Electrical Engineering Bsc. Thesis

Localisation and Communication for Autonomous Deci-Zebros:

Localisation and System Integration

Ronald Schotman (4228243)
Marck Wolleswinkel (4280490)

Supervisors:
Chris Verhoeven (EWI, TUDelft)
Edwin Hakkenes (EWI, TUDelft)
Ronald Bos (EWI, TUDelft)
Daniël Booms (EWI, TUDelft)

TU Delft
June 2017

Abstract

This document describes the Bachelor Graduation Project of the Communication and System integration group of the Zebro Localisation and Communication Module. The module has been designed for a Zebro which is a six legged robot with the purpose of working in swarm-related behaviour. To make this behaviour possible a new Localisation module for Zebro as well as a communication module has been designed with a corresponding processing unit. This thesis consists of two parts; the first part covers the development of a communication module capable of autonomous communication in an homogeneous network where no predefined routes are available, also known as a mesh network. The second part covers the integration of the communication, ranging and processing subgroups from which all the components are to be placed on a printed circuit board.

Preface

The authors, Ronald Schotman and Marck Wolleswinkel, have written this document as part of their thesis for their BAP poject (Bachelor Afstudeer Project, Bachelor Graduation project) ”Localisation and Communication for Autonomous Deci-Zebros: Localisation and System Integration” for the degree of Bachelor of Science in Electrical Engineering at the Delft University of Technology. The idea and setup for the graduation project came from the Zebro Project Group, EWI, and is part of the TU Delft Robotics Institute.

Delft, *Ronald Schotman*
June, 2017 *Marck Wolleswinkel*

Acknowledgements

We would like to thank Chris Verhoeven, Edwin Hakkenes, Ronald Bos and Daniël Booms for their coordination and continued support during the development of the project. Without their help we would not have been able to accomplish the amount of work done over the span of 8 weeks and their valuable support and suggestions contributed much to the improvement and completion of this project. We would also like to thank the TU Delft, and the department of micro electronics for providing us with a lab room and all the resources to work with. The project has truly been a unique experience and has learnt us more than we could have imagined.

Contents

Introduction 13
1.1 State-of-the-art analysis 14
1.2 Problem definition 15

Programme of Requirements 17
2.1 Functional Requirements 17
2.2 System Requirements 18
2.3 Performance Requirements 18
2.4 Development Requirements 19

System Design 21
3.1 Communication network 21
3.1.1 Mesh networking 21
3.1.2 MANET 22
3.1.3 DigiMesh 23
3.2 Distance estimation using the communication channel 24
3.2.1 RSSI 25
3.2.2 Round trip time 26
3.2.3 TDOA 26
3.2.4 'Cricket' localisation 27
3.3 Communication protocol 28
3.3.1 Communication error handling 28
3.4 Predefined packet scheme 28
3.5 Turn-based communication 31

Communication prototyping 33
4.1 Prototyping boards 33
4.2 Monitoring 36
4.2.1 Configurability 37
4.3 Results 37
4.4 Future goals 37

PCB design and System integration 39
5.1 Schematics 39
5.1.1 System overview 39
5.1.2 Components 40
5.2 The PCB 46
5.2.1 Footprints 46
5.2.2 Layout 46
5.2.3 Layers 46
5.2.4 Routing 47
5.3 Main components 47
5.3.1 The Buck converter 47
5.3.2 The Ranging PCB 48
5.3.3 The Communication PCB 48
5.3.4 System integration 50

Discussion 51

Conclusions 53
7.1 Functional Requirements 53
7.2 System Requirements 54
7.3 Performance Requirements 55
7.4 Development Requirements 55

Future work 57

References 59

Appendices 61

Micro controller firmware - main.c 63

Monitoring software - MainForm.cs 73

Bill of Materials 81

Schematic 83

Chapter 1
Introduction

The *Zebro project* is a *swarm robotics* research group at the TU Delft. Its mission is "To develop self-deploying, inexpensive, extremely miniaturised and autonomous roving robots which cooperate in swarms capable of functioning in a wide spectrum of topologies and environments that can quickly provide information with the help of distributed sensor systems and supports payloads suitable for a wide range of missions." It started in September 2013, and has now been absorbed by the TU Delft Robotics Institute as part of their research endeavours into swarm robotics.

The idea of swarm robotics is inspired by nature, in particular by small insects or birds who exhibit so-called *swarm behaviour*. As a group, a swarm can accomplish complex tasks, despite the fact they are not considered 'smart' individually. By all following the same simple set of rules, the swarm as a whole can exhibit *emergent behaviour*, behaviour that is not explicitly programmed into any one of its members. This idea that a group of simple individuals can deal with more complex tasks is the cornerstone of swarm robotics. Furthermore, an entire swarm is very resilient, as every individual member is expendable. This means that swarm robotics has some unique applications, such as a disaster area that is inaccessible to humans or large robots, due to a hazardous or arduous environment.

Currently, the Zebro project is working to redesign the robots to (further) improve producibility, reparability, stability, and cost. Being able to produce useful Zebro units on a large scale will provide fertile ground for swarm robotics research efforts. There are several different versions of the Zebro, each with a different form factor and design goal. Our efforts are focused on the *Deci Zebro*, which is the size of an A4 sheet of paper and is designed to be extremely modular.

Features that are still highly desired by the Zebro project are the ability for a Zebro to localise itself with respect to its neighbours and the ability to let Zebros communicate with each other. Expanding on this, the overall objective of our group is to develop this, which has been nicknamed the 'Localisation and Communication module' of the Deci Zebro project.

At the very beginning, our group decided to split the project into three distinguished parts:

- **Ranging**
  This group is responsible for providing a set of distances to other Zebros in the Zebros vicinity.
- **Processing**
  This group processes the distance data from the ranging group to create a local model of the swarm around the Zebro.
- **Communication and system integration**
  This group is responsible for the communication system and for the integration of both the ranging submodule and processing submodule into one single modular Zebro-module.

This thesis will cover the design and implementation of the communication and system integration group.

1.1 State-of-the-art analysis

Wireless communication exists in many different forms nowadays, ranging from simple low frequency ad hoc networks to communication with satellites. Because of the great demand on wireless connectivity between devices, numerous standard protocols are available for engineers to integrate in their design. A few of the most common are discussed and assessed for suitability in the localisation and communication module.

Bluetooth

A commonly used and well-known communication protocol is the Bluetooth standard. Bluetooth was designed with intentions for portable equipment and robust data transmission capabilities. Over the last two decades, Bluetooth has proven to serve well in simple applications where two devices devices need to connect with each other with easy coupling and low data rates. It is however possible to set up ad hoc networks with Bluetooth. [1] [2] At the moment there are no exact specifications for Bluetooth working with more than a handful of devices, but there is a large community trying to develop these, resulting in a few time-efficient algorithms such as BlueStars, BlueMesh, and BlueMIS that seem fast enough for combined ranging and communication. [3].
However, the complexity of a Bluetooth connection and, even more, a Bluetooth network make it unsuitable for application in the Zebro project. This is also due to the fact that Bluetooth was designed to operate in a peer-to-peer topology, which is not valid for this appliation.

Wi-Fi

Probably the most used communication protocol nowadays is the Wi-Fi. Just like Bluetooth, Wi-Fi makes use of the IEEE802.11 standard. Wi-Fi was intended to replace (high speed) cabling for local area network access. Wi-Fi suits better than Bluetooth in applications where high bandwidth is desirable and some degree of client configuration. With Wi-Fi the transmission rate is inversely proportional to transmission range which also corresponds to the transmit power level. [4] This is why WiFi is usually combined with electrical devices that are grid bound.
Furthermore, just as with Bluetooth, the high complexity of Wi-Fi makes it not really suitable for application in the Zebro project.

ZigBee

ZigBee is a widely used specification for wireless communication networks [5]. Just like Wi-Fi and Bluetooth it can work in different frequency bands, from low RF bands (868 MHz) as well as the 2.4 GHz bands. Most implementations with ZigBee use a point-to-point topology [6] but ZigBee also defines a mesh network topology called ZigBee Mesh [7] [8]. This is a mesh implementation that requires a coordinator to handle routing and maintain proper setup of the network. An alternative to ZigBee Mesh is DigiMesh. This is a mesh topology using ZigBee but with homogeneous nodes and no need for coordinators or routers [9].

1.2 Problem definition

Based on the project description set out by the Zebro project, which will be discussed in the Programme of Requirements chapter 2, and the division into submodules, the localisation and system integration group has the following tasks:

- Realise wireless communication between neighbouring Zebros
- Integrate this communication system together with the submodules designed by the ranging and processing group into a single module for the Zebro.

As is described in section 1.1, many standards for wireless communication already exist. It is therefor important to note that it is not the goal of this project to create a new wireless communication standard, but to seamlessly implement a 'proven technology' onto the Zebro project. This will still require basic structures like a packet format and communication protocol to be designed specifically for the application with the Zebro.

This thesis will cover the two tasks mentioned above. First, the design and prototyping of the communication system will be discussed and thereafter the integration of the whole project will be covered.

Chapter 2
Programme of Requirements

The ultimate goal of the module that is to be designed for the Deci Zebro is for the Zebro to be able to determine its position relative to its neighbours. This functionality is used to create the swarming behaviour of the Zebro.
Furthermore, since this localisation module will most likely require some form of communication with the other Zebros in the swarm, a communication submodule must also be included in the design. Apart from the communication required for the localisation system, the communication submodule can also be used for broadcasting data by instructions of the central brain of the Zebro.

All of the above added with some additional requirement is shown below.

2.1 Functional Requirements

[F-1] The module should be able to locate multiple neighbouring Zebros

- **[F-1.1]** The module should be able to be used on and locate charging stations and similar units
- **[F-1.2]** The module should be able to distinguish different users of the module

[F-2] The module should be able to receive and transmit packets to neighbours

- **[F-2.1]** The module should be able to make contact with at least 3 neighbouring Zebros
- **[F-2.2]** The module should be able to broadcast housekeeping data for inspection by users

2.2 System Requirements

[S-1] The mechanical, power, and digital connections must conform to the specifications outlined in [10]
[S-2] The module must conform to the Zebrobus protocol outlined in [11]
[S-3] The module must respect the modularity of Zebro modules

- **[S-3.1]** The module should be a slave to the Zebro bus and cannot make requests itself
- **[S-3.2]** Use of the module cannot inhibit the functionality of other modules on the Zebro
- **[S-3.3]** With the exception of data broadcast on the Zebro bus, the module should collect data exclusively using its own sensors
- **[S-3.4]** The module should communicate its data to the Zebro by responding to requests over the Zebro bus
- **[S-3.5]** The modules should be interchangeable between robots and easily replaceable

[S-4] The module should be able to be used in indoor environments
[S-5] The communications network should be homogeneous

- **[S-5.1]** Communications cannot rely on a predetermined master node or network hub
- **[S-5.2]** An individual Zebro within range should be able to join the communications network at any time
- **[S-5.3]** Any individual Zebro in the network should be able to leave without affecting network functionality

[S-6] The module should be usable on stationary elements, such as a charging station, without any changes to the hardware
[S-7] The module connects to the Zebro power system (see **[S-1]**)

- **[S-7.1]** The power system is rated at 13 V to 19 V [1]

[S-8] The module should be able to provide debugging information over the Zebro bus

2.3 Performance Requirements

[P-1] The module should be able to locate other Zebros with an accuracy of 10 cm
[P-2] The module should be able to locate Zebros within a range of at least 5 m
[P-3] The distance data provided to the Zebro brain should have a resolution of 10 cm
[P-4] The module should be able to locate Zebros in all directions of the horizontal plane

[1] There are no specifications yet for the maximum drawn power/current

2.4 Development Requirements

[D-1] The module production cost shall not exceed €150 per unit
[D-2] The module PCB should be able to be fabricated using a pick-and-place service

Chapter 3
System Design

3.1 Communication network

3.1.1 Mesh networking

Requirement **[F-2]** of the program of requirements (chapter 2) describes a network where every in the network is able to transmit to and receive data from every node in its vicinity. In addition, no predefined routes are available, since the network can be set up and changed at any time with different combinations of nodes. This specific type of network is described by the *mesh network* topology, which is illustrated in figure 3.1. In a mesh network, each node has a connection with (at least some of) its neighbours and relays the data it receives to all nodes it has a connection with. Consequently, each node in the mesh network will receive the original message after a number of hops, depending on how far the original sender is from the receiving node.

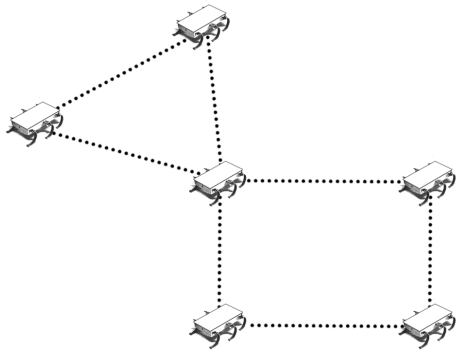


Fig. 3.1 A mesh network of Zebros

Because there are no pre-existing routes, routing is done on an ad-hoc basis. This also means that nodes can join a network at any time. Also, when a node leaves a network, the network is self-healing and re-establishes the necessary routes to connect all nodes in the network (figure 3.2)

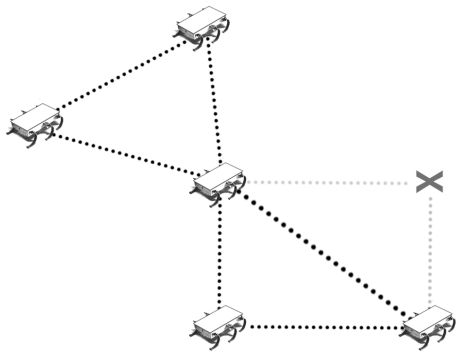


Fig. 3.2 The network is self-healing when a node leaves the network

3.1.2 MANET

In addition to describing a mesh network topology as explained above, requirement **[S-5.1]** also specificies that the network should be autonomous. This is however not an inherent property of mesh networks. Most implementations rely on a number of coordinators and routers to set up the necessary routes between nodes and to coordinate the package flow through the network.
This is however a problem, because it implies that there are nodes in the network that are 'superior' and would cause the network to fail if they were removed or not present. The requirement clearly states that all nodes should have an equivalent role and function and that any combination of 2 or more arbitrary nodes should be able to communicate.
The mesh topology that is going to be used is therefore further specified and will in fact become a MANET, or Mobile Ad hoc Network. As described in [12], a MANET is an infrastructure-less network of nodes that can communicate with each other while moving. Because of this movement, the links between the nodes have to be constantly updated in order to ensure the overall connectivity in the entire network.
Expanding on this, AMNETs are MANETs with the additional constrained that they are autonomous, so without the use dedicated nodes for coordination or routing , as defined in [13].

3.1.3 DigiMesh

One problem that still remains is that although MANET and AMNET describe network structures that are well-suited for the Zebro application, they do not refer to a specific practical implementation. Furthermore, there are not many practical implementations on the market right now that support this type of mesh structure.
One option for a mesh implementation is the ZigBee© Mesh (figure 3.3), as mentioned in 1.1. However this is an implementation which requires designated nodes to act as routers in order to maintain the network. Since this is in violation with requirement **[S-5]**, it is not suitable for implementation in the Zebro project.

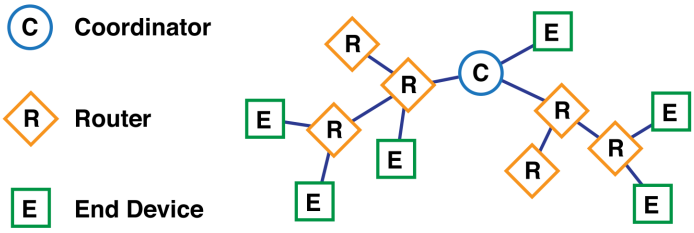


Fig. 3.3 The network structure of ZigBee© Mesh. Image from [9]

But there is a feasible competitor of the ZigBee Mesh which is developed by Digi International. It is called DigiMesh™ and it is a proprietary technology based on their XBee module ([9], figure 3.4). The most important difference between ZigBee Mesh and DigiMesh can easily be seen by comparing figure 3.3 with figure 3.4. In a DigiMesh, all nodes have the same role and the operation of the network works with every combination and amount of nodes. Also no network setup is required, other than pre-defining the channel all the devices will operate in.

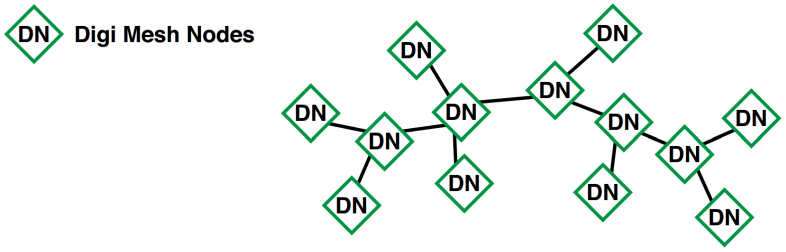


Fig. 3.4 The network structure of DigiMesh. Image from [9]

As mentioned before, DigiMesh^{TM} is a proprietary technology and is therfore only available from Digi International itself. Because it is not desirable to be singularly dependent on one supplier for the availability and moreover the price of the hardware, the implementation of the communication device in the module is done in a modular fashion, which is shown in figure 3.5. This allows for the DigiMesh module to be replaced by an alternative mesh network module when a better alternative is found or developed. The only requirement is that the data communication is done via a serial data connection (the Tx and Rx lines in figure 3.5). However, because DigiMesh is well-suited for application in the Zebro project, for the scope of this project it will be used as the communication device.

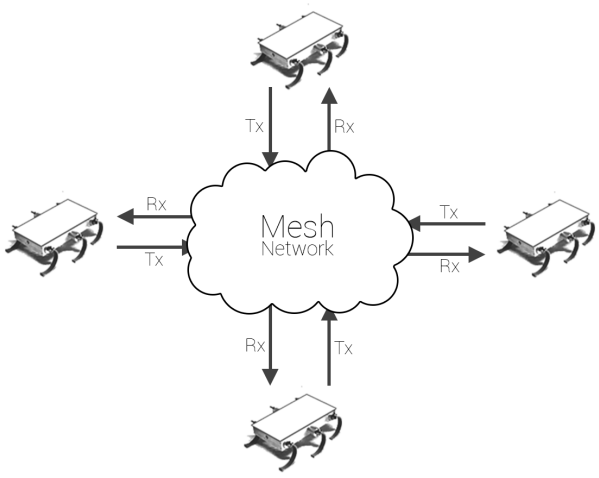


Fig. 3.5 The choice of communication device is modular

3.2 Distance estimation using the communication channel

Although it is the objective of the ranging subgroup to determine the distance to neighbouring Zebros, there are also ways to extract this distance with the use of the communication device. In the following section, a few of these methods are discussed and also explained why they are suitable or unsuitable for the application.

3.2.1 RSSI

When a wireless channel is used to communicate, every packet has an RSSI value. RSSI stands for Received Signal Strength Indicator and it is a measurement of power regarding the signal containing that packet. This received power can be related to the distance from the sender to the receiver using the Friis equation (3.1)

$$\frac{P_r}{P_t} = (\frac{\lambda}{4\pi R})^2 G_r G_t \tag{3.1}$$

where P_r is the received signal power, P_t is the transmitted signal power, λ is the electromagnetic wavelength of the signal, G_r is the receiving antenna gain, G_t the transmitting antenna gain and R the line-of-sight distance between the antennas. Since all other parameters are (practically) constant and known, it should be possible to determine R using P_r.
However, this is where one of the main advantages of a mesh network becomes a disadvantage. Since there is no guarantee that every node in the mesh network will receive the original message from the sender, each node, after receiving the message, will resend the message to the other nodes. This allows for messages to be transmitted far beyond the range of a single node and moreover insures that that chances of missing a packet decrease significantly.
The problem with this when it comes to RSSI is evident; because the package that is received by the node might not be the original send by the sender, the RSSI value that is associated with the package can not be used to determine the distance between the original sender and the receiver (using equation 3.1), because the power has been boosted by the last node that hopped the message.

A possible way to avoid this is to look only at the RSSI of the *first* packet that has been received. Since the direct path from the original sender to the receiver is always the shortest (when a message is hopped it is also delayed because of bit-error checking) this message will contain the information from the original transmitter.
This is unfortunately not as trivial as it sounds. Firstly, there is a possibility that the receiver is beyond the direct range of the original sender, so the first message that is received is not the original message, but instead a hop of the first node that is in range of the receiver. But the main problem is in isolating the RSSI value of the first packet. A mesh node only passes its data through to the central processor when it is not receiving packet hops anymore. So by the time the data reaches the central processor, the RSSI of the first packet received is long gone, because the RSSI values are not stored in a buffer by the mesh node.

Because of this, using RSSI in a mesh network topology is not suitable for determining range between individual nodes.

3.2.2 Round trip time

Instead of looking at the received power, it is also possible look at the time it takes for a package to travel through the ether from sender to receiver and back. This can be done when the receiver returns an acknowledgement indicating the package has been received. Since the speed with which the packages travel is (quite accurately) known, a distance can be extracted using equation 3.2

$$d = \Delta t * v \tag{3.2}$$

where d is the distance, Δt the time between the sending of the original message and the receiving of the acknowledgement and v the speed of the carrier waveform. Because the data is send via radio waves this speed is almost equivalent to the speed of light, which is 299.792.458 m/s. In order to measure a distance with a resolution of 10 cm (which is reasonable for the Zebro project), time differences of $2 * 0.1/299792458 = 6.6 * 10^{-10}$ s $= 0.66$ ns would have to be measured. Note that the distance has been doubled, because of the round trip. An average micro controller works at a maximum clock rate of 20 MHz, which gives an maximum detection resolution of 50 ns. This is equivalent to a distance resolution of $5 * 10^{-8} * 299792458 = 14.99$ m for the roundtrip, which means that only distances in multiples of 7.5 m can be calculated.
This shows that the round trip time of a package is not suitable for determining distance either. Of course, more advanced hardware could be used to detect smaller differences in time, but this goes with a steep increase in hardware costs, which makes it not viable for implementation in the Zebro project.

3.2.3 TDOA

Time Difference of Arrival is another method for determining distance similar to time of flight. However, instead of measuring the time it takes for a package to travel from the sender to the receiver and back, the distance determination is based on the difference in arrival time between different receivers. This can be applied with practically any form of travelling wave, but it is often done using (ultra)sound waves.
One setback in this approach however is that the basic algorithm requires a number of nodes to be anchored, which means that they are stationary and that their positions are known. This is a problem, since one of the requirements states that the localisation must be performed without the use of any infrastructure or other type of static device.
Still, it is possible to use TDOA to determine the location of neighbouring nodes using a Multidimensional Scaling Algorithm[1]. However, this would require that all

[1] https://en.wikipedia.org/wiki/multidimensional_scaling

nodes share the data they measured with each other, which would require much bandwidth from the communication module. This is not desirable because this would mean that the communication channel will be blocked most of the time for the localisation features and can not be used to transmit other data from the Zebro itself.

3.2.4 'Cricket' localisation

As discussed in section 3.2.3, TDOA is not really suited for the application mostly because of the amount of data that has to me transmitted in the network, because a device cannot determine the difference in arrival time using only its own data.
An alternative but similar method to tackle this problem is proposed by the Cricket localisation technique, developed at MIT ([14]). The concept is illustrated in figure 3.6.

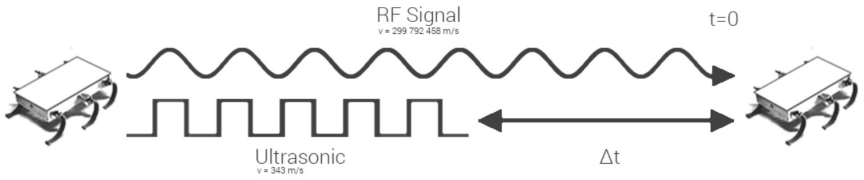


Fig. 3.6 The concept of the Cricket localisation system

Instead of only sending an acoustic signal (in this case ultrasonic), a node also sends a communication signal, containing information about its idea and other necessary system parameters. Because of the difference in speed between an acoustic wave ($v = 343$m/s) and a radio wave ($v = 299.792.458$m/s), there is a difference in arrival time between the two signals at each receiving node. Now, a node immediately has all the necessary information to perform a TDOA and determine its distance from the sender.

The ranging subgroup has developed the necessary module to realise the ultrasonic part of the localisation system, more information can be found in [15]. The next step to determine positions of neighbours with this distance data is done by the processing subgroup, more information can be found in [16].

3.3 Communication protocol

3.3.1 Communication error handling

Since the network consist of wireless interconnections between the nodes, there is a significant probability, especially with increased distance, that the communication is subject to bit errors and other communication errors. This means that the packet arriving at the destination could be corrupt or not even arrive at all. Many different schemes are available to compensate for this, a few will be discussed.

3.3.1.1 Acknowledgement

Wireless communication systems often use an acknowledgement system, where the receiver replies to the sender when it has received a package. This way, by means of a predefined timeout, the sender can easily verify whether the message send has been received by the receiver and if not, it can be resended. This is however not a viable option for the mesh setup described earlier. For one, since the sender is not certain to know how many devices there are in the network, it is unable to determine whether there are nodes that have not received the message or are simply not present in the network. Even more important is the fact that this acknowledgement procedure would have to be a carefully structured turn-based system, because only one device is allowed to send at any given moment in time. This would mean that every message has to be sent and checked individually for each receiver, which would result in a time consuming acknowledgement scheme for every transmission, especially when each message is relatively short. On top of that, it might also happen that the acknowledgement of the recipient is not received by the receiver for some reason which would cause the sender to repeat the original message unnecessarily, taking up even more time. This is therefore not the desired method to solve possible communication errors.

3.4 Predefined packet scheme

A very simple solution to reduce the chance of message failures would be to send each message multiple times. Since the processes resulting in the communication errors can be modelled as random stochastic process, the probability that 2 or more consecutive messages fail is significantly lower than the failure of a single message, the chances of one node missing a message are significantly decreased with the increase of the number of repeats.

However, it is still not ruled out that a message received has been corrupted by bit

errors or been cut short. Because of this, it is of great importance that a node can detect when a message is corrupted. When handled incorrectly, interpreting data from corrupted messages could result in malfunctioning of the device, in which a manual reset is necessary and since this might not always be possible (especially when the device is not reachable because it is on the moon) it could render the device useless. The first step to detect the validity of the message is to define a standard format for packets. This way, a receiver can easily identify whether the packet is in right format. Note that the protocol used (UART) works with chars so each part of a packet is 1 char or 8 bytes. The format used is displayed in table 3.4.

1 char	1 char	1 char	1 or more char	1 char	1 char
Start (STX)	Sender ID	Length	Data	CRC	End (ETX)

Table 3.1 Communication packet format

From table 3.4 it can be easily determined that a packet has a minimum length of 6 chars (given that a packet has at least 1 byte of data). If a packet received is shorter than this, it is definitely corrupt and must be rejected.
The packet format consists of the following components:

- **Start**. This is ASCII char 0x02. Every message must start with this char. It is reserved, so no other part of the message can contain this char.
- **Sender ID**. The ID of the sender is stored in this char.
- **Length**. The length of the *whole* message, which is a value between 5 and a predefined maximum number. The value is encoded as an ASCII char, so 5 corresponds with char 0x35 etc.
- **Data**. The actual data. Note that this can be in any format, as long as it does not use the reserved characters (STX and ETX).
- **CRC**. Cyclic Redundancy Check, this is a commonly used error detection mechanism. The value contains the result of a polynomial division on the content of the packet ([17]). If the package is intact, the value should match the one that will be calculated on the same package on the receiver side. If not, the packet has been corrupted and it will be discarded.
- **End**. This is ASCII char 0x03. Every message must end with this char. It is reserved, so no other part of the message can contain this char.

To verify the format of an incoming message, a state machine is used. Initially, the device is in an idle state, waiting for a packet to arrive. As soon as the first char of a packet has been received. Note that by the packet format of table 3.4, this can only be a start char (0x02). If it is any other char, the message is corrupted, so the rest of the package is ignored.
If the character is indeed an STX char, the state machine moves to the next state in which it will receive the ID of the sender. After this, it will receive the length of the package, which is important to determine the validity of the data. Because the data part of the packet is the only thing of variable length, the complete packet length

is directly related to the data length. After receiving the length, the state machine enters a state where it will store the incoming chars until one of the following events occurs:

1. The incoming char is the end char (0x03) and the number of data chars (plus 1 CRC char) received corresponds with what was calculated based on the packet length.
2. The incoming char is the end char (0x03), but the number of data chars (plus 1 CRC char) received does not correspond with what was calculated based on the packet length. The message is either too short or too long but, in both cases, invalid.

If the message is determined to be invalid, the content is discarded and the state machine goes back to its idle state, waiting for a new packet to arrive. If the message is not found to be invalid, it can be concluded that the package format is correct and that the message has the correct length. This does however not rule out that there have been bit errors in the data part, which especially with long messages is not unlikely. This is where the CRC char comes in. Before sending the message, the sender computes a CRC over the message and stores the result in the CRC char. When the message is received, the receiver performs the same computation on the data it received (excluding the CRC and ETX chars). If the message is corrupted, the result will not match with the CRC char and the message is discarded.

3.5 Turn-based communication

Because the network structure used only allows for one node to be sending at any moment in time, there must be a system in the communication protocol that determines when a node is allowed to use the network to transmit data.
The most straight-forward way to accomplish such a system is by creating a turn-based structure. Just like playing a board game, nodes await their turn and only transmit when it is their turn. When it is not their turn, they receive and interpret the messages from the others. In this approach, all nodes are hierarchical equivalent and no master or coordinator is required. Each node needs only to know its own ID and the maximum number of nodes in the network. Figure 3.7 illustrates the system using 4 nodes.

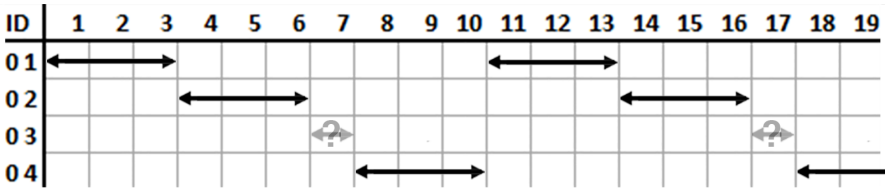


Fig. 3.7 Turn-based control diagram

In this example, each node has a time slot of $3t$ per turn and a wait time-out of t. At the beginning, it is assumed that node 1 is allowed to start at $t = 0$. As soon as node 1 starts to transmit data, all other node switch to receiving mode. Since the predefined package format described in 3.4 contains the ID of the sender, all nodes in the network have knowledge of the ID of the node who's turn it is. This is essential for determining the next node. After the time slot of node 1 has elapsed, all 4 nodes go into a wait mode. They determine the node that is next (in this case node 2) and compare that ID to their own ID. If a node finds it is next, it starts to transmit and this automatically causes all other nodes to switch to receiving mode.

Now assume that node 3 is not present or has turned off for some reason. After the time slot of node 2 has elapsed, all nodes switch to the wait mode and wait for the next node (node 3) to start its communication. However, since no active node has ID 3, nothing happens. Therefore after a predefined wait time-out, all nodes increase the next-node ID and check if they match that ID. Now node 4 concludes it is next and starts to transmit, which causes the remaining nodes to go to receiving mode.

It is required that the system runs this turn-based scheme in a loop, so after the last device is done transmitting, the first one should start again. This is why it is important to have knowledge of the maximum number of nodes possible in the net-

work. When node 4 is done transmitting and all devices go into wait mode, the nodes realise that they have reached the highest ID possible in the current network. So, instead of increasing the ID, they all return to the first ID of the sequence and thus node 1 starts to transmit again and the whole system repeats.

Figure 3.8 shows the transition between states for a single node in the network.

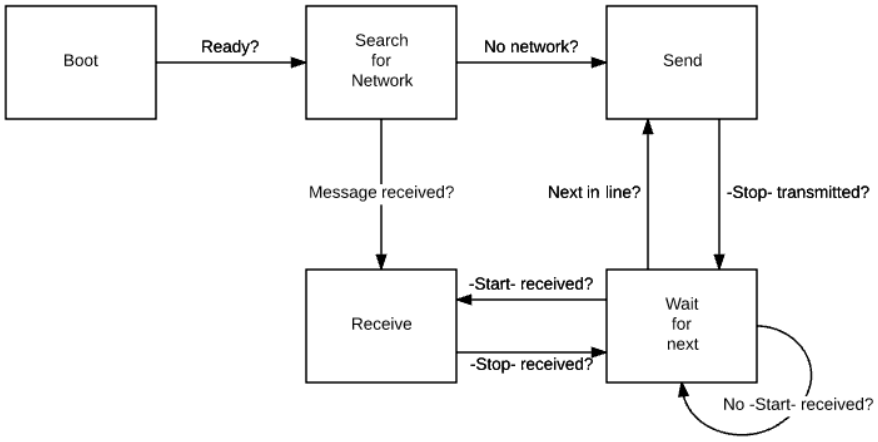


Fig. 3.8 Turn-based control diagram

When the node is powered up, it will go into the *Boot* state. Here all important system parameters and peripherals of the micro controller (timers, UART, interrupts) are initialised. When this is done, the node continues into the *SearchForNetwork* state. In this state it will wait and listen to communication from an already existing communication cycle. When it receives a transmission, it immediately goes to the *Receive* state and joins the active communication cycle. If no message has been received after a predefined timeout, the node is apparently the first and will start its own localisation cycle by going to the *Send* state.
When the node enters the *Send* state, it will transmit a predefined -*Start*- command, which will indicate to the other nodes that they must go to receiving mode. After this, the node has a predefined time to use the communication channel for broadcasting its data, before it must send a -*Stop*- command. When the other nodes receive this -*Stop*- command, all (including the sender) go into the *WaitForNext* state, in which they wait until the receive the next -*Start*- command. When a node realises it is the next one to transmit, it will go to the *Send* state and the whole loop starts again.

Chapter 4
Communication prototyping

4.1 Prototyping boards

To create a simple proof-of-concept prototype for the communication system described in chapter 3, a total of 4 prototyping boards have been developed to be able to do tests with the communication protocol and turn-based communication system. Figure 4.1 shows one of these protoboards.

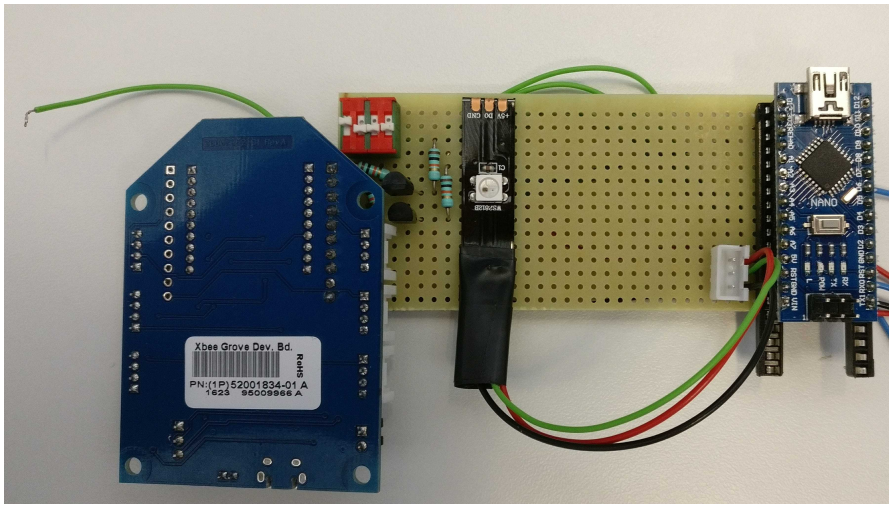


Fig. 4.1 One of the four prototyping boards

The prototyping boards contain the following elements:

- **Arduino Nano**
  The Nano is used as an easy way to prototype with the microcontroller. It al-

ready has all the necessary peripherals like a crystal, voltage regulator and programming header to make debugging code easy and fast.

- **XBee S2C DigiMesh**
  As explained in chapter 3.1.3, an XBee module will be used as the wireless communication channel. Because the footprint of this module does not fit on a prototyping board, a XBee Grove development board is used with header pins to connect the XBee module to the other hardware
- **Level shifters**
  Because the Nano operates at 5 V and the XBee module at 3.3 V, level shifting is required to connect the Rx and Tx lines of the Nano and the XBee. More about these level shifters is explained in System Integration, chapter 5.1.2
- **DIP switch**
  As explained in chapter 3.5, each node needs to have an unique ID. To keep the testing with the prototyping boards simple and efficient, each board has 4 dip switches with which an ID for that board can be chosen (ranging from 0 to 15).
- **WS2812 LED**
  In order to give a real-time indication of the state of the device, a single WS2812 LED is included, which can light up in any colour. More about the WS2812 LED is explained in System Integration, chapter 5.1.2

After developing the necessary code in C, the 4 prototyping boards were able to successfully communicate with each other and follow the desired turn-based protocol as described in chapter 3.5. Figure 4.2 shows the 4 prototyping boards working in harmony.

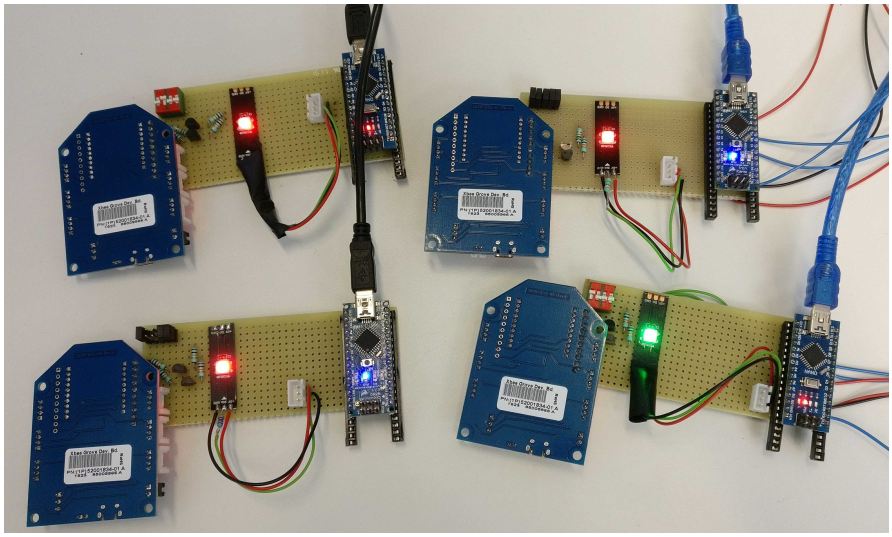


Fig. 4.2 All 4 prototyping boards communicating with each other

When the LED is red it indicates that the device is in its *receive* state. Green corresponds to the *send* state and blue is assigned to the *wait* state. Figure 4.3 shows that when one node is disconnected, the others will continue to communicate. When it is the turn of the node that has just been turned off, the other nodes will go into the *wait* state until a start command arrives or when the predefined timeout elapses. Because the node is turend off, a start command for its ID will never be send, so after the timeout elapses, the next ID in line sends it start command and the communication continues.

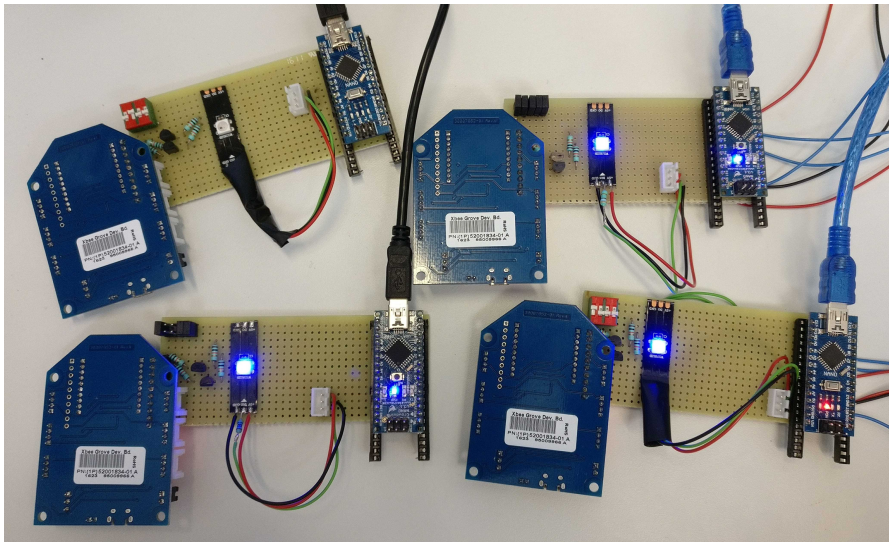


Fig. 4.3 One board is switched off, the other nodes are in the *wait* state

4.2 Monitoring

In order to monitor and debug the behaviour of the 4 prototyping boards, a PC application was developed to monitor the current state of and communication in the network. A screen shot of the application is give in figure 4.4

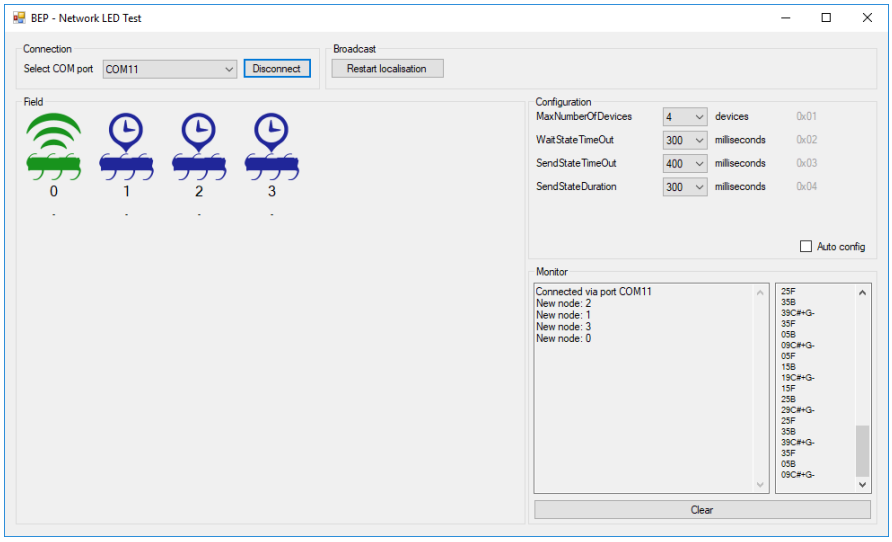


Fig. 4.4 First version of the PC application

The main function of the PC application is to give the user insight to the current state of the network. In the first version of the firmware with which the prototyping boards are programmed, the nodes in the network are simply broadcasting their ID and current configuration in the turn-based communication loop, explained in chapter 3.5. No other information is available to the micro controller yet, so after sending its configuration, the node simply waits for a fixed number of milliseconds (*SendStateDuration*). After that, it sends a message indicating its turn is over and so the next node can start transmitting. If the other nodes do not receive this finish-message before a predefined timeout (*SendStateTimeOut*), it is assumed that the node has left the network for whatever reason and the communication loop continues onto the next node. Note that this means that *SendStateTimeOut* has to be longer than *SendStateDuration* to ensure stable behaviour.
If the next node does not start the communication before a predefined timeout (*WaitStateTimeOut*) the next node in line is allowed to start. Just the same, when it does not start communication before the timeout (*SendStateTimeOut*), the next node is selected and so on.
As explained in 3.5, it is required to know the maximum number of nodes possible in the network. Because the system must be easily extensible to more nodes, this property is also software-definable on the micro controller (*MaxNumberOfDevices*)

4.2.1 Configurability

The 4 parameters currently available on a node (*MaxNumberOfDevices, WaitStateTimeOut, SendStateTimeOut, SendStateDuration*) can be programmed using the PC software. However, since it is desirable that the nodes retain their configuration values after a power down, the values are stored on the micro controller's EEPROM.
Eventually, it might also be desirable for the network to perform checks that ensure that all nodes in the network have the same configuration. For instance, when the *WaitStateTimeOut* of one node is shorter than that of the others, it can cause network failure, because the node might start communicating out of turn. At the time this thesis was written, this functionality has however not been developed yet.

4.3 Results

After a few iterations of bug fixing on the firmware, the 4 prototyping boards were able to seamlessly communicate in the desired protocol. Also, removing and adding nodes to the live network worked as desired. Furthermore, programming the EEPROM via the PC software turned out to be a hassle at first, because doing this during a nodes *send* state would often cause it to expire the *SendStateTimeOut*, which meant the rest of the network would continue and that particular node would be out of sync. Fortunately, this problem could be tackled by switching the node to its *wait-for-network* state after configuration.

4.4 Future goals

At the time this thesis was written, a few goals were yet to be achieved.

- The CRC as explained in section 3.4 had not yet been included in the package protocol. To ensure errors are detected, this remains a desirable feature that will be added in the future.
- As explained in section 4.2.1, the ability for nodes to adjust their configuration to the current network configuration was yet to be implemented. This would further improve network stability.
- The hardware and firmware integration with the ranging subgroup had been tested with the prototyping boards, but this still left a few issues that had yet to be tackled
- The firmware integration with the processing subgroup still had to be tested. Since this would first require the integration with the ranging subgroup, it had not been tested yet.
- As will be explained in chapter 5, the ultimate goal of the project is to deliver one integrated system. This would mean that the components on the prototyp-

ing boards would be combined on a single PCB. Since the lead times on PCB manufacturing are quite long, there had not yet been a chance to test this.

Chapter 5
PCB design and System integration

This part of the thesis will cover the system integration of the all the participating groups in the project. Next to the communication module a ranging part and a processing part was developed. To be able to seamlessly integrate the other parts of the project a printed circuit board, PCB, is needed. The PCB is designed using KiCad. KiCad is a free software programme for electronic design automation (EDA). KiCad facilitates firstly the design of schematics for electronic circuits and afterwards the conversion to PCB designs and Gerber files. We'll first start by describing the designed circuit to later on specify the design choices in the circuit and on the PCB.

5.1 Schematics

The full electrical schematic consists out of 114 electrical components. Below a summary of all the chosen parts is given and a bill of materials has been placed in the Appendix. In figure 5.1 a block schematic is shown to give an overview of the total system.

5.1.1 System overview

As shown in figure 5.1 the system can be divided into multiple parts. At the top and at the centre of the system we'll find the micro-controller with the power supply next to it. The microcontroller communicates with an XBEE S2C and and a Zebro bus interface. Further more the microcontroller sends RGB data to a led ring and collects data from a compass, temperature sensor, current sensor and voltage sensor. A localisation module created by another group, from now on referred as the ranging group, sends and recieves data from the microconrtoller for localisation processing. This "external" part is to be integrated in to our final product. In the specification of the system components we'll first cover the microcrontroller.

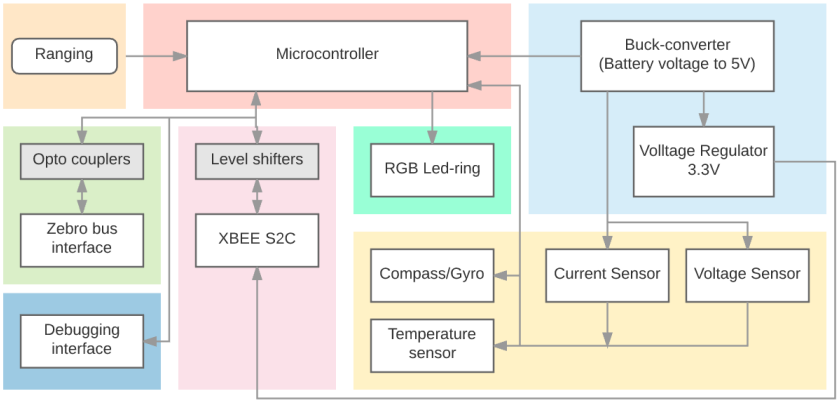


Fig. 5.1 A block representation of the total system

5.1.2 Components

ATMega Chip For the microcontroller the ATmega644AP was chosen. The AT644 is an 8-bit Atmel Microcontroller with 64K Bytes in-System Programmable Flash. Furthermore the microcontroller has 8 channels of 10bit ADCs and 2 UART ports, which is ideal for ZigBee communcation in combination with internal communication over a Zebro bus. The AT644PA has an internal clock of 1MHz which tends to be inaccurate. To be able to compute at higher speeds for communication and processing purposes -and to provide a more accurate clock, we chose to add an external crystal oscillator to the microcontroller that would boost the clock 16 times to a frequency of 16Mhz. The ATmega644PA has an operating voltage of 1.8 to 5.5V and has a speed grade of 4.5 - 5.5V @ 0 - 20MHz, [**?**] because of this we chose a supply voltage of 5V.

Power supply To supply the communication module with the internal Zebro batteries a powersupply was needed. The choice was made to design a step-down buck converter that could scale down the battery voltage, ranging from 14 to 18V, to a steady 5V. Later on the 5V supply would be again be scaled down to 3.3V for the Xbee® Module. For the Buck converter a LM2596 3.0 A, Step-Down Switching Regulator was chosen manufactured by ON Semiconductors [**?**]. One would note that the possible amperage for this regulator is pretty high for our case, but this was deliberately chosen because the buck converter would supply a WS2812 Led ring next to the ATmega644AP microcontroller and other components. The LM2596 has

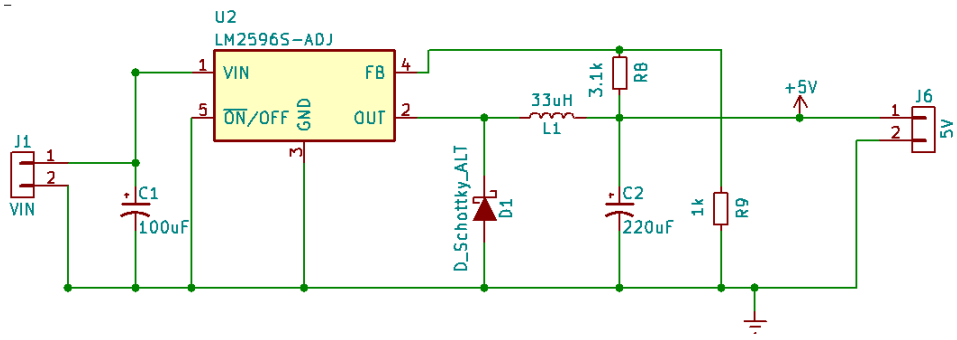


Fig. 5.2 The Buck Converter Circuit

an adjustable output voltage range of 1.23 V to 37 V and a wide input voltage range up to 40 V with good line and load regulation. A feature that also made the LM2596 attractive is that it was decently compact and easily adjustable. A schematic of the buck converter is given in figure 5.2. To set the output voltage, only the two resistors, R8 and R9, had to be adjusted according to the following formula:

$$V_{out} = 1,23 * (1 + \frac{R8}{R9}) \tag{5.1}$$

The chosen values for R8 and R8 are $3,1k\Omega$ and $1k\Omega$. For the Xbee-S2C® power supply the LM3480IM3-3.3 fixed LDO voltage regulator was chosen. This LDO also has a wide input voltage that could also easily be implemented in the system to ensure a more robust design.

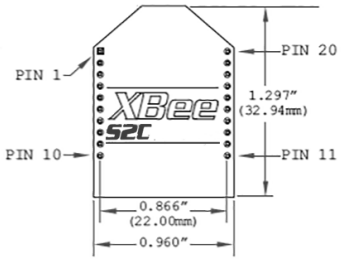


Fig. 5.3 The through-hole XBee® S2C ZigBee® RF Module dimensions

Xbee® Module For the antenna the XBee® S2C ZigBee® RF Module was chosen. The XBee modules provide wireless connectivity to end-point devices in ZigBee mesh networks. A through-hole variant was chosen for the XBee to enable easy placement and removal of the component from the PCB with the use of headers and to reserve some space for other components placed below. As mentioned earlier the XBee is powered with 3.3 Volts and uses UART to communicate with the microcontroller at 120 mA and +18 dBm transmitting power. [**?**]

Level shifters For the data link between the ATMega microcontroller and the XBee-S2C level shifters were needed. Level shifters are used when logic signals go from one voltage domain to another voltage domain. In our case RX and TX signals are being exchanged, where the ATMega microcontroller supplies a 5V logic signal and the XBee a 3.3V signal. To solve this problem a level shifter will scale the signal for the XBee component coming from the ATMega chip so that it can read logic-1 or logic-0 correctly.

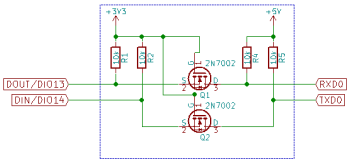


Fig. 5.4 The 5-3.3V Level shifters

Led ring The communication module must be able to visualise the information produced by the ranging and processing groups. To do this and ensure a higher level of Zebro-Human interaction there was chosen to implement a 24-bit LED ring. The LED ring is of type WS2812 5050 with integrated drivers. The only thing that has to be added to the circuit to drive the LED control signal is a series 400Ω resistor to protect the drivers.

Sensors One of the requirements of the project were that the Zebro should be able to transmit housekeeping data such as temperature, current and voltage levels. To enable the zebro to do so, three sensors were added to the board:

Current sensor: The current sensor was easily created by firstly measuring the voltage over a small resistance and converting this differential (floating) voltage to one with ground. After this the small voltage of around 50mV would be amplified to circa 5V with an opamp and presented to the microcontroller. Protection measures were taken by adding diode at the end connected to a 5V supply to prevent possible damage to the microcontroller.

Voltage sensor: Similar to the current sensor, the voltage sensor was implemented by adding a basic voltage divider to get VS+ from 20V to 5V again with protection by a diode connected to a 5V supply.

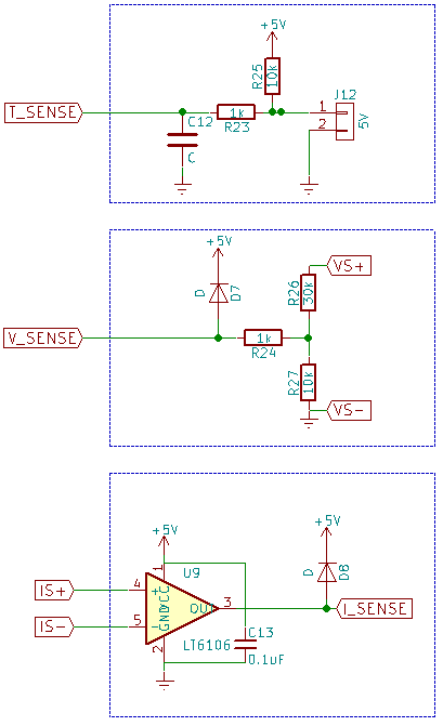


Fig. 5.5 The temperature, voltage and current sensors.

Temperature sensor: For the design of the temperature sensor thermistors were used. Most types of thermistors have a Negative Temperature Coefficient of resistance or (NTC), which implies that their resistance value goes down when the temperature is increased. An NTC thermistor provides a very high resistance at low temperatures. As temperature increases, the resistance drops quickly. Because an NTC thermistor experiences such a large change in resistance per C, small changes in temperature are reflected very fast and with high accuracy (0.05 to 1.5 C). A capacitance was set parallel to the NTC to create a RC filter which would filter out most of the noise and thereby ensure that only a slow increase or decrease of temperature could be measured on the input of the ATMega microcontroller.

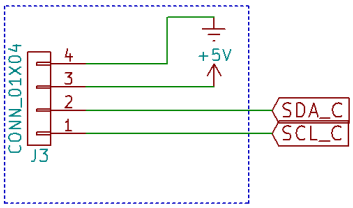


Fig. 5.6 The gyroscope connectors

Gyroscope and Compass For the gyroscope and compasss the MPU-9250 was chosen. The MPU-9250, at 5V, has a low power consumption and houses a 3-Axis gyroscope, accelerometer and magnetometer from Asahi Kasei Microdevices Corporation. The MPU-9250 also houses a dedicated I2C sensor bus, making it particularly effective for our use. The chips allows precision tracking of both fast and slow motions by offering a user-programmable gyroscope with full-scale range up to 2000 degrees per second. The small board with dimensions of 40mm x 20mm is to be mounted on the board with a 4 pin JST connector providing 5V, GND and both SDA and SCL.

Debugging leds and pins To facilitate the process of testing and debugging, the choice was made to assign as much headers as possible to all the non occupied microcontroller pins. Furthermore 4 LEDS were integrated into the design with four different colours to make it easier to distinguish them from each other.

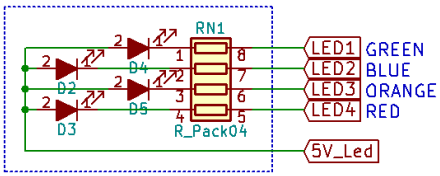


Fig. 5.7 The debugging leds

Zebro bus interface Because the module is to be integrated on a Zebro system, some of the standardised communication with the main module of the Zebro had to be incorporated. The interface for this is described by the Zebro project in [10] and [11]. The interface has 2 components, a I2C communication interface refered to as the Zebro-bus and an interrupt pin. The

Zebro-bus is used for the communication between the submodule and the brain of the Zebro. The brain acts as an I2C-master and all modules are set up as I2C-slaves. The interrupt is a safety feature that enables any module in the Zebro system to send a 'distress' signal. When the (NOT)Interrupt pin becomes low, the brain can decide to shutdown the system and find out which module triggered the interrupt and why.

Optocouplers One important aspect of any electrical device is safety. In terms of electrical safety and also durability, it is important to provide electrical isolation between modules whenever possible. A common and practical way to do this is to use optocouplers. These integrated circuits consists of a LED and a photo-diode. When the LED is on, the photo-diode starts to conduct, when it's off, there is no conduction. Because light is used as the carrier, there is galvanic isolation between both sides of the optocoupler, with values typically around 5 kV. This way, when a module fails and high currents or voltage are present, the rest of the system is protected.

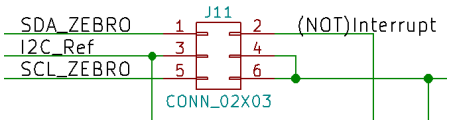


Fig. 5.8 The Zebro interface

Ranging circuit For the bachelor graduation project, another group had designed a ranging circuit for localisation. A more in depth specification of the design choices can be found in their thesis [15]. For ranging an ultrasonic transceiver was needed. An ultrasonic transceiver can transmit and receive signals at a frequency of 40 kHz, also known as ultrasonic. The entire ranging circuit, designed and developed by the ranging group in the project consists therefore out of two parts: a transmitting circuit and a receiving circuit. We'll start with the latter.

Receiving circuit: To make sure that received signals have a sufficient amplitude to be processed by the ATMega micro controller, a signal processing unit is needed. Received signals with a relatively small amplitude will first be amplified by a detection circuit with operational amplifiers. To ensure sufficient amplification the choice was made to implement a two stage amplification as shown in the circuit below. The ultrasonic transceiver is biased at 4.5V which is half of the supply voltage because received signals are symmetrical around 0 Volt. Biasing in this manner will result in an amplification of both negative an positive signals and a peak to peak voltage of 9V.

After the two stages of amplification the signal enters a peak detection circuit. The peak detection circuit functions as a an aid for the comperator in the next stage. By charging a capacitor at the input of the peak detection circuit a saw tooth signal is created that will not drop below a input DC voltage. After the peak detection

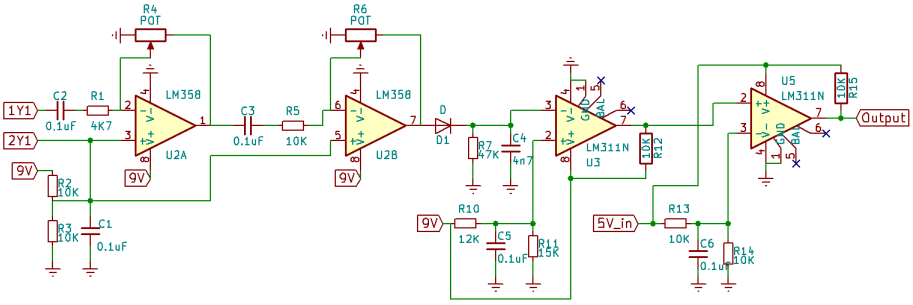


Fig. 5.9 The Receiving Circuit

circuit the signal is fed to a comperator wich compares the input to a DC threshold voltage. The threshold value is placed between the values of the voltage when the circuit is not receiving anything or the minimum value when the circuit does receive a signal. By doing this the circuit is able to detect falling edges when the ultrasonic transceiver receives a signal, hereby producing a ”clean” zero or 5V output. As shown in the figure 5.9, the sytem is complimented with several capacitors to stabilise the output voltage in the interconnecting stages of the circuit.

Transmitting circuit: For the transmission of signals the MIC4428 is used made by Micrel. The MIC4428 functions as an ultrasonic transmitter driver circuit by converting a 40kHz, 5V, PWM signal from the ATmega microcontroller pins to a 30V peak-to-peak transmission signal. The transmission circuit works by switching and inverting a 15V supply with an H-bridge. The schematics are shown below. For a more in depth elaboration of the schematics please consult the thesis written by the ranging group. [15]

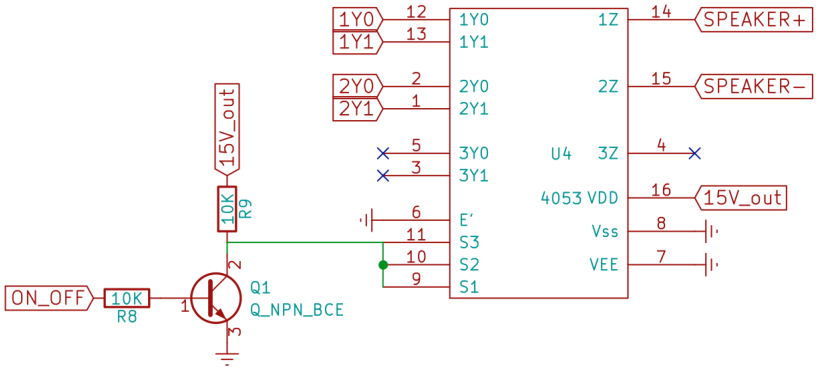


Fig. 5.10 The Transmission Circuit

5.2 The PCB

5.2.1 Footprints

After making several iterations of prototypes on protoboards a final design was chosen and put into an schematic using the Eeschema tool. With testing finished we started the design of the PCB and the layout of the footprints. Footprints give the size and dimensions of the used components. In a schematic they are usually not displayed as they are of no importance during the design of the electrical circuitry. Footprints are used during the design of the layout of the printed circuit board and provides the user with a blueprint as guide during the process of component placement. After completing the schematics in EEschema the components are linked to their respective footprints in CvPCB. After this a so called ”Netlist” can be generated that consists out of all the associated components and footprints together with their interconnecting wiring.

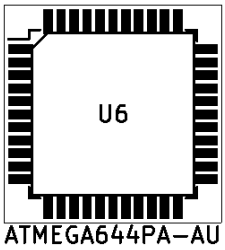


Fig. 5.11 An example of a footprint, the ATMega Microcontroller

5.2.2 Layout

After the footprints were chosen it was time to determine the ideal locations of the components on the board. During this process it is very important to keep a few things in mind. The datasheet of the buck-converter dictates that all the capacitors should be placed near each other. Furhtermore the crystal oscillator that is used to clock the microcontroller at 16Mhz should be placed as close as possible to the ATmega chip. This is because long tracks might cause distortion and clocks tend to be very sensitive to these.

5.2.3 Layers

The designed PCB will consist out of different layers. On the top level there is a silk screen wich is used to provide neccessary textual information. Below the Silk screen there are two copper layers. Two copper layers are used to ensure that all components can be connected to each other without crossing each other which will make the device malfunction. The top and bottom layers are present throughout the entire PCB, where the aim was to use the bottom layer as much as possible for a ground plane to minimise interference of other signals.

5.2.4 Routing

Because the PCB consists out of two layers, multi-level outing, or tracing, was possible. Routing is the process of laying down electrical traces to connnect the components of the board with each other. To avoid spacing problems it was needed to set the track width of the traces no bigger than the width of the microcontroller pins. During the routing process we also had to keep in mind that if we were to make traces cross each other on the two different levels; this should be tried to be done in a perpendicular way, minimizing the amount distortion. The use of of vias should also be kept to a minimum to minimise distortion of the signal, the minimisation of vias would also spare a lot of time later on if we would choose to mill out prints. This is because after milling out prints it would be needed to manually metalise the vias. As required, the Zebro bus signals had to optically coupled, this required carefully selected areas with separate ground plains.

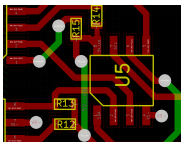


Fig. 5.12 An example of tracks and vias used for routing the PCB

5.3 Main components

5.3.1 The Buck converter

Since the buck converter provides the supply voltage for nearly all the components on the PCB, this part was designed and tested first to make sure that the selected components would function sufficiently together. The buck converter was relatively simple to design and only consisted out of 7 components. For the sake of simplicity we chose to use only one layer because the ground were easy to connect and this would save milling time during production. A plot of the buck converter is shown to the right. The buck converter was milled out of a FR4 single sided ($35\mu m$) copper plate and tested. The tests consisted out of cases where the buck converter was fed by a voltage of 12V, the minimum possible battery voltage, and in the other case with 17V, the maximum battery voltage. The buck converter was also connected to an Arduino Uno to simulate a load. As can be seen from the figure below the output of the buck converter was practically the same at 12V input as at 17V input, rendering it sufficient for our PCB.

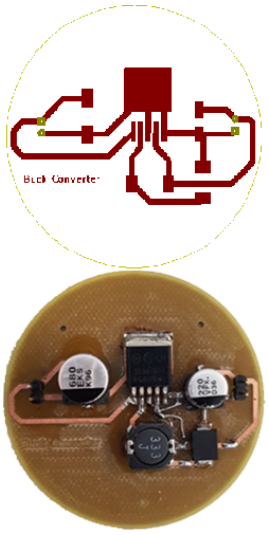


Fig. 5.13 PCB layout of the Buck converter and the produced board

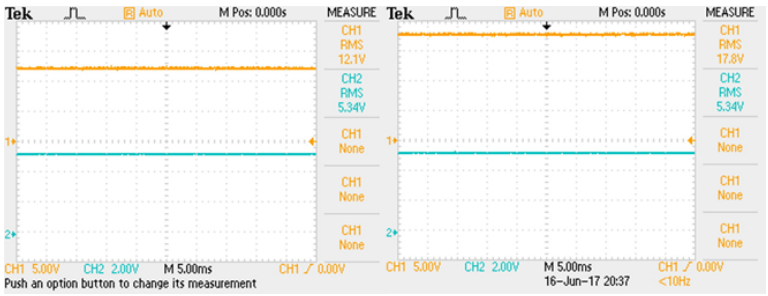


Fig. 5.14 Measured output values of the Buck converter compared to input voltage

5.3.2 The Ranging PCB

As mentioned earlier, another group had been working on the schematics for their ranging circuit. We assisted them with the developement of their PCB. Just as the buck converter the ranging pcb was relatively easy to design and consisted out of 41 components. The buck converter was milled out of a FR4 double sided ($35\mu m$) copper plates and tested was tested with signal generators and oscilloscopes. The results of these tests can be found in their report. [15]

5.3.3 The Communication PCB

To mechanically fasten the communication circuit to the Zebro a twist lock connection is required. This twist lock mechanism has a circular top surface with a diameter of 8 cm. To seamlessly fit the PCB onto the lock mechanism, the choice was made to make the PCB circular too with a diameter of 7,9cm. As mentioned earlier the communication module consists out several key components; the ATmega644AP microcontroller, the power supply, the Zebro Bus interface and the XBEE S2C.

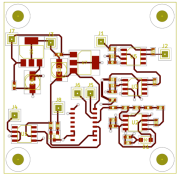


Fig. 5.15 THe PCB layout of the ranging module

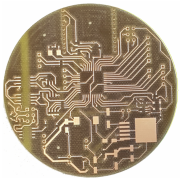


Fig. 5.16 The final milled out version of the communication module

At the the start of the communication PCB design these components were placed first, starting with the microcontroller. The ATMega chip was placed conveniently at the centre of the PCB with the external oscillator nearby. After this the power supply components were placed at the bottom with the corresponding voltage and current sensing circuits. The Zebro bus communication interface was placed at the left side of the circuit, making

sure the two different ground planes were clearly separated. A simplified overview of the PCB is shown in figure 5.17 with the coloured zones. Compared to all the other components, the XBee is the largest with dimensions of 2.43cm x 3.29cm. Because of the area constraints and shear size of the component; it was placed centered to the top of the board with the ATMega chip mirrored on the other side for easier routing. Because the XBEE module contains a RF antenna a keepout area was needed, this required us to place the component as depicted in the figure. The dimensions of the created vias, tracks and drills is given in the table below. Table 5.1.

Table 5.1 Track and hole widths communication PCB

Track width	Via width	Via drill size	Headers and Xbee hole width
0,500 mm 19.69 mils	0,80 mm 31,5 mils	0,50 mm 19.7 mils	1 mm 39,4 mils

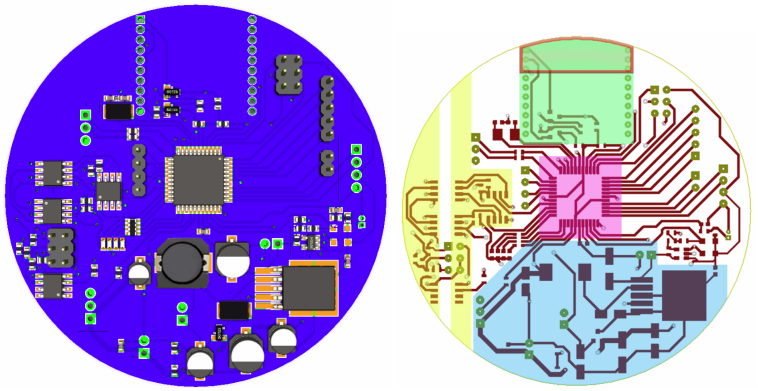


Fig. 5.17 (Left) a render of the communication PCB from PCBview. (Right) 4 coloured zones of the -Pink- ATmega644AP microcontroller, the -Blue- Power supply, the -Yellow- Zebro Bus interface and the -Green- XBEE S2C module with the keepout zone

Because the micro controllers are soldered onto the board, it is important that reprogramming them on-the-fly is possible. For this, a dedicated header is included known as an In Circuit System Programmer (ICSP) header. With the proper programming tool it is now possible to program the micro controller when it is soldered in the circuit, which is very helpful when debugging or when firmware updates are released

5.3.4 System integration

To finally combine the circuit designed by the ranging, processing and communication groups we chose to use two layers of PCB's that are stackable. By keeping the communication module on a separate layer it would be possible to easily remove and modify the ranging circuits and provide the possibility to only use the communication module for certain Zebro set ups. The two PCB layers are to be stacked on top of each other using M3 PCB risers and the entire stack would be placed, if needed, under an ultrasonic transducer with a 3D printed housing. See figure 5.18 for an impression. The design is chosen in this way to keep everything modular and provide the possibility to add or remove any PCB levels as long these have 4 M3 holes for the risers.

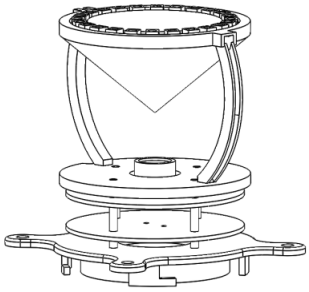


Fig. 5.18 3D render of the combined Communication-Ranging module

Unfortunately we did not have enough time to fully integrate all the components in PCBnew and order to the PCB's online in time for this thesis. Our estimation is that this is possible with two weeks of extra work. A mock-up of the two final PCB's are given below in figure 5.19, showing the communication components on the bottom level PCB and the ranging and processing components on the top level. We compared the available space for the ranging components on the top level and the size of the previously milled out print of ranging and expect that there will be no issues whatsoever concerning available footprint space.

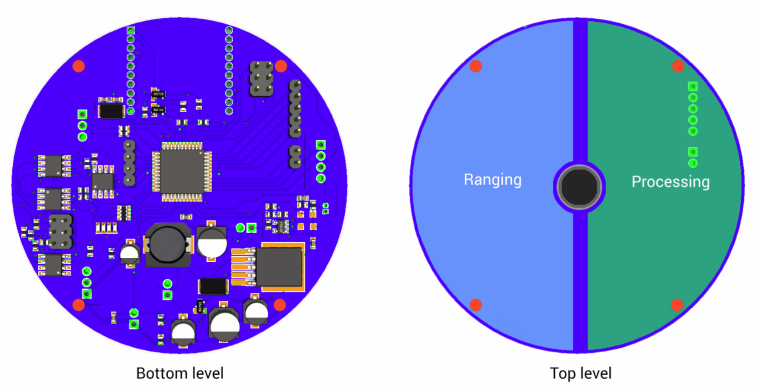


Fig. 5.19 A muck-up of the final PCB with two seprate layers for Communciation and Ranging/Processing

Chapter 6
Discussion

Communication At the beginning of the project our aim was to develop a communication module capable of autonomous communication with neighbouring zebro's. When we first started doing literature research it was immediately clear that the developement of an own mesh network with all the routing protocols was a very challenging and time consuming task that could not be done within 8 weeks of a project. Within a week of research DigiMesh was found to be the most efficient and effective networking application, with an easy set-up, low power consumption and hardware that we were familiar with. The use of XBee with DigiMesh has it's pros and cons. The XBee modules for instance are easy to install, provide a robust communication network and do not need any specific programming to work with DigiMesh. This comes at a cost however; DigiMesh is a proprietary software which makes it harder to understand the "guts" of the programme and to modify it. By using DigiMesh we are also forced to purchase moderately expensive XBee modules, instead of creating our own communication hardware.

Group dynamics After 2 weeks of working on the project, the team was separated into three groups, processing, ranging and communication. Next, we ordered the XBee modules and and got them up and running the week after. The communication algorithm, the testing and the GUI soon followed, giving us the impression we were making good progress. At that time we reached a bottleneck where in we had to wait for the other groups to provide their work for us to continue with out work. Therefore we decided to let Marck help the ranging group that was having hardware problems at the moment and to broaden the scope of our project by including system integration in our thesis. Ronald started focusing on system integration by mapping all the schematics and footprints in KiCad and by milling out test circuits. Together with Marck all the preparations were made for the fabrication of the PCBs and the plan is to assemble these prints in the coming two weeks. We'll continue with our future work in the next section.

Chapter 7
Conclusions

To verify whether the module we developed during the project is viable for application in the ZEBRO project, a check to see if it meets all the requirements set out in chapter 2 is done,. Because not all of the requirements were applicable on the communication and system integration module, a distinction is made as to which ones were relevant for this sub module. Below the programme of requirements is listed once again with details about if the requiremennt is met.

7.1 Functional Requirements

[F-1] The module should be able to locate multiple neighbouring Zebros
This requirement is part of the ranging and processing subgroup

[F-2] The module should be able to receive and transmit packets to neighbours *Yes, we have set up a communication network and are able to send and receive packets to all the neighbouring zebros*

- **[F-2.1]** The module should be able to make contact with at least 3 neighbouring Zebros
  Yes, the module is capable of connecting with every Zebro in its network
- **[F-2.2]** The module should be able to broadcast housekeeping data for inspection by users
  Yes, any content can be sent via the communication module, this includes housekeepingdata

7.2 System Requirements

[S-1] The mechanical, power, and digital connections must conform to the specifications outlined in [10]
The module is designed to conform with this

[S-2] The module must conform to the Zebrobus protocol outlined in [11]
The module has a dedicated I2C connection on board as described by the Zebrobus protocol

[S-3] The module must respect the modularity of Zebro modules

- **[S-3.1]** The module should be a slave to the Zebro bus and cannot make requests itself
- **[S-3.2]** Use of the module cannot inhibit the functionality of other modules on the Zebro
  Yes, This requirement is met
- **[S-3.3]** With the exception of data broadcast on the Zebro bus, the module should collect data exclusively using its own sensors
  Yes, All necessary data (current, voltage) is measured on the module itself
- **[S-3.4]** The module should communicate its data to the Zebro by responding to requests over the Zebro bus *Yes, The module has a dedicated I2C connection on board as described by the Zebrobus protocol*
- **[S-3.5]** The modules should be interchangeable between robots and easily replaceable *Yes, The mechanical form factor of the module conforms to this*

[S-4] The module should be able to be used in indoor environments
No (sub)system which can not function indoors has been used

[S-5] The communications network should be homogeneous

- **[S-5.1]** Communications cannot rely on a predetermined master node or network hub
  DigiMesh operates with universal node types without masters or hubs
- **[S-5.2]** An individual Zebro within range should be able to join the communications network at any time
  DigiMesh allows for nodes to join the network live
- **[S-5.3]** Any individual Zebro in the network should be able to leave without affecting network functionality
  DigiMesh is self-healing and allows for nodes to leave the network live

[S-6] The module should be usable on stationary elements, such as a charging station, without any changes to the hardware
The module is universal for all types of ZEBRO objects

[S-7] The module connects to the Zebro power system (see **[S-1]**)

- **[S-7.1]** The power system is rated at 13 V to 19 V *Conversion to operating voltages (5V, 3.3V) is done on the module*

[S-8] The module should be able to provide debugging information over the Zebro bus *This is possible and the module even has a dedicated UART connector for debugging*

7.3 Performance Requirements

These requirements are part of the ranging and processing subgroup

7.4 Development Requirements

[D-1] The module production cost shall not exceed €150 per unit *The costs of the communication and system integration part does not exceed this (by far). The estimated production cost is estimated at €60, including PCB manufacturing*

[D-2] The module PCB should be able to be fabricated using a pick-and-place service *Almost all components are SMD (which is pick-and-placable)*

This shows that the submodule meets all the relevant requirements.

Chapter 8
Future work

In the couple of weeks that still remain for our project, our planning is to finalise the work done on the system integration part, with as an ultimate goal to be able to demonstrate at least 3 working modules which are able to locate each other and visualise this on the LED ring. Figure 8.1 gives an example of what this could look like.

Here the colors indicate the proximity to the neighbour and the position on the LED ring indicates the angle at which the neighbour is with respect to itself. Assuming that the cable entry (top of the image) is the front of the module and the color red indicates a neighbour closeby, green at medium range and blue far away, the image shows that four neighbours have been detected. Two are closeby, at angles of approximately 30 and 225 degrees, one is at medium range at an angle of 300 degrees and one is far away at an angle of 105 degrees.

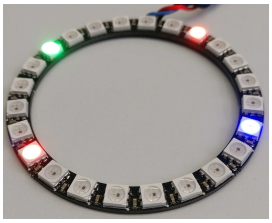


Fig. 8.1 The LED ring indicates its neighbours

In order to achieve this goal, some work still has to been done.

- Integration between the communication sub module and the ranging sub module has to be optimised. As explained in section 4.4 there are still some problems that have to be overcome for the two sub modules to fluently work together.
- The programming code from the processing subgroup is ready for implementation, but there has not yet been an opportunity to test this code on a micro controller or to verify the integration on the micro controller already running the communication protocol.
- The hardware necessary for the processing sub group (a sensor which determines magnetic heading) has to integrated into the subsystems. At first, it was decided to do this all on one micro controller, but unfortunately the complexity

of the operations necessary to extract the relevant data from the raw sensor data would cause the performance of the entire to drop significantly. To this end, it was decided that a dedicated micro controller would be used for the processing of this sensor data. However, this micro controller also has to be integrated to the system.

- To make testing possible, the PCB as described in chapter 5 has to be ordered and assembled with the actual hardware. At this point it is not longer viable to use the prototyping boards of chapter 4.1, because the micro controller on the Arduino Nanos is different from the one that will be used on the final PCB. The latter has more memory, pins and peripherals which are necessary for the combined firmware of the three subsystems. With the communication protocols properly working and the sub components fully tested we aim to produce the two PCBs within two weeks. The schematics of all the subgroups have all ready been put together and the bill of materials has been finalised too. The schematic of the entire system can be found in the appendix.

References

1. C. M. Yu, S. K. Hung, and Y. C. Chen, "Forming mesh topology for bluetooth ad hoc networks," in *2013 IEEE International Symposium on Consumer Electronics (ISCE)*, June 2013, pp. 123–124.
2. J. A. Prasetyo, A. Yushev, and A. Sikora, "Investigations on the performance of bluetooth enabled mesh networking," in *2016 3rd International Symposium on Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*, Sept 2016, pp. 56–61.
3. A. Jedda, A. Casteigts, G.-V. Jourdan, and H. T. Mouftah, "Bluetooth scatternet formation from a time-efficiency perspective," *Wireless Networks*, vol. 20, no. 5, pp. 1133–1156, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11276-013-0664-z
4. Y. Wei, M. Song, and J. Song, "An aodv-improved routing based on power control in wifi mesh networks," in *2008 Canadian Conference on Electrical and Computer Engineering*, May 2008, pp. 001 349–001 352.
5. S. Farahani, Ed., *ZigBee Wireless Networks and Transceivers*. Burlington: Newnes, 2008. [Online]. Available: http://www.sciencedirect.com/science/book/9780750683937
6. S. Choudhury, P. Kuchhal, and R. Singh, "Zigbee and bluetooth network based sensory data acquisition system," *Procedia Computer Science*, vol. 48, pp. 367 – 372, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915007048
7. A. Fernandes, M. S. Couceiro, D. Portugal, J. Machado Santos, and R. P. Rocha, "Ad hoc communication in teams of mobile robots using zigbee technology," *Computer Applications in Engineering Education*, vol. 23, no. 5, pp. 733–745, 2015. [Online]. Available: http://dx.doi.org/10.1002/cae.21646
8. Z. Yi, H. Hou, Z. Dong, X. He, Z. Lv, C. Wang, and A. Tang, "Zigbee technology application in wireless communication mesh network of ice disaster," *Procedia Computer Science*, vol. 52, pp. 1206 – 1211, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091500959X
9. D. International. (2015) Wireless mesh networking, zigbee vs. digimesh. [Online]. Available: https://www.digi.com/pdf/wp_zigbeevsdigimesh.pdf
10. *DeciZebro module interface specifications*, The Zebro Project, 2017.
11. P. de Vaere and D. Booms, *ZebroBus Standard Specification*, v1 ed., The Zebro Project, june 2016.
12. S. Bauk, D. G. Gonzalez, A. Schmeink, and Z. Z. Avramovic, "Manet vs. zigbee: Some simulation experiments at the seaport environment," *JITA Journal of Information Technology and Applications*, vol. 6, no. 2, pp. 63–72, Dec. 2016.
13. W. L. Shen, C. S. Chen, K. C. J. Lin, and K. A. Hua, "Autonomous mobile mesh networks," *IEEE Transactions on Mobile Computing*, vol. 13, no. 2, pp. 364–376, Feb 2014.
14. N. B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The cricket location-support system," August 2000. [Online]. Available: http://nms.lcs.mit.edu/papers/cricket.pdf

15. K. Kouwenhoven and S. Verkamman, “Development of omnidirectional distance sensing module for the decizebro,” Technische Universiteit Delft, 2017.
16. M. Chandi and J. Carpay, “Intra-swarm localization from noisy distance measurements,” Technische Universiteit Delft, 2017.
17. Wikipedia, “Computation of cyclic redundancy checks,” 2017, [Online; accessed 16-June-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Computation_of_cyclic_redundancy_checks

Appendices

Appendix A
Micro controller firmware - main.c

```
#define F_CPU 16000000

#define MAX_PACKET_SIZE 16              // Maximum size of communication package
#define RECEIVE_BUFFER_SIZE 32  // Receive buffer size

#define STX             2
#define ETX             3

// EEPROM ADDRESSING
#define ADDRESS_Initialised                     0x00

#define ADDRESS_MaxNumberOfDevices      0x01
#define ADDRESS_WaitStateTimeOut        0x02
#define ADDRESS_SendStateTimeOut        0x03
#define ADDRESS_SendStateDuration       0x04

#define TEMP_OFFSET -7

#include <util/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "light_ws2812.h"

// Debug
#define PIN11_ON()      PORTB |= (1<<3)
#define PIN11_OFF()     PORTB &= ~(1<<3)
#define PIN12_ON()      PORTB |= (1<<4)
#define PIN12_OFF()     PORTB &= ~(1<<4)
#define PIN13_ON()      PORTB |= (1<<5)
#define PIN13_OFF()     PORTB &= ~(1<<5)

// Dipswitch macro's - note that these are temporary
#define DIP_1   (PIND & (1<<3))
#define DIP_2   (PIND & (1<<4))
#define DIP_3   (PIND & (1<<5))
#define DIP_4   (PIND & (1<<6))

// WS2812
struct cRGB color;
struct cRGB led[24];
const struct cRGB colBlack      = {.r = 0, .g = 0, .b = 0};
const struct cRGB colRed        = {.r = 255, .g = 0, .b = 0};
```

```
const struct cRGB colGreen      = {.r = 0, .g = 255, .b = 0};
const struct cRGB colBlue       = {.r = 0, .g = 0, .b = 255};
const struct cRGB colYellow     = {.r = 255, .g = 255, .b = 0};
const struct cRGB colWhite      = {.r = 255, .g = 255, .b = 255};


// UART
#define BAUDRATE 9600
#define BAUD_PRESCALLER (((F_CPU / (BAUDRATE * 16UL))) - 1)
char rx_flag = 0;
char uart0_data;
char uart0_status;
char str[10], uart0_data_str[10];
unsigned char uart0_data_str_cnt;

// States
typedef enum {BOOT, WAITFORJOIN, RECEIVE, SEND, SENDWAITFORCONFIG, SENDWAIT, WAITFORNEXT, MAINTENANCE} states;
states state, next_state = BOOT;
unsigned int last_addr_in_loc = 0;
unsigned int next_addr_in_loc = 0;

// Communication protocol
typedef enum {START, SENDERID, LENGTH, DATA} comm_states;
comm_states comm_state, next_comm_state = START;
char received[RECEIVE_BUFFER_SIZE], sendBuffer[MAX_PACKET_SIZE], messageData[MAX_PACKET_SIZE - 4], messageReceived
unsigned int receivedInsertIndex, receivedReadIndex;
int messageSenderID, messageLength, messageDataIndex;
const char BEGIN[1] = "B";
const char FINISH[1] = "F";


// System parameters
unsigned int addr;
unsigned int timer = 0;
uint8_t MaxNumberOfDevices, WaitStateTimeOut, SendStateTimeOut, SendStateDuration;
uint32_t MaxNumberOfDevicesVal, WaitStateTimeOutVal, SendStateTimeOutVal, SendStateDurationVal;

// Timer
const unsigned int regAfterOverflow = 65520;    // Timer period of approximately 1 ms

void InitUART();
void InitIO();
void InitTimer();
void ReadEEPROM();
void InterpretEEPROM();
void WriteEEPROM();
void InitADC();
void InitSystem();

void SetLEDColor();
void SendConfiguration();
void SendMessage();
void ReceiveMessage();
void DebugReceiveState();

int ReadSystemTemperature();

int charToInt(char in) { return in - 48; }
char intToChar(int in) { return in + 48; }

void USART_send(unsigned char data);

int main(void)
{
        InitIO();
        InitUART();
        InitTimer();
```

```
ReadEEPROM();
InitADC();

InitSystem();

sei();            // Enable global interrupts

while (1)
{
        ReceiveMessage();

        switch (state)
        {
                case BOOT:
                        timer = 0;
                        next_state = WAITFORJOIN;
                break;

                case WAITFORJOIN:
                if (timer >= (MaxNumberOfDevicesVal + 1) * SendStateTimeOutVal)
                {
                        timer = 0;
                        next_state = SEND;
                }
                if (messageReceived == 1)
                {
                        messageReceived = 0;
                        if (messageData[0] == BEGIN[0]) // todo can also be stop
                        {
                                last_addr_in_loc = messageSenderID;
                                next_addr_in_loc = last_addr_in_loc + 1;
                                if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;

                                timer = 0;
                                next_state = RECEIVE;
                        }
                        if (messageData[0] == FINISH[0]) // todo can also be stop
                        {
                                last_addr_in_loc = messageSenderID;
                                next_addr_in_loc = last_addr_in_loc + 1;
                                if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;

                                timer = 0;
                                next_state = WAITFORNEXT;
                        }
                }
                break;

                case SEND:
                // Send BEGIN
                SendMessage(BEGIN,1);

                // Broadcast current configuration
                SendConfiguration();

                timer = 0;
                next_state = SENDWAITFORCONFIG;
                break;
                case SENDWAITFORCONFIG:
                if (messageReceived == 1)
                {
                        messageReceived = 0;
                        if (messageSenderID == -1 && messageData[0] == 'C') // Received config, interpret
                        {
                                MaxNumberOfDevices = messageData[1];
                                WaitStateTimeOut = messageData[2];
                                SendStateTimeOut = messageData[3];
```

```
                SendStateDuration = messageData[4];

                WriteEEPROM();
                InterpretEEPROM();

                SendConfiguration();
                next_state = SENDWAIT;
        }
}
if (timer >= SendStateDurationVal) {next_state = SENDWAIT;}
break;
case SENDWAIT:
if (timer >= SendStateDurationVal)
{
        SendMessage(FINISH,1);

        last_addr_in_loc = addr;        // Own address
        next_addr_in_loc = last_addr_in_loc + 1;
        if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;

        timer = 0;
        next_state = WAITFORNEXT;
}
break;

case RECEIVE:
if (messageReceived == 1)
{
        messageReceived = 0;
        if (messageData[0] == FINISH[0])
        {
                last_addr_in_loc = messageSenderID;
                next_addr_in_loc = last_addr_in_loc + 1;
                if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;
                timer = 0;
                next_state = WAITFORNEXT;
        }
}
else if (timer >= 2 * SendStateTimeOutVal)
{
        timer = 0;
        next_state = WAITFORNEXT;
}
break;

case WAITFORNEXT:
if (next_addr_in_loc == addr)
{
        timer = 0;
        next_state = SEND;
}
else if (messageReceived == 1)
{
        messageReceived = 0;
        if (messageData[0] == BEGIN[0])
        {
                last_addr_in_loc = messageSenderID;
                next_addr_in_loc = last_addr_in_loc + 1;
                if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;

                timer = 0;
                next_state = RECEIVE;
        }
}
else if (timer >= WaitStateTimeOutVal)
{
        timer = 0;
```

```
                                next_addr_in_loc = next_addr_in_loc + 1;
                                if (next_addr_in_loc >= MaxNumberOfDevicesVal) next_addr_in_loc = 0;
                        }

                        break;
                        case MAINTENANCE:
                        timer = 0;
                        next_addr_in_loc = 0;
                        next_state = WAITFORNEXT;
                        break;
                }

                if (messageReceived == 1) messageReceived = 0; // Clear message
                state = next_state;

                SetLEDColor();
        }
}


// Initialize IO
void InitIO()
{
        // DIP switches - set as input and enable pull-up
        DDRD &= ~((1<<3)|(1<<4)|(1<<5)|(1<<6));
        PORTD |= (1<<3)|(1<<4)|(1<<5)|(1<<6);

        // LED test
        PORTB &= ~((1<<3)|(1<<4)|(1<<5));         // Pin low
        DDRB |= (1<<3)|(1<<4)|(1<<5);             // Set as outputs
}

// Initialize UART
void InitUART()
{
        UBRR0H = (uint8_t)(BAUD_PRESCALLER>>8);
        UBRR0L = (uint8_t)(BAUD_PRESCALLER);
        UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
        UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
}

// Initialize timer
void InitTimer()                                                    // Time frame of 1 mS
{
        TCNT1 = regAfterOverflow;
        TCCR1A = 0x00;
        TCCR1B = (1<<CS10) | (1<<CS12);;  // Timer mode with 1024 prescalar
        TIMSK1 = (1 << TOIE1) ;   // Enable timer1 overflow interrupt(TOIE1)
}

// Initialize values stored in EEPROM
void ReadEEPROM()
{
        MaxNumberOfDevices = eeprom_read_byte((uint8_t*)ADDRESS_MaxNumberOfDevices);
        WaitStateTimeOut = eeprom_read_byte((uint8_t*)ADDRESS_WaitStateTimeOut);
        SendStateTimeOut = eeprom_read_byte((uint8_t*)ADDRESS_SendStateTimeOut);
        SendStateDuration = eeprom_read_byte((uint8_t*)ADDRESS_SendStateDuration);

        // Initialize default values when EEPROM has been erased
        char hasBeenInit =  eeprom_read_byte((uint8_t*) ADDRESS_Initialised);
        if (hasBeenInit != 'Y')
        {
                MaxNumberOfDevices = 35;
                WaitStateTimeOut = 61;
                SendStateTimeOut = 61;
                SendStateDuration = 41;
                WriteEEPROM();
```

```
                eeprom_write_byte((uint8_t*)ADDRESS_Initialised, 'Y');
        }

        InterpretEEPROM();
}

void InterpretEEPROM()
{
        MaxNumberOfDevicesVal = MaxNumberOfDevices - 31;
        WaitStateTimeOutVal = (WaitStateTimeOut - 32) * 25 + 25;
        SendStateTimeOutVal = (SendStateTimeOut - 32) * 25 + 25;
        SendStateDurationVal = (SendStateDuration - 32) * 25 + 25;
}

void WriteEEPROM()
{
        eeprom_write_byte((uint8_t*)ADDRESS_MaxNumberOfDevices, MaxNumberOfDevices);
        eeprom_write_byte((uint8_t*)ADDRESS_WaitStateTimeOut, WaitStateTimeOut);
        eeprom_write_byte((uint8_t*)ADDRESS_SendStateTimeOut, SendStateTimeOut);
        eeprom_write_byte((uint8_t*)ADDRESS_SendStateDuration, SendStateDuration);
}
// Initialize ADC
void InitADC()
{
        //ADC Multiplexer Selection Register
        ADMUX = 0;
        ADMUX |= (1 << REFS1) | (1 << REFS0);  //Internal 2.56V Voltage Reference with external capacitor
        //Temperature Sensor - 1000

        //ADC Control and Status Register A
        ADCSRA = 0;
        ADCSRA |= (1 << ADEN);  //Enable the ADC
        ADCSRA |= (1 << ADPS2);  //ADC Prescaler - 16 (16MHz -> 1MHz)

        //ADC Control and Status Register B
        ADCSRB = 0;
}

int16_t ReadADC(uint8_t channel)
{
        ADMUX |= (channel & 0x0F);
        ADCSRA |= (1 << ADSC);  //Start temperature conversion
        while (ADCSRA & (1 << ADSC));  //Wait for conversion to finish
        return ADC;
}

int ReadSystemTemperature()
{
        return ReadADC(0b1000) - 303 / 1.0;
}

// Initialize system parameters
void InitSystem()
{
        // Determine own serial number
        // TEMP: DIP 1 and 2 determine address
        if (!DIP_1 && !DIP_2)   addr = 0;
        if (!DIP_1 && DIP_2)    addr = 1;
        if (DIP_1 && !DIP_2)    addr = 2;
        if (DIP_1 && DIP_2)             addr = 3;
}

// Color WS2812 LED(s) with current state
void SetLEDColor()
{
        switch (state)
        {
```

```
                case BOOT:
                color = colBlack;
                break;
                case WAITFORJOIN:
                color = colWhite;
                break;
                case RECEIVE:
                color = colRed;
                break;
                case SEND:
                case SENDWAITFORCONFIG:
                case SENDWAIT:
                color = colGreen;
                break;
                case WAITFORNEXT:
                color = colBlue;
                break;
                case MAINTENANCE:
                color = colYellow;
                break;
        }

        for (int i = 0; i < 24; i++) { led[i] = color; }
        ws2812_setleds(led,24);
}

// Send the device current configuration
void SendConfiguration()
{
        // Denote the configuration by a 'C'
        char str[5];
        str[0] = 'C';
        str[1] = MaxNumberOfDevices;     // The contents correspons with the EEPROM stored values
        str[2] = WaitStateTimeOut;       // FOR NOW: dummy values to check programming functionalitey
        str[3] = SendStateTimeOut;
        str[4] = SendStateDuration;

        SendMessage(str, 5);
}

// Create message and send it
void SendMessage(char input[], unsigned int length)
{
        if (length <= MAX_PACKET_SIZE - 4)
        {
                int i = 0,j = 0;
                // Create message
                sendBuffer[i] = STX;                            // START
                i++;
                sendBuffer[i] = intToChar(addr);        // Sender ID
                i++;
                sendBuffer[i] = intToChar(length + 4);  // Length of package
                i++;
                while (i < length + 3)
                {
                        sendBuffer[i] = input[i - 3];   // Data
                        i++;
                }
                sendBuffer[i] = ETX;                            // END

                while (j <= i)
                {
                        USART_send(sendBuffer[j]);
                        j++;
                }
        }
```

```
}

// Interprets the packet just received
void ReceiveMessage()
{
        if (rx_flag == 1) // This signals a yet unprocessed incoming char
        {
                char incomingChar = received[receivedReadIndex];

                switch (comm_state)      // The current comm_state indicates what we EXPECT to receive
                {
                        case START:
                        if(incomingChar == STX)
                        {
                                next_comm_state = SENDERID;
                        }
                        else
                        {       // Message invalid
                                next_comm_state = START;
                        }
                        break;
                        case SENDERID:
                        if(incomingChar >= intToChar(-1) && incomingChar <= intToChar(MaxNumberOfDevicesVal))
                            // ID should be between 0 and 9 - 10 id's possible, -1 is the debug PC
                        {
                                messageSenderID = charToInt(incomingChar);
                                next_comm_state = LENGTH;
                        }
                        else
                        {       // Message invalid
                                next_comm_state = START;
                        }
                        break;
                        case LENGTH:
                        if(incomingChar >= '4' && incomingChar <= intToChar(MAX_PACKET_SIZE))
                            // Length can be between 4 and 8
                        {
                                messageLength = charToInt(incomingChar);
                                messageDataIndex = 0;
                                next_comm_state = DATA;
                        }
                        else
                        {       // Message invalid
                                next_comm_state = START;
                        }
                        break;
                        case DATA:
                        if(incomingChar == ETX) // todo packet to short
                        {
                                messageReceived = 1;
                                next_comm_state = START;
                        }
                        else if(messageDataIndex < (messageLength - 4))
                        {
                                messageData[messageDataIndex] = incomingChar;
                                messageDataIndex++;
                                next_comm_state = DATA;
                        }
                        else
                        {
                                // End of data, but no ETX char received, so message invalid
                                messageReceived = 0;
                                next_comm_state = START;
                        }
                        break;
                }
```

```
                receivedReadIndex++;                                    // Increase buffer index
                if (receivedReadIndex >= RECEIVE_BUFFER_SIZE) receivedReadIndex = 0;

                comm_state = next_comm_state;
                rx_flag = 0;
        }

        //DebugReceiveState();
}

// TEMP: debug receive state
void DebugReceiveState()
{
        switch (comm_state)
        {
                case START:
                PIN13_ON();
                PIN12_OFF();
                PIN11_OFF();
                break;

                case SENDERID:
                PIN13_OFF();
                PIN12_ON();
                PIN11_OFF();
                break;

                case LENGTH:
                PIN13_ON();
                PIN12_ON();
                PIN11_OFF();
                break;

                case DATA:
                PIN13_OFF();
                PIN12_OFF();
                PIN11_ON();
                break;
        }
}

// Send UART char
void USART_send(unsigned char data)
{
        while(!(UCSR0A & (1<<UDRE0)));
        UDR0 = data;
}

// Interrupt for timer overflow
ISR (TIMER1_OVF_vect)     // Timer1 ISR
{
        timer++;
        TCNT1 = regAfterOverflow;
}

// Interrupt for receiving UART
ISR(USART_RX_vect)
{
        uart0_status = UCSR0A;                                  // Read the status from the UCSRA register
        received[receivedInsertIndex] = UDR0;   // Read data from UDR data register into a cyclic buffer
        receivedInsertIndex++;                                  // Increase buffer index
        if (receivedInsertIndex >= RECEIVE_BUFFER_SIZE) receivedInsertIndex = 0;
        rx_flag = 1;
}
```

Appendix B
Monitoring software - MainForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace NetworkLEDTest
{
    public partial class MainForm : Form
    {
        private struct Configuration
        {
            private int maxNumberOfDevices;      // devices
            private int waitStateTimeOut;        // milliseconds
            private int sendStateTimeOut;        // milliseconds
            private int sendStateDuration;       // milliseconds

            private char GetConfigChar(int index)
            {
                switch (index)
                {
                    case 0:
                        return CalculateConfigCharFromValue(maxNumberOfDevices, -32, 1, 1);
                    case 1:
                        return CalculateConfigCharFromValue(waitStateTimeOut, -32, 25, 25);
                    case 2:
                        return CalculateConfigCharFromValue(sendStateTimeOut, -32, 25, 25);
                    case 3:
                        return CalculateConfigCharFromValue(sendStateDuration, -32, 25, 25);
                    default:
                        return (char)0;
                }
            }

            private char CalculateConfigCharFromValue(int value, int prescalingOffset, int scaling,
                int postscalingOffset)
            {
                return (char)(((value - postscalingOffset) / scaling) - prescalingOffset);
            }
```

```
        public void ReadValuesFromForm(MainForm form)
        {
            maxNumberOfDevices = int.Parse(form.cboMaxNumberOfDevices.SelectedItem.ToString());
            waitStateTimeOut = int.Parse(form.cboWaitStateTimeOut.SelectedItem.ToString());
            sendStateTimeOut = int.Parse(form.cboSendStateTimeOut.SelectedItem.ToString());
            sendStateDuration = int.Parse(form.cboSendStateDuration.SelectedItem.ToString());
        }

        public char[] GetConfigChars()
        {
            char[] str = new char[4];
            for (int i = 0; i < 4; i++)
            {
                str[i] = GetConfigChar(i);
            }
            return str;
        }
    }
    Configuration config = new Configuration();

    const char STX = (char)2;
    const char ETX = (char)3;

    // Colors
    Color colInactive = Color.LightGray;
    Color colWaiting = Color.RoyalBlue;
    Color colSending = Color.MediumSeaGreen;

    private List<Node> lstNodes = new List<Node>();

    private Queue<int> queNodes = new Queue<int>();

    private SerialPort objSerialPort;
    private List<Message> lstMessages = new List<Message>();
    private string strReceivedBuffer;
    private int intCurrentLocIndex = -1;
    private int maxNumberOfDevices = 4;

    public MainForm()
    {
        InitializeComponent();
        CheckForIllegalCrossThreadCalls = false;

        InitUI();

        ClearField(this, null);
        ClearMonitor(this, null);

        ListSerialPorts();
    }

    private void InitUI()
    {
        for (int i = 1; i < 224; i++) cboMaxNumberOfDevices.Items.Add(i.ToString());
        for (int i = 25; i < 2400; i += 25) cboWaitStateTimeOut.Items.Add(i.ToString());
        for (int i = 25; i < 2400; i += 25) cboSendStateTimeOut.Items.Add(i.ToString());
        for (int i = 25; i < 2400; i += 25) cboSendStateDuration.Items.Add(i.ToString());

        cboMaxNumberOfDevices.SelectedIndex = cboMaxNumberOfDevices.Items.IndexOf("4");
        cboWaitStateTimeOut.SelectedIndex = cboWaitStateTimeOut.Items.IndexOf("300");
        cboSendStateTimeOut.SelectedIndex = cboSendStateTimeOut.Items.IndexOf("400");
        cboSendStateDuration.SelectedIndex = cboSendStateDuration.Items.IndexOf("300");
    }

    private void ListSerialPorts()
    {
        // Get a list of serial port names and add them to form control
```

```
        cboPorts.Items.Clear();
        IEnumerable<string> ports = SerialPort.GetPortNames().OrderBy(q => q);
        foreach (string port in ports)
        {
            cboPorts.Items.Add(port);
        }
        if (cboPorts.Items.Count > 0) { cboPorts.SelectedIndex = 0; }
    }

    private void OpenCloseSerialPort(object sender, EventArgs e)
    {
        if (objSerialPort != null && objSerialPort.IsOpen)
        {
            objSerialPort.Close();
            WriteLog("Comport closed");
            btnConnect.Text = "Connect";
        }
        else
        {
            try
            {
                queNodes.Clear();

                if (cboPorts.SelectedItem != null)
                {
                    objSerialPort = new SerialPort(cboPorts.SelectedItem.ToString())
                    {
                        BaudRate = 57600,
                        Parity = Parity.None,
                        StopBits = StopBits.One,
                        DataBits = 8,
                        Handshake = Handshake.None,
                        RtsEnable = true
                    };
                    objSerialPort.DataReceived += new SerialDataReceivedEventHandler(DataReceived);

                    objSerialPort.Open();

                    WriteLog("Connected via port " + cboPorts.SelectedItem.ToString());

                    btnConnect.Text = "Disconnect";
                }
                else { MessageBox.Show("Select serial port first!", "No port"); }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error while opening serial port", MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
            }
        }
    }

    private void DataReceived(object sender, SerialDataReceivedEventArgs e)
    {
        string indata = "";
        List<int> array = new List<int>();
        while(objSerialPort.BytesToRead > 0)
        {
            int val = objSerialPort.ReadByte();
            array.Add(val);
            indata += (char)val;
        }

        strReceivedBuffer += indata;

        indata = indata.Replace(((char)03).ToString(), Environment.NewLine);
        txtReceived.AppendText(indata);
```

```
while (strReceivedBuffer.Contains(STX) && strReceivedBuffer.Contains(ETX))
{
    int stxIndex = strReceivedBuffer.IndexOf(STX);
    if (stxIndex > 0)
    {
        strReceivedBuffer = strReceivedBuffer.Substring(stxIndex);
        stxIndex = strReceivedBuffer.IndexOf(STX);
    }

    if (stxIndex == 0)
    {
        int etxIndex = strReceivedBuffer.IndexOf(ETX);

        if (etxIndex > 0)
        {
            Message msg = new Message(strReceivedBuffer.Substring(0, etxIndex + 1));
            lstMessages.Add(msg);

            // Does this node exist?
            if (lstNodes.Any(p => p.id == msg.SenderID()))
            {
                if (msg.data == "B") // begin
                {
                    lstNodes.FirstOrDefault(p => p.id ==
                        msg.SenderID()).SwitchNodeState(NodeStates.TRANSMITTING);

                    // Check if nodes have disappeared
                    if (queNodes.Contains(msg.SenderID()))
                    {
                        // Make nodes idle until next
                        while (queNodes.Peek() != msg.SenderID())
                        {
                            int _id = queNodes.Dequeue();
                            lstNodes.FirstOrDefault(p => p.id == _id)
                                .SwitchNodeState(NodeStates.IDLE);
                            WriteLog("Node lost: " + _id);
                        }
                        queNodes.Dequeue();
                        queNodes.Enqueue(msg.SenderID());
                    }
                    else
                    {
                        WriteLog("New node: " + msg.SenderID());
                        queNodes.Enqueue(msg.SenderID());
                    }

                    if (intCurrentLocIndex == -1) intCurrentLocIndex = msg.SenderID();
                    if (intCurrentLocIndex != msg.SenderID() &&
                        lstNodes.Any(p => p.id == intCurrentLocIndex))
                    {
                        intCurrentLocIndex = msg.SenderID();
                    }
                }

                if (msg.data[0] == "C") // config
                {
                    if (chkAutoConfig.Checked)
                    {
                        config.ReadValuesFromForm(this);
                        if (new string(config.GetConfigChars()) != msg.data.Substring(1))
                        {
                            lstNodes.FirstOrDefault(p => p.id == msg.SenderID())
                                .SwitchNodeState(NodeStates.MAINTENANCE);
                            SendConfig(msg.SenderID().ToString());
                        }
                    }
```

```
                        }


                        if (msg.data == "F") // finish
                        {
                            lstNodes.FirstOrDefault(p => p.id == msg.SenderID())
                                .SwitchNodeState(NodeStates.WAITING);

                            if (intCurrentLocIndex == -1) intCurrentLocIndex = msg.SenderID() + 1;
                            else intCurrentLocIndex++;
                            if (intCurrentLocIndex >= maxNumberOfDevices - 1) intCurrentLocIndex = 0;
                        }
                        if (msg.data[0] == 'T')
                        {
                            lstNodes.FirstOrDefault(p => p.id == msg.SenderID())
                                .UpdateTemperature(int.Parse(msg.data.Substring(1)));
                        }

                        strReceivedBuffer = strReceivedBuffer.Substring(etxIndex);

                    }
                    else
                    {
                        // No node, should not happen, so pause program
                        Debugger.Break();
                    }
                }
            }
        }

    }

    private void RefreshSerialPorts(object sender, EventArgs e)
    {
        ListSerialPorts();
    }

    private void ClearMonitor(object sender, EventArgs e)
    {
        txtReceived.Text = "";
        txtLog.Text = "";
        lstMessages.Clear();
    }

    private void ClearField(object sender, EventArgs e)
    {
        flpNodes.Controls.Clear();
        lstNodes.Clear();

        for (int i = 0; i < maxNumberOfDevices; i++)
        {
            // Add node
            Node nod = new Node(i);
            lstNodes.Add(nod);

            if (this.InvokeRequired)
            {
                this.BeginInvoke((MethodInvoker)delegate ()
                {
                    flpNodes.Controls.Add(nod.objGUIObject);

                });
            }
            else
            {
                flpNodes.Controls.Add(nod.objGUIObject);
            }
```

```
        }
    }

    private void SendConfig(string targetID)
    {
        string sendBuffer = "";

        sendBuffer += (char)2;
        sendBuffer += '/';      // <- ID of the maintainer
        // Length
        sendBuffer += 'C';      // Data
        sendBuffer += new string(config.GetConfigChars());
        sendBuffer += (char)3;

        sendBuffer = sendBuffer.Insert(2, (sendBuffer.Length + 1).ToString());

        objSerialPort.Write(sendBuffer);
        WriteLog("Send config to " + targetID);
    }

    private void SwitchToMaintenanceMode(object sender, EventArgs e)
    {
        lstNodes.ForEach(p => p.SwitchNodeState(NodeStates.MAINTENANCE));

        string sendBuffer = "";

        sendBuffer += (char)2;
        sendBuffer += '/';
        sendBuffer += '5';
        sendBuffer += 'X';
        sendBuffer += (char)3;

        objSerialPort.Write(sendBuffer);
        WriteLog("Switched to maintenance mode!");
    }

    private void FinishMaintenanceMode(object sender, EventArgs e)
    {
        ClearField(this, null);
        queNodes.Clear();

        string sendBuffer = "";

        sendBuffer += (char)2;
        sendBuffer += '/';
        sendBuffer += '5';
        sendBuffer += 'R';
        sendBuffer += (char)3;

        objSerialPort.Write(sendBuffer);
        WriteLog("Exit maintenance mode...");
    }

    private void WriteLog(string message)
    {
        txtLog.AppendText(message + Environment.NewLine);
    }

    private void chkAutoConfig_CheckedChanged(object sender, EventArgs e)
    {
        cboMaxNumberOfDevices.Enabled = !chkAutoConfig.Checked;
        cboWaitStateTimeOut.Enabled = !chkAutoConfig.Checked;
        cboSendStateTimeOut.Enabled = !chkAutoConfig.Checked;
        cboSendStateDuration.Enabled = !chkAutoConfig.Checked;
    }

    private void cboMaxNumberOfDevices_SelectedIndexChanged(object sender, EventArgs e)
```

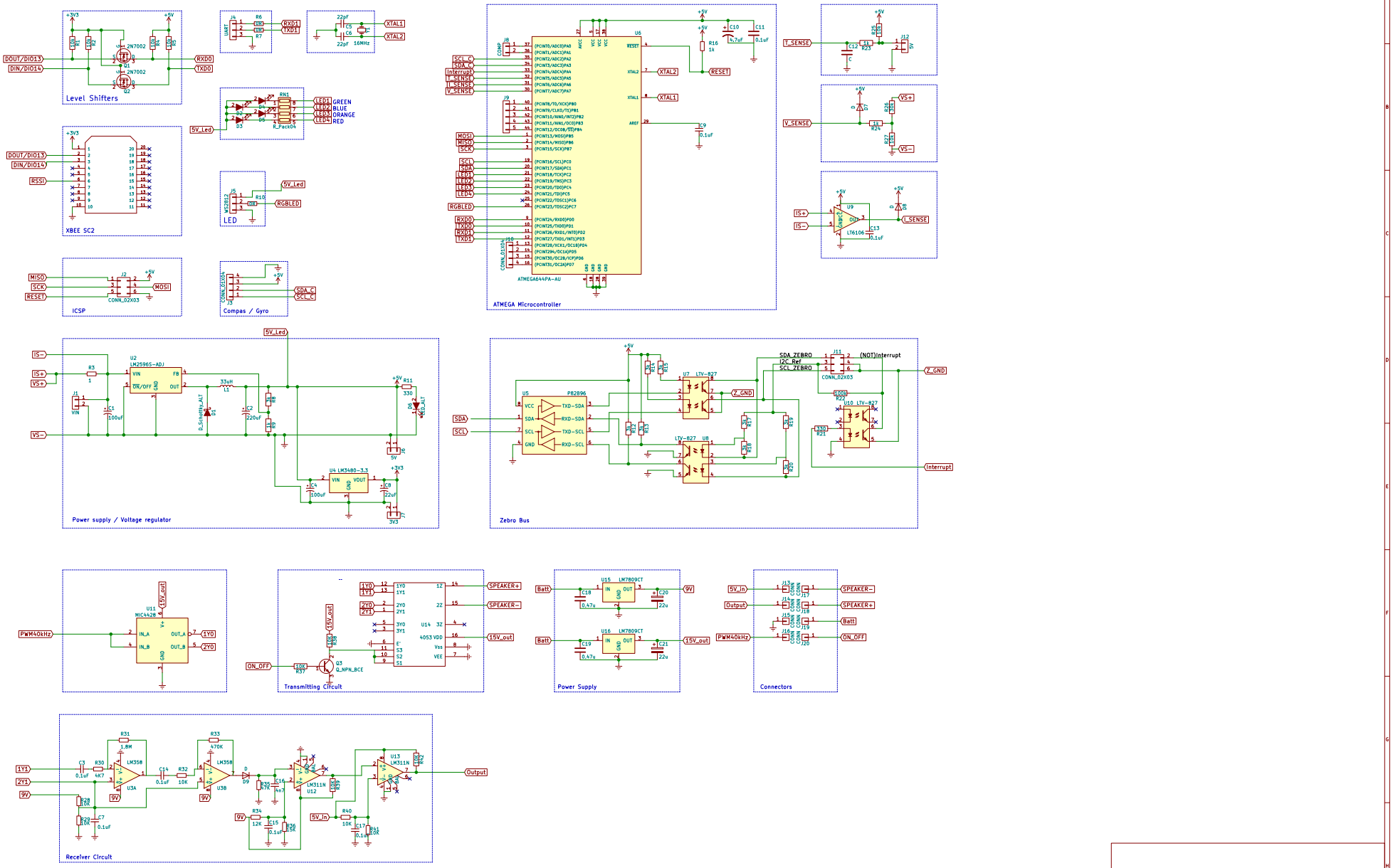
```
        {
            maxNumberOfDevices = int.Parse(cboMaxNumberOfDevices.SelectedItem.ToString());
            ClearField(this, null);
        }
    }

    public class Message
    {
        public Message(string input)
        {
            try
            {
                senderID = input[1];
                data = input.Substring(3, input.Length - 4);
            }
            catch { }
        }

        public string source;

        public char senderID;
        public string data;

        public int SenderID() { return int.Parse(senderID.ToString()); }
    }

    public enum NodeStates { IDLE, TRANSMITTING, WAITING, MAINTENANCE }

    public class Node
    {
        public Node(int _id)
        {
            id = _id;

            // Create panel
            objGUIObject = new Panel();
            objGUIObject.Size = new Size(75, 125);

            objImage = new PictureBox();
            objImage.Size = new Size(75, 75);
            objImage.Location = new Point(0, 0);
            objImage.SizeMode = PictureBoxSizeMode.StretchImage;

            objIDLabel = new Label();
            objIDLabel.Size = new Size(75, 25);
            objIDLabel.Location = new Point(0, 75);
            objIDLabel.AutoSize = false;
            objIDLabel.Text = id.ToString();
            objIDLabel.TextAlign = ContentAlignment.MiddleCenter;
            objIDLabel.Font = new Font(objIDLabel.Font.FontFamily, 12);

            objTempLabel = new Label();
            objTempLabel.Size = new Size(75, 25);
            objTempLabel.Location = new Point(0, 100);
            objTempLabel.AutoSize = false;
            objTempLabel.Text = "-";
            objTempLabel.TextAlign = ContentAlignment.MiddleCenter;
            objTempLabel.Font = new Font(objIDLabel.Font.FontFamily, 8);

            // Context Menu
            lblInfo = new ToolStripMenuItem();
            lblInfo.Text = "ID = " + _id;
            lblInfo.Enabled = false;

            btnMaintenance = new ToolStripMenuItem();
            btnMaintenance.Text = "Maintenance...";
```

```
            cmsContextMenu = new ContextMenuStrip();
            cmsContextMenu.Items.Add(lblInfo);
            cmsContextMenu.Items.Add(new ToolStripSeparator());
            cmsContextMenu.Items.Add(btnMaintenance);

            objGUIObject.Controls.Add(objImage);
            objGUIObject.Controls.Add(objIDLabel);
            objGUIObject.Controls.Add(objTempLabel);
            objGUIObject.ContextMenuStrip = cmsContextMenu;

            SwitchNodeState(NodeStates.IDLE);
        }

        public int id;
        public Panel objGUIObject;
        private PictureBox objImage;
        private Label objIDLabel;
        private Label objTempLabel;

        private ContextMenuStrip cmsContextMenu;
        private ToolStripMenuItem lblInfo;
        private ToolStripMenuItem btnMaintenance;

        public void UpdateTemperature(int temp)
        {
            objTempLabel.Text = temp + "␣ C";
        }

        public void SwitchNodeState(NodeStates newState)
        {
            switch (newState)
            {
                case NodeStates.IDLE:
                    objImage.Image = NetworkLEDTest.Properties.Resources.zebro_idle;
                    break;
                case NodeStates.TRANSMITTING:
                    objImage.Image = NetworkLEDTest.Properties.Resources.zebro_transmitting;
                    break;
                case NodeStates.WAITING:
                    objImage.Image = NetworkLEDTest.Properties.Resources.zebro_waiting;
                    break;
                case NodeStates.MAINTENANCE:
                    objImage.Image = NetworkLEDTest.Properties.Resources.zebro_maintenance;
                    break;
            }
        }
    }
}
```

Appendix C
Bill of Materials

Category	Quantity	Product description	Footprint	Value	Manufacturer	Designators	Farnell code	Price	Total
MCU	1	ATMEGA644PA-AU microcontroller	TQFP-44		Microchip	U2	1972106	€ 6.43	€ 6.43
IC	1	P82B96TD I2C bus interface	SOIC-8		NXP	U10	8906068	€ 2.33	€ 2.33
	1	LM2596 Switching Buck Regulator	TO-263-5		ON semiconductor	U9	2534165	€ 2.52	€ 2.52
	1	LM3480IM3-3.3 voltage regulator	SOT-23-3		Texas Instruments	U4	1469102	€ 0.78	€ 0.78
	1	LM3480IM3-5.0 voltage regulator	SOT-23-3		Texas Instruments	U3	3160634	€ 0.82	€ 0.82
	1	XB24CDMPIT-001	DIP-20-X		Digi International	U1	-	€ 15.91	€ 15.91
	3	ILD213T	SOIC-8		Vishay	U5	1045451	€ 0.71	€ 2.13
	1	ZXCT1086E5TA Current Sense Amplifier	SOT-23-5		DIODES	U8	1904029	€ 0.80	€ 0.80
Resistor	1	EXBV8V102JV 50V 63mW	R_array	1k	Panasonic	RN1	2060162	€ 0.07	€ 0.07
	14	MC0063W0603510K 50V 63mW	R0603	10k	Multicomp	R1,R2,R4,R5,R25 ,R27,R12,R13,R1	9331700	€ 0.01	€ 0.11
	3	MC0603SAF0000T5E 50V 100mW	R0603	0R	Multicomp	R6,R7,R10	2309111	€ 0.01	€ 0.03
	5	MC0063W060311K 50V 63mW	R0603	1k	Multicomp	R9,R11,R16,R23, R24	9330380	€ 0.01	€ 0.04
	1	MC0063W0603130K 50V 63mW	R0603	30k	Multicomp	R26	9330984	€ 0.01	€ 0.01
	1	MC0063W060311R0 50V 63mW	R0603	1R	Multicomp	R3	2141505	€ 0.01	€ 0.01
	1	MC0063W060353K3 50V 63mW 5%	R0603	3k	Multicomp	R8	9332022	€ 0.01	€ 0.01
	1	MCWR06X3300FTL 50V 100mW 1%	R0603	330R	Multicomp	R21	2447339	€ 0.00	€ 0.00
	1	MC0063W06035100K 50V 63mW 5%	R0603	100k	Multicomp	R22	9331719	€ 0.01	€ 0.01
Kerco	2	MC0603N220J500CT 50V 5%	C0603	22p	Multicomp	C5,C6	1759057	€ 0.01	€ 0.02
	4	MC0603B104K500CT 50V 10% X7R	C0603	100n	Multicomp	C9,C11,C12,C13	1759122	€ 0.01	€ 0.04
Elco	3	EEEFK1V101XP 35V	c_elec_6.3x7.7	100u	Panasonic	C1,C3,C4	1850114	€ 0.24	€ 0.73
	2	EEEFK1V220R 35V	c_elec_5x5.3	22u	Panasonic	C7,C8	9695834	€ 0.36	€ 0.71
	1	EEEFK1V221P 35V	c_elec_8x10.5	220u	Panasonic	C2	9695877	€ 0.71	€ 0.71
	1	EEEFK1V4R7R 35V	c_elec_4x5.3	4.7u	Panasonic	C10	9695818	€ 0.17	€ 0.17
Inductor	1	MSS1048-333MLC 2.8A	L1812	33u	Coilcraft	L1	2288270	€ 0.94	€ 0.94
Crystal	1	QC5CB16.0000F18B23M 330ppm 18pF	SMD_5032	16 MHz		Y1	2508603	€ 0.54	€ 0.54
LED	1	VLMB1300-GS08 20mA	Vishay	Blue		D3	2251459	€ 0.25	€ 0.25
	2	VLMTG1300-GS08 20mA	Vishay	Green		D2,D6	2251490	€ 0.27	€ 0.53
	1	VLMO1300-GS08 20mA	Vishay	Orange		D4	2251473	€ 0.29	€ 0.29
	1	VLMS1300-GS08 20mA	Vishay	Red		D5	2251484	€ 0.25	€ 0.25
Diode	1	SS54 Schottky Rectifier 40V 5A	DO-214AB		Multicomp	D1	1861427	€ 0.24	€ 0.24
	2	1N4148	SOD-123F		Multicomp	D7, D8	1621821	€ 0.07	€ 0.15
Transistor	2	2N7002-7-F	SOT-23		DIODES	Q1,Q2	1713823	€ 0.18	€ 0.36
									€ 37.91

Appendix D
Schematic


Level Shifters
XBEE SC2
ICSP
LED
Compas / Gyro
ATMEGA Microcontroller
Power supply / Voltage regulator
Zebro Bus
Transmitting Circuit
Power Supply
Connectors
Receiver Circuit
ATMEGA644PA-AU
LM2596S-ADJ
LM3480-3.3
MIC4428
LM7809CT
LM358
LM311N
LTV-827
P82B96
4053
Q_NPN_BCE
2N7002
RN1
R_Pack04
CONN_02X03
CONN_01X04
WS2812
Sheet: /
File: PCB_Frees.sch
Title:
Size: A2
Date:
Rev:
KiCad E.D.A. kicad 4.0.6
Id: 1/1