to extend and interact with the hardware through so called shields and modules. These shields and modules can be input or output devices which interacts with the microcontroller on the Arduino though the digital or analogue headers on the Arduino board. These headers can however be used for other purposes e.g. powering a LED or reading analogue and digital input such as button presses or various sensor. Examples uses for shields are LCD, Ethernet, SDCard, motor control, and relay control (see [**ArduinoShields**] for more).

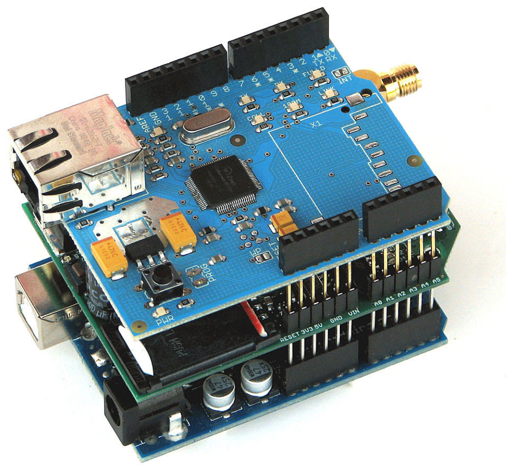


**Figure 3.1:** An Arduino UNO with a stack of two shields on top

### Arduino Uno Pins

The Ardiuno Uno has a total of 20 pins related to input or output, all remaining pins relate to powering the device. 14 of the pins are digital I/O pins, whereof 6 can provide 8-bit **Pulse Width Modulation** (**PWM**) output using a predefined function called analogWrite(). PWM is a method of acquiring an analog like result using a digital resource. Digital control creates a square wave by disabling and enabling the power to a pin. analogWrite() takes a value between 0 - 255, this value determines how long the signal is on and off per cycle.

The six remaining pins relating to I/O are analog input pins. Each digital pin has an internal pull-up resistor providing 20 - 50k ohm. Furthermore some of the digital pins are in some way different from the rest. Pin 0 and 1 on the board, also marked with RXD and TXD, are used for receiving and transmitting **transistor transistor logic** (**TTL**) serial data.

Pins 2 and 3 are interrupt pins, using the function attachInterrupt() to provide an external interrupt. As mentioned 6 pins can provide a PWM output, these are pins 3, 5, 6, 9, 10 and 11. Pins 10 (SS), 11 (MOSI), 12 (MISO) and 13 (SCK) can be used to support

**Serial Peripheral Interface** (**SPI**) communication, each providing one of the four logic signals. SPI is an interface used for short distance communication providing full-duplex communication using a slave-master type architecture. The four pins mentioned each relate to four logic signals the SPI bus uses. SS, slave select, is an output from the master, MOSI, master output - slave input, is a message from master to slave, MISO, master input - slave output, is a message from slave to master, and lastly SCLK, serial clock, which is an output from master and is the clock which the slaves synchronise. To use SPI communication one must implement the SPI library.

Another type of communication is also supported by certain pins, pin A4 or SDA pin and pin A5 or SCL pin. These two pins can support **Two Wire Interface** (**TWI**) communication using the Wire library. TWI is a derivative of the $I^2C$ serial bus, and is essentially the same as $I^2C$. Like SPI this is a serial bus which implements a master-slave type architecture, however this allows for more masters. Unlike SPI this is a two-wire rather than four-wire system only using a Serial Data Line (SDL) and a Serial Clock Line (SCL). For more infomation on the pins on an Arduino see [**ArduinoUNO**].

## 3.3 Radio Frequency Module

In order to achieve wireless communication between Arduino devices a Radio Frequency (RF) module is used. The module used in this project communicates at 433 MHz; its datasheet specifies the range is 20 to 200 meters, depending on the input voltage, which ranges from 3.5 to 12 volts. The Arduino provides 3.3 volts and 5 volts output, and the 5 volts output will be used. The maximum bandwidth is 4 kilobytes/secound. The signal is sent using **Amplitude Shift Keying (ASK)**. These values will be tested, to ensure the reliability of the information.

The RF system consists of a transmitting module (tx) and a receiving module (rx), see 3.2. Both have the possibility of connecting an antenna to it in order to increase the range of the device. The transmitting module have three pins to connect to it. The pins are VCC, meaning the power supply pin, the GND meaning ground, and DATA. The DATA pin is turned on by having the same difference in voltage between GND and DATA as VCC and GND. The receiving module has the same pins as the transmitting module, however the DATA pins is being read from instead of transmitted to by the Arduino.

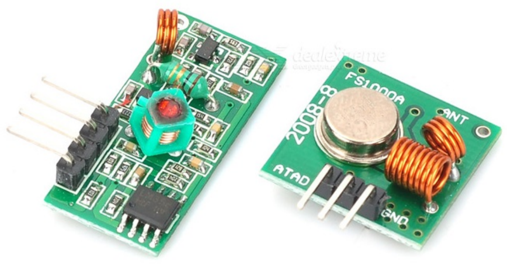


**Figure 3.2:** The 433MHz RF receiver (left) and transmitter (right)

## Antenna

It is possible to attach an antenna to the RF module. The benefits of using an antenna is to better send and/or receive signals, this can improve both maximum distance and accuracy of transmissions. Attaching an antenna is done by soldering the antenna to the ANT port on both the receiver and the transmitter. In Figure 3.2 these are visible; one in the upper left corner on the receiver and one in the upper right corner on the transmitter.

A wire acts like a monopole antenna, which works by acting like an open resonator for the signal, it's length can greatly affect its effect. Candidates for the length of this can be calculated based on the frequency of the signal, for the RF modules this is 433 MHz. The formula for a quarter-monopole antenna is:

$$l = \frac{c}{f * 4} \tag{3.1}$$

where c is the speed of light, and f is the frequency of the wave. Similarly the formula for a half-monopole antenna is:

$$l = \frac{c}{f * 2} \tag{3.2}$$

Using the frequency, 433 MHz, the quarter- and half-monopole antenna lengths are 17.3 cm and 34.6 cm respectively. [**AntennaLength**]

The efficiency of using such antennas will be tested in Section 3.6.

## Amplitude Shift Keying

As mentioned the RF module sends its signal using ASK. ASK is a form of modulation where the digital input is a radio wave where the digital high (also known as "1") is a greater amplitude than the digital low (also known as "0"). This principle can also be used with light, here digital high would be a short pulse, and a digital low would be the absence of light. To use ASK the devices must be synchronised meaning that the time period of a single bit must be known by the receiver, in order for it to demodulate it. In Figure 3.3 is an example of how a radio wave would look like while using ASK, here the period for each signal is twice the peiod of the wave. [**ASKnFSK**]

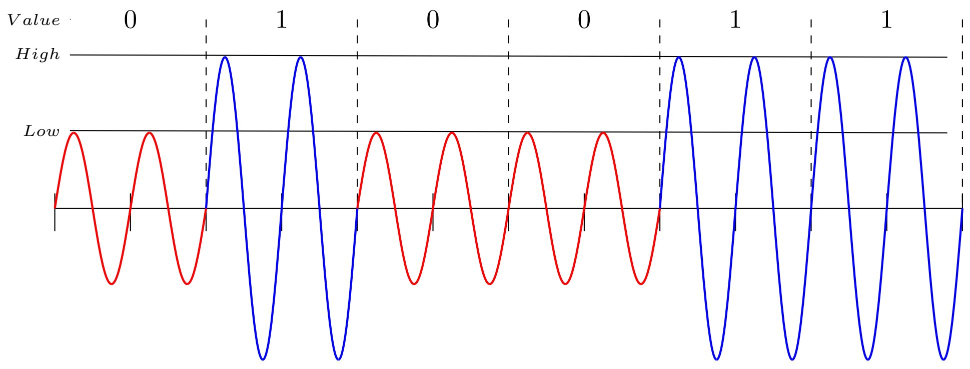


**Figure 3.3:** ASK: Digital values are signaled by the amplitude of the radio wave.

## 3.4 Other components

This subsection briefly covers some of the other components for the project.

### Light Emitting Diode

A **Light Emitting Diode** (**LED**) is a small light-source which can come in a variety of different colours including those hidden from the human eye such as infrared and ultraviolet. The LED is connected to electricity through an anode (often the long leg) and a cathode (short leg), once connected current flows and the LED to emits light. They typically require very little power.

### Switch

Another relatively simple piece of hardware used is a switch. The switches mentioned throughout this paper establish a connection between two disconnected conductors when held down, allowing a flow of electrons to pass through the switch. This can be read by the Arduino and used as an input device.

## 3.5 Software for Arduinos

This section will discuss how software is distributed to the Arduino, and also present some of the possible libraries for Arduinos.

### 3.5.1 Toolchain

When programming for the Arduino, it is possible to use the Arduino IDE, developed by the Arduino company. This IDE uses a specific version of the GCC compiler, which targets the AVR microcontrollers. The C or C++ code is first checked for errors and compiled to object code, which then is linked with the standard Arduino libraries and any custom libraries. The linked object code is then compiled to a single Intel Hex file, which is loaded onto the Arduino using the bootloader already situated on the microcontroller. A USB to serial converter is also implemented directly on most of the Arduino boards, since it is through the serial connection that the microcontroller communicates. This toolchain is also shown on Figure 3.4.[**2015ArduinoToolchain**]

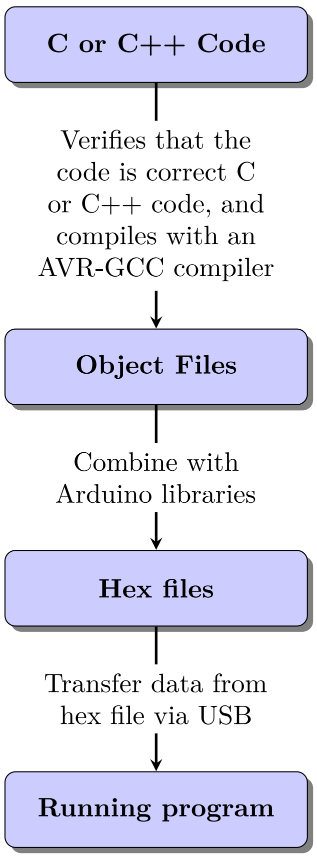

C or C++ Code
Verifies that the code is correct C or C++ code, and compiles with an AVR-GCC compiler
Object Files
Combine with Arduino libraries
Hex files
Transfer data from hex file via USB
Running program

**Figure 3.4:** Flow diagram showing the flow of the source files onto the Arduino

To show how to upload code to an Arduino, some screenshots have been taken to show

how this is done through the Arduino IDE. First to upload any code to the Arduino, the kind of Arduino being used has to be specified, and how to do this can be seen on figure 3.5 (a).

Secondly the Port where the Arudino being uploaded to is plugged in needs to be specified, and a screenshot of this can be seen on figure Figure 3.5 (b).

When the code is ready to be uploaded to the Arduino a simple press of the arrow in the top of the IDE will upload and run the code. This can be seen on figure Figure 3.5 (c), the code being run on this screenshot is a simple program which turns on an LED on pin 13 of the Arduino with a varying delay, which is also printed to the serial monitor. The serial monitor is of great use when debugging with the Arduino as it makes it possible to know what values different variables of the program have at run-time.

The code is split into three parts. The function `setup()` is run once when the program starts, and here is where different pins are often set to be input or output. The function `loop()` is then run continuously as long as the Arduino is turned on. The last part in this example is a global variable `int i` which is set to global in this very simple example.

It is of course also possible to program in many other languages for the Arduino as long as a compiler which compiles for the AVR processors is used. An example of this could be Java. NanoVM is an implementation of some of the Java virtual machine for the AVR processors, it does not contain all of its features because of the limited resources the microcontrollers possesses. According to the creator NanoVM will never contain all the features of the JVM, and C is still a much more powerful language when working with microcontrollers. Therefore it is also chosen for the software being developed in this project. [**NanoVM**]

### 3.5.2 Libraries

This section will describe different libraries used for the project.

#### RadioHead

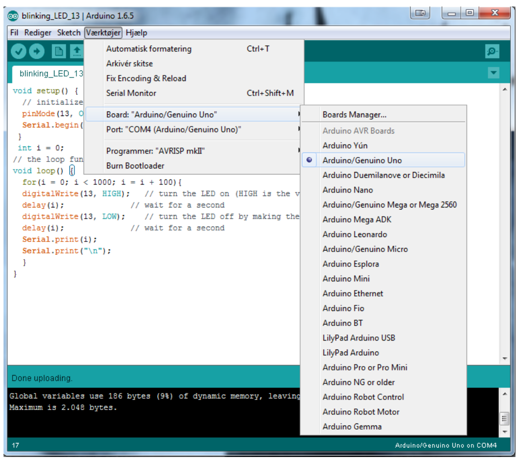
In order to use the RF modules with good results, some way of synchronising and encoding the transmissions can be used to reduce faults, like package loss. A library called RadioHead does exactly that, and this section explores how it works.

RadioHead transmits each message, using ASK, in the following way: Each byte (8 bits) gets encoded as two 6 bit symbols, the 6 bit symbols will never contain 3 of the same signal in a row (high or low), this is to ensure that the receiver can decode the signal properly as it needs to differentiate the high and low signals. When a message for example the string 'Hello', several steps aret taken, and they are as follows:

First a preamble consisting of the the binary pattern '1010' six times. This is so the receiver can synchronise its timer to the timer of the sender. Then the pattern '111000' once, and then '101100', and then the start symbol '101100111000'. This is the start code of the message. Every four bit from now is encoded as the six bit encoding mentioned before. The next byte (effectively 12 bits) is the message length including itself and the check-sum bytes. Then the message 'Hello', this doesn't need to be terminated as the length is known, has the size 5 bytes ($5 * 12bits/byte = 60bits$). Then a check-sum to ensure the integrity of the message, consisting of 24 bits after encoding.

This brings the total size of the packet to 144 bit, while the message is 5 bytes or 40 bits.

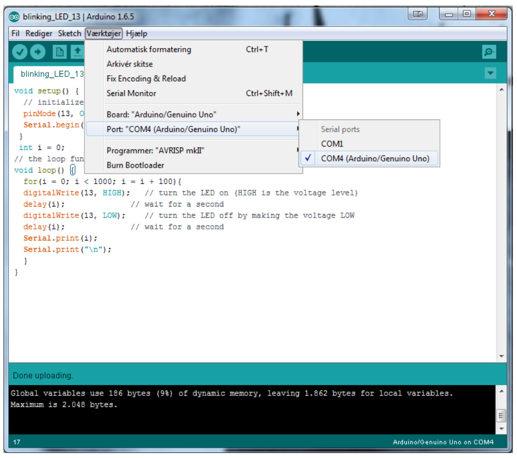
The recommended bit-rate for the transceiver is 2000 bit/sec, this means the message



```
void setup() {
  // initialize
  pinMode(13, O
  Serial.begin(
 }
 int i = 0;
// the loop fun
void loop() {
  for(i = 0; i < 1000; i = i + 100){
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the v
  delay(i);               // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the
  delay(i);               // wait for a second
  Serial.print(i);
  Serial.print("\n");
  }
}
```

```
Global variables use 186 bytes (9%) of dynamic memory, leaving
Maximum is 2.048 bytes.
```

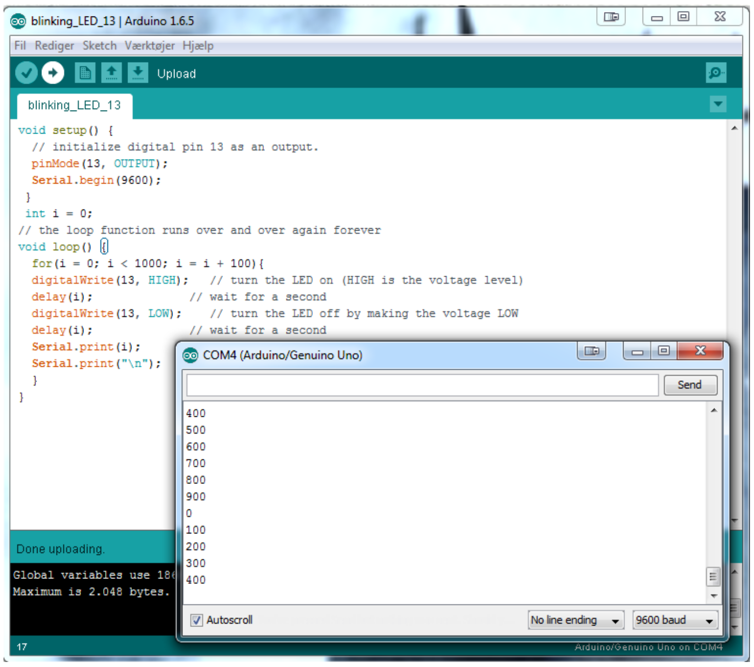
(a) Choose Arduino


blinking_LED_13 | Arduino 1.6.5
Fil Rediger Sketch Værktøjer Hjælp
Automatisk formatering Ctrl+T
Arkivér skitse
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board: "Arduino/Genuino Uno"
Port: "COM4 (Arduino/Genuino Uno)"
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM1
✓ COM4 (Arduino/Genuino Uno)
blinking_LED_13
Done uploading.
17
Arduino/Genuino Uno on COM4

```
void setup() {
  // initialize
  pinMode(13, O
  Serial.begin(
 }
 int i = 0;
// the loop fun
void loop() {
  for(i = 0; i < 1000; i = i + 100){
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(i);               // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(i);               // wait for a second
  Serial.print(i);
  Serial.print("\n");
  }
}
```

```
Global variables use 186 bytes (9%) of dynamic memory, leaving 1.862 bytes for local variables.
Maximum is 2.048 bytes.
```

(b) Choose Port


blinking_LED_13 | Arduino 1.6.5
Fil Rediger Sketch Værktøjer Hjælp
Upload
blinking_LED_13
Done uploading.
17
Arduino/Genuino Uno on COM4
COM4 (Arduino/Genuino Uno)
Send
Autoscroll
No line ending
9600 baud

```
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
  Serial.begin(9600);
 }
 int i = 0;
// the loop function runs over and over again forever
void loop() {
  for(i = 0; i < 1000; i = i + 100){
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(i);               // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(i);               // wait for a second
  Serial.print(i);
  Serial.print("\n");
  }
}
```

```
Global variables use 18
Maximum is 2.048 bytes.
```

```
400
500
600
700
800
900
0
100
200
300
400
```

(c) Serial monitor after uploading code

**Figure 3.5:** Screenshots from the Arduino IDE.

**Table 3.1:** A table view of the content of transmitting ‘Hello’ with RadioHead.

| Name | Purpose | Content | Size [bit] |
|---|---|---|---|
| Preamble pt. 1 | Syncronization | ‘1010’ * 6 | 24 |
| Preamble pt. 2 | Syncronization | ‘111000’ | 6 |
| Preamble pt. 3 | Syncronization | ‘101100’ | 6 |
| Start symbol | Indicate start of transmission | ‘101100111000’ | 12 |
| Message length | To indicate the length of the message | 4to6(‘01101000’) | 12 |
| Message | The payload | 4to6(‘Hello’) | 60 |
| Checksum | To verify the integrity | 4to6(‘???’) | 12 |

will take at least:

$$\frac{144bit}{2000\frac{bit}{sec}} = 0.072sec \tag{3.3}$$

## Timing

After calculating the contents of the package to be sent, the Arduino needs to write it to the RF transmitter. To time this the built in “timer1” on the Arduino is used, to trigger an interrupt with a relatively high consistency. Every eighth interrupt the signal is changed to the next bit and written via a digital pin. On the receiving end, the signal is sampled eight times pr. period, if five or more of these eight is high then, it is declared a 0 bit, otherwise it is declared a 1 bit.

## Timer

The library Timer is used to perform repeated actions in a given period. There are functions to call a function repeatedly, to wait a given period and then call a function, to toggle the state of a pin repeatedly and finally to pulse a pin repeatedly. This can be useful as i.e. pulsing an LED can be performed without a call to the delay function and halting other functions of the program. The library works by creating internal objects which will be updated every time a call to update is performed. Therefore it is necessary to call the update function often, for this to work. To stop an event one must save the id returned when the event was started and use this to call the stop method.

The function prototypes used in this library is:

```
int8_t every(unsigned long period, void (*callback)(void));
int8_t every(unsigned long period, void (*callback)(void), int repeatCount);
int8_t after(unsigned long duration, void (*callback)(void));
int8_t oscillate(uint8_t pin, unsigned long period, uint8_t startingValue);
int8_t oscillate(uint8_t pin, unsigned long period, uint8_t startingValue,
  int repeatCount);
int8_t pulse(uint8_t pin, unsigned long period, uint8_t startingValue);
int8_t pulseImmediate(uint8_t pin, unsigned long period, uint8_t pulseValue)
  ;
void stop(int8_t id);
void update(void);
void update(unsigned long now);
```

## 3.6 Hardware Tests

In order to get knowledge of and experience with the hardware three tests have been devised. The first test is conducted to validate the results of the following tests, and it scrutinizes the hardware of the same type for inconsistencies. The following tests examines for the speed of the devices and the reliability with antennas attached to the RF-modules of different lengths. The full tests is documented in **??**, **??** and **??**; this section will present the results of the tests, and explore its implication on the project.

### Radio Frequency Module Difference Test (full test in ??)

As previously stated this test is used to validate all other tests involving the radio frequency modules. Because seemingly identical hardware can act very different in practice, due to minuscule differences introduced in the manufacturing process, it is necessary to be able to provide some guarantee that the differences does not effect the use of the devices to significantly. In other words to be able to compare results gathered with different radio frequency modules, one must be confident that a given result can be reproduced by any of the RF modules. There is of course a margin of error which must be accepted, or else none of the modules would be declared suited for use in further testing.

The results of this test shows that there are three modules, which has a significant amount of package loss. These three modules are two receivers and one transmitter, and will be excluded from any other tests. Because of this exclusion the project group can guarantee that other test results are not significantly influenced by differences in the radio frequency modules.

### Radio Frequency Module Reception Test (full test in ??)

To provide better reliability in reception of the messages sent with the radio frequency modules, different antennas is tested at different distances. The goal is to find the optimal antenna length i.e. the antenna length that causes the least package loss. Both transmitters and receivers are fitted with varying lengths of antennas, in a way so all combinations of antenna lengths are tested. In the test the distances that were tested ranged from 2 to 28 metres between the transmitter and the receiver, and the varying antenna lengths were:

**0 cm** No external antenna

**12 cm** An arbitrary antenna length

**17.3 cm** A quarter-monopole antenna (at 433 MHz)

The results of this test shows that the theory, which states that the quarter-monopole antenna at 433 MHz should be 17.3 cm, holds up in practice. At all distances the antenna length of 17.3 cm gives a significantly lower package loss percent. In fact, when the transmitter is fitted with a 17.3 cm antenna even the receiver with no external antenna has a relatively low package loss percent. However all the tests also shows that the arbitrary antenna length does not work well and actually introduces package loss in comparison to no external antenna and the 17.3 cm antenna.

Because of these results all radio frequency modules will be fitted with antennas of 17.3 cm length. This will give the most reliable transmissions and receptions between devices using said modules.

### RadioHead Time Sent Test (full test in ??)

The last hardware test will assess the speed at which a message of $n$ bytes can be sent using the communication library for Arduino called RadioHead. This test is performed to extract a formulae, which can be used to calculate said speed; as it is advantageous to know when a given message was sent, in order to better synchronise devices.

The results of the test shows that the time it takes to send a message is linearly proportional with size of the message in bytes. Moreover the following equation can be derived from the results:

$$f(n) = 6.0101 * n + 65.7826 \tag{3.4}$$

Where $n$ is the number of bytes in the message and $f(n)$ is the time it takes to send the message in milliseconds. This equation shows that any transmission has an overhead of 65.7826 milliseconds, and that for every byte it takes an aditional 6.0101 milliseconds.

# Part II

# Designing and Implementing the Protcol

# Chapter 4

# System Requirements

This chapter will sum up the requirements made for the protocol in accordance with the analysis in the previous chapters.

Requirements for a system are the descriptions of what a system should be able to do, and also the constraints the system faces. In general there exists two kinds of software requirements, functional requirements and non-functional requirements. Functional requirements can be simplified to how a system should react when given certain input, or how it should react in certain situations. Non-functional requirements can be simplified to being the constraints on the system, e.g. timing constraints or constraints created because of hardware. This definition is based on the work by **SEBook** in his book *Software Engineering.* For this system, there are a few non-functional requirements. All which causes more functional requirements.

Non-functional requirements according to the information gathered can be expressed as:

*1)* Devices can malfunction, run out of power or in some other way die.

*2)* Devices have limited transmission range.

*3)* Devices use a single radio frequency.

*4)* Packages can be corrupted or invalidated in some other way.

*5)* Transmission is not instantaneous.

*6)* Clock may become less accurate the longer it runs.

Functional requirements according to the information gathered in the analysis in the previous chapters and by the non-functional requirements can be expressed as the following:

*a)* Being the first device in the network, this device should create the network so new devices can connect to the network in a time slot available to new devices.

*b)* A device should be able to join the network, and announce that is is now part of the network, such that other devices know it is connected.

*c)* A device should be able to transmit messages in its specific time slot.

*d)* A device should be able to receive messages when it is not the device's turn to transmit messages.

*e)* A device should be able to execute user code so that it can use its actuators to perform certain tasks.

*f)* Devices should be able to tell when a device has been removed from the network. Pertains to non-functional requirement *1*

*g)* The devices should be able to repeat a messages from other devices. Pertains to non-functional requirement *2*

*h)* The devices should be able to reliably communicate despite a packet loss up to 5%. Pertains to non-functional requirement *3*

*i)* A device should not transmit when the current time slot is not that device's dedicated time slot in order to avoid jamming. Pertains to non-functional requirement *4*

*j)* The network should be able to sync relatively often in an effort to avoid desynchronize. Pertains to non functional requirements *5* and *6*.

These requirements can all be tested when the project has reached its end, to see whether what was desired has been achieved, and if so how. In the next section the development plan for the project is specified.

# Chapter 5

# Development Plan & Problem Exploration

This chapter will investigate how the protocol can be perceived mathematically as a graph, and how it can be divided into simpler subproblems which can be solved iteratively.

The requirements stated in the previous chapter, can be modelled as a graph problem where a weighted directed graph $G = (V, E)$ describes the network. The vertices, $V$, are the devices and the edges, $E$, are the communication paths between the devices. The weight, $W(v_1, v_2)$, is the probability of a message from $v_1$ successfully being received by $v_2$. An example of this can be seen on Figure 5.1. For this project only networks which can be depicted as a tree will be considered, since a forest essentially describes at least two separate networks. Figure 5.1 is an example of a strongly connected network which can be described as: $E \subseteq V \times ]0,1] \times V$ where for any $v_i, v_j$ there exists a path from $v_i$ to $v_j$ with the reliability $r$ which is a number in the range $]0,1]$.

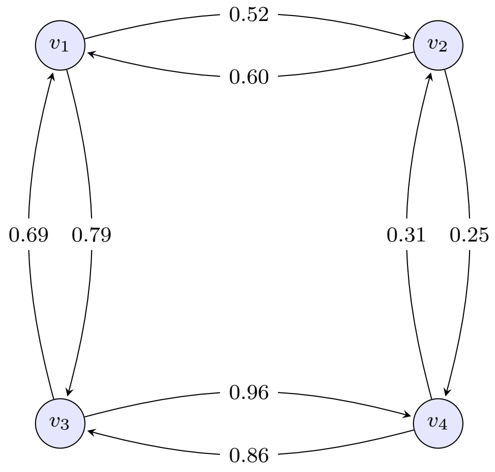


**Figure 5.1:** An example of a strongly connected unreliable network with probabilities modelled as a graph.

This is a complex problem to solve and many issues will become apparent throughout analysis of the problem. In order to simplify the problem some assumptions are made about the network. We assume that the devices will always be in range of each other; this means that all vertices on the graph have a non-zero edge between all other vertices on the graph. Secondly during development of the system, the system could be set up to give ~100 % probability of successfully receiving the message, by wiring the Arduinos together instead of using radio communication; this gives more room to work on the baseline communication issues without having to consider the unreliability of the radio frequency modules. Furthermore the networks can be set up as either fully connected graphs or strongly connected graphs. This creates the four sub problems which can be modelled as graphs:

**CCRC-problem:** Completely connected reliable communication problem

$$\text{iff } \forall\, v_i, v_j \in V\, \exists (v_i, r, v_j) \in E \mid r = 1 \tag{5.1}$$

The CCRC-problem describes how all vertices are directly connected with each other, creating a completely connected graph, and all the weights of the edges are also 1, which means that all transmissions will be received.

**CCUC-problem:** Completely connected unreliable communication problem

$$\text{iff } \forall\, v_i, v_j \in V\, \exists (v_i, r, v_j) \in E \mid 0 < r \leq 1 \tag{5.2}$$

The CCUC-problem describes a completely connected graph however in this scenario there is not a 100% chance the transmission will be received; all vertices are still in range of each-other.

**SCRC-problem:** Strongly connected reliable communication problem

$$\begin{gathered}\text{iff } \forall\, v_i, v_j \in V \text{ there is a directed path from } v_i \text{ to } v_j \wedge \\ \forall\, e \in E \mid W(v_q, v_r) = 1\end{gathered} \tag{5.3}$$

The SCRC-problem describes a strongly connected network meaning at least one vertex has an indirect connection to another rather than a direct connection, i.e. there still exists a path from every vertex to every other vertex but at least one pair of vertices requires one or more mediating vertices. In this example communication is once again reliable such that all transmissions will be received.

**SCUC-problem:** Strongly connected unreliable communication problem

$$\begin{gathered}\text{iff } \forall\, v_i, v_j \in V \text{ there is a directed path from } v_i \text{ to } v_j \wedge \\ \forall\, e \in E \mid 0 < W(v_q, v_r) \leq 1\end{gathered} \tag{5.4}$$

The SCUC-problem describes the most realistic scenario, a strongly connected network with a chance of not receiving transmissions.

In order to simplify the development of the project, the problems will be handled seperately and in order. This means starting with the CCRC-problem and working towards solving the SCUC-problem. Thus the solution will go from a narrow minded solution to a more broad and general solution each iteration. This approach increases the probability of having a working solution at the end of this project without over- or underestimating the workload.

# Chapter 6

# Design of the protocol

## 6.1 Protocol

The development of a protocol is influenced by the purpose of the protocol. For the protocol documented in this paper, the primary objective is to create a completely connected reliable communication network of Arduinos i.e. a network of Arduinos able to communicate with an arbitrary number of devices and allowing devices to join or leave. The protocol should general in purpose, with the possiblites of using it for different usecases. This should be archieved by designing the protocol as a libary and thus allowing usecode for the speicfic usecases.

### 6.1.1 Payload

The first matter to discuss will be the data to be sent between Arduinos i.e. the payload. When developing for an Arduino it is important to design for the limited resources available. Therefore it is advised to keep the amount of resources used small. Additionally the transmission time increases as the size of the payload increases, this will further escalate the risk of packet loss. Thus a balance between the size of the payload, the speed of transmission and the reliability of transmissions should be met.

The amount of data in the payload largely depends on the complexity of the network itself, and the devices that make up the network. An example would be a sensory network which observe one thing, e.g. a fire alarm. The only variable required here, aside from those used for communicating, is whether or not a fire has started, which would result in very little data transmission.

This is one use case, however more complex networks should also be supported. In an effort to allow for such a variety in networks an abstraction of the data will be made, and as such the only consistent data being sent in a transmission will be the information needed for communicating on the network. A figure showing the design of the payload can be seen on Figure 6.1.

| | | | |
|---|---|---|---|
| **Current slot** | **Slot Count** | **Address** | **Data** |

**Figure 6.1:** Payload with information needed to communicate on the network, along with some data to be used by the Arduinos in the user code.

The payload has the four variables: slot, slot count, address and timestamp, which is to be used to communicate in the network. The slot is used to indicate which slot the frame is currently in, while slot count is the total number of slots. The payload allows the protocol to abstract from what data is being sent, and as such any data can be send as long as it is of reasonable size. Examples of data could be sending a message to turn on some LED or maybe requesting other devices for their sensor outputs. Address is much like a mac address and is unique for all Arduinos, and is used to differentiate between Arduinos to tell them apart. This could instead be included in the Data as some implementations with the protocol might not need to differentiate between the Arduinos, however this is a more general solution for the protocol. Since the size of the address is finite, and will most likely be implemented as a 1 byte unsigned integer. A 1 byte address would allow 256 addresses, however a few may be reserved for special uses. The timestamp is used to synchronize the units. This payload is reasonably small, and the size of the data field will depend on the user code on the device.

### 6.1.2 Frames

While the concept of TDMA is widespread, a general implementation is not; which is partly why so many variations of TDMA exist. For this section we will focus on the design and requirements set by the frame as well as an individual time slot. The design of the frame itself refers to the relative location of each occupied time slot including one or more unused slots which new devices would seize. The specifics of adding and removing devices from the network and how those actions affect the frame will be discussed in sections referring to those individual problems. This section will focus on the design of the frame and not how events in the network may alter it. Depending on which of the model from Chapter 5 we take root in, different problems are worth considering. For the base example, completely connected and reliable communication, the only real concern is when to place the open slot. In a larger network it will take longer for a device to join allowing for possible more devices to be activated within a single frame, thus resulting in several devices attempting to join. However if we once again return to the example of a fire alarm, several devices turning on within seconds of each other in a large network seems an unlikely scenario for other than an initial setup of the network. Also since the only effect is a slight delay in joining the network, and this would rarely occur outside of the startup network, it is less significant; as such the design will only contain one empty slot at a time as portrayed on Figure 6.2. As the figure also shows each device will have one slot per frame, in the case where the network is not completely connected but strongly connected one may consider changing this.

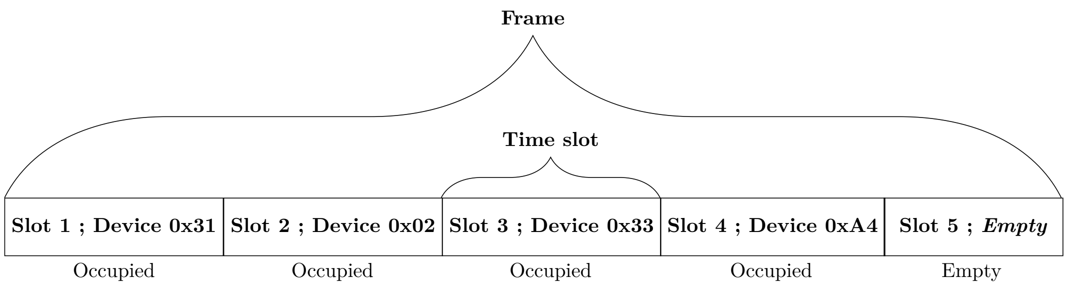


**Figure 6.2:** A visual representation of the frame with 5 time slots where of one is empty

## Time Slot

For each time slot only one device may transmit, as such it is important that the time slot provides enough time for a device to transmit data, and equally as important that any computations can be executed before the next time slot. As such each time slot is composed of two phases; communication and processing. In the communication phase the device will either have to transmit or receive data thus the length of this phase should be long enough to receive data from the device that may take the longest to transmit, i.e. this phase has to be uniform across all time slots. The length of the communication phase is denoted by: $\delta_{com}$.

For the processing phase one should be able to ensure that a transmission is possible for each time slot. To achieve this several possibilities of how to determine the length of this phase present themselves; in an effort to broaden the applicability of the protocol it has been chosen that this too, should be uniform and determined by a worst case scenario. The processing phase in each time slot will be determined by comparing worst time scenarios for each device and using the worst of these. The length of the processing phase is denoted by: $\delta_{proc}$. This ensures that a device can perform its computation within its own time slot, while allowing for the opportunity to receive and compute commands when it is not the turn of the device, Figure 6.3 represents the design of the time slot. This time slot design will also be dynamic, which means that in the case a device joins with a worst case computation longer than what the time slot provides in computation time, every time slot in the frame will have to be altered. The total time of each timeslot is denoted by: $\delta$.

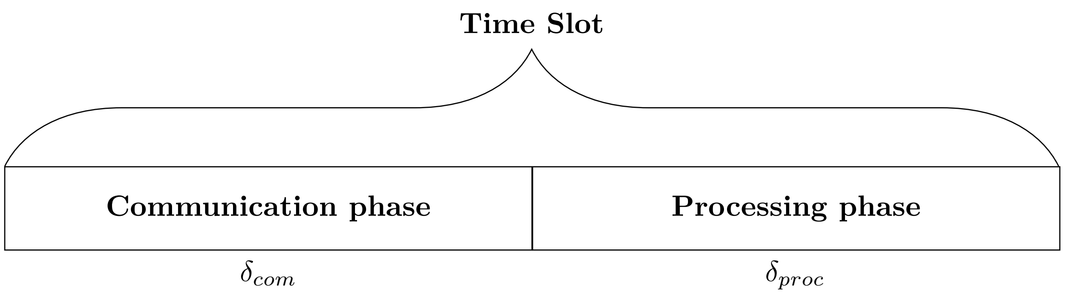


**Figure 6.3:** A visual representation of the time slot, $\delta_{com}$ and $\delta_{proc}$ denotes the time dedicated to each phase

## 6.2 Pseudocode Description

This section presents key components of the designed TDMA-based protocol. First an informal description is given followed by relevant flowcharts and pseudocode examples. The model of the protocol is divided into the two phases - the initilization phase and the loop phase.

These phases will be described in the following subsections using the same model for a general device. That device can only know its local variables and what it receives via the ‘received’ function.

| Name | Type | Description | Constraint |
|---|---|---|---|
| $n$ | integer | The number of slots in the network | $2 \leq n \leq 256$ |
| $k$ | integer | The slot claimed by this device | $1 \leq k \leq n-1$ |
| $i$ | integer | The current slot | $1 \leq i \leq n$ |
| $\delta$ | integer | The duration of a time slot | $\delta = \delta_{com} + \delta_{proc}$ |
| $\delta_{com}$ | integer | The duration of a time slot reserved for communication | $0 < \delta_{com}$ |
| $\delta_{proc}$ | integer | The duration of a time slot reserved for processing | $0 < \delta_{proc}$ |
| $t_0$ | integer | The minimum time to determine if a network exists | $3 \times \delta \leq t_0$ |
| $x$ | clock | A clock used for timing, and always running | |

**Table 6.1:** The local variables every device has access to.

The devices store their knowledge in their local variables as seen in Table 6.1. The aim of the network should be to manipulate those values while running, to satisfy the statements seen in Table 6.2. This should ensure that two devices will not send messages at the same time.

$$\begin{array}{lll}
(a) & \forall \{d_a, d_b\} \subset Network: & d_a.k \neq d_b.k \\
(b) & \forall \{d_a, d_b\} \subset Network: & d_a.n = d_b.n \\
(c) & \forall \{d_a, d_b\} \subset Network: & d_a.i = d_b.i \\
(d) & \forall\, d \in Network: & d.k \neq n \\
(e) & \forall\, k \in \mathbb{N}_{<n}: & \exists!\, d \in Network: d.k = k
\end{array}$$

**Table 6.2:** The requested situation where $Network$ is the set of devices currently connected in a network.

This is accomplished by having the devices agree on the number of slots and current slot while no devices have the same slot. Finally, no device should occupy the last slot as this is used by new devices to connect to the network.

The devices are assumed to have an implementation of the ‘receive’ and ‘transmit’ function. The ‘transmit’ function is simple, it sends a message over radio waves containing the parameters parsed into it. The ‘received’ function is harder, it takes some parameters which are set to the received values if any was received. It then returns true if a message was received; otherwise false.

### 6.2.1 General Case

During the lifetime of the devices, they should spend most of their time in this general case or in the user code. This code determines whose turn it is, synchronises devices, and handles communication.

As seen in Figure 6.4 the flow of the code is a straightforward loop with a few logical checks. ‘Make payload’ and ‘Process message’ are the only complicated processes. ‘Make payload’ collects the known data into a package, that is ready to send. ‘Process message’ then receives one of such packages and unpacks it. Unpacking would usually involve storing the data on the device.

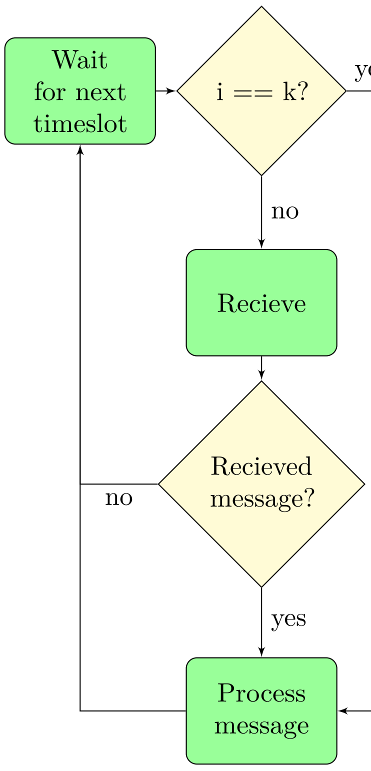


**Figure 6.4:** Flow diagram showing how a device acts when connected to a network

This loop should only be stopped when the network is reconfigured. Reconfiguration of the network involves setting up and connecting to a network. These processes is described in **??**.

```
 1 procedure mainLoop()
 2     while true do
 3         loopstart:
 4         run userCode() until $x \geq \delta_{proc}$
 5         $i \leftarrow (i \bmod n) + 1$ // Update current slot
 6         if $i = k$ then
 7             makePayload() // Updates the data to be sendt
 8             transmit($i$, $n$, $id$, $data$)
 9             $x \leftarrow 0$ //Reset the clock to synchronize
10         else
11             while $x \leq \delta$ do
12                 if received($i'$, $n'$, $id'$, $data'$) then
13                     $x \leftarrow 0$ //Reset the clock to synchronize
14                     processMessage($id'$, $data'$)
15                     goto loopstart
16                 end
17             end
18             $x \leftarrow 0$ //Reset the clock to synchronize
19         end
20     end
```

**Listing 6.1:** Pseudocode example of the main loop

The full pseudocode for the general case can be seen in Listing 6.1. In the code, there are a few extra details but the main structure is the same. The code describes how a device knows whether it is its turn or not.

What is worth noting is the clock $x$ which controls the timing. It is reset at three different places in the code; on line 9, 13, and 18. The first two resets on line 9 and 13 are just before sending and just after receiving a message. This is the closest point in the program we can synchronize to.

The last reset on line 13 is when no message was received in the timeslot. In this case, the device should reset after the calculated $\delta_{com}$ duration. In a completely reliable network, this should only occur in the empty slot.

### 6.2.2 Initializing

When first turning on a device, which should be part of the network, the first task of the device must be to try to connect to an existing network. First the device will be set to listening for any signal for a given time period. If the device does not pick up any transmission during this period the device will start initialising its own network.

When a device creates a network, the network contains two times slots; one for the device itself and one empty slot, which allows another device to connect to the network. When the frame is in the device's own time slot it will transmit so new devices can detect said network, when they are listening in their startup period. A flow diagram of this setup can be seen on **??**

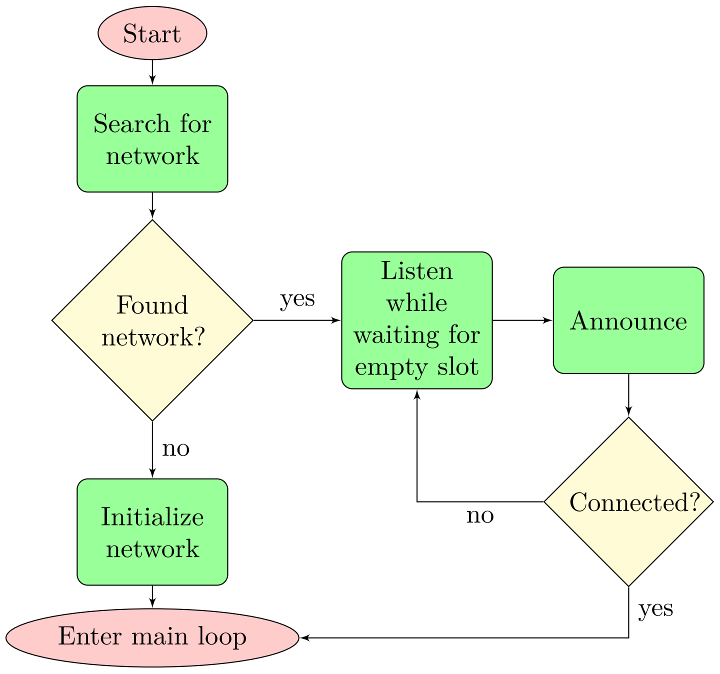


**Figure 6.5:** Flow diagram showing how a device acts during the Initialization phase

`Initialize()`, the function in **??**, performs the initialization of the device. This includes searching for a network or establishing its own network if no network can be detected. **??** introduces are new local variable $t_0$, this constant decides when the device should give up listening for a network and instead establish a network. For the completely connected reliable communication case represented in Chapter 5 $t_0$ has to be at least the length of $2 \times \delta + \delta_{com}$, however it being slightly longer taking guard time and the chance of a slight miss synchronization into consideration $3 \times \delta$ will be considered the minimum. The minimum is calculated from the case where the device is turned on after the $n-1$'th slot has started, but not yet ended. It is worth noting here that this minimum is only true for this case, both strongly connected and non-reliable communication present their own reasons for this not being true for those cases.

If a network is found prior to $x$ exceeding $t_0$ the device will process the data and use this to properly attain a slot in the network as is seen on lines 7 - 9 in **??**. Should it occur that no network is found the device will start its own network with the initial configuration that lines 14 - 16 in **??** shows.

```
procedure Initialize()
    $x \leftarrow 0$
    while $x \leq t_0$ do
        if received($i'$, $n'$, $id'$, $data'$) then
            $x \leftarrow 0$ //Reset the clock to synchronize
            //Prepare variables for transmit in emptyslot
            $n \leftarrow n' + 1$ // Increment number of slots
            $k \leftarrow n'$ // Claim empty slot
            $i \leftarrow i'$ // Get current slot
            mainLoop()
        end
    end
    // Create new network
    $n \leftarrow 2$ // Two slots; one for this device and one empty
    $k \leftarrow 1$ // Claim the first slot
    $i \leftarrow 0$ // Set current timeslot to one before me
    $x \leftarrow 0$ // Reset clock
    mainLoop()
```

**Listing 6.2:** Pseudocode example of the special case procedure Initialize()

## Multiple Startup Issue

A major complication that may be encountered when starting a device is the possibility of multiple devices starting at once. In order to guarantee a working network starts the devices cannot create a network at the same time or they will simply be jamming each other such that no new device will ever join. The CCRC-problem has until now assumed perfect communication as its first priority is guaranteeing that a network can be established and transmit messages in an established network using single frequency transmitters and receivers. As such for the CCRC-problem another assumption will be added, only one device will be turned on within a frame, which in turn nullifies this issue. The CCUC-problem focuses more on the uncertainties of communication in the network and will in keeping with that also consider this issue.

# Chapter 7

# Model-checking in UPPAAL

Model-checking consists of verifying whether or not certain properties hold for a given modelled system, or protocol, i.e. can the system deadlock or end up in other unwanted states. Doing this by hand can be a faulty and exhaustive process, therefore tools have been invented to assist in this process. In the following sections the tool UPPAAL will be described and used to model and verify the protocol used in this project. The tools might not be able to accurately describe reality as there are made some assumptions, however using a tool like this is a great help in producing a correctly working protocol.

## 7.1 UPPAAL

UPPAAL have been used to find problems with and verify a wide range of protocols and applications. This includes and is not limited to:

*a)* TDMA Protocol Start-Up Mechanism [**Lonn:1997:FVT:826040.827011**]

*b)* Bang & Olufsen audio/video protocol [**Havelund97formalmodeling**]

*c)* Car supervision system [**gebremichael2004formal**]

The UPPAAL application is a GUI interface made in Java, and the verification is done with an engine written in C++. UPPAAL works by consisting of several models each of these is a timed automata, which is a finite-state machine with clocks. All clocks in the system progress synchronously. This means that a UPPAAL model consists of states and edges or transition, each of these edges can fire.

In UPPAAL it is possible to make queries, to verify whether certain properties are true or not for a given model. It has a special syntax, for example: **A[] not deadlock** which

What is UPPAAL? [**tutorial04**]

*"UPPAAL is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. [...] The tool is designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronisation."*

asks if for all states there is no deadlock. **E<>** can be used instead of **A**[] and this asks if there exists a path of transitions so that a property is true. One final example is **A<>** which express that for all paths the property will at some point in time be true.

### Limitations of Model-Checking

It may not be possible to represent all variables a real system is affected by in a model-checker like UPPAAL. It is not possible to guarantee that the protocol, the model and the application are equivalent, this means that even if the protocol and/or model is correct then the implemented application may not be. Deciding this is an undecidable problem within computer science, often referred to as $EQ_{TM}$, explained further in *Introduction to the Theory of Computation* by Michael Sipser [**Sipser**]. Therefore some assumptions must be made. The first assumption for our project is that our protocol, model and application will be equivalent. Other assumptions for the implementation of the protocol are the following:

*a*) The network is strongly connected

*b*) Any transmission sent is also received by the other devices

*c*) There is some mechanism to synchronise the clocks of the devices

*d*) Only one new devices is introduced to the network at a time.

## 7.2 UPPAAL Models

This section will specify what an UPPAAL model consists of, how it is constructed and how it works.

An UPPAAL model consists of one or more templates, which are timed automatas, all templates can have local code, while it is also possible to create global code available to all templates, which are simply called declarations in the UPPAAL GUI. Both the global and the local code, which is written in a language made for UPPAAL, can have variables and functions which can be called by instances of their respective templates. The UPPAAL modelling language has certain special types like `clock` and `channels`. A clock is what makes the automatas timed, and is a variable which can consist of natural numbers, and it is continuously incremented to indicate that time is passing. A channel is used to synchronise instances of templates so edges fire at the same time. Instances of the templates are created in a different section called system declarations, which is used to setup the system for testing. The templates all have an initial state like other automatas. All states and edges can have certain properties to control the transitions in the automata which is how the model for the protocol can be checked. The states in UPPAAL templates are often called locations instead of states.

The properties of a state are the following:

**Name**
These are used to refer to certain states from either the code for the model, or in the verification aspect of UPPAAL. It is also useful for documentation of the model. The color of names are dark red in all UPPAAL models.

**Invariants**
Are used to keep a template in the state until an expression no longer holds true. The invariants often make use of the clock type available in UPPAAL to make sure a template will leave a state after a certain time has passed. The color of invariants are a pink-ish color in all UPPAAL models.

Other types of properties for states are toggled variables which change how a state is computed:

**Initial** states indicates where the template will start, and works just like in other state Automata which has to have an Initial state. This is marked by a double circle on the state.

**Urgent** states freeze time for the template, which means that all clocks for a template in an urgent state will not change. This is marked by a U on the state.

**Committed** states also freeze time, but it does so for the entire system, as if any template is in a commited state, the next edge to be fired has to be an edge from a committed state. This is marked by a C on the state.

Edges can have the following properties:

**Select** is used to non-deterministically give an identifier a value in a range, and is shown with a green-yellow color.

**Guard** is used to make sure an edge can be used unless a certain expression is upheld. Guards are shown in a green color.

**Sync** is used with the channel variables to make sure an edge is fired in all available templates. If an edge with the sync `chan!` is fired all templates which are in a state with a legal transition `chan?` has to be fired. This makes sure all templates make a transition when available to. Syncs are shown in a light-blue color.

**Update** is used to change variables when an edge is fired, either to give them a certain value or maybe to increment or decrement variables. Updates are shown in a purple color.

Now that UPPAAL has been introduced, it is time to look at the model of the protocol, so the protocol can be verified to uphold the requirements from Chapter 4

## 7.3 The Model

This section will showcase the model made to express the design from Chapter 6. For simplicity the introduction of the model in this chapter has split the model into two parts, an initializing phase and the main loop. The complete model can be seen in **??** along with the code for the global and local declarations, including the functions used throughout the model.

The first part to be shown will be the initialization, which can be seen on **??**

The initial state has a clock `startup`, which makes sure that only one of the devices will be leaving the initial state at the same time. When one of the devices leave this state, `startup` will reset, and another will leave the state when the `startup` once again reaches the desired time. It has to be mentioned that this `Startup_Time` has to be increased as the

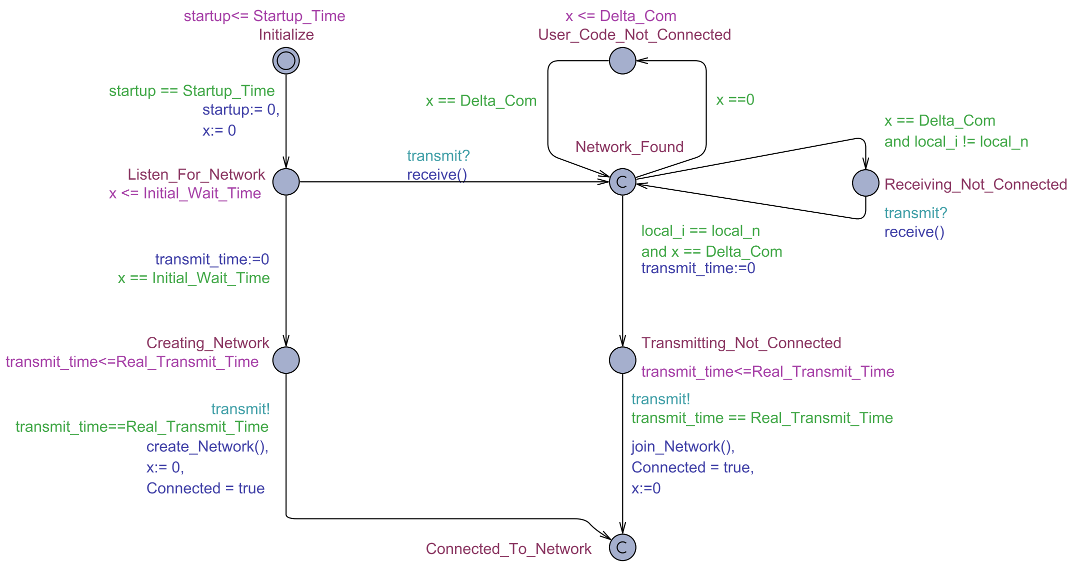


**Figure 7.1:** UPPAAL Model showing how the devices initialize.

number of devices in the system increases. This is because when the frame of the network becomes larger, `Startup_Time` has to be increased to make sure that only one device has been released at a time for when en empty time-slot occurs.

With the current values in the model creating a network with six devices is doable. While the other devices are waiting to be released, the device which was the first to leave will listen for a network, as there is obviously no network yet the device will fire the edge towards the `Creating_Network` state. The device will setup the initial values for the network, change its boolean `Connected` to true, and start the main loop. When the next device leaves the initial stage, it will when it is listening for a network be successful as a network has just been created, which means that the device will move towards the committed state `Network_Found` and reset the clock `x` to zero. This state is committed as its purpose is simply context switching between receiving and performing user code. When it has just received a transmission, the device which just transmitted will according to Chapter 6 be performing user code, and so should the device trying to connect.

When `x` is equal to `Delta_Com`, which is the time given to a device to perform user code, the device will change to the state `Network_Found` once again. In this state when `x` is not zero, the device checks whether `i`, which is the current time-slot, is the empty-time slot, if it is the device will transmit, and ultimately connect to the network, increasing the number of time-slots in the network according to the specifications from Chapter 6. If the empty slot was not the current time-slot, the device will instead go to the state `Receiving_Not_Connected` where it will receive the other devices' transmissions, and once again reset `x` to zero, and perform user code, until eventually the empty-slot occurs, where it will connect to the network.

The committed state `Connected_To_Network` has 3 edge leaving it, one for receiving, one for transmitting, and one of executing user code. The model can be seen on **??**.

It works the roughly the same as the state `Network_Found`, except the guard checking whether it is the empty time-slot instead checks whether it is the device's time-slot. Another change is when the device is in the state `Receiving`, if nothing has been transmitted and the clock `x` is equal to `Delta` the device will go increment its `i_local` value and go perform user

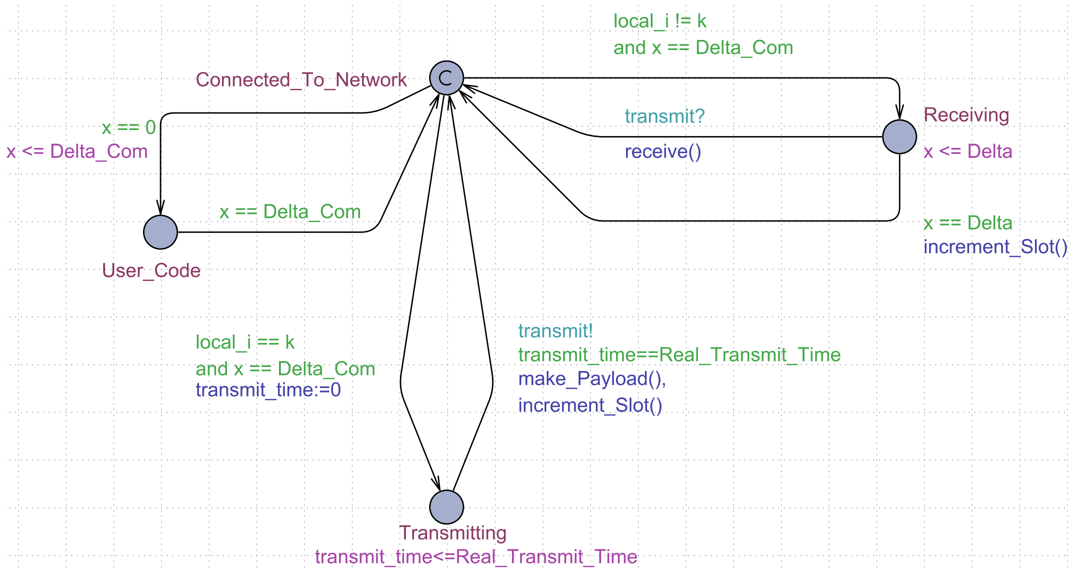


**Figure 7.2:** UPPAAL Model showing the devices' main loop.

code. This case happens whenever the empty slot is the current time-slot and no new device is trying to connect to the network. For a specification of all the functions being used in this model, please have a look in **??**.

## 7.4 Verifying the Model

As mentioned in **??** it is possible to make queries to verify that some given properties are true. Section 6.2 contains statements which should be true for a correctly connected network. These statements can be written as queries in UPPAAL, which the tool will then determine to be correct or not. The queries in UPPAAL can be seen on **??**.

```
1. A<> forall(i: id_t) Device(i). Connected
2. A[] not deadlock
3. A[] forall (i : id_t) forall (j : id_t) Device(i). Transmitting and
        Device(j). Transmitting imply i == j
4. A<> forall(i : id_t) forall (j : id_t) (Device(i). User_Code and
        Device(j). User_Code and Device(i).local_i == Device(j).local_i)
5. A<> forall(i : id_t) forall (j : id_t) Device(i). local_n == Device(j).local_n
6. A<> forall(i : id_t) forall (j :id_t) Device(i). k != Device(j). local_n
7. A<> forall(i : id_t) forall (j : id_t) Device(i). k == Device(j).k imply i == j
8. A<> n == N+1
9. A<> forall(i : id_t) Device(i). k < n and Device(i). k > 0
```

**Listing 7.1:** Queries for the UPPAAL Model

All of these queries will yield a true result when run with 6 devices on the model which has been shown. More queries have been added to verify the model, an example is query 2. A brief explanation of each query will be given below.

**Query 1.** asks if for all possible state transitions, will it at some point be true that all devices in the system will be connected.

**Query 2.** asks if there are no deadlocks in the model.

**Query 3.** determines whether two different devices can be in the state `transmitting` at the same time.

**Query 4.** will decide whether the devices local number of i, is the same when they are all in the state `User`$_C ode.asksifalldeviceshasthesamevalueofninthelocalvariablen.$

**Query 5** **Query 6.** asks if it false that any device has taken the empty time-slot as their own time-slot.

**Query 7.** asks if any two devices have taken the same time-slot k.

**Query 8.** asks if at some point is it always true that the number of time slots in the network is one larger than the number of devices in the system.

**Query 9.** asks if all the time-slots in the network are less than n and larger than 0.

The queries 7-9 together make up the statement (e) from **??**, since if no devices have the same time-slot k and if the number of devices is one less than the number of time-slots and all time-slots are in the range [1,n-1] all number in this range must be a time-slot in the network.

Since all of these statements hold true, it is concluded that the model does indeed do what it was designed to do, and an implementation according to the specification should theoretically work and be possible to create.

# Chapter 8

# Implementation

# Chapter 9

# Iterations of the Problem

This chapter will investigate the expansions of the protocol discussed in Chapter 5. The protocol developed so far has been for the *completely connected reliable communication graph* The first one to be designed in this chapter is for the *completely connected communication graph* which means it is not 100% reliable.

## 9.1 Completely Connected Communication Graph

Write this at some point

# Chapter 10

# Evaluation

# Chapter 11

# Discussion

# Chapter 12

# Conclusion

# Chapter 13

# Future Works

# Part III

# Appendix

# Appendix A

# Radio Frequency Module Difference Test

In tests using the RF Modules the difference in build quality and performance between the modules is a significant factor, and could be a source of error. Consequently a test must be performed, which investigates these differences, so that the other test results can be validated.

## Test setup

To test all RF modules available to the group two Arduinos are used; one as receiver and the other as transmitter. The RF modules are then switched in and out of breadboards connected to said Arduinos, so that both receiver and tranmitter modules can be tested independently. Furthermore a third Arduino is used to verify the results of each module. The software on the Arduinos is the same as in the test in **??**, i.e. a transmitting part which sends 100 unique packages and a receiving part which registers any package loss. Firstly modules is randomly tested to find a transmitter that shows very little to no package loss, thereafter every receiver is tested using the "good" transmitter; with "good" meaning a package loss very close to 0 %, maximum 2 %. The same thing is done to test the transmitters, i.e. a "good" receiver is used. This approach is then repeated multiple times with different "good" receivers and transmitters, to ensure a more fair test environment, with multiple combinations of modules.

## Results

The results have been plotted on two bar charts: one with the results of the different transmitters;

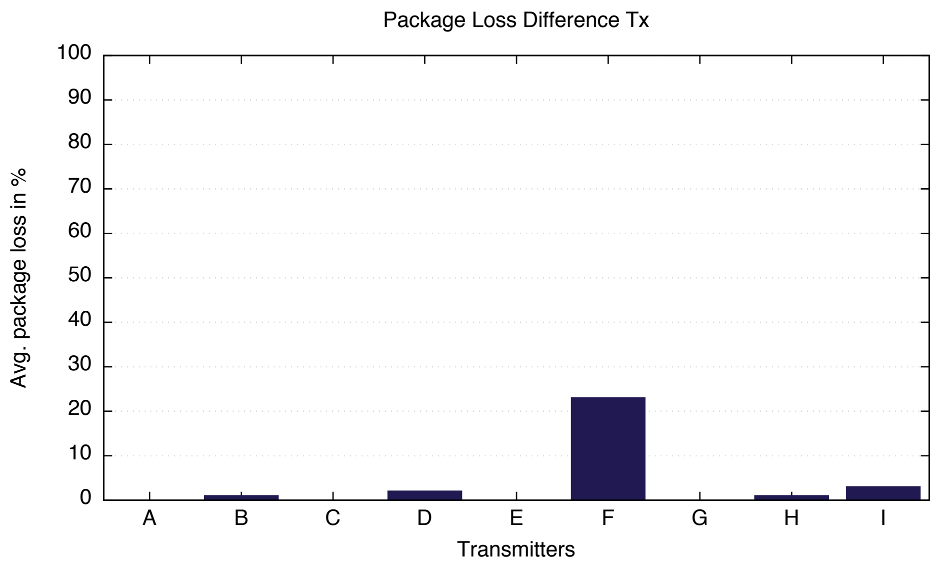


**Figure A.1:** The average package loss inflicted by the transmitters.

and one with the results of the different receivers.

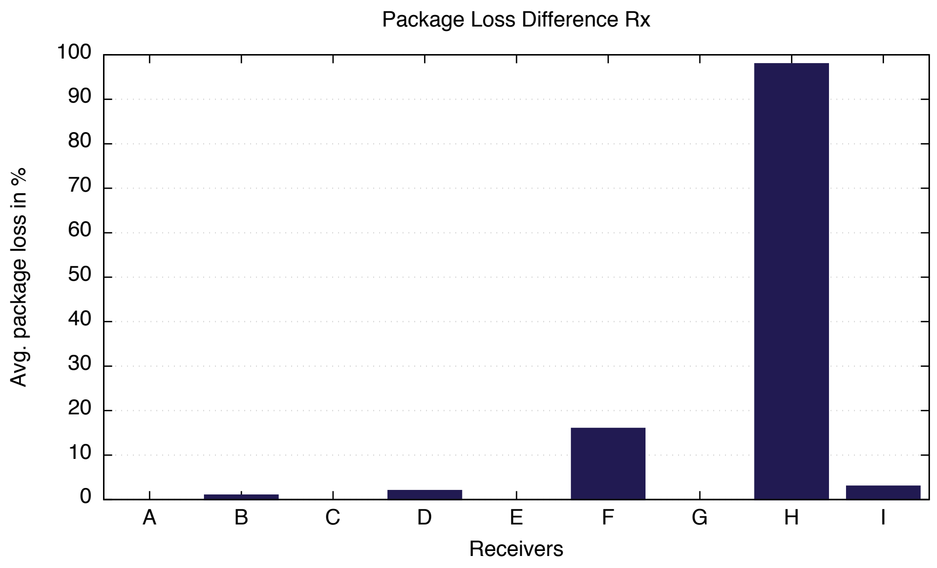


**Figure A.2:** The average package loss inflicted by the receivers.

## Conclusion

When looking at the results of this test, it is clear that some of the modules were significantly worse at respectively transmitting and receiving data. Transmitter F inflicted an average package loss over 20 %, and receiver H failed to receive more than 2 % of the packages on average. Moreover receiver F had an average package loss just below 20 %. However to ensure validity of other tests using the RF Modules, the group will exclude the three faulty modules (transmitter F, receiver H, and receiver F) from these tests.

# Appendix B

# Radio Frequency Module Reception Test

In order to test how the Radio Frequency Modules (RFM) works, a test have been devised. The purpose of this test is to test the range of the RFM's with different antennas and across different ranges. The test will use the RadioHead libary as described in 3.5.2. Built into RadioHead is a CRC check, this insures the integrity of the message received.

## Test setup

We will test three different antennas

*a*) 0 cm (No external antenna);

*b*) 12 cm (An arbitrary length);

*c*) 17.3 cm (Using Equation (3.1))

Each will be used on both the receiver and the transmitter. This yields six combinations which will be tested. Multiple receivers can receive the same message, therefore all receivers can be tested at the same time. We have tested with distances from 2 metres to 28 metres, except for the test with a 0 cm antenna on the transmitter where a high package loss was occurring even at ranges of 14 metres. The transmitter was placed on a table 1 meter above the ground, and a graphical circuit diagram of an Arduino fitted with a transmitter can be seen on **??**. As is shown on the figure an LED was also connected to the Arduino. This LED was used as a visual aid blinking every time a package was send, this has no real influence on the test and was used as a convenience for those who tested.

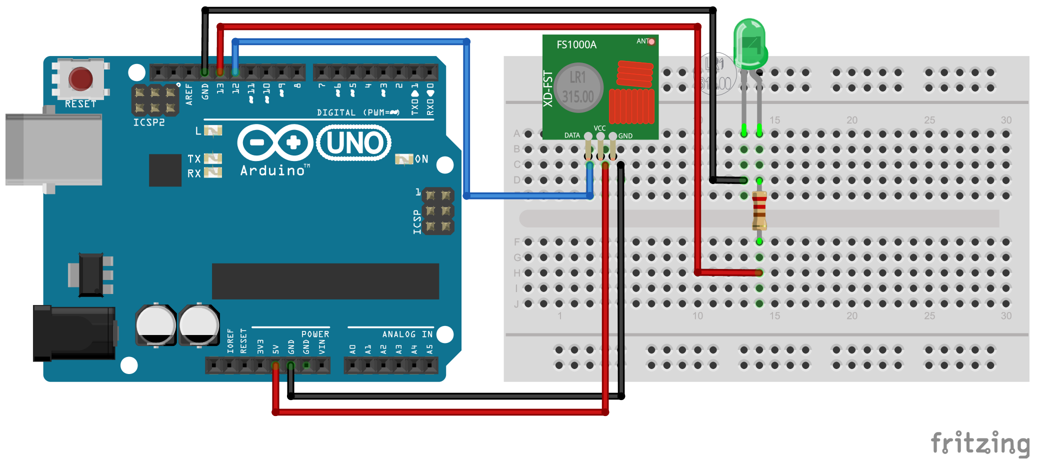


**Figure B.1:** Graphical circuit diagram of an Arduino with a transmitter.

The receivers were placed on a table 1 meter above the ground, each 15 cm away from the others, and a graphical circuit diagram of this setup can be seen on **??**. Similarly to the transmitter an LED was applied here blinking whenever a package was received.

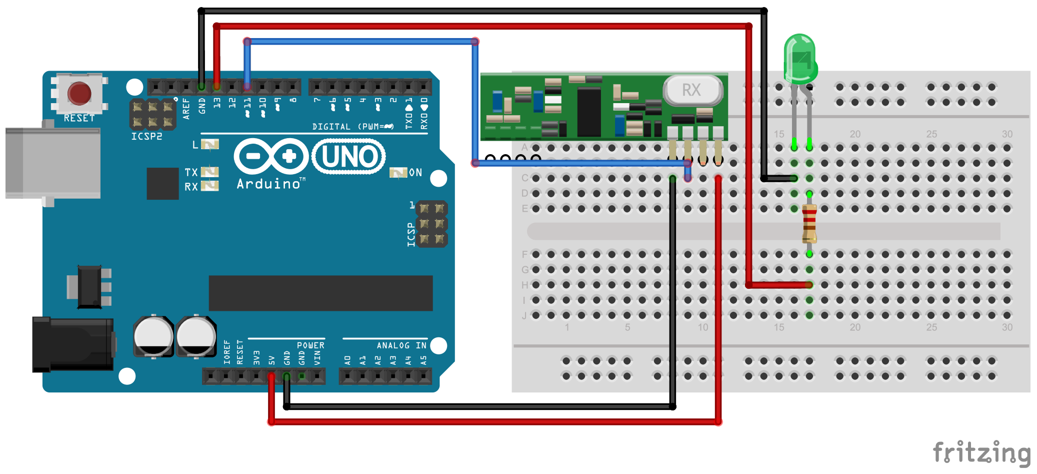


**Figure B.2:** Graphical circuit diagram of an Arduino with a receiver.

First the transmitters and receivers were 2 meters from each-other then a test with 100 packets containing 16 bytes was sent, the contents of a single message was the index of the packages being sent. The receivers counted the messages received and wrote the final result to the computer over USB. This was repeated every 2 metres. The code run on the Arduinos can be found in **??** and **??**.

## Results

The results of the test have been plotted in three separate graphs, this approach has been chosen so that it is easier to focus on the difference in antenna length on the receiving end.

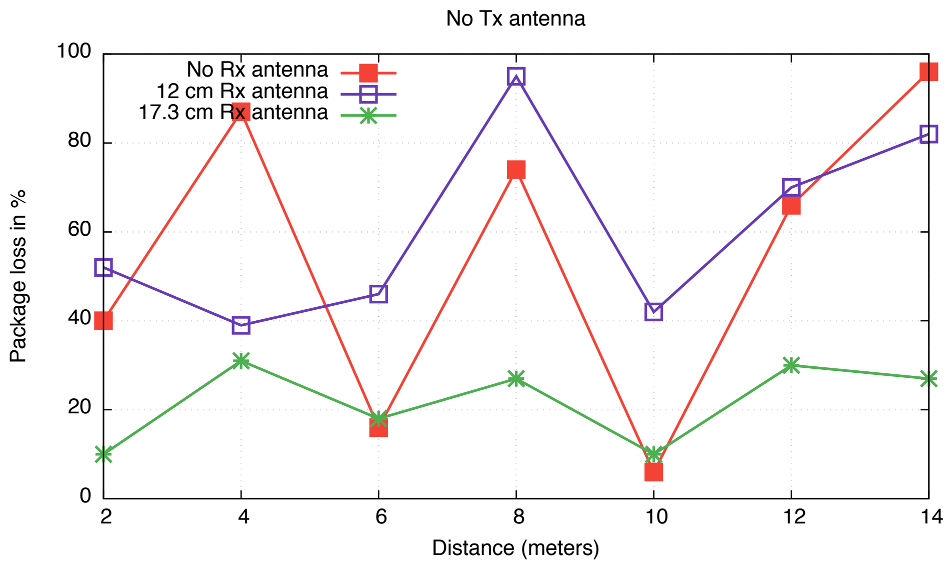


**Figure B.3:** Package loss percentage at different distances with no antenna on the transmitter.

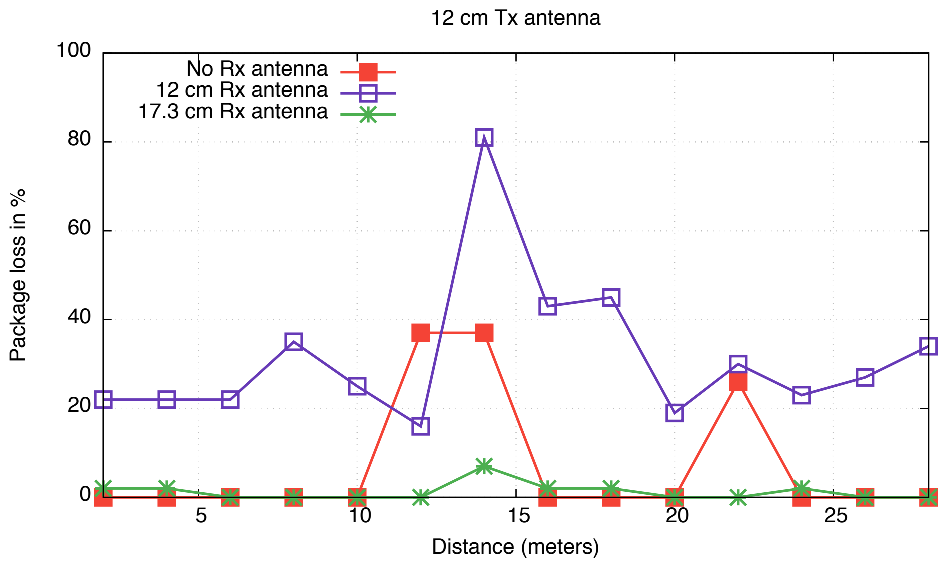


**Figure B.4:** Package loss percentage at different distances with 12 cm antenna on the transmitter.

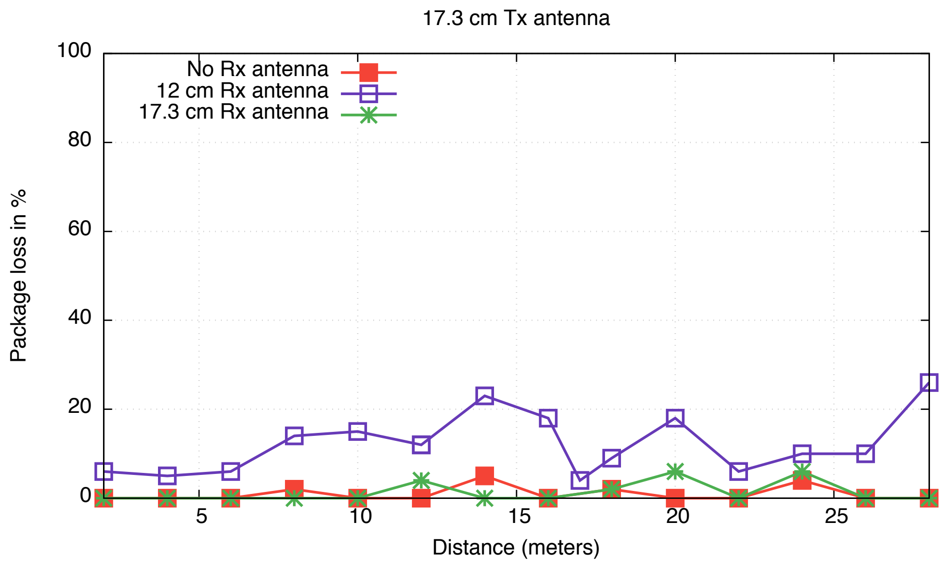


**Figure B.5:** Package loss percentage at different distances with 17.3 cm antenna on the transmitter.

## Analysis

Across all three graphs the plots representing the receiver with an antenna length of 17.3 cm consistently shows the lowest package loss. The only place where the results are different to distinguish from each other is in the graph where the transmitter has an antenna length of 17.3 cm (see **??**); here all the different lengths of receiver antennas demonstrates a low package loss percentage compared to the other graphs. This could indicate that the transmitter and its antenna has a significant impact on the reliability of the transmission of data. In fact the package loss of the receiver not fitted with any external antenna is indistinguishable from the receiver with 17.3 cm external antenna.

Furthermore the three graphs indicate that the antenna length of 12 cm on the receiver does not help the reception, on the contrary it worsens the ability to receive messages. However the antenna length of 12 cm does not affect the reliability as negatively when it is fitted on the transmitter (see **??**), this could be due to an error in the test setup, or just the fact that the transmitter is able to sent a much stronger signal as soon as an external antenna is introduced.

**Table B.1:** Table of average package-loss in percentage for varying lengths of antennas.

| | | Receiver | | |
|---|---|---|---|---|
| | | 0 cm | 12 cm | 17.3 cm |
| Transmitter | 0 cm | 55.00% | 60.86% | 21.86% |
| | 12 cm | 7.14% | 31.71% | 1.21% |
| | 17.3 cm | 0.93% | 12.71% | 1.29% |

In **??** it is strongly indicated that the transmitter and whether it has an external antenna highly affects the reliability of the transmitted signal as we can observe that the lack of an external antenna on the transmitter produces a high package loss percentage on the receiving end.

An average of the results from the graphs has been calculated and omitted into a table which can be found on **??**. This shows that using antennas of the length 17.3 cm will cause the transmissions to have a package loss of  1 %. The results will be further concluded upon later.

## Sources of error

There are multiple possible source of error within the test, which all needs to be considered in order to conclude upon the results. The most significant and the ones that are most likely to occur are as follows:

**Objects placed in the way of the signal**

In order for the results to represent real world scenarios one could test the package loss with various different objects in between the transmitter and receiver. E.g. humans, construction objects, furniture. However as this test was looking for quantitative data the results were gathered with no object in the way of the signal, so that the conditions and data of the tests would be comparable.

**Signal interference**

This category of errors is one of the most remarkable since any form of signal interference potentially could invalidate the data and even produce false positives. To avoid as much signal interference as possible, the test was conducted in a basement with relatively thick concrete walls, and equipment that uses the same frequency e.i. 433 MHz was kept away from the test setup.

**Inaccuracy in distance and antenna length**

The margin of error on the measured distance between the probes is with in a few centimetres, and since each probing was done in increments of two meters it is to be considered insignificant. On the other hand the tolerable margin of error when considering the length of the antennas is much smaller than two centimetres. Such varying in antenna length would have drastic consequences due to the antenna length's dependence upon the wavelength of the signal. However the antennas used in this test were carefully measured to specification of 12 and 17.3 cm. It is however noteworthy that the internal antennas of both the transmitter and receiver modules may affect the results.

**Difference in RF modules**

To be able to compare the results of the different antennas, one must assure that the difference in results is not induced by differences in the RF modules. However by testing all the used RF modules in **??**, this source of error is to be considered close to non existent, since all the RF modules used in this test, showed less package loss than 2 %. This source of errors is hereby deemed insignificant for the results of this test.

## Conclusion

The 17.3 cm antenna on the transmitter has the biggest impact on the package loss rate. This is in line with the theory of using a quarter-wave monopole antenna. This implies that using a 17.3 cm antenna on both the receiver and transmitter will greatly benefit the connection, both for range, and reliability. The only downside is the size will increase, however this trade-off is very well worth it for the project at whole. The amount of package loss still occurring with the 17.3 cm antennas shows that it is not certain that a message being transmitted will be received. For the project this means that the technique used to control the communication between the Arduinos needs to take into account that a transmission might not be received.

# Appendix C

# RadioHead Time Sent Test

The purpose of this test is to determine how long it takes to send a message of a given length. This information is useful at later points during a real-time analysis. The result is therefor a formulae which given the message length outputs the time it takes to send said message.

## Theory

As described in Section 3.5.2 there is an overhead of 104 bits for each message, along with the 4-to-6 bit conversion. The bit-rate is 2000 bits pr. second, which gives the theoretical minimum time it takes to send a message:

$$t(n, c) = c + \frac{104bit + n * \frac{6}{4} * \frac{8bit}{1byte}}{2000\frac{bit}{second}}$$

Where $n$ is the length of the message in bytes, $c$ is some constant in seconds which is the processing of the message. This equation could result in the following examples:

$$t(10byte, 0.01second) = 0.122second$$
$$t(20byte, 0.01second) = 0.182second$$
$$t(30byte, 0.01second) = 0.242second$$
$$t(40byte, 0.01second) = 0.302second$$

## Test Setup

The hardware setup used in this test is the same as the one used in the test presented in **??**. However this test was only performed at the distance of 1 metre, but this should not affect the time it takes to send a message, as this is determined by the timer on the Arduino.

### Software used in the test

The software used on the Arduinos for this test was different than the other test in **??**. The full source code is in **??** and **??**. In the software the message lengths of 1 to 60 bytes was set as the range to be tested. This can be seen in the core of the transmitter code, which is as follows:

```
void loop()
{
    driver.send((uint8_t *)msg, len);
    driver.waitPacketSent();
    len = (len % 60) + 1;
    delay(100); // In milliseconds
}
```

The transmitter will send a message of length `len`, wait 100 milliseconds, then send a message of length `len + 1`, and redefine `len` to `len + 1`. If `len` is 60 then the next value will be 1. There is placed a delay, of 100 milliseconds, in the end of the loop so the receiver has some time to process the received message; this will have to be subtracted when processing the test results.

The core of the receiver code is as follows:

```
void loop()
{
  uint8_t buf[RH_ASK_MAX_MESSAGE_LEN];
  uint8_t buflen = sizeof(buf);

  if (driver.recv(buf, &buflen)) // Non-blocking
  {
    Serial.print(buflen);
    Serial.print(", ");
    Serial.println(millis() - timer);

    timer = millis();
  }
}
```

The `millis` function returns the time in milliseconds since the Arduino was powered on. This code prints the length of a package received and the time, in milliseconds, since the last package was received.

## Results

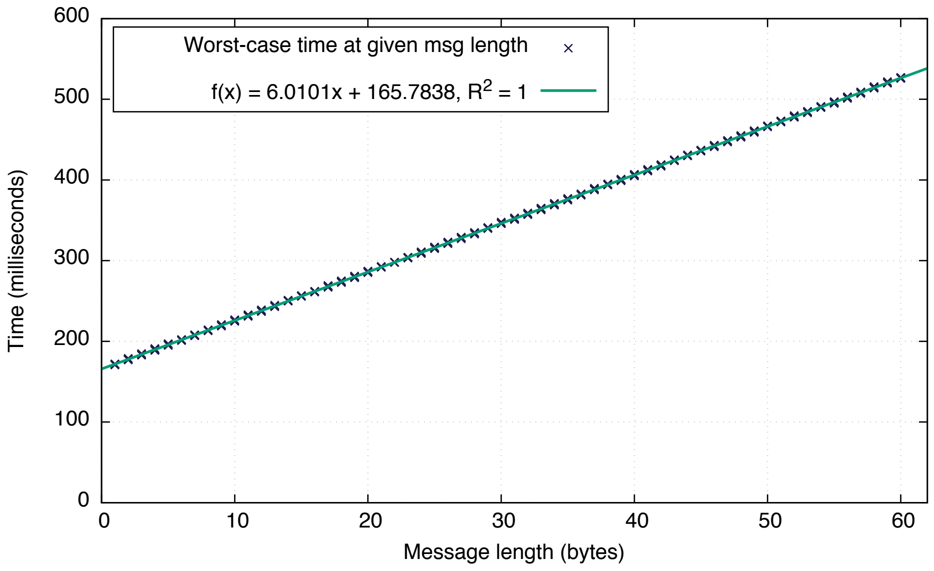


**Figure C.1:** Result of running the test.

These results are gathered over 800 data points, so each point is the worst-case of at least 10 samples. In this plot the 100 milliseconds delay as seen in the transmitter code is present. The worst-case value for 1 byte is 171 milliseconds, and for 60 bytes it its 526 milliseconds. Linear regression was performed on the data collected which yielded the formulae $f(x) = 6.0101 * x + 165.7826$ with $R^2 = 1$. The time it takes so send a message of length x bytes is therefore $f(x) = 6.0101 * x + 65.7826$, when the 100 millisecond delay is removed.

## Conclusion

This test shows that minimising the message length has a linear impact on the time it takes to send a message. Furthermore the time it takes to send such a message can be estimated in milliseconds, using the formulae $f(x) = 6.0101 * x + 65.7826$, where x is the length of the message in bytes. This result will impact the timing of the system as a whole.

# Appendix D

# Code for Transmitter in Package Loss Test

```
#include <Timer.h>
#include <RH_ASK.h>
#include <SPI.h> // Not actually used but needed to compile
#define STAT_LED 13
#define SEND_LED 2
#define TEST_LEN 100
#define TEST_INTERVAL 200

Timer t;
RH_ASK driver;
int pid = 0;
uint32_t cnt = 0;

void setup() {
    pinMode(STAT_LED, OUTPUT);
    pinMode(SEND_LED, OUTPUT);
    #ifdef VERBOSE
    Serial.begin(9600);
    if (!driver.init())
         Serial.println("init failed");
    Serial.println("Ready!");
    #endif
    digitalWrite(STAT_LED, HIGH);
    pid = t.every(TEST_INTERVAL, sendNext);
}

void loop() {
    t.update();
    if(cnt > TEST_LEN){
      t.stop(pid);
      digitalWrite(13, LOW);
    }
}

void sendNext() {
  t.pulse(SEND_LED, 150, LOW);
  uint8_t msg[4];
  *(uint32_t*)msg = cnt++;
  driver.send(msg, sizeof(msg));
  driver.waitPacketSent();
  #ifdef VERBOSE
  Serial.print("Pkg # ");
  Serial.println(*(uint32_t*)msg);
  #endif
}
```

# Appendix E

# Code for Receiver in Package Loss Test

```
#include <RH_ASK.h>
#include <Timer.h>
#include <SPI.h> // Not actualy used but needed to compile
#define RECV_LED 2
#define LOSS_LED 3
#define STAT_LED 13
#define TEST_LEN 100
#define TEST_INTERVAL 210 // 10 millis extra per interval, just to be sure
#define ANT_LEN 17

RH_ASK driver;
Timer t;

void setup() {
    pinMode(STAT_LED, OUTPUT);
    pinMode(RECV_LED, OUTPUT);
    pinMode(LOSS_LED, OUTPUT);

    Serial.begin(9600); // Debugging only
    if (!driver.init())
         Serial.println("init failed");
    digitalWrite(STAT_LED, HIGH);
}

uint32_t cnt = 0;
uint32_t lastcnt = 0;
uint8_t missed = 0;
#ifdef TEST_LEN
uint32_t starttime = 0;
#endif

```

```
void loop() {
    t.update();
    uint8_t buf[RH_ASK_MAX_MESSAGE_LEN];
    uint8_t buflen = sizeof(buf);

    if (driver.recv(buf, &buflen)) {
      t.pulse(RECV_LED, 150, LOW);
      cnt =  *(uint32_t*)buf;
      if(cnt == lastcnt + 1) {
        lastcnt = cnt;
      } else if(cnt < lastcnt || cnt == 0) {
        lastcnt = 0;
        missed = 0;
        #ifdef TEST_LEN
        starttime = millis();
        #endif
        #ifndef VERBOSE
        Serial.println("Transmission reset");
        #endif
      } else {
        t.pulse(LOSS_LED, 500, LOW);
        uint8_t curmis = missed;
        missed += cnt - lastcnt;
        curmis = missed - curmis;
        lastcnt = cnt;
        #ifdef VERBOSE
        Serial.print("Missed ");
        Serial.print(curmis);
        Serial.print(" pkgs - ");
        Serial.print(missed);
        Serial.print(" total loss (");
        Serial.print(((double)missed/(double)cnt)*100);
        Serial.println(" %)");
        #endif
      }
      #ifdef TEST_LEN
      if(cnt + 1 == TEST_LEN || millis() > (starttime + TEST_INTERVAL *
    TEST_LEN)){
        Serial.print(ANT_LEN);
        Serial.print(" \t ");
        Serial.print(missed);
        Serial.print(" \t ");
        Serial.println(TEST_LEN);
        digitalWrite(STAT_LED, LOW);
        while(1){
          if (driver.recv(buf, &buflen))
            if(*(uint32_t*)buf < TEST_LEN){
              t.oscillate(STAT_LED, 100, HIGH, 5);
              lastcnt = 0;
              missed = 0;
              starttime = millis();
              break;
            }
        }
      }
      #endif
    }
}
```

# Appendix F

# Delay Test Receiver

```
#include <RH_ASK.h>
#include <SPI.h> // Not actualy used but needed to compile

RH_ASK driver;

unsigned long timer = millis();

void setup()
{
  Serial.begin(9600); // Debugging only
  if (!driver.init())
    Serial.println("init failed");
}

void loop()
{
  uint8_t buf[RH_ASK_MAX_MESSAGE_LEN];
  uint8_t buflen = sizeof(buf);

  if (driver.recv(buf, &buflen)) // Non-blocking
  {
    Serial.print(buflen);
    Serial.print(", ");
    Serial.println(millis() - timer);

    timer = millis();
  }
```

# Appendix G

# Delay Test Transmitter

```
#include <RH_ASK.h>
#include <SPI.h>

RH_ASK driver;

void setup()
{
  Serial.begin(9600);
  if (!driver.init())
    Serial.println("init failed");
}

uint8_t len = 1;
const char *msg = "0123456789
    abcdefghijklmnopqrstuvwzyzABCDEFGHINJKLMNOPQRSTUVWXYZ";

void loop()
{
  driver.send((uint8_t *)msg, len);
  driver.waitPacketSent();
  len = (len % 60) + 1;
  delay(100);
}
```

# Appendix H

# UPPAAL Model

In this appendix the UPPAAL model along with its code can be seen.

```
// Place global declarations here.
int n = 0;                                  // Number of Timeslots connected
int i = 0;                                  // Current time slot in the frame
const int Delta = 80;                       //Timeslot Length
const int Delta_Com = Delta/2;
const int Real_Transmit_Time = Delta/3;
const int Initial_Wait_Time = Delta*3;
int Startup_Time = Initial_Wait_Time*2;
clock startup;

//Channel
broadcast chan transmit;

//Device Creation
const int N = 6;
typedef int [1, N] id_t;
```

**Listing H.1:** Code for the global declarations.

```
// Place local declarations here.
clock x;
clock transmit_time;
int k = -1;                       //Timeslot
bool Connected = false;

// Local copies of globals
int local_n = 0;                  // Number of devices connected
int local_i = 0;                 // Current time slot in the frame
```

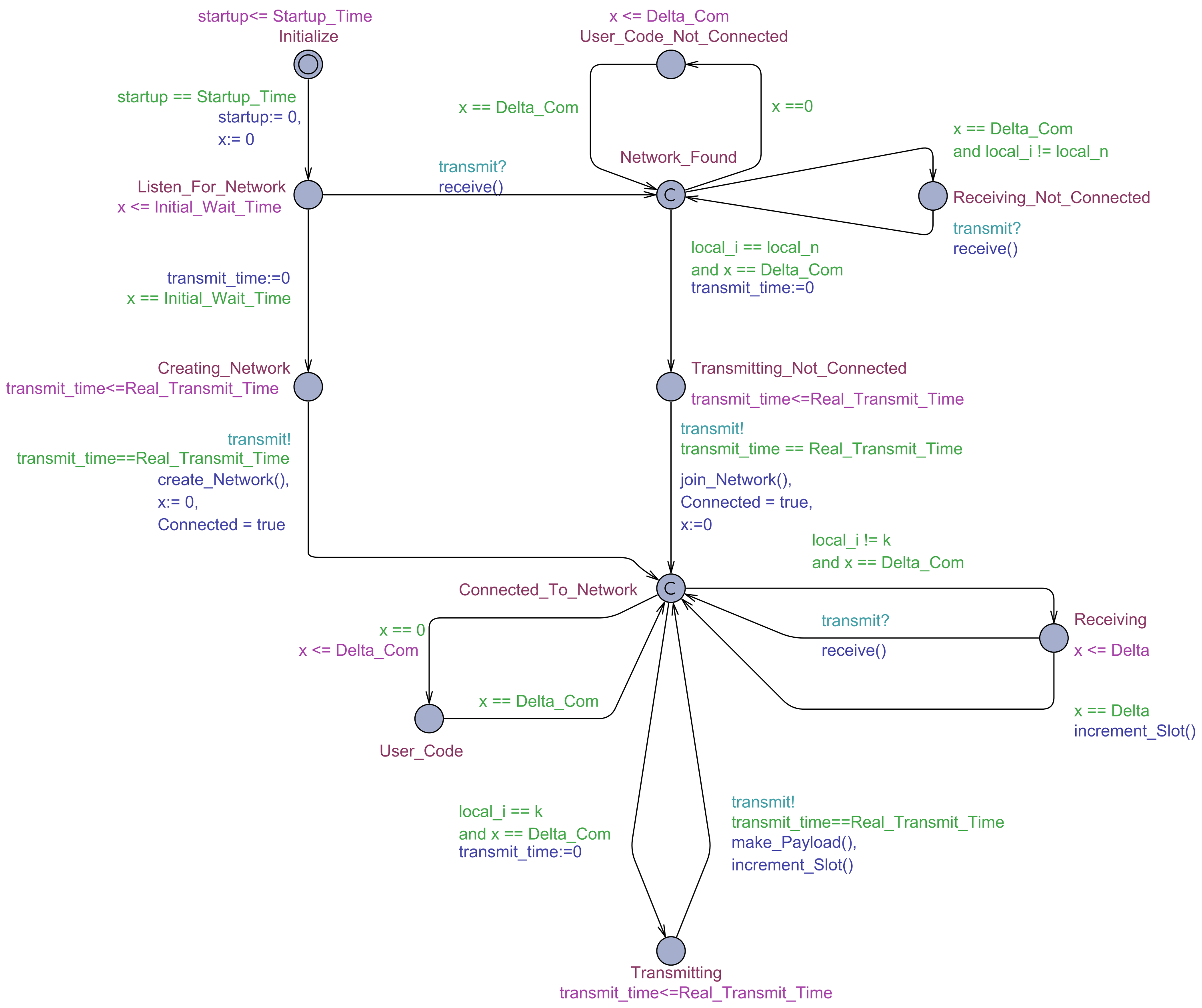


**Figure H.1:** UPPAAL Model

```
void increment_Slot(){
local_i = (local_i % local_n)+1;
x:=0;
}

void receive ()
{
local_i = i;
local_n = n;
increment_Slot();
}

void join_Network(){
k=n;
i = 1;
local_i = 1;
n = n+1;
local_n = local_n+1;
}

void create_Network(){
i = 1;
n = 2;
k = 1;
local_i = 1;
local_n = 2;
}

void make_Payload(){
i = local_i;
n = local_n;
}
```

**Listing H.2:** Code for the local declarations for Device.

```
// Place template instantiations here.
// List one or more processes to be composed into a system.
system Device;
```

**Listing H.3:** Code for system declarations.