University of Southern Denmark

MSc in Engineering - Electronics
2. Semester Project

Monitoring System for the SDU Go-Kart

Authors:
230390 Martin Brøchner Andersen
151088 Catalin Ionut Ntemkas
030192 Mikkel Skaarup Jaedicke
100589 Thomas S. Christensen

Supervisor:
Leon Bonde Larsen

Date: 19-12-2016

Preface

Acknowledgment

This project would not have been possible without the invaluable assistance and patience of our supervisor. A Tremendous thank you to Leon Bonde Larsen. The process was simplified by Assistant Professor Kjeld Jensen from which the sensory equipment used in the report was supplied.

Catalin Ionut Ntemkas

Martin Brøchner Andersen

Mikkel Skaarup Jaedicke

Thomas Søndergaard Christensen

The report, source code, data, plotting script and simulations can be found at:

`github.com/mikkeljae/SEM1PRO_ELECTRONICS`

Abstract

A go-kart has been supplied as a development platform for student projects at University of Southern Denmark. In the interest of enabling more complex projects, a unified data collection system is necessary. This is developed throughout this report. An analysis is done to determine the requirements of such a system. It was found that a two-part network is suitable for this application. The two parts are a CAN network and an ad-hoc WiFi network. A custom protocol for use on CAN, GoCAN, is developed. GoCAN supports up to 16 sensors from which data can be monitored on a remote monitoring station using the WiFi connection. The system was only partially implemented since a connection between the CAN network and Linux was not achieved.

Contents

List of Figures

List of Tables

Part I
Preface

1 Introduction

As a part of the Master's in Engineering in Electronics at University of Southern Denmark (SDU), an electric go-kart is made available to the students. The go-kart serves as a development platform around which many of the projects throughout the programme are to be centered. The theme of these projects spans several areas of science and engineering. These could include power electronics components, computer vision, path finding or, as in the case of this report, networking on embedded systems. While the types of projects differ wildly, mostly, they will have one thing in common. A need for reliable collection of data.

Many tasks are simply too time consuming or difficult for a single project which means that it may be necessary to have different development teams working on closely related projects. Having a unified strategy for data collection on the development platform will greatly ease collaboration between projects. This project aims to create a backbone for data collection for the go-kart that will fulfil this need. Modularity is a key aspect in making a system that can be useful in as many situations as possible.

The report was created using an analysis driven approach. As such an emphasis was set on determining the requirements of the system. Having a firm understanding of the requirements allowed the authors to perform informed decisions on the implementation of the system. In order to better focus the work, daily meatings were held, discussing the progress and problems of yesterday as well as the focus of the coming day for each individual. It was found that following this method lowered the amount of time a problem existed without the awareness of the remaining members of the group.

2 System Description

This project is aimed at developing a backbone for data collection on the SDU go-kart. As mentioned, the go-kart is intended as a development platform to be used by engineering students at SDU. Some projects may implement new sensory equipment while others may work on improving the inverter. Common for all of them is that they require access to the data that their system is producing. Providing a unified structure for gathering data from the go-kart will greatly ease the development on the platform, especially across different projects with different developers. A system will be developed which provides a simple method for accessing data on the go-kart while driving. Live access of parameters while driving, realistically, requires some form of wireless communication between a monitoring system and the go-kart, see figure 2.1.

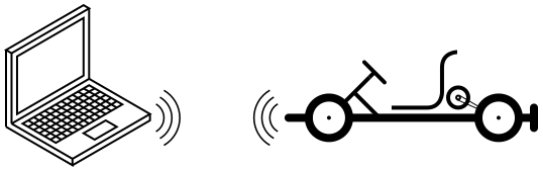


Figure 2.1: SDU go-kart transferring data to a stationary computer wirelessly.

While the system should be able to provide a live feed of the data being collected on the system, it should also log all data during a drive, with the possibility of transferring it later. This will enable the user to do more advanced data analysis on the dataset than what can be achieved by monitoring the data. In some cases, certain parameters may be irrelevant to the test being performed. Transmitting these parameters should be avoidable by providing the possibility to start and stop data collection from specific data producers.

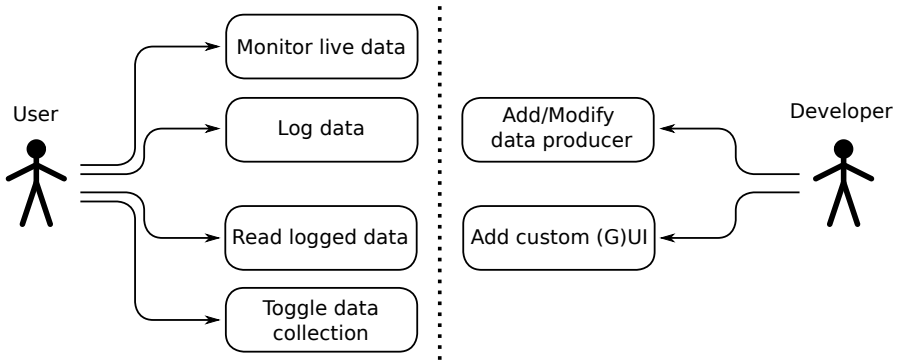


Figure 2.2: Use case diagram for the system.

As the goal of this system is to ease development on the go-kart, it should not only ease data collection, but also allow for simple addition of new sensors or other equipment. Different projects may also have different requirements in terms of the presentation of data. For this reason it should be possible to add a custom (G)UI of that projects design. These requirements introduce a distinction between the actors expected to work with the system, the user

and the developer. The user will use the system to extract data from the system while the developer will work on integrating new data producers into the system. A use case narrative for each of the use cases is shown in appendix A.

Part II
Analysis

3 Introduction

A thorough system analysis is needed in order to design a system that meets the needs expressed by the use cases. This section will present the analysis of the complete system, touching on the following topics:

- **Parameters of interest:** In order to create a well-suited system it is necessary to gain knowledge of the type of parameters that may be of interest to monitor.
- **Hardware for monitoring parameters:** Having a list of parameters, what hardware is necessary to collect and monitor these parameters.
- **Data rate:** An investigation of the amount of data that can be expected from data producers connected to the system is required.
- **Data collection:** A number of sensors and other data producers is expected to exist on the go-kart. It is necessary to analyse how data from these is collected most effectively.
- **Embedded platform:** An investigation of what requirements the platform running the go-kart part of the system should satisfy.
- **Wireless network:** Data is required to be transmitted wirelessly from the go-kart to the monitoring station. An investigation of the appropriate technology will be conducted.
- **Local storage:** In order to log data, some form of local storage is required. This investigation will examine what options exist on the available platform to choose the appropriate solution.
- **Monitoring station interface:** An analysis of the appropriate interface on the stationary part of the system will be conducted.

4 Parameters of Interest

In developing new hardware for the go-kart or evaluating current hardware, it is necessary to be able to monitor associated parameters. This section will investigate what parameters need to be logged in order to provide a useful and complete logging of the behavior of the go-kart. The parameters in question fall into three categories; physical parameters, electrical parameters and mechanical parameters. Clearly, developers may, in addition to the aforementioned categories, also have programmatic information, such as debug information from a motor controller. This section will deal only with parameters that describe the state of the go-kart. The three categories are dealt with in turn in the remainder of this section.

4.1 Physical Parameters

This category comprises all information about the motion of the go-kart.

- **Position, absolute:** Providing a means to record the absolute position of the go-kart is a useful feature in certain fields. Especially any form of localization and path finding will be able to put this information to use. The absolute position of the go-kart can be recorded using a GPS module and possibly by using a known starting coordinate and information about the relative movement of the go-kart.

- **Position, relative:** The relative position of the kart can be used to infer the absolute position of the go-kart. Additionally it can provide a means to analyze a drivers performance on track the or detect drift while cornering. The relative position includes both translational, as well as rotational information. This information can be gathered using an inertial measurement unit (IMU). An IMU is a compound device, comprised of an accelerometer, a gyroscope and, in some cases, a magnetometer.

- **Velocity:** The velocity of the go-kart is key in optimizing lap-times; clearly, it is desirable to monitor this parameter. It can be extracted by reading the motor encoders. However, as the driving wheels are prone to slippage when cornering, this would give an inaccurate reading of the actual velocity of the go-kart. Instead, a simple encoder can be mounted on either one, or both of the front wheels as these are free-running and independent. Once the rotational speed of the axle is known, the velocity of the go-kart can be inferred using the tire diameter.

- **Acceleration and forces:** It may be of interest to monitor the forces exerted on the go-kart, or its acceleration, as it drives on the track. This information is already provided by the accelerometer in the IMU mentioned above and as such provides no additional complication.

Three sensors are mentioned in this section. A GPS, an IMU and an encoder. In order to limit the scope of the project only the GPS and the IMU will be implemented.

4.2 Electrical Parameters

This category comprises all information about the electrical aspects of the go-kart.

- **Motor currents:** Providing a means of monitoring the currents flowing through the motor allows the user to calculate the torque exerted by the motor as well as the power draw of the motor. Measuring the currents could also prove to be an invaluable debugging tool when developing a new inverter for the go-kart.

- **Throttle position:** The throttle on the go-kart is connected to a potentiometer. Measuring the voltage output of this potentiometer provides a simple way of monitoring the position of the throttle.

- **Desired currents:** Based on the current throttle position a set of desired currents are calculated. Monitoring these allows spotting any discrepancies between the desired and the actual currents. This information can be used in debugging a current controller.
- **Voltage from inverter:** Voltage can be used to calculate the power going into the motor.
- **Battery voltage:** As the go-kart is electrical, naturally, it has a battery. Monitoring the current battery status could give the user an indication of how much driving time is left.
- **Motor angle:** Knowing the angle of the motor at all times gives a means of calculating more accurately the currents at specific times.

These parameters are all available from the Sevcon Gen4 motor controller mounted on the go-kart. This controller has a CANopen interface from which this data can be extracted.

4.3 Mechanical Information

This category comprises all information about the mechanical aspects of the go-kart.

- **Steering wheel angle:** Monitoring the angle of the steering wheel allows analysing the performance of the driver. In addition, with an actuator, it opens up for the possibility of remote control of the go-kart. Similarly to monitoring the velocity, the steering wheel angle can be monitored by adding an encoder to the steering column.
- **Braking pedal position:** The braking system on the go-kart is similar to that of an ordinary car. The braking disc is mounted on the driving axle and the braking calipers are connected to the brake pedal by a series hydraulic lines. Monitoring its actuation allows analysing the performance of the driver and as mentioned above, may potentially allow for remote control of the go-kart.
- **Temperature:** Certain parts of the go-kart directly impact driving performance with changing temperature. Additionally, overheating the motor, the batteries or tyres may cause expensive or dangerous failures.

As both monitoring of the steering wheel angle and the braking pedal position would require mechanical changes to the go-kart, they are beyond the scope of this project and as such will not be implemented. The Sevcon gen4 provides the temperature of the motor and its internal motor control circuitry. All other temperatures would require installation of thermometers on the vehicle.

4.4 Conclusion

As mentioned in the beginning of the section, the parameters discussed throughout this section are limited to those that explain the physical aspects of the go-kart. A future project may include the introduction of a data producer

which does not directly gather data from a sensor, but generates a filtered output of other sensory equipment. It is not possible to account for every parameter that a future developer may wish to add. In order to provide a proof of concept for the system it was decided to implement the control for three data producers; the Sevcon Gen4 motor controller, an IMU and a GPS.

5 Hardware for Monitoring Parameters

In section 4 an overview of the different parameters that may be of interest for logging is given. It was concluded that three components would suffice as a proof of concept; the Sevcon Gen4 motor controller, an IMU and a GPS. This section will explore in more detail what requirements and communication schemes exist for each of the components.

5.1 Sevcon Gen4 Motor Controller

The Sevcon Gen4 motor driver is a general purpose AC motor driver. This means, it can be used for both asynchronous and synchronous motors of a wide range of sizes. The controller can be set up for a particular motor and peripherals, and run without interfacing to another computer. However, it is also possible to read data from it while running, and in some cases set values. It is therefore possible to use this to access the electrical parameters.

The Sevcon communicates through CANopen, which is a GoCAN protocol running on top of CAN. This allows reading from and writing to a vast array of registers of varying length, which is described in further detail in section 13.3

5.2 Inertial Measurement Unit (IMU)

IMU's, generally, exist in two versions. A 6D and a 9D version (Dimensions). Both include an accelerometer and a gyroscope, each adding three dimensions. In addition to these the 9D IMU includes a magnetometer, enabling measurement of absolute direction, as opposed to the relative measurement of direction given by the gyroscope. The requirement in terms of each of these parts is given as:

- **Accelerometer [m/s^2]:** As the name implies, the accelerometer measures accelerations. That is, when the component changes velocity the acceleration exerted on the IMU is measured. Professional drivers using professional grade go-karts driving upwards of 250 $\frac{km}{h}$ can reach up to 2-3 g's of acceleration. The go-kart available in this project has a maximum speed of 50 $\frac{km}{h}$. Clearly, the forces exerted on this platform will be lower, however, a minimum requirement of $\pm$ 3g will be set for the accelerometer in the IMU. This will ensure that bumps in the road does not saturate the accelerometer.
- **Gyroscope [$\frac{^\circ}{s}$]:** A gyroscope in this case, consists of three discs spinning at high velocity. When the IMU turns in around any of its axis, the change will be detected on one of the gyroscopes. The gyroscope

part of the IMU will mostly be useful for detecting change in inclination, turning the go-kart, or spinning out. In the two former cases, the rate of change is fairly low, about 90 $\frac{^\circ}{s}$ for a sharp turn. But in case the driver spins out, or oversteers, this number is much higher. Therefore the minimum requirement of $\pm 360 \frac{^\circ}{s}$.

- **Magnetometer [T]:** The magnetometer measures the magnetic flux density through the IMU. This is done by three hall sensors measuring the flux density along three axis. This can be used to calculate the absolute orientation of the go-kart, both which way is north and which way is up. This type of sensor is notoriously noise sensitive. The flux density of the Earth's magnetic field[1] ranges up to 65µT. By comparison, this is several orders of magnitude lower than the magnetic field generated by the motor, so it is important to place the IMU away from the motor and high power wires. The scale of the magnetometer would need to be at least ±75µT.

The accelerometer measures the first derivative of translation with respect to time. The gyroscope measures rotation. This leaves three elements, that can be calculated: first integral of translation (position), translation itself (velocity), first integral of rotation (yaw, pitch and roll) and the first derivative of rotation. The position is measured by the GPS, and the others can be calculated by data fusion of the IMU measurements.

Vectornav VN-100

The VN-100 from Vectornav was available from the university. It is a 9D IMU with built-in data processing to deduce more elements, that are not directly measurable. Its ranges are generally much larger than the requirements, but at the same time, has sufficiently fine resolution. The resolution is listed in table 1.

Sensor	Range	Bit	LSB
Accelerometer	±16g	16	< 0.5g
Gyroscope	$\pm 2000 \frac{^\circ}{s}$	18	$< 0.02 \frac{^\circ}{s}$
Magnetometer	±250µT	12	0.15µT

Table 1: List of the resolution of the internal ADCs on the Vectornav VN-100

Apart from the raw data listed above, the VN-100 can calculate other parameters, such as quaternions, yaw, pitch, roll, absolute heading and altitude. It can also make the accelerometer data zero base, meaning when the go-kart is standing still, 0g will be output from the IMU, rather than the 1g caused by the gravitational pull of earth.

The Vectornav comes in a rugged, waterproof casing with a USB cable going out. A library is available for use on Linux.

5.3 Global Positioning System (GPS)

It was possible to borrow a GPS sensor from U-blox, the NEO-6P. This has a data rate of 5 Hz, and a Circular Error Probability (CEP) of down to 2 m. This means, that any position measurement has a 50 % chance of being within a radius of 2 m from the actual position. This is very good compared to other GPS modules, but is barely precise enough to determine which side of a go-kart track, the go-kart starts in. Further precision is however a possibility using a base station.

This is a precise and fast GPS module, and will give a good platform for data fusion, should later projects want to work with this. Furthermore, a GPS module would only work outdoors, and the nearest go-kart track to SDU is indoors, in which case a local positioning system would be preferred.

The U-blox NEO-6P is mounted on a custom made PCB with a minUSB port and a connector for the antenna. Thus a platform with USB drivers can use this module

5.4 Conclusion

Three sensors have been selected for this project. The main reason for selecting these sensors is that they are readily available. The selected sensors are listed in table 2.

Table 2: Table of selected sensors.

Sensor	**Brand**	**Interface**
Motor controller	Sevcon Gen4	CAN open
IMU	Vectornav VN-100	USB
GPS	U-blox NEO-6p	USB

6 Data Rate

This section will explore the amount of data that can be expected from a data producer connected to the system. Many of these parameters of interest pose different requirements in terms of the desired sample rate as well as the size of each sample. A temperature sensor, for instance, does not require nearly the same sampling frequency as an accelerometer. Due to these differences the data rate expected from one data producer may differ wildly from another. In order to provide a safe estimate, the expected data rate is calculated using a worst case sensor. That is, the type of sensor which sets the highest requirements in terms of sampling frequency as well as data size. Again, it is not possible to foresee every application that may be developed in the future. Despite of this the IMU chosen for this project, see section 5.2, is considered as the worst-case. It is is a 9-axis sensor that provides data at a rate up to 300 Hz at 32 bit floating point precision. This would result in a data rate of:

$$300 \cdot 32 \cdot 9 = 86.4\,\mathrm{Kb/s}$$

As previously mentioned, this is the assumed worst case for the system. As such there may be only a few data producers providing data at this rate. Most other producers will provide only limited data in relation to the IMU, either due to a much lower sampling frequency or an overall smaller data size. A GPS generally provides an update only at a few Hz. An encoder, while it may have a reasonably high sampling rate, it is most likely not close to 32 bit resolution.

Assuming five sensors running at the assumed worst case rate and ten running at half that rate, the resulting data rate will be 864 kb/s.

7 Data Collection

As described in previous sections, the system will have to support the use of several data producers. In addition to this, it should also be possible to easily add additional data producers at a later time. It is possible to implement the support of many producers on a single platform, however, eventually the platform would likely run out of I/O for additional support. The single-platform approach would also likely not meet the computational requirements set by the increasing amount of equipment. This implies that a multi-platform system is the appropriate approach for this system. Each platform, or node, would then implement the functionality of one or more data producers.

The data needs to be collected from all of the nodes, then logged and transmitted wirelessly to the monitoring station. The type of network depends greatly on the number of nodes that it has to support. An ongoing project on SDU, Formula Student [2], has a similar platform for which a network has been developed. That project, which is to develop a race car, has a network of seven physical nodes, distributed around the car, which each handle data producers and consumers in that area. The physical distribution of the nodes means that the responsibility of a given group of sensors can be given to a node based on their placement on the car. The parameters found in section 4 can be measured using five different types of sensors. This project does, however, aim to give users the ability to add their own nodes. For this reason the limit of supported nodes is set to 16. It is worth noting that the structure of each node is irrelevant, so long as it adheres to the networks communication standards. This means that a node can potentially comprise several (different) sensors or perhaps a complete separate network of nodes.

7.1 Network Topology

There are various network topologies that can be used to setup the required node network for this project. These include among others the bus, ring and star. Before choosing a topology, a brief description of the purpose and functionality of the network as well as an overview of their advantages and disadvantages are needed.

The purpose of this network is to accommodate multiple nodes, such as sensors, sub-networks and in general data-producers. These nodes need to be

able to transfer their data and receive commands, as specified in the use cases. The communication between the various nodes does not require a central hub. Furthermore, in the case that one node fails, the network as a whole should still be operative. Since it is a multi-node network and it may require more nodes in the future, scalability is also required.

Network Topologies

Networks can be described by different patterns, or topologies, some of which might have several technologies. Some of these topologies, that are practical to implement, are listed bellow.

- **Bus:** On this network, all nodes communicate through a common bus. This means that one node communicates to all other nodes at the same time, and that only one node can transmit at a time.
- **Ring:** All the nodes in this type of network form a ring, where each node is connected to its two neighbours. Messages travel along the ring.
- **Star:** The star topology is a centralized type of networking. All nodes are connected to a central hub, handling all the communication between them. It offers high bandwidth and is easy to implement. It can be scalable to a certain point, since the hub can be upgraded to handle more connections.

Suitable Topology

A star network requires extra hardware in the form of a hub, which has a limited number of ports. This either has to be large enough to a predefined maximum number of nodes, or it will have to be replaced to accommodate a more nodes. This takes away scalability and modularity.

Both the bus and ring topology fulfill the requirements suitably in terms of scalability and hardware requirements. Technologies of both types will be explored in order to determine which suits the project better.

7.2 Networking technology

In the previous section it was found that the bus and ring topologies in particular lend themselves well to the implementation of this system. A number of different technologies exist, that implement these topologies. This section explores a few of the ones that are considered for this project to find the one best suited.

Selecting Technologies

Different networking technologies exist in use today, such as CAN and Powerlink among others. Powerlink is the network topology used by Formula Student; it is a ring type network, using Ethernet between neighbouring nodes. Messages are transmitted as a pulse train from one node to its neighbour, who then appends its data to the end of this train. Message trains are sent out at

fixed time intervals. However, message trains need to be fully received by one node before it can be transmitted with its data appended, and because of this it can be an issue synchronizing the nodes. There can be very little overhead, and with the Ethernet cables and ports setting the limits, the bandwidth is very large – 100 MB/s on Formula Student. However, this does require all nodes to have two Ethernet ports with direct access to an FPGA, and that is rare to find on an evaluation board.

CAN is a bus topology widely used in the automotive industry with data rates up to 1Mbit/s for small network lengths. This is slightly less than the requirement calculated in section 6, however that number was based on the highest possible sample rate of the IMU, along with the inefficient data type, which is 32 bit float. Because CAN is a bus topology, it is easy to add another node to the bus, as one would just tap into the bus anywhere. It is because of this tapping into the bus, that it is not suitable for higher data rates, and if the bus breaks at any one point, communication is lost on the entire bus. Unlike Powerlink, this runs without a dedicated master. It is not nearly as fast as Powerlink, and does not have the redundancy of a ring network. Due to the use of buffers, the CAN bus is not a real time bus, but synchronization is easier.

The open source project openPowerlink offers FPGA implementation of Powerlink, and external modules can be utilized to handle the hardware part of the network. This will allow evaluation boards with less than two Ethernet ports to work in the ring network. CAN also requires one external piece of hardware, but does not necessarily require an FPGA to run. According to the data rate calculations in section 6, the CAN network has enough bandwidth to handle the requirements of this project.

In comparing CAN and Powerlink, CAN was the better fit for this project. CAN requires less hardware to implement, and the hardware it requires is cheaper than that of Powerlink. Furthermore, Powerlink is proprietary, which poses a problem in implementation, that CAN does not.

7.3 Controller Area Network (CAN)

The CAN protocol was originally developed in the 1980's by Bosch. It is a multi-master network, where each node connects to a common bus. All nodes are able to broadcast data to all other nodes. The CAN protocol includes an overhead of 47 bits per message. The data sent in a message can vary in size from 0 to 8 bytes. This is described in detail in section 13.2. The bus offers 1 Mbit/s at up 40 m of length.

Some hardware is necessary in order to properly realise a CAN network. The structure of each node can be seen in figure 7.1. The following paragraphs explain the physical parts of this structure as well as the requirements of the protocol.

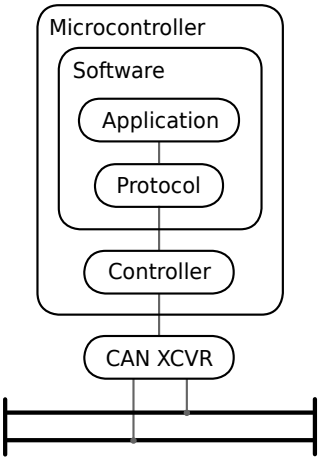


Figure 7.1: CAN node architecture

The CAN Bus

The bus is shown in figure 7.2. It is a differential voltage bus. This means that the value on the bus is determined by the voltage difference between the two wires, rather than the absolute voltage of either wire. The bus has to be made with twisted pair wires with a characteristic impedance of 120Ω and terminated at each end with 120Ω resistors. This increases the EMC of the bus, as inductive noise present on one wire is likely also present on the other, and the noise will cancel itself out.

Because it iss a differential bus, if the bus is broken at any point, no communication will be received, even if the receiving node still has a galvanic connection to the transmitting node. Alternatively, it is possible to terminate each node, but this greatly reduces transmission speed.

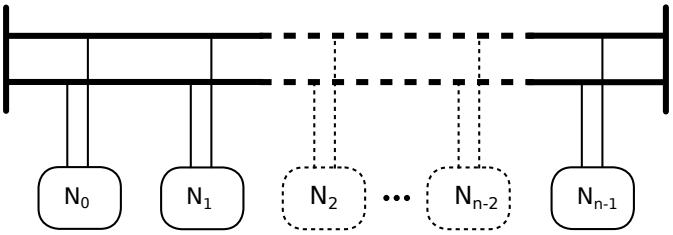


Figure 7.2: Overview of the network structure.

The CAN Transceiver (XCVR)

As CAN uses differential voltages, it is not possible to implement directly in micro controllers. Typically a micro controller has digital ports putting out either 0V, or a fixed voltage level defined as high. That is, depending on the voltage level, the signal is perceived as either high or low. As mentioned

above, the absolute voltage of either wire of the CAN bus does not matter, but difference in voltage does. As there are multiple nodes on the same bus, it is also important for one node to be able to dominate the others. Specifically for CAN, `0` is dominant, which means if one node tries to transmit `1` while another node transmits `0`, the `0` is transmitted. By checking if the bus matches what a node transmits, it can detect if there is another dominant node occupying the bus.

The transceivers translate the Tx signal from the CAN controller to the differential bus signal, and simultaneously translates the bus signal to the Rx signal going into the controller. That means, that while a node is sending, Tx and Rx are the same.

The CAN Controller

This element can be standalone hardware, but it is in many cases built into the micro controller. If the CAN controller notices that the Rx it receives is not the same as the Tx it sends out, it will know that it is being dominated. Due to the tight timing demands, this cannot be implemented in software. The major advantage of having the controller built into the micro controller is that it would otherwise require additional communication, like UART or SPI, for the communication between the microprocessor and controller. A CAN controller (both standalone and built-in) has an input and output FIFO, meaning that the CAN bus communication can operate asynchronously. This does, however, mean that it can not be used as a real time network. Asynchronous operation is necessary as there is only one bus and it is possible that nodes will attempt to transmit messages simultaneously.

Protocol

The CAN standard, apart from a variety of hardware requirements, also defines a protocol that nodes on a CAN bus should adhere to. Part of this protocol is a well defined message frame. A detailed description of this frame can be seen in section 13.2.

It should be possible to toggle the data collection of individual nodes. This means that it is necessary to have a protocol that allows addressing each node on the bus individually. The message frame of the CAN protocol does include a message ID, however this ID holds no information about the sender of the message. In CAN a table of all message ID's and their meaning is used to decode messages. Achieving the desired functionality will require an adaptation of the protocol.

One such adaption already exists in the form of CANopen. CANopen is a widely used standard that builds upon CAN. [6] shows that CAN accounts for the physical and data-link layers in the OSI model while CANopen accounts for the network to application layer. It is already used in the communication with the Sevcon Gen4 and as such, using this protocol would mean that the Sevcon could be connected directly on the bus. CANopen is quite extensive, implementing many features that are not necessarily beneficial in

this application. The many features also adds to the complexity of learning how to use the system. It also adds a significant amount of overhead for each message, taking up bandwidth which could have otherwise been used for data.

Another option is to create a custom protocol, using CAN as the physical and data-link layers. Designing a custom protocol means that the overhead can be reduced to the bare minimum. This would allow for a smaller, simpler protocol. As described in section 7, the protocol should support up to 16 nodes and allow for addition of new nodes. Since it is required to toggle the data production from a node, it should also be possible to address every node individually.
Due to the added complexity and overhead of CANopen, it is decided to develop a custom protocol. This is explored in depth in section 13.3. In that section, it is also being described why CANopen is a not an ideal fit for this project.

8 Embedded Platform

As described in section 7, a CAN network will be created to support the nodes on the go-kart. Each node is some type of embedded platform, possibly connected to a sensor or other data producing unit. From the perspective of the network the only requirement for the embedded platform is that it has a CAN controller. In principle the network could consist of 16 different embedded platforms.

8.1 Nodes

In section 5, it was chosen to use an IMU and a GPS, both using a USB interface. Additionally, one node has to transfer data wirelessly from the go-kart to a stationary monitoring system. In section 14, WiFi is chosen as the wireless protocol, and this will be handled on a separate node. Both USB and WiFi are implemented in Linux. Thus choosing an embedded platform supporting such an operating system will greatly simplify the code that needs to be written for all nodes. The authors attend a course on embedded software design in Linux, as such, this is a natural choice.

As a part of the aforementioned course, the authors have access to Zybos. This platform fits the requirements set for a node and being accessible, this is the EP chosen for the project.

8.2 Zynq Board (Zybo)

The Zybo has a Xilinx Zynq Z-7010 chip, which has a processing system (PS) part and a programmable logic (PL) part. The PS consists of a dual-core ARM Cortex-A9 processor and I/O peripherals, including two CAN controllers and USB. The PL consists of a Xilinx 7-series Field Programmable Gate Array (FPGA). The PL and PS are connected through the AXI bus. The Zybo itself is made as a development platform with several buttons,

switches, LEDs, connections for USB, Ethernet, HDMI and several PMOD connectors.

Linux on the Zybo

Several Linux distribution are configured to run on the Zybo. The authors have experience with the Xillinux[4] distribution, therefore this will be used. Xillinux is based upon the digilent Linux distribution which is built on the Xilinx distribution which is based upon Ubuntu 12.04. Xillinux architecture implements the Xillybus which is a DMA-based bus between the PL and PS. Xillybus is implemented as an IP core in PL and a corresponding Xillybus driver in the Xillinux kernel. In PL the IP core is interfaced through standard FIFO buffers. In PS the Xillybus is reached in userspace through `/dev/xillybus_<bus-name>`.

CAN Controller on the Zybo

The Zybo provides two different types of CAN Controllers. Xilinx provides an AXI Core implementing a CAN controller in PL and it has a built-in CAN Controller in the PS. The AXI core CAN Controller is available in the Vivado Suite, but it cannot actually be bitstreamed without obtaining the appropriate license from Xilinx. The CAN Controller in PS is readily usable and will therefore be used in this project.

8.3 Power Requirements

The Zybos need to be supplied with 5 V, which subsequently powers the sensors, the CAN bus and any other peripherals present on the go-kart. The Sevcon has a 5V output capable of supplying up to 100mA, which is insufficient. It is therefore necessary to use a DC-DC converter to supply the nodes. Some tests along with datasheet lookups are used to determine the power requirements for the system.

By measurement, it was found that one Zybo draws a maximum of 475mA while running Linux. The CAN transceivers consume up to 40 mA[3]. The GPS module consumes up to 40 mA, the IMU consumes up to 56 mA. Generally it is assumed that the sensors do not consume more than 75 mA per node, which brings the total current consumption up to 600 mA. For 16 nodes, this brings the total current up to 9.6 A.

The source for this DC-DC converter will be the batteries on the go-kart. These vary depending on charge from 57.6 down to 40 V, so the DC-DC converter would need to be operable in this entire range.

Parameter	Requirement
Input voltage	40-57.6 V
Output voltage	5 V
Output Current	9.6 A

8.4 Conclusion

A node is not hardware specific. It can be run on any type of embedded platform, so long as it is capable of communication over CAN. Throughout this project the nodes will be run on the Zybo platform. This will ease the development as Linux includes support for both USB and WiFi. An analysis of the power draw of a fully populated network revealed that a supply of upwards of 10A is required.

9 Wireless transmission

Communication between the go-kart and the monitoring station is required to be wireless. This section aims to determine what requirements exist for the wireless connection. Upon determining the requirements, a suitable technology will be found.

9.1 Range

The range of the transmission is determined by the length of the test track. Normally the SDU kart is tested on one of the parking lots at SDU. This "track" is 50m long and 20m wide. There are no appreciable obstacles for the transmission on the parking lot. This test track sets a minimum requirement that the wireless setup should be able to transmit data at 55m with no obstacles.

Actual go-kart tracks are usually larger than the parking lot in question though. The nearest go-kart track is *Odense gokart Hal*, which is thought to be an average size indoor go-kart track. The track is about 70m long and 40m in width with no obstacles other than the barriers. If the wireless transmitter and receiver are placed above the barriers then the barriers will not be an obstruction to the transmission. This track sets a minimum requirement of at least 80m transmission.

9.2 Bandwidth and compatibility

The CANbus has a bandwidth of up to 1Mbit/s. The wireless technology therefore has to at least match this figure reliably.
It should be possible to change the monitoring station, therefore the chosen wireless transmission technology should be compatible with standard Linux computers. The technology should be compatible with both standard Linux computers and the zybo board. Both have USB ports and Ethernet ports as standard, therefore the chosen hardware should have one of those.

9.3 Technologies

Bluetooth fits the compatibility requirements. Bluetooth 5.0 has a maximum speed of 50Mbit/s, which is sufficient. The range of typical class 2 Bluetooth device is, however, only 10m [14]. This range is definitely not enough for this application.

WiFi is also compatible with standard computers running Linux and typical WiFi units have speeds that are a lot higher than required. WiFi can be operating in the 2.4GHz band and in the 5GHz band. 2.4GHz units have the highest range, and offer higher compatibility. The 802.11n protocol generally has the best range compared to the other 802.11 protocols [15]. Depending on the antenna, WiFi has sufficient range and bandwidth. As the connection should be able to function in areas which do not necessarily provide internet access, it is required that the hardware chosen should support the creation of an ad-hoc network.

The authors have access to two different types of hardware which fit these requirements:

- **PMOD WiFi:** Digilent Inc. has a WiFi module designed specifically for use with the PMOD interface on the Zybo. Communication with this module is done in the PL of the Zynq chip, using the SPI protocol. According to the marketing material [**?**], it can support 1-2 Mbit/s at 400m, which is more than sufficient for this system.

- **USB WiFi-Dongle:** The TP-LINK TL-WN722N is a WiFi dongle with a USB interface. One requirement states that any hardware should be compatible with Linux. The TL-WN722N uses the Atheros AR9271 chipset, which is on the list of supported devices for the ath9k_htc driver [5]. It adheres to the 802.11n protocol and is also capable of supporting an ad-hoc network.

Of these two options, the latter is chosen. Since the nodes are already running Linux, doing the VHDL implementation necessary to support the PMOD module, adds unnecessary complexity to the system. While the Xillybus does simplify this issue greatly, having the WiFi on Linux natively means that all programming can be done in C/C++. This will most likely decrease development time.

9.4 Conclusion

Table 3: Minimum requirements for wireless transmission.

80m transmission range
1 Mbit/s bandwidth
802.11n protocol
USB or Ethernet
ad-hoc network compatability

This section produced the requirements shown in table 3. It was chosen to use the TP-LINK TL-WN722N. This is a WiFi dongle which uses a USB interface and can support ad-hoc networks. It adheres to the 802.11n protocol and has sufficient bandwidth.

10 Local Storage

In addition to transmitting data to a stationary computer, the data should be stored on the go-kart. It should be possible to transmit this data through WiFi upon request. This will ensure, that no data is lost if the go-kart gets out of range of the WiFi. The storage will be done on the WiFi node as transmitting a potentially large log over the CAN bus could potentially be inefficient.

Each Zybo has an SD card which holds the operating system. These particular SD cards have more than 3 GB of un-partitioned space which can be used for log storage. Linux allows for an easy way to handle files, which would ease the task of logging data at fixed intervals, or to read the log back. At this point, it is not clear how much data will be on the bus, which means that it is not clear whether this is enough. The assumption is made that data will be stored only for one drive session. That is, upon starting a new drive, the old log is deleted. This assumption enables calculating the maximum writing speed possible while not running out of space.

From experience, it is known, that the go-kart has battery for half an hour driving sessions. From this knowledge, the data rate can be calculated as follows:

$$\frac{3\text{GB}}{30\text{min}} = \frac{2.58 \cdot 10^{10}\text{bit}}{1800\text{s}} \approx 13.7\text{Mb/s} \tag{10.1}$$

With a data rate this high, it is unlikely that the go-kart runs out of storage space before it runs out of battery.

11 Software on Stationary Computer

The stationary computer needs to receive data through WiFi and decode the received messages using the protocol that is to be developed. A user should be able to access the data through an API. This API should act as the front end of the underlying network. As a proof of concept, a simple UI should be implemented to show how data can be presented to the user. The software also needs the ability to let the user send a command to the on-vehicle network to start and stop specific nodes. As with the previous parts of the system, this program should be compatible with Linux. The described software is outlined in figure 11.1.

11.1 Protocol

As described in section 7.3, a custom protocol will be developed to complement CAN. Any data received through the WiFi will be encoded using this protocol. It is the responsibility of this software to decode the software in order to present it to the user in a readable format. In order to maintain the modularity of the system, the decoding should be done in such a way that future developers can easily add new nodes, and with them, new decoding for data types.

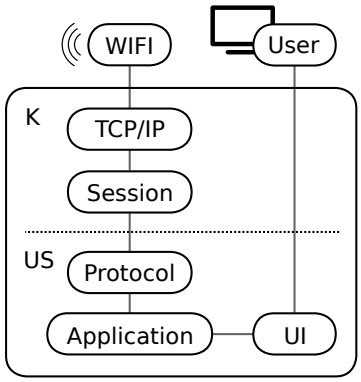


Figure 11.1: Structure of software on stationary computer.

11.2 Application

The system is intended to function as a monitoring system of go-kart data. Presentation of the data is not the focus of the project and as such only a rudimentary UI is developed to show the functionality. The software should however be designed in such a way that a developer can easily create a custom (G)UI for the system, which fits the needs of that particular project. In order to create the most seamless interface for this functionality the software should implement an API which acts as a front end for the underlying network. The API-approach has the added benefit of allowing the user to use the incoming data in any way that may suit their project.

12 Conclusion

In the analysis, a number of decisions were taken in order to give a preliminary design.

A number of parameters of interest were explored, eventually resulting in three types of sensors to be implemented, an IMU, a GPS and the motor controller. These were the VN-100 from Vectornav, the NEO-6P from U-blox and the Sevcon Gen4, all of which were available from SDU. Based on these sensors, the expected data rate was found to be 86.4 kb/s. It was decided to make a distributed network of embedded platforms. CAN bus was selected to handle communication between these.

For implementation, the embedded platform has been selected to be the Zybo from Diligent, running Xillinux. The wireless data connection between the go-kart and the stationary computer will be handled using WiFi. The SD card with Linux mounted in each Zybo has sufficient free storage for logging data at a high rate. A Linux based computer will be used to receive data wirelessly from the go-kart, and present it to a user.

These decisions resulted in a list of requirements, as seen in section 12.1. A full overview of the system is displayed in table 12.1.

12.1 System Requirements

Table 4: System requirements.

Functional	
1	Read data from sensors/data producers
2	Timestamps all data
3	Transmit data wirelessly between Zybo and monitoring station
4	Log data to SD card
5	Transfer logs to monitoring station
6	Start/stop transmission from nodes
7	Present data to user
8	Provide an API for data access
9	Must continue to function on node failure
Operational	
10	A developer can add new nodes
11	A developer can modify existing nodes
12	A user can start/stop data transmission from nodes
13	A user can request a data log
14	A developer can add API for specific sensor
15	A developer can add custum (G)UI
Quality of Service	
16	Can support 16 nodes
17	Has wireless range of > 80 m
18	CAN network has 1 Mb/s data bandwidth
19	Timestamps with a precision of 1 ms
Design	
20	Must allow for integration of new nodes
21	Software must be modular to allow for simple integration of nodes
Interfaces	
22	USB
23	5 V DC
24	WiFi
25	CAN
26	CANopen
27	SD card

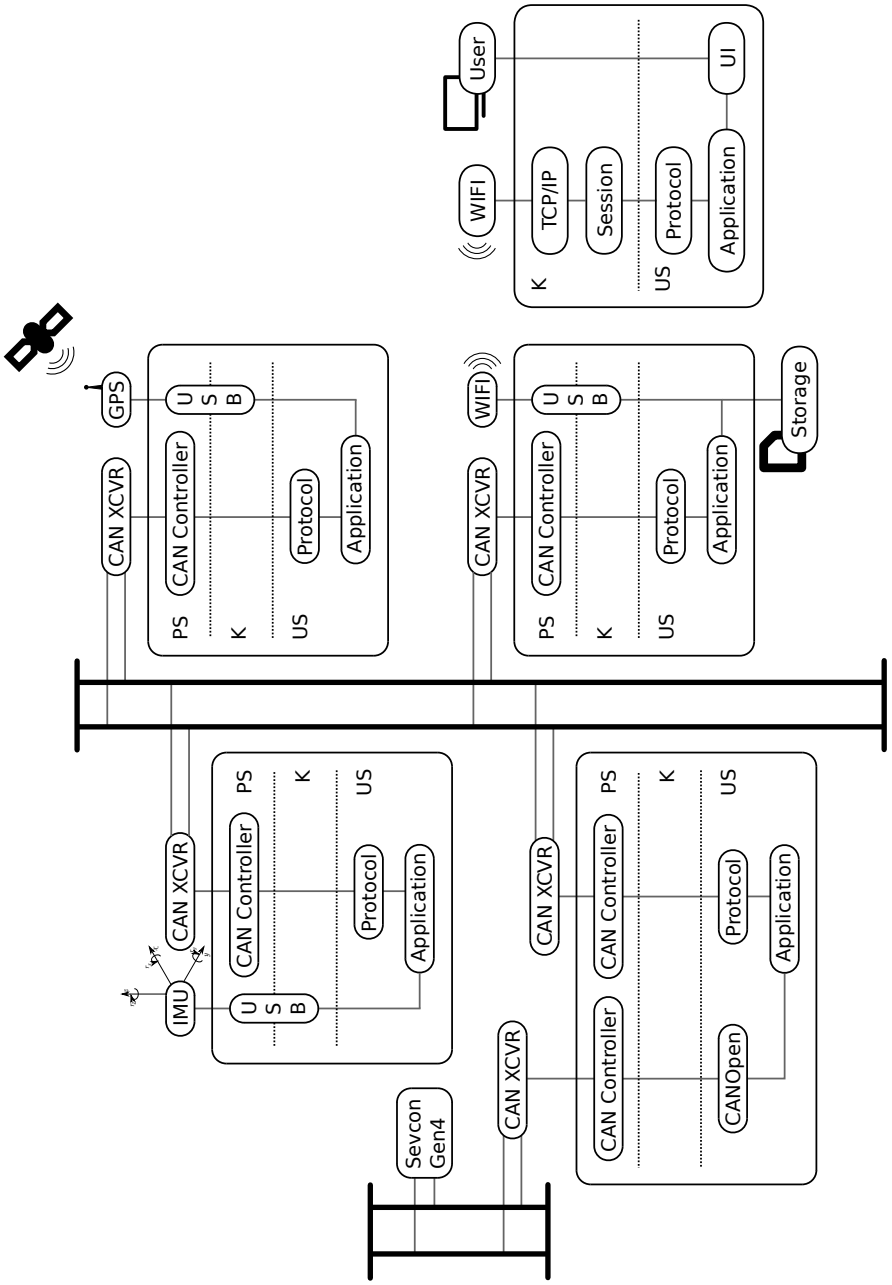


Figure 12.1: An overview of the entire system to be implemented. The four nodes are depicted with their PS level, Kernel and User Space. Two CAN busses are displayed by bold, parallel lines. The stationary computer is depicted with a WiFi connection to the WiFi node on the right side. The Sevcon Gen4 is depicted on the left.

Part III
Implementation

The analysis part lead to a table of requirements and a design of the desired system. This part will describe the implementation of the various subparts necessary in realising the system. All of the required technologies and hardware were decided upon in the analysis. Only the technologies and hardware mentioned in section 12 will be considered in this part.

13 On-Vehicle Network

The CAN protocol is described in this section, starting with the implementation of the physical layer. After that, the CAN frame is described, and finally how this is used to create a protocol for this project.

13.1 Physical Layer

The physical layer has three main parts: The CAN controller, the CAN transceiver and the bus itself.

The Zybo supports CAN, and has a CAN controller in its PS. The CAN transceiver is connected to the controller by Rx and Tx voltage signals, and connects to the bus through two differential ports. The transceiver must support the standard ISO11898-2, as this is what the Sevcon Gen4 implements. The device SN65HVD232 from Texas Instruments support this standard, and is supplied with 3.3 V, so it powered by the Zybo while still communicating with the Sevcon even though its CAN bus uses 5V. According to TI [3], this family of 3.3V transceivers are compatible and interoperable with other 5V transceivers, so long as they support the same standard.

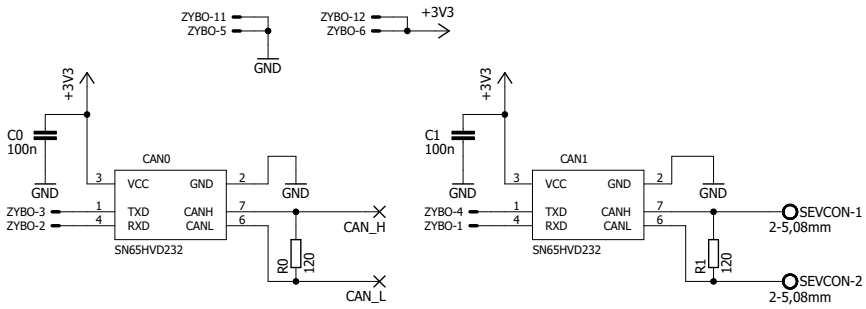


Figure 13.1: Schematic of the two CAN transceivers. One board is build for two transceivers, each with pads for termination.

PCBs for the CAN transceivers are reasonably simple in design and while they can be procured, it was decided to create a custom transceiver. This way it is possible to create a more mechanically solid stack. The transceivers have been mounted on boards, that plug directly into the Zybo's 12-pin PMOD

connector, which will be stacked on top of each other, see figure 13.2.

The board has been designed to use the MIO ports, available in the PMOD JF. This is necessary to utilize the built-in CAN controllers. Although the schematic contains two transceivers and two termination resistors, most of these devices are not mounted. Four boards will be made, all containing CAN0, C0 and the Zybo connector. One board will also include the resistor R0 - this is the bottom-most board. Another board will include R0, R1, CAN1, C1 and the SEVCON connector. Ths is placed on top, to allow the SEVCON connector's screws to be accessible.

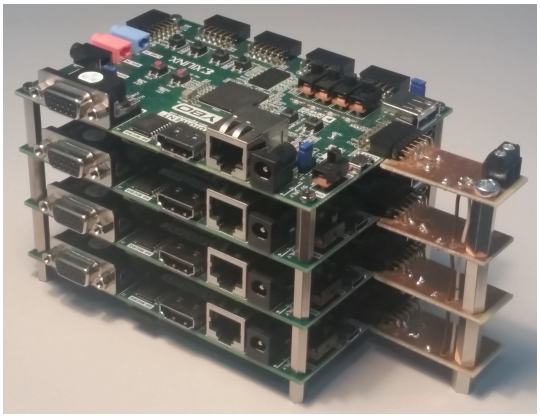


Figure 13.2: The CAN stack fully populated.

The wires of the CAN bus can be seen on figure 13.2 as a white and brown wire going through the copper PCBs. Because the bus is so short, it is not practical to make twisted pair. However, the short length of the bus allows this without issue. The wires are also not designed for a specific characteristic impedance.

13.2 CAN Message Frame

Messages sent over CAN is put into a frame. One frame contains all the parts in figure 13.3. A `0` input to the transceiver will result in a voltage difference on the CAN bus, and a `1` input will result in no difference. A `0` will always be dominant. This is used to ensure that two nodes do not transmit at the same time. If a node tries to write a `1`, but simultaneously reads back a `0`, it detects that the bus is occupied by a node with higher priority. The CAN message consists of 47 bits of overhead and 0-8 bytes of data. The frame of the message is described here.

- SOF: **S**tart **O**f **F**rame: One dominant bit to start a frame.
- Message ID: 11 bit message identifier provides information about what the message contains. Subscribers will recognize a message by this ID.
- RTR: **R**emote **T**ransmission **R**equest: set to `1` if a transmission is requested from another node. Because it is a publisher-subscriber network, this will always be `0`.

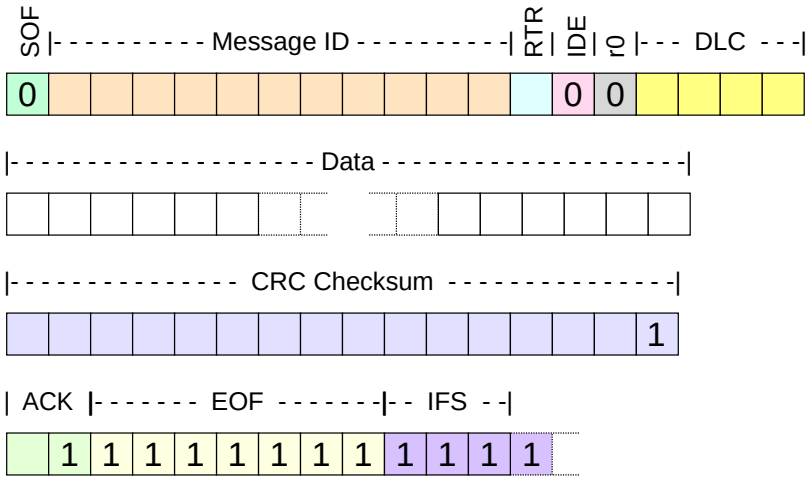


Figure 13.3: Bitwise representation of a CAN frame.

- IDE: **ID E**xtended: Set to `1` if extended ID is in use. This adds 18 more bits of message identifiers immediately after this bit. At the moment, this is not needed, and should always be 0.
- r0: Bit reserved for future use, always set to zero.
- DLC: **D**ata **L**ength **C**ode: how many bytes, of data, the frame contains. Ranging from 0-8.
- Data: Raw data.
- CRC Checksum: **C**yclic **R**edundancy **C**heck: 15 bit checksum based on what was written in the Data portion. The last bit is a delimiter, and is always `1`.
- ACK: **ACK**nowledge: The transmitting node will set the first bit of ACK to be recessive (`1`). Each receiving node will calculate CRC based on the data, and compare it to the CRC checksum. If no error is detected, it will set the bit to dominant (`0`). On a multi-node network, if one node confirms the data, the transmitting node cannot know if any other node did not receive the data. However, because of the nature of a differential voltage bus it is nearly impossible that two nodes would not receive the same signal, so long as the bus is correctly impedance matched and distances do not exceed the specification. The last bit of ACK is a delimiter, and always `1`.
- EOF: **E**nd **O**f **F**rame: Pattern signifies the end of a frame, the pattern is 7 recessive bits.
- IFS: **I**nter**F**rame **S**pace: an arbitrary length of recessive bits, at least three bits long. After this point, a new frame may start by setting a zero.

Because `0` dominates `1`, it is possible to prioritize messages. Lower message IDs are given higher priority. If two nodes start writing a message at the same time, the first node to write a `1` when the other node writes `0` will realize that another node has right of way, and will stop transmitting and wait for the current frame to end. Once the this bus is available, the node will attmept to send the message again. It might then be blocked again by another node, and there is a risk, that it will never send. It is the developer's responsibility to ensure that this does not happen.

Another automatic feature of CAN is bit stuffing. If either `0` or `1` is written to the controller 5 consecutive times, a bit of the opposite polarity stuffed into the stream. This is done to ensure that receiving nodes remain synchronized with the transmitting node, as there is no clock signal. That means, that a frame can be extended by upwards of 20 % if the data requires it. This is however not done to the EOF and IFS parts. This means, that the controllers may go out of sync at the end of each frame, this is irrelevant since the frame is finished.

13.3 Go-Kart CAN protocol (GoCAN)

Due to the requirements of the on-vehicle network, it was decided to formulate a new protocol specifically for this project. While CANopen could be used, it is a vast protocol with many features that are of no use in this system. An elaboration of the choice to make a custom protocol over using CANopen is done in this section. Additionally, the design of the go-kart CAN protocol, or GoCAN for short, is also presented. The requirements of GoCAN are listed below.

- Simple and easy to learn.
- Support for 16 nodes.
- One node must be able to receive/transmit data to/from multiple inputs/outputs.
- Publisher-subscriber architecture.
- Variable data length - beyond 8 bytes per message.
- Expandable – must be able to add more nodes without affecting the existing ones.
- Commands: Start/stop broadcasting.
- Commands can be send to specific or all nodes.
- Timestamp data.

The CANopen Protocol

As mentioned before, CANopen is a high level protocol that runs on top of the CAN protocol. It is open source project, that runs on the top 5 layers of the OSI model[6]. It includes several types of protocols, most interesting are

the SDO and PDO. By splitting the 11 bit Message ID into a 4 bit function code, followed by 7 bits of node ID, meaning that a node can communicate 16 different functions to 127 different nodes.

The Service Data Object (SDO) can read from or write to any registers on any node. It can handle 127 different nodes with each 65536 different indices, each with 256 sub indices, in total, more than 2 billion addresses with each 4 bytes of data. The way CANopen supports this, is by using the first four bytes of the data portion of the frame for metadata, as described in figure 13.4. This of course means that the overhead from one message increases from 47 bits to 79 bits, or 247 % of the maximum data size.

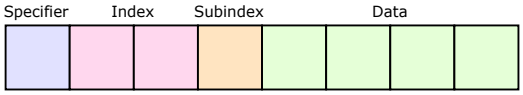


Figure 13.4: "The data section of the CAN frame is split into three parts: one byte for the specifier, three bytes for the node index and subindex, and four bytes for the actual data in the transfer."[6]

This protocol is not ideal for this network. In part because it's request based, but more because of the enormous overhead. For instance, the IMU produces 4 byte data types for each of its 9 axes. With an overhead of 79 bits, this comes to 999 bits, and at 400 Hz, that node alone will take up 40 % of the bandwidth. In addition to this, all transmits would need to be requested, so the total bandwidth comes close to 70 %. The SDO is generally used to define or read parameters for that node, that are not bound to change over time. It should not be used for broadcasting data.

The Process Data Object (PDO) is a protocol used to transmit specific indices with a higher throughput. In this protocol, certain indices are mapped to one of 8 PDOs per node. This way, a node can identify data only by the CAN message ID, without using any of the data portion for overhead. It is also possible to group multiple values together, and to transmit without request, either at fixed time or event driven. Using PDO, it is possible to group 8 of the axes together and send the 9th by itself. This brings the data from the IMU down to 523 bits to transmit its 9 axis data.

Each node on the CANopen bus has its own Object Dictionary (OD) specifying its addresses. Some of these addresses are reserved, but since the Message ID specifies which node it is, there's no risk of overlapping. If then one part is replaced by a different make or model, then it might be necessary to reconfigure other nodes, and the system loses modularity.

For all its nice features, the CANopen protocols do not fit the requirements outlined in the beginning of this section. It includes a lot of complexity, which unnecessary for this project. Instead it has been decided to develop a custom protocol that relies more on the automatic functionality of the CAN protocol.

The GoCAN Protocol

The basic CAN frame is retained for this protocol. GoCAN, mostly, is a reinterpretation of the message ID of the frame.

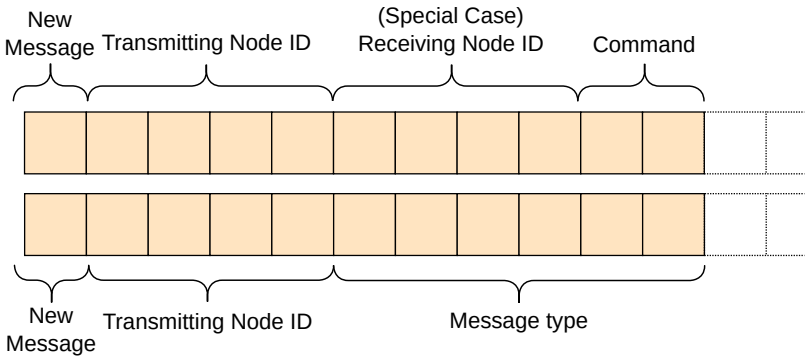


Figure 13.5: Naming convention for GoCAN. The top frame is the format when the receiver is a sensor node. This includes the node ID of the receiving node and a command. The bottom frame is the format when the receiver is the WiFi node. This has only the node ID of the sending node and the message type of the associated data.

The message ID is different depending on the recipient. A message can be addressed to either a sensor node or the WiFi node. The two cases are shown in figure 13.5. The first 5 bits are common for the two cases, bit 1 indicates whether this is the first frame of the message, bits 2-5 indicate the address of the transmitting node. If a node needs to send more than 8 bytes in one message, the new message bit can be set to `0`, thus blocking other nodes with higher priority until the entire message has been transmitted. If the message is addressed to a sensor node, bits 6 through 9 indicates the node ID of the sensor node while bits 10 and 11 hold the command. Command types are "Start broadcasting", "Stop broadcasting" and "Synchronize". These will be explained in more detail in section 13.4. If the recipient message ID is the ID of the WiFi node, then the command applies to all nodes. If the message is addressed to the WiFi node, bits 6 through 11 hold the message type of the associated data. This will allow for up to 64 different message types for each node. It is the responsibility of the developer to utilize these correctly.

As mentioned, there are two types of nodes:

- **The WiFi node:** Acts as the link between the user on the monitoring station and the network. It transfers data to the monitoring station and distributes commands on the network.
- **Sensor nodes:** Any sensor or data producing unit will be associated with a node. A sensor node will produce data and act according to received commands.

The four node IDs currently in use:
The WiFi node is given the lowest node ID since this will result in the highest priority. This is necessary when issuing commands. The IMU produces short data packages at a high rate, and the GPS node produces data packages at a

WiFi node:	0b0001
IMU node:	0b0011
Sevcon node:	0b0111
GPS node:	0b1110

lower rate. Therefore it is of less concern if the GPS message is delayed due to obstruction from other nodes.

13.4 Functions of the GoCAN Protocol

Some additional functionality is required in GoCAN to fulfill the requirements set. This functionality is described throughout this section.

Timestamps

As mentioned in section **??** the CAN protocol is not real time, therefore all data on the bus must be timestamped before transmission. The resolution of the timestamps has been decided to be 1 ms, as this leaves sufficient accuracy for the sensors analysed in this project. This presents several challenges, in terms of what the time reference is and how to convey timestamps with adequate precision without creating too much overhead. With internet access, the Epoch timestamp is available. This is infeasible in bare-metal code. Instead a counter will be implemented on each node, incrementing an 32 bit integer every 1 ms. This gives 49.6 days before overflowing.

Because a standard 32 bit integer takes 4 bytes, it would introduce a large overhead if the timestamp is sent with every message. Instead, each timestamp will determine the time of all subsequent data messages, until a new timestamp is sent. As an example, the pseudo code below describes the transmission of all nine axis of the IMU with the same timestamp. Additionally, the IMU can transmit pressure and temperature measurements, but as these are slower signals, they would be transmitted at a lower rate.

```
1  Send Timestamp
2  Send Ax, Ay, Az
3  Send Gx, Gy, Gz
4  Send Mx, My, Mz
5  Wait
6  Send Timestamp
7  Send Ax, Ay, Az
8  Send Gx, Gy, Gz
9  Send Mx, My, Mz
10 Send Pressure, Temperature
11 Wait
```

Lines 2-4 are measurements taken at the timestamp in line 1. The WiFi node appends the corresponding timestamps to each of these nine data points with the corresponding timestamp. Lines 7-10 refer to the timestamp at line 6.

Commands

The WiFi node is capable of issuing commands on the network. As described in section 13.3, it is possible to start and stop any specific or all other nodes from the WiFi node.

Synchronization

Another specific command is the synchronization command. When the system starts all nodes will start polling their Rx FIFO for this sync command. When this command is received, the node will start the millisecond timer.

Multiframe Messages

Due to the limited data length of a CAN frame, it is likely necessary to support multiple frames per message. Generally it is better to use all 8 bytes of data in one frame, to reduce the relative size of the overhead, and in case a dataset isn't easily split into 8 byte portions, it might be easier to bundle it all together and send as multiple full frames. In the message ID of all but the first frame of a multi-frame message the `new message`, to make it highest priority. That way a message will not be interrupted by other messages from other nodes. The construction and interpretation of these multiframe messages are described in sections 15.1 and 17.2.

Datatype and Scaling

As most sensor data comes from sources of limited resolution, the optimal solution would be to send only the number of bits that are measured. In some cases however that is not possible, and a better solution is to round up or down to the nearest byte, and send the data as a fixed point data type. Intead of defining a new datatype, an integer of appropriate length will be used instead, and the object dictionary will define the scaling.

As an example, the message described in table 5 contains four data points of 2 bytes. This contains the current measurements on the three phases along with the electrical position of the rotor. An example is described in table 5.

Message ID	DLC	I_a	I_b	I_c	Ω_e
10100001001	1000	414	-1545	1131	13090
		25.9 A	-96.6 A	70.7 A	1.31 rad

Table 5: Example of message data from line 17 of table 14

According to table 14, there are two scaling factors defined, one for currents and one for angle. The bottom line of table 5 shows the resolved measurements.

13.5 Utilizing the XCanPs

A CAN controller needs to be used when connecting a Zybo to a CAN bus. XCanPs is a CAN controller built-in to the Zybo. This section will seek

to develop a PL architecture for the Zybo that allows for utilization of the XCanPs. Furthermore it will describe the implementation of software that has the following functionality.

- Receiving and sending of frames.
- Creating the message ID.
- Decoding of the message ID.
- Handling interrupts from buttons and a GPIO port.
- Controlling the LEDs.
- Accepting and ignoring messages according to a subscription list.

Some of the functionalities are not needed in the finished version of the system, but will be implemented for testing and debugging purposes.

Architecture

To use the XCanPs these need to be enabled in the Zynq PS. Interrupts were also enabled along with adding an AXI GPIO port connected to a PMOD connector to allow for externally triggered interrupts to the system. Additional AXI GPIO cores for LEDs and buttons were added for debugging and testing purposes. The architecture can be seen it figure 13.6. It is simple, but adequate to meet the purpose of basic communication between nodes.

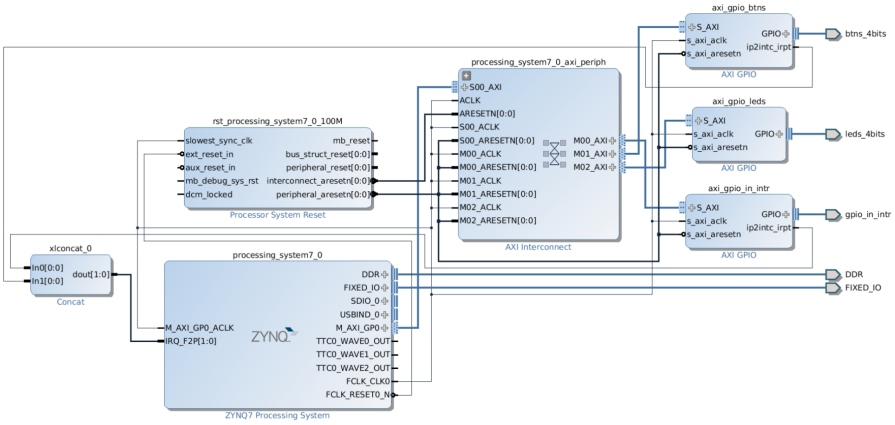


Figure 13.6: Block diagram of the architecture in Vivado.

Software

Xilinx provides example code for the XCanPs in `xcanps_polled_example.c` [7]. This is used as a basis for the developed software. The code was rewritten to give the wanted functionality.

Sending Frames

Figure 13.7 shows the procedure of sending a frame to the CAN network containing data, which makes use of the protocol function `createMsgID()`. Its purpose is to create a message ID by putting together the new message bit, the node ID and the message type. In the final implementation of the system the message ID would be provided by the software running in the userspace. Afterwards the massage ID and dummy data are put into a `TxFrame`. The `TxFrame` will be sent when the FIFO has space. The actual sending is done with a call to the XCanPs function `XCanPs_Send()`.

Receiving Frames

The procedure of receiving a frame is shown in figure 13.8. The node calls the `RecvFrame()` function and waits in a loop until it receives a frame. Then it checks the subscriptions in order to determine if the message is addressed to the node using the function `amISubscribed()`. This function makes use of an array containing the messages IDs that the node should accept when a message is being transmitted through the network.

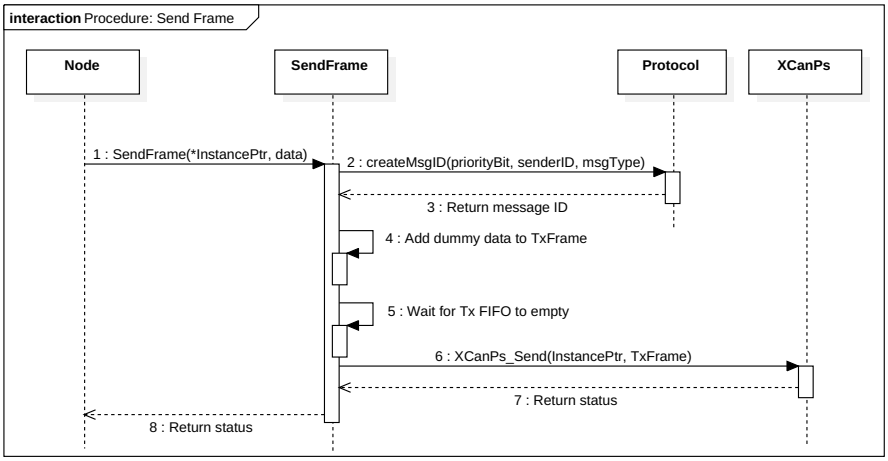


Figure 13.7: The sequence diagram of the process of sending a frame.

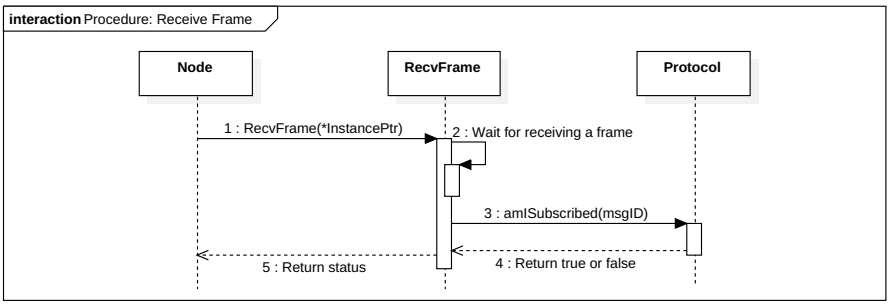


Figure 13.8: The sequence diagram of the process of receiving a frame.

13.6 Accessing CAN Controller from Linux

Utilizing the XCanPs from bare-metal code is not enough for all nodes as some needs to run Linux as described in the analysis part. This section describes the steps tried in order gain access to the CAN controller from Linux. According to Xilinx documentation, this can be done by following their Linux CAN driver guide [8].

Enabling the CAN Controller Drivers

To access the CAN controller the appropriate drivers needs to be enabled in the kernel. Specifically, the Kconfig file with the path shown in code 13.1 needed to be configured. The entry at line 128 was changed as seen in code 13.2. Originally, lines 130 and 131 were as seen in code 13.3.

Code 13.1: CAN Kconfig pathfile.

```
/usr/src/kernels/3.12.0-xillinux-1.3/drivers/net/can
```

Code 13.2: Kconfig file contents from line 128.

```
128 config CAN_XILINXCAN
129    tristate "Xilinx CAN"
130    depends on NET [=y] && CAN_DEV [=y] && CAN [=y] &&
131          (ARCH_ZYNQ || MICROBLAZE [=y])
132    default y
```

Code 13.3: Original content of lines 130 and 131.

```
130    depends on CAN && (ARCH_ZYNQ || MICROBLAZE)
131    default n
```

Unfortunately, the above steps did not enable the drivers successfully.

Changing Device Tree Settings

The next step of the process was to modify the device tree settings file, requiring an entry for the CAN PS to be inserted. The necessary file was located under the boot folder named as seen in code 13.4. The modifications can be seen in code 13.5 for CAN controllers as well as for the AXI CAN core.

Code 13.4: Device tree settings file and its path.

```
/boot/xillinux-1.3-zybo.dts
```

Code 13.5: Device tree settings changes.

```
1 zynq_can_0: can@e0008000 {
2         compatible = "xlnx,zynq-can-1.0";
3         clocks = <&clkc 19>, <&clkc 36>;
4         clock-names = "can_clk", "pclk";
5         reg = <0xe0008000 0x1000>;
6         interrupts = <0 28 4>;
7         interrupt-parent = <&intc>;
8         tx-fifo-depth = <0x40>;
9         rx-fifo-depth = <0x40>;
```

```
10      };
11  axi_can_0: axi-can@40000000 {
12          compatible = "xlnx,axi-can-1.00.a";
13          clocks = <&clkc 0>, <&clkc 1>;
14          clock-names = "can_clk", "s_axi_aclk";
15          reg = <0x40000000 0x10000>;
16          interrupt-parent = <&intc>;
17          interrupts = <0 59 1>;
18          tx-fifo-depth = <0x40>;
19          rx-fifo-depth = <0x40>;
20          };
```

It was expected that the changes took effect after rebooting the system, which was not the case.

Conclusion

After the changes, the CAN devices were still not visible in Linux and thus, accessing the CAN controllers was unsuccessful.
Research was done by the authors in why this could not be accomplished. It was found that there were some naming inconsistencies between the guide and the Device Tree Settings file present in Linux running on the Zybo. Specifically, all the devices' names in the file are prepended with ps7. A new set of changes with this naming convention was done, but with no success.

Another approach to this, was to build a newer version of the kernel containing these changes and to build an entirely new Linux system for the Zybo. This approach was considered to be infeasible as the kernel needed to be patched with Xillinux patches which are not compatible with a newer version of the kernel.

13.7 Object Dictionary

In order to ensure that all message IDs are unique, an Object Dictionary is declared for GoCAN. The name is inspired from CANopen, but unlike the CANopen Object Dictionary, this refers to the messages being transmitted on the bus, and not the parameters or variables of the node. Because of this, some variables appear in several message IDs, as it might be desirable to reconfigure some nodes to send different data, and the node should group data together to reduce the number of frames.

At this time, there are four nodes in the Object Dictionary with their node IDs: WiFi "0001", IMU "0011", Sevcon "0111" and GPS "1110". The WiFi node can issue commands to other nodes, and as such its portion of the OD is different. Common for the other three nodes is that the message type 0x01 is a four byte timestamp. Other than that, data messages start from 0x08.

Table 14 in appendix B on page 72 contains the Object Dictionary in its current form.

14 Off-Vehicle Network

The off-kart network is done using WiFi. It was chosen to use this method as all modern laptops are equipped with the capability of connecting to such a network. Hereafter the challenge lies in establishing an ad-hoc network between a Zybo and a PC. This network is created using a USB-WiFi dongle on the Zybo. The following script is run to bring down the device, change the network type to ad-hoc, bring up the device, create the ad-hoc network and, finally, assign an IP-address to the device. Some commands require super user access. For clarity, `sudo` is left out from all calls throughout this section.

```
>>service network-manager stop
>>ip link set wlan0 down
>>iw wlan0 set type ibss
>>ip link set wlan0 up
>>iw wlan0 ibss join go-kart 2412
>>ip addr add wlan0 196.178.10.10
```

It was verified using the ping command from the Zybo to the PC and vice versa.
Upon setting up the ad-hoc network, a socket connection is created between the Zybo and the PC using socat

```
zybo>>socat - tcp-listen:2048
pc>>socat - tcp:196.178.10.10
```

Messages can be passed through this connection by piping data to it.
Following is a description of the steps taken to reach the approach explained above.

14.1 Setting Up the Ad-Hoc Network

Since the Zybo is already running a Linux distribution which provides all of the necessary drivers, it was decided to use a WiFi dongle to create the ad-hoc network on the Zybo. The consequences of this choice are discussed below.

The first step was to ensure that the correct drivers are present on the system. The USB/WiFi dongle is a TP-LINK TL-WN722N. This device uses the Atheros AR9271 chipset and is on the list of supported devices for the ath9k_htc driver (available on Xillinux) on [5]. It is also capable of supporting ad-hoc networks, a crucial feature in setting up this network. Running `dmesg` prints the kernel log revealing, amongst other things, what drivers are loaded.

```
>> dmesg | grep ath
[10.329564] usb 1-1: ath9k_htc: Firmware htc_9271.fw
    requested
[10.332438] usbcore: registered new interface driver
    ath9k_htc
```

Additionally, the dongle also shows up in the list of USB devices:

```
>> lsusb | grep Ath
```

```
Bus 001 Device 002: ID 0cf3:9271 Atheros Communications,
   Inc. AR9271 802.11n
```

As per these commands the operating system (OS) has correctly detected and loaded the driver. The `iproute2` package contains the utilities used to manipulate TCP/IP connections.

```
>> ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
   state UNKNOWN mode DEFAULT group default
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   pfifo_fast state UP mode DEFAULT group default qlen 1000
   link/ether 00:0a:35:00:01:22 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq
   state DOWN mode DEFAULT group default qlen 1000
   link/ether ec:08:6b:1b:41:b3 brd ff:ff:ff:ff:ff:ff
```

The device has been given the identifier `wlan0`. It can be brought up by:

```
>> ip link set wlan0 up
>> ip link show wlan0
3: wlan0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   mq state DOWN mode DEFAULT group default qlen 1000
   link/ether ec:08:6b:1b:41:b3 brd ff:ff:ff:ff:ff:ff
```

At this point the device is up, as can be seen on the list of flags:

```
<NO-CARRIER,BROADCAST,MULTICAST,UP>
```

However, the `NO-CARRIER` flag is also set. This flag indicates a fault on the physical layer, i.e. a disconnected network cable or similar. Running a scan on the device shows that it is able to correctly detect nearby networks:

```
>> iw dev wlp2s0b1 scan | grep SSID
SSID: SDU-WEBLOGIN
SSID: eduroam
SSID: SDU-GUEST
```

This was believed to be a software related issue, as it could detect networks, but not correctly connect to them. In an attempt to isolate the error the dongle was connected to a PC. The dongle worked flawlessly on the PC and connected to the internet after bringing down the standard interface. Clearly, there are differences between the two systems, one or more of which were causing one to work while the other did not. In order to eliminate these differences the versions of the involved software was brought up to date. The PC is running Arch Linux, a distribution using rolling release. This means that, generally, the software will always be at the latest stable release. The Xillinux distribution running on the Zybo is based on the Ubuntu 12.04 release, the long term support version from 2012, see [4]. As such, much of the software is outdated. The `iproute2` package as well as the `network-manager` were updated to the latest version. Other than provide some experience with building and searching for dependencies, it did little to alleviate the issues.

It was decided to continue attempting to set up the ad-hoc network on the PC, expecting that this process may provide some insight into the issues on

the Zybo, and how to fix them. Initially the drivers were verified using the same procedure as mentioned previously; for brevity, this is left out. From experience, the first step was to disable the network manager. A network manager is particularly useful when a user wants to connect to the internet or similar, basic operations. It automatically detects hardware and manages connections in the area. The automatic setup means that it will often overwrite any changes made by the user. For this reason the network manager is disabled:

```
>> systemctl connman.service stop
```

`connman` is the network manager used on the system. In order to create an ad-hoc network it is necessary to change the mode of the interface to IBSS (Independent Basic Service Set):

```
>> ip link set wlp2s0b1 down
>> iw wlp2s0b1 set type ibss
```

The interface needs an address assigned to it in order to communicate on the network.

```
>> ip addr add 192.168.10.9 dev wlp2s0b1
>> ip link set wlp2s0b1 up
```

Finally, after having brought up the device, the ad-hoc network can be created:

```
>> iw wlp2s0b1 ibss join go-kart 2412
```

This commands creates an ibss (ad-hoc) network called go-kart on frequency 2.412 GHz. Repeating these steps on a different system, assigning a different address to it, makes it possible to ping between the systems:

```
>> ping 192.168.10.11
PING 192.168.10.11 (192.168.10.11) 56(84) bytes of data.
64 bytes from 192.168.10.11: icmp_seq=1 ttl=64 time=0.592 ms
64 bytes from 192.168.10.11: icmp_seq=2 ttl=64 time=0.638 ms
```

Having this up and running, it was attempted to run the above sequence of commands on the Zybo. This allowed the Zybo to correctly create the network. Trying to ping the board on its assigned IP-adress, however, yielded a `no route to host` message. Looking into the routing tables reveals a discrepancy:

```
>> ip route show
default via 192.168.10.1 dev eth0 metric 100
169.254.0.0/16 dev eth0 scope link metric 1000
192.168.10.0/24 dev eth0 proto kernel scope link
   src 192.168.10.10
192.168.10.0/24 dev wlan0 proto kernel scope link
   src 192.168.10.10
```

As can be seen, messages on eth0 and wlan0 are both routed to the same address space, 192.168.10.0/24. Changing the address space to something different corrected the problem. In this case 196.178.10.0/24 was chosen. Finally, a short bash script is written to quickly set up the network:

```
>> service network-manager stop
>> ip link set wlan0 down
>> iw wlan0 set type ibss
>> ip link set wlan0 up
>> iw wlan0 ibss join go-kart 2412
>> ip addr add wlan0 196.178.10.10
```

Since the system will not necessarily have an interface when it is mounted on the go-kart, it will have to automatically run this script on startup. On Xillinux, this can be achived by placing the script in `/etc/init.d/<script_name>`.

14.2 Making the Connection

Having correctly set up the network, the next step is to make a connection between the PC and the Zybo. This can be done using a variety of methods. For this project, three approaches were considered:

- C.
- Boost.
- Socat.

C has already been used in other parts of the project. For this reason, this was the first option to be considered. This option has the benefit of relying only on the standard C libraries. Although [9] was used as a reference in setting up the connection, it proved difficult to create a bi-directional connection across the ad-hoc network.

Boost is a widely used C++ library. As the authors had prior knowledge with this library it became the next choice. ASIO is part of Boost and contains all of the tools necessary to set up a TCP/IP connection. Boost provides a multitude of different examples and tutorials on how to use ASIO, specifically the chat server from [10] was modified and formed the basis for the program used in setting up the connection. This approach was not without problems, however. As mentioned previously, Xillinux is based on an older version of Ubuntu, as such, the version of CMake and Boost available on the platform is insufficient. In updating the software, one significant issue emerged; the size of the library. Boost takes up 684MB of space, making it infeasible for use on the Zybo.

Socketcat is a command line utility that can be used to establish a bi-directional connection between two hosts. As opposed to the two prior solutions, socat is implemented and verified by the developers of the utility. In addition, this utility provides exactly the function needed, as opposed to boost, which is a collection of libraries spanning everything from mathematical algorithms, complex data structures, sockets and numerous other fields. Generally, socat can be called using

```
>> socat [options] <address> <address>
```

An address is specified by a keyword and can be a range of different types. Of interest in this project is `TCP` and `STDIO`. `STDIO` can also be represented simply by `'-'`. The `TCP` address requires two arguments, the ip address and the port. A simple message passing service can be set up in two terminals by first setting up a tcp listener in terminal one:

```
>> socat - tcp-listen:2048
```

This command will redirect all incoming data from stdio to port 2048 and conversely, all incoming data from port 2048 is redirected to stdio. Connecting to port 2048 on localhost can be done from terminal two:

```
>> socat - tcp:localhost:2048
```

The message written in terminal one will appear in terminal two and vice versa. Using this functionality, the WiFi node can be connected to a PC using pipes. Piping is commonly used on Linux systems to connect the I/O of programs. The Linux philosophy is that one program should do one thing and do it well. If more functionality is needed, multiple single-purpose programs are interconnected using pipes.

14.3 Conclusion

Throughout this section a connection was made between a PC and the Zybo. It was chosen to use a USB/WiFi-dongle to carry the signal on the Zybo. The TP-LINK TL-WN722N, a general consumer dongle, was chosen for two reasons: The chipset of the dongle is supported by the available drivers and it is capable of creating ad-hoc networks. Using the `iproute2` package, an ad-hoc network was set up from the the Zybo and connected to from a PC. A small script was created to automatically set up the ad-hoc network upon booting the system. Three different approaches were attempted in actually creating a connection across the WiFi network, C, Boost and finally socat. Clearly, analysis should have been done before deciding on a method prior to starting the implementation of the socket connection. C was discarded due to complexity and the time constraints of the project. While Boost ASIO was made to function on the PC, the size of the library made it infeasible for use on an embedded platform. Finally, it was decided to use the Linux command line utility socat for creating the connection between the PC and the Zybo.

15 Nodes

The on-vehicle network generally has two types of nodes, a generic sensor node and the WiFi node. This section will seek to design and implement software that adheres to the requirements listed in the table of requirements. In general all nodes in the system needs to:

- Get data from CAN network.
- Send data using a CAN controller.

As described earlier, these responsibilities are implemented in the bare-metal CAN program described in section 13.6 therefore these responsibilities will be omitted from the following sections.

15.1 Sensor Node

The requirements state that it should be simple to add new sensor nodes to the system. To realize this the node software should be designed to be modular. It should be easily identified what software and what interface a developer of a new sensor node must adhere to. An example of a sensor node architecture can be seen in figure 15.1.

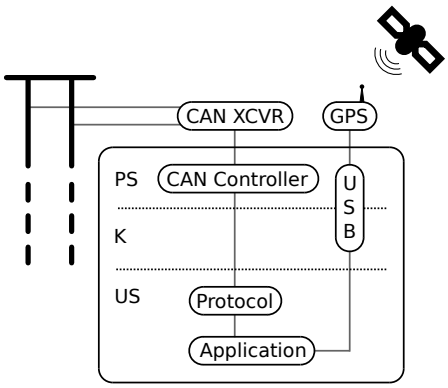


Figure 15.1: GPS node implemented on the Zybo.

This specific node has a GPS attached to it connected through a USB interface, but in general it could be any kind of data producing unit connected through any kind of interface. Due to time constraints, only the GPS sensor software was adapted to function with the designed node software.
It is found that a sensor node has the following responsibilities:

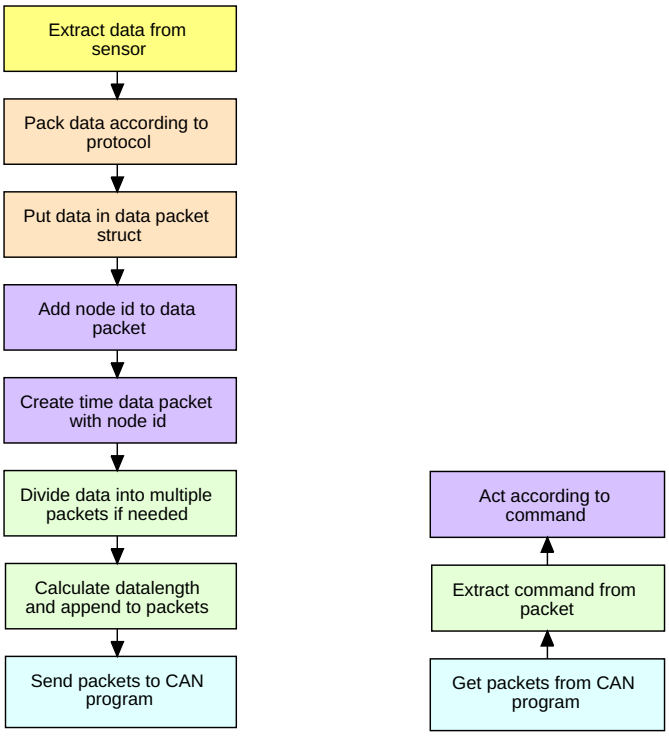
- Get data from associated sensor.
- Pack data according to GoCAN.
- Construct and send CAN packet.
- Receive and execute incoming commands.

A CAN packet is defined as the `struct` in code 15.2.

Code 15.1: Struct for data packet.

```
struct data_packet {
  std::bitset<1> nw_msg;
  std::bitset<4> node_id;
  std::bitset<4> dlc;
  std::bitset<6> messagetype;
  std::vector<bool> data;
                                  ⋮
};
```

The three first responsibilities listed describe a flow which is presented in more detail on the flow diagram on figure 15.2a.
It is required to design the software with modularity in mind. For this, it is necessary to determine which tasks are common for all nodes and which are



(a) Outgoing data. Data from a sensor, to the CAN program.

(b) Incoming data. Data from the CAN program, to the node.

Figure 15.2: Block diagram showing handling of outgoing and incoming data. The boxes are colored according to which class implements the functionality. Yellow is the `GPS` class, orange is the `GPS_Packer` class, purple is the `Node` class, green is the `Protocol` class and blue is the `CAN_link class`.

application specific. The application specific tasks, in this case, are found to be extracting data from sensor and to pack data according to GoCAN. In principle the task of packing data according to protocol is generic, but the method for packing data depends on data types and message types for the specific application. The remaining tasks are common for all sensor nodes in the system. The procedure when data is going from the CAN program to the node is shown in figure 15.2b. In this case, all tasks are found to be common for all sensor nodes.

Class diagram

Based on the previous analysis, a class diagram was developed and can be seen in figure 15.3. The main responsibilities of the classes are shown in figure 15.2. The classes `GPS` and `GPS_Packer` are application specific and should be developed for each specific application. In a generic sensor node `GPS` would be a `<sensor>` class and `GPS_Packer` would be a `<sensor>_Packer` class. The classes `CAN_Link`, `Protocol` and `Node` are agnostic with respect to the type of data they receive from the sensor and to the CAN network. They are generic classes and should be reused when developing new nodes.

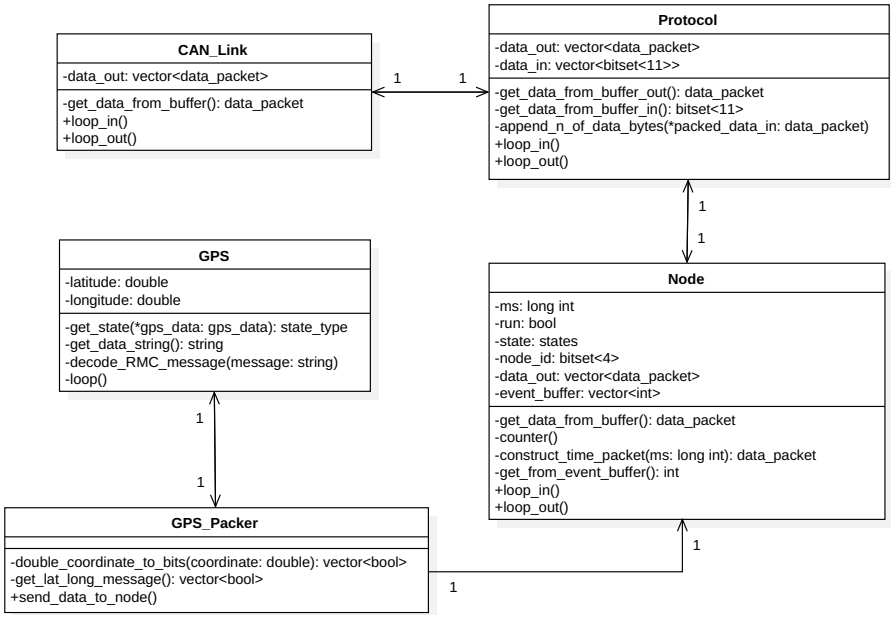


Figure 15.3: Class diagram showing sensor node software.

A struct containing all fields of a data packet in boolean data types is defined in listing 15.2.

Code 15.2: Struct for data packet.

```
struct data_packet {
  std::bitset<1> nw_msg;
  std::bitset<4> node_id;
  std::bitset<4> dlc;
  std::bitset<6> message type;
  std::vector<bool> data;
                                  ⋮
};
```

It will be passed between the classes and they will each add their own information.
The classes and their responsibilities will be explained here in more detail.

GPS class

The `GPS` class needs to extract data from a connected GPS unit and update its own variables with that data. The specific GPS unit used in this project has a USB interface and uses the NMEA protocol to format data.

GPS_Packer

The `GPS_Packer` class is also sensor specific and is the link between the sensor and the data agnostic node. It is hard-coded with the message types that the sensor is allowed to send onto the network. It has the responsibility

of packing data according to the developed protocol. The data field in the `data_packet` struct is populated with the GPS data and the message type in binary form. Once packed, the struct is passed to the `Node` class. The reason for making a separate class for the packer and not putting the functionality into the `GPS` class is that if the specification of the protocol or message types change, only this class needs to be modified. Similarly, if the GPS module is replaced, the `GPS_Packer` class does not need to be altered.

Node class

The Node class receives `data_packets` from `GPS_Packer`. Upon receiving a `data_packet`, it will set its node ID. Before sending a message, the associated time packet has to be sent. In order to create the time packet the class holds a timer which increments each millisecond. The timer is reset upon receiving a synchronization message. The class receives start, stop or synchronize events from the protocol class and reacts to those accordingly.

Protocol class

The `Protocol` class receives `data_packets` from `Node`. If it receives `data_ packets` with more than eight bytes in the data field, it will split the packet into multiple `data_packets`. The additional `data_packets` must have the same node ID and message type, but the new message, `nw_msg`, bit should be cleared on all but the first message. The class also needs to append the number of bytes in the data field, `dlc`, for each `data_packet`. On `data_packets` coming from the CAN network the last two bits of the message type is used to indicate the type of command sent to the node.

CAN_link class

The `CAN_link` class has the responsibility of transferring and receiving data to and from the CAN program. As this interface has not yet been implemented this class makes use of stdio. That is, data from the sensor will be printed to the shell and data to the node is written to the shell.

Passing data between classes

Communication between classes is realised by using the producer-consumer pattern. As the name implies one class produces data and puts this in a queue where another class consumes by taking data out of the queue. To get the producer and consumer functions to run in parallel they are run in separate threads. All queues are mutex protected to make the software thread-safe.

Node class functionality

The functionalities of `Node` on outgoing data is implemented using a state machine. It is shown in figure 15.4.
When the variable `run` is equal to `1`, the node should output sensor data. `run` is being updated by the thread that handles incoming commands. It will go to the clear state and clear all variables and wait for data. When data

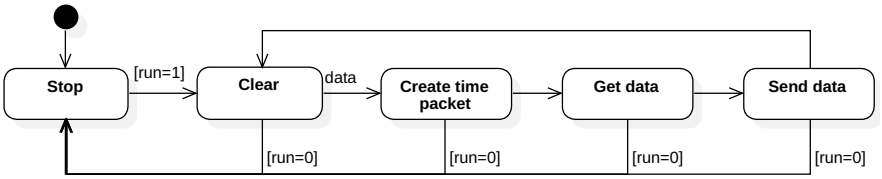


Figure 15.4: State machine implementing the functionality of `Node`.

is present it will move through the states, create the time packet, get data and send data. If at any point `run` is set to `0` the state machine will go to the stop state. In the stop state the input queue of `data_packet` will be cleared.

15.2 WiFi Node

The node with a WiFi connection to the stationary computer is a special node in the system. There should be only one and it should collect all CAN frames, log them and transfer them using WiFi. Because of time constraints the software designed in this has not been implemented in code. The WiFi node architecture can be seen in figure 15.5.

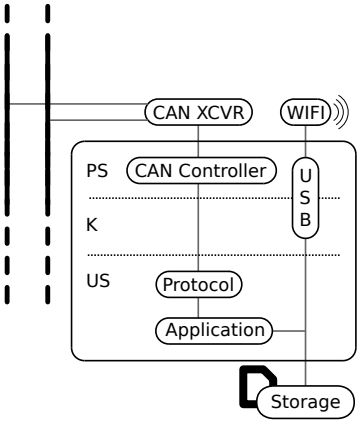


Figure 15.5: WiFi node architecture.

From the analysis it is clear that the WiFi node has the following responsibilities:

- Log all data to SD card.
- Transmit and receive data through WiFi.
- Pack data according GoCAN.
- Send log file through WiFi upon request.
- Manage Timestamps.
- Merge multi frame packets.

It is found that the responsibilities can be grouped into normal operation mode and sending log file mode. This should be implemented as a state machine as shown in figure 15.6. In normal operation mode the WiFi node should handle data coming from the CAN network and the data coming from the WiFi connection. If it receives a `send_log` command from the stationary computer it should go into the sending log mode and stay there until the log file has been sent.

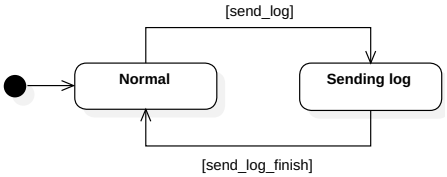
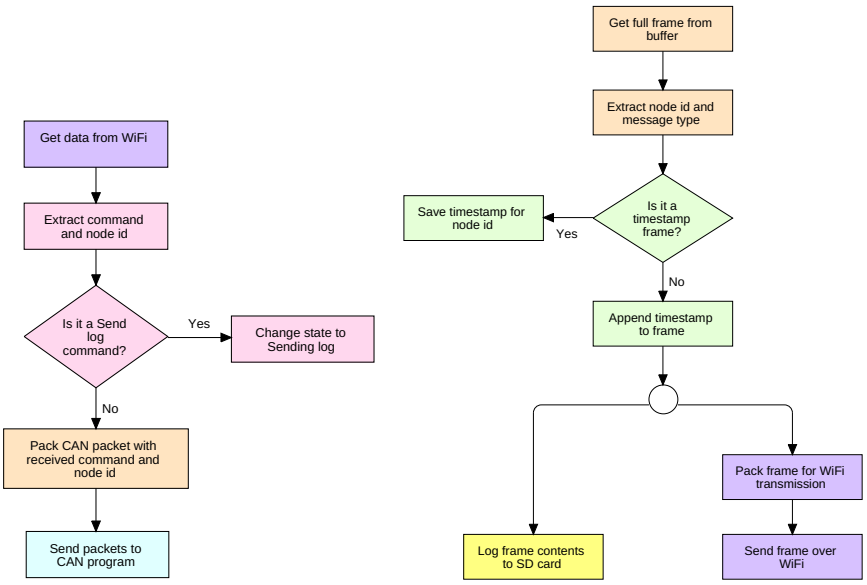


Figure 15.6: State machine in WiFi node software.

In normal operation mode it should handle all incoming data from the WiFi connection as shown in figure 15.7a. As it can be seen the WiFi node will only send frames to the CAN network when it receives commands to do so from the stationary computer.



(a) Handling of data received through WiFi.

(b) Handling of data received from CAN network.

Figure 15.7: Block diagram showing handling of outgoing and ingoing data. The boxes are colored according to which class implements the functionality. Purple is the `WiFi` class, pink is the `State_Machine` class, orange is the `Protocol` class, green is the `Timestamper` class, blue is the `CAN_link` class and yellow is the `Logger` class.

In normal operation mode the WiFi node needs to receive CAN frames from

the CAN program. Some frames contain data that is split into multiple frames. In order to log and timestamp data correctly these frames needs to be merged together. The `nw_msg` field in the messageid will be `0` if a received frame needs to be merged with the previously received frame. A state machine handling merging of frames can be seen in figure 15.8.

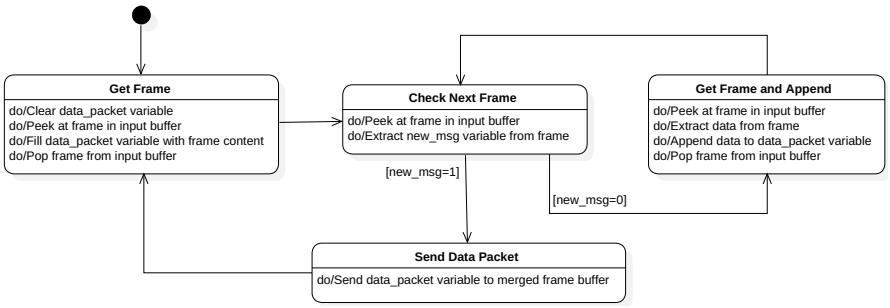


Figure 15.8: Merging multi-frame packets together.

The mentioned input buffer is the buffer where the CAN program puts its received frames. When the frames are correctly merged they will be put in the merged frame buffer. The frames in this buffer then needs to be timestamped correctly, logged and sent through WiFi. This flow is depicted in figure 15.7b.

Class diagram

Based on the previous analysis a class diagram was developed and can be seen in figure 15.9. The classes main responsibilities are shown in figure 15.7.

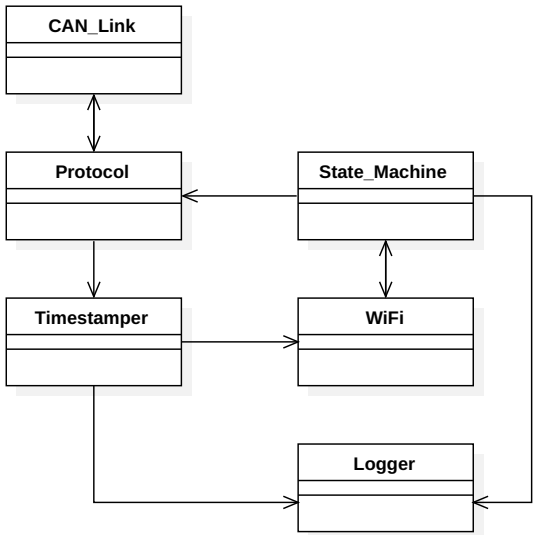


Figure 15.9: Class diagram showing the design of the WiFi node software.

WiFi class

The `WiFi` class needs to receive and transmit data through the connected WiFi dongle. It receives data from the `Timestamper` class in normal mode and data from `State_Machine` in sending log mode. The `State_Machine` also sets if it should transmit the data it gets from the `Timestamper` class.

State_Machine class

The `State_Machine` class implements the state machine of figure 15.6. When in sending log mode it needs to get a log file from the `Logger` class and give it to the WiFi class. When in normal mode it needs to extract commands and node IDs from the received WiFi data and give it to the `Protocol` class.

Protocol class

The Protocol class needs to get frames from the full frame buffer and extract node ID and message type. It also needs to pack CAN frames with received commands and node ID from the `State_Machine` class.

CAN_link class

The `CAN_link` class has the responsibility of transferring and receiving data to and from the CAN program.

Logger class

The Logger class needs to log all received data to a SD card. It also needs to read the logged file and transfer it to the `State_Machine` class, when in sending log mode.

Timestamper class

The `Timestamper` class needs to do the timestamp management described earlier. It needs to keep the latest received timestamp for each node in the network. When it receives data from a node it should append the latest received timestamp from that node.

15.3 Timestamping

The reason for timestamping at the nodes is that CAN is not a real-time network. Therefore timestamping when frames are received would be of no use, as the latency of the frames is unknown. The accuracy of a timestamp given at the node depends on the system on which it is runs. Implementing the timekeeping on bare-metal code with access to hardware timers is ideal as it allows for a very high accuracy.

In the node software designed in this section, the timekeeping is done by running the `counter` shown in code 15.3 in a separate thread. The thread will increment `ms` every 1ms. The problem with this method is that the thread is running in Linux userspace and rely on being scheduled to the processor.

As it is a userspace program it will have lower priority than OS specific processes. Even if the specific thread is given a higher priority the scheduler will still handle it in a non-deterministic way.

Code 15.3: Declaration of counter function.

```
void Node::counter(void){
   long int temp_ms;
   while(1){
      std::this_thread::sleep_for(std::chrono::milliseconds
          (1));
      temp_ms = get_ms();
      temp_ms++;
      set_ms(temp_ms);
   }
}
```

Real Time Linux

Using the `CONFIG_PREEMPT_RT` patch on a Linux kernel significantly increases the real-time performance [16]. The latency after applying the patch varies between systems, but generally with a worst case of a few hundred microseconds [17]. The patches were successfully used on a Ubuntu kernel. To be able to use the kernel on the Zybo it was needed to patch the kernel further with Digilent and Xillinux patches. As those patches do not fit with the `CONFIG_PREEMPT_RT` patch set it was deemed infeasible to continue as the authors have no prior experience in writing patches for the linux kernel.

16 Sensors

This section will describe the implementation of the sensors selected during the analysis. Of the three types of sensors, only the GPS was made to work.

16.1 Interfacing with Sevcon

The Sevcon Gen4, currently on the go-kart, is compatible with CAN. However, as it is a general purpose motor driver, it cannot be programmed to use GoCAN. For this reason, and to make the network unaffected by replacement of the motor driver, it has been decided to use a Zybo to interface with the Sevcon.

Physical Connection

Communication with the Sevcon is done through a high speed CAN bus that needs to adhere to ISO11898-2. As described in section 13.1, one transceiver board has two transceivers along with a terminal so that one Zybo can connect to the Sevcon using the second CAN controller.

Sevcon Object Dictionary

The Sevcon utilizes CAN open, which means all of its parameters are listed in an object dictionary. Because the Sevcon is a general purpose AC motor driver, its object dictionary is very large, and holds a lot of objects that are irrelevant for this particular setup, such as motor slip, and speed control parameters. The object directory is documented in a 1400+ line Excel file. Some objects of interest listed in table 6.

Parameters	Index-subindex
Motor Temperature	4600h-3
Measured Id	4600h-7
Measured Iq	4600h-8
Measured Vd	4600h-9
Measured Vq	4600h-10
Target Id	4600h-5
Target Iq	4600h-6
Encoder Read-out	4630h-9 to 12
Throttle value	2620h
Velocity	606Ch

Table 6: List of some of the parameters readable through CANopen

For the most part, these values have 16 bit resolution, which means they can be grouped together four at a time in a process data object. The fact that a value can be mapped to a PDO means, that it can be transmitted to the Zybo at fixed time intervals or whenever it is updated. The Encoder Read-out sin/cosine encoder position, so it needs to be converted to mechanical angle. This is done using equation 16.1

$$\Omega_m = \text{atan2}(\cos, \sin) \tag{16.1}$$

These adaptations need to be done to make the Sevcon node a generic motor driver node. That way it would be possible to use this system with a custom made inverter.

16.2 Interfacing with the IMU

The used IMU is a VectorNav vn-100 IMU. The physical interface to the IMU is a usb cable and when connected to Linux it shows up in `/dev/`. VectorNav provides an extensive C an C++ library for both Windows and Linux use [11]. This library was used to confirm that the module functions as expected, however, due to time constraints, it was not used further.

16.3 Interfacing with the GPS

The used GPS is a u-blox NEO-6P GPS module. The physical interface to the GPS is a usb cable and the output adheres to the NMEA standard and is made of 8 different NMEA sentences. The RMC sentence contains all essential information, that being position, velocity and time. Therefore the implemented GPS class, that has the responsibility of interfacing the

GPS, only needs to decode RMC sentences. An example of a RMC sentence is shown in code 16.1. The extracted information shows that the time is 09:11:23:00, the module is active, latitude is 55 degrees 22.03929 minutes North, longitude is 10 degrees 25.91037 minutes East, the speed is 0.348 knots, the date is 7th of November 2016 and checksum is 7C. For the sake of simplicity all coordinates are converted to degrees with decimals.

Code 16.1: RMC sentence.

```
1 $GPRMC,091123.00,A,5522.03929,N,01025.91037,E
      ,0.348,,071116,,,
2     A*7C
```

Service virtualization

The GPS only produces interesting data when it receives data from a number of satellites. This means that the GPS antenna needs to be outside, which is not very practical when developing software. Therefore the output from the GPS with the antenna outside was piped into a file. This file was then read by a program with a fixed time interval thus making a service virtualization of the GPS. This service virtualization was used when developing software for the GPS.

17 System Front End

The requirements state that it should be simple to add additional sensors to the system. Adding a sensor includes creating a way of easily accessing and showing the data on the observing system. This section will explain the design of the front end that will provide this functionality.

17.1 Data Format

Depending on where the data comes from it may be inherently different. The interpretation of data is therefore not uniform across the entire system. For this reason, it is necessary to create a system that is agnostic with respect to the type and amount of data being handled. In section 13.3 a description of the node and message identification system is given. The node ID and message type identifiers are decided by the implementer and together they provide a unique, 11 bit identifier, the message ID, for the type of data in the message. Since the message ID is capable of uniquely identifying the data, it will be used in storing the data. Additionally each message will be associated with a 4 byte timestamp. This timestamp is given as the time in milliseconds since synchronization and is associated with message type 1.

17.2 Front End Architecture

An overview of the functionality can be seen in figure 17.1. As can be seen, this architecture provides the link between the receiving program (socat) and a potential GUI. Upon receiving a message from the go-kart, socat will pipe the raw message to the interpreter. The interpreter will proceed to extract

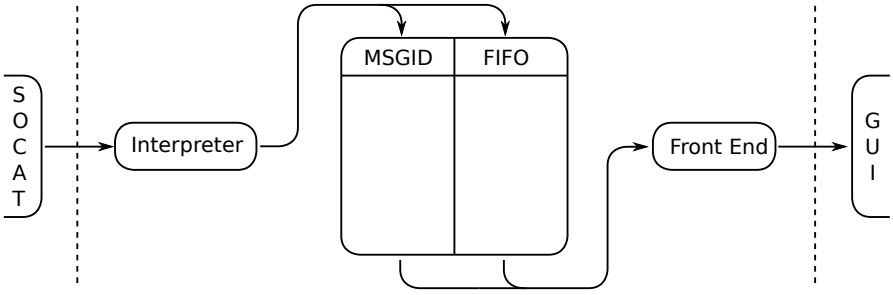


Figure 17.1: Overview of the front end functionality. Messages are received over WiFi from the go-kart. An interpreter reads the message to determine the message ID (MSGID). A fifo of data and timestamps is associated with each message ID. All information is made available to a GUI through the front end.

the message ID, the timestamp and the data. These are then put into a fifo buffer which will contain the latest 1024 data points for a given message ID. One of the requirements of this system is that it should be easily expandable. A consequence of this is that adding a new node on the system should require as few changes to the code as possible. Making this front end agnostic with respect to the type of data means that no changes are required when adding different nodes with different data types. However, full agnosticity throughout the entire front end would mean that the GUI programmer needs to know how to handle the binary data. Instead, it was decided to let the front end consist of a number of functions, written specifically for each node. The GPS node, for instance, may have a `get_latitude()` function. In this manner, an API can be created which provides the GUI with access to the parameters that needs to be shown. The complete code can be found on the associated github repository in `code/frontend/`. As a note, throughout this section it is assumed that the messages are of string type. Clearly, this is hugely inefficient as these are string representations of binary numbers. In testing the software this approach simplified the process greatly and was due to time pressure. It is unlikely that this choice will pose any issues in terms of performance since this code will be running on a PC, a platform that is several orders of magnitude faster than the Zybo.
From figure 17.1, a class diagram has been created. This can be seen in figure 17.2. The functionality is implemented in four classes, each of which will be explained in more detail below.

`receiver` class

This class is run in a separate thread. It continuously listens to std::io, waiting for input from the go-kart. Upon receiving a message, it is put into a message buffer maintained by interpreter.

`interpreter` class

As mentioned, this class holds a message buffer, it is shown in code 17.1. The buffer can be accessed from the receiver thread indirectly by a call to `put()` and the interpreter thread continously checks if data is present in the buffer.

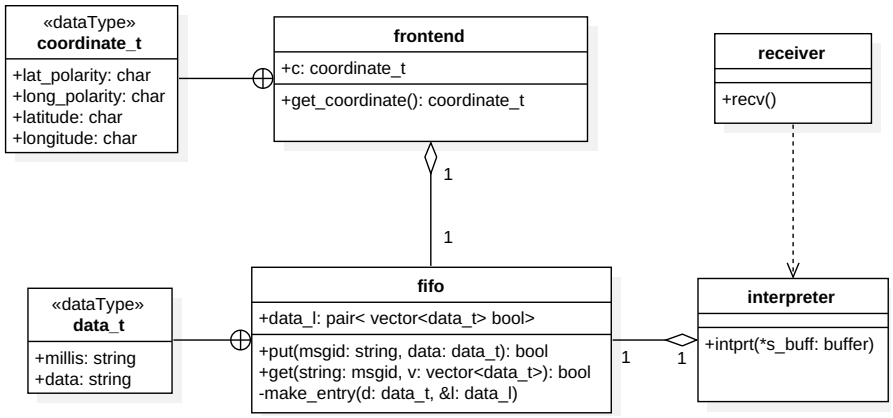


Figure 17.2: Class diagram of the front end.

Since these threads are run simultaneously it is necessary to use a mutex to avoid race conditions. Whenever a message is put in the buffer, it is split into its three components, a 10 bit message id, a 4 byte timestamp and a string of data of arbitrary length. In figure 17.3 is a depiction of the message type.

Figure 17.3: Depiction of the message sent over WiFi. The message ID is the same 11 bit identifier used in the CAN network. The timestamp is a 4 byte value containing the number of milliseconds since synchronization. The data field can be of arbitrary length.

Once split, the information is stored using from the `fifo::put()`.

Code 17.1: Buffer for holding incoming messages.

```
struct buffer
{
  std::vector<std::string> strings;
  std::mutex mutex;
};
```

`fifo` class

This class implements a hashmap and functions to insert and extract data from the map. Each message ID has a flag associated with it to signal whenever new data is available. The hashmap uses the message ID as the key and the data is `std::pair<std::vector<data_t>, bool>`. That is, each message ID is associated with a list of `data_t` and a `bool` used as a flag. As with the buffer in the interpreter class, the fifo will be accessed from two threads running simultaneously and as such, mutexing is necessary. Mutexing is done on the `put()` and `get()` functions directly. The `put()` function is shown in code 17.2. It takes a message ID and a data frame as arguments. The `data_t` type is a struct containing a timestamp and a string of data. Initially it will attempt to lock the mutex, if it fails, the function

returns false and the caller will have to try again until successful. When the mutex is locked, it searches through the hashmap for the message id. If the search fails, this is the first message from that node, with that message type and a new entry is made in the hashmap. If the search succeeds the new data point is pushed onto the end of the data list. If there are 1024 points in the list already, the first element is deleted. Before returning true to indicate that the data has been put into the fifo, the flag is set high and the mutex is unlocked.

Code 17.2: Function used to insert new data into the hashmap.

```
bool fifo::put(std::string msgid, data_t data)
{
   if(mutex.try_lock())
   {
      if (buffer.find(msgid) == buffer.end())
      {
         data_l list;
         make_entry(data,list);

         buffer.emplace(msgid,list);
      }
      else
      {
         buffer[msgid].first.push_back(data);

         if(buffer[msgid].first.size() > 1023)
         {
            buffer[msgid].first.erase(buffer[msgid].first.
                begin());
         }
      }
      sem_set(msgid);
      mutex.unlock();
      return true;
   }
   return false;
}
```

`get()` works similarly by first locking the mutex and then checking for the existence of the message id. If the message ID is present, the flag is cleared, the data list is copied to the address given in the function call and true is returned to indicate success.

`frontend` class

As mentioned previously, the front end is a collection of functions which interpret and present the data collected in the fifo for use in a potential GUI. This aims to hide some of the complexity of the underlying system from the GUI programmer. Any number of functions can be added to this class depending on the needs of the sensor. The general structure of a function can be seen in code 17.3.

Code 17.3: Function template for accesing data in fifo.

```
<PARAMETER_TYPE> frontend::get_<PARAMETER>()
{
   std::vector<data_t> v;

   while(!fifo_o->get(<MESSAGE_ID>, v));

   if (!v.empty())
   {
      double t = std::stoi(v.back().millis);
      std::string d = v.back().data;

      /*Decipher d*/
   }

   return <PARAMETER_TYPE>;
}
```

Part IV
Verification

The complete system was not made to function, however, most of the subsystems were. Throughout this section tests will be designed and carried out in order to verify that the various subsystems live up to the requirements set in section 12.1.

Service virtualisation is used extensively in verifying the various systems. This is the process of emulating one piece of hardware or software such that another can act on the input given it.

Four parts are tested throughout this section, the CAN network, the WiFi, the node software and the front end. These parts constitute both the system on the go-kart as well as the system on the monitoring station.

18 CAN Bus

With the CAN controller available on the PS of the Zybo, the CAN bus will be tested to determine the transmission capabilities of the network. The transceivers are able to work at up to 8 Mb/s, while the CAN controllers on the Zybo only guarantee functionality up to 1 Mb/s.

The tests are performed using the design from figure 13.6, on page 31 as the FPGA part, though not all parts of this design is used for each test. Software was developed for bare-metal purpose, and because of this, not all tests were performed using interrupts, as the Zybo had no other task that it could be interrupted from.
The tests performed will be:

- **Basic communication**: verifying the ability for basic communication between nodes, while confirming, that the CAN stack works.
- **Latency tests**: Measuring the time it takes to send one full frame.
- **Bandwidth**: Determining how much raw data can be transmitted per unit time.
- **Priority when multiple node sending**: Ensuring, that higher message ID make way for the lower ones.

18.1 Basic Communication

In order to verify that the basic communication of a CAN network works on the developed CAN bus a test needs to be conducted. The method of the test was to implement the functionality of sending the input value of a pressed button to the network and to receive frames from the network, decode the frames and turn on the appropriate LED. The flow of event for sending a

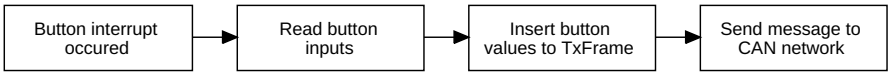


Figure 18.1: Flow chart with button interrupts.

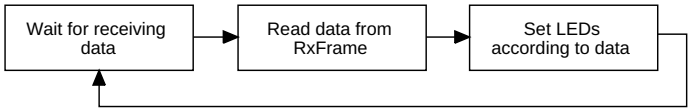


Figure 18.2: Flow chart for the process of receiving data.

frame is shown in figure 18.1 and the flow for the receiving a frame is shown in figure 18.2.
All Zybos involved in the test was programmed with the architecture shown in figure 13.6 and programmed with both the functionality of sending and receiving frames.

When pressing a button on a Zybo the appropriate LED on the other LED turned on, verifying that frames were packed, transmitted, received and unpacked correctly. The test was also performed with multiple Zybos connected to the CAN bus, in this case pressing a button on a Zybo turned on the appropriate LEDs on the rest of the Zybos.

18.2 Latency Test

For this test, only two Zybos are needed. One node prepares a frame for transmission, then sets a GPIO pin high. It then performs the necessary checks and writes the message details to the TX FIFO, and then sets its GPIO pin low.

The other node will then wait for the full message frame to be received, and then set its GPIO pin high. Once the metadata of the message (data length and message ID) has been intepreted, the GPIO pin goes low again. Using an oscilloscope, it is possible to measure the time it takes to transfer a message.

The test will be performed for an 8 byte frame. The messages will be constructed, so that bit stuffing doesn't occur by writing 0xA1 to 0xA8. The resulting voltage measurements are displayed ion figure 18.3.

The time from the red voltage goes low, to the time the blue voltage goes high is 87.5µs. This is of course very dependent on the particular controller used for this test, which according to the datasheet can work up to 1 MHz. Measuring this test shows, that bit come at 1.25 MHz. The software used for basis of this test does allow to adjust a pre-scaler, so that the controller works faster, but it is not able to receive frames at a higher rate than 1.25 MHz.

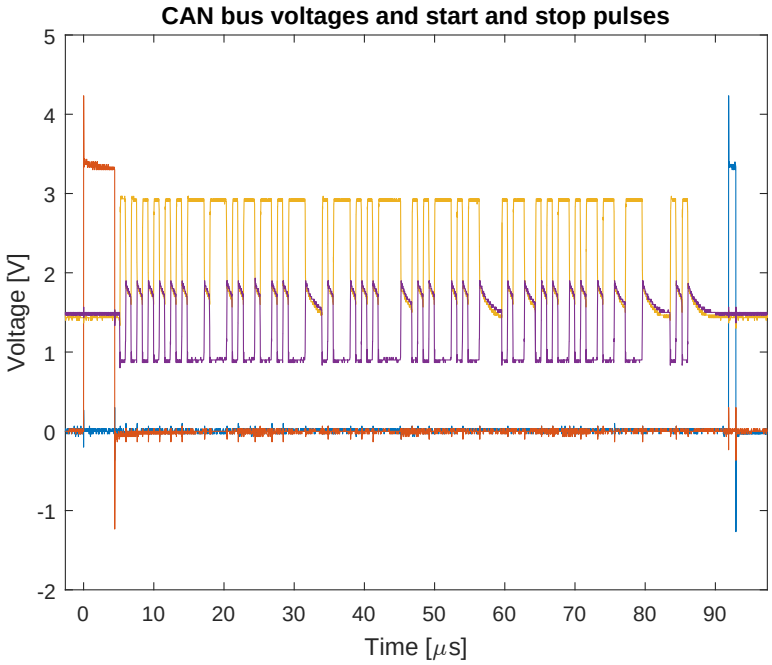


Figure 18.3: Start and stop pulses for an 8 byte CAN message. Yellow indicates the voltage of the CAN_H, purple indicates the voltage of CAN_L.

The CAN bus voltage can be interpreted to bits, to show what's actually being transmitted. This is done by measuring the voltage difference between the yellow and purple graphs, keeping in mind, that a difference in voltage corresponds to a zero, while no difference corresponds to a one. The CAN frame is displayed and interpreted ion figure 18.4.

The data portion of the frame is bitwise big endian, but bytewise little endian. The message to be sent was codes as two 32 bit unsigned integers: 0xA1A2A3A4 and 0xA5A6A7A8. The controller then swapped the bytes around, causing the data to become little endian. This is a convention implemented by the CAN controller, and as the same endianness is applied when sending and receiving the frame, it is irrelevant. The message was received correctly.

The CRC is calculated automatically by the controller. Unfortunately it ends on five consecutive `1`'s, meaning that a `0` must be stuffed in- betweenfore the delimiter, causing the frame to be one bit longer.

Additionally this controller uses 5 bits for the IFS part of the frame, rather than the mandatory minimum of 3 bits.

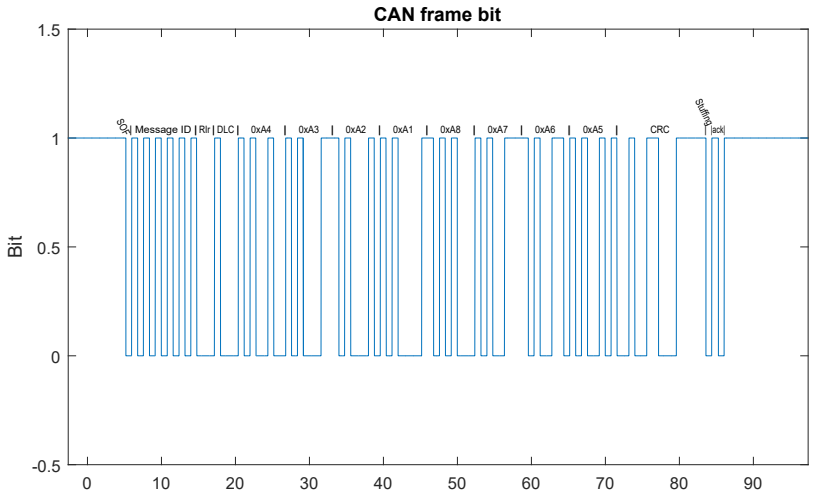


Figure 18.4: Bit interpretation of the CAN frame. RIr is the tree bits: RTR, ID Extention and r0, which are all 0.

18.3 Bandwidth

This will be calculated, as the faster controllers are not available. Bandwidth is considering the amount of net data being transmitted per unit time, when excluding the overhead. Bandwidth will be calculated for 8 byte frames, and bit stuffing will be omitted.

The maximum operating data rate for the CAN protocol is 1 Mb/s. As mentioned in section 13.2, the CAN frame has 47 bits of overhead. Including 8 bytes of data, this comes up to 111 bits. Time per frame is 111µs. As each frame contains 8 bytes of data, the data rate becomes:

$$\frac{8}{1.39 \cdot 10^{-5}} = 72072 \tag{18.1}$$

That means, that the effective transfer rate is 70.4 kB/s, equal to 563 kb/s. This is significantly less than stated in the requirements.

18.4 Message Priority

This test is based on the CAN polled example, where two Zybos will transmit data. The two transmitting Zybos will prepare a CAN frame with the same data content, but different message IDs.

- Zybo A will send message ID 0b10100100000.
- Zybo B will send massage ID 0b10101000000.

I.e. only the fifth and sixth bits have been swapped, meaning that Zybo A will have the higher priority. Both of these Zybos will prepare their respective Tx frame, and continuously poll the PMOD port. The PMOD connection must be configured to have pull down, to ensure that unconnected pins are

set low. Using a DC voltage source, the GPIO port of each Zybo will be set high simultaneously.

When the PMOD returns a non-zero value, each Zybo will call the XCanPs_Send function twice. This will fill two frames with 8 bytes of data into the Tx FIFOifo of each Zybo. Because of the asynchronous nature of the CAN protocol, it is still somewhat random which message gets transmitted first, so the frame will be sent twice. This means, that after the first frame is sent from either one of the Zybos, both zybos will start transmitting at the same time. In this case one Zybo must stop transmitting when it detects, that it has the lower priority. This is shown ion figure 18.5.

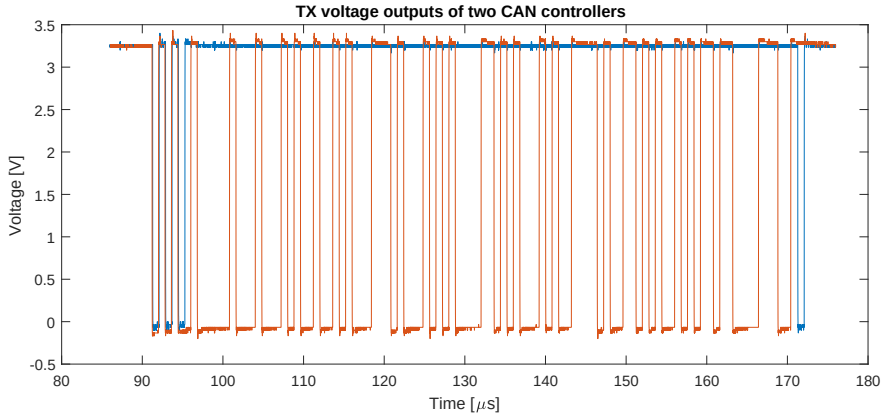


Figure 18.5: Voltage measurements taken at the TX pin of Zybo A (red) and Zybo B (blue).

The displayed frame is the second of four frames sent for this test, which is why the graph starts at 80µs. Both Zybos start sending their frame simultaneously, and as there is no difference until the fifth bit of the message ID, neither Zybo is aware that the other is also sending. At the time 95µs, Zybo B (blue line ion figure 18.5) sends out high, whilst receiving low, meaning that another node is sending as well. It will then cancel this attempt to transmit, and wait until the current frame has been transmitted before it will try again. Also note the Zybo B writing the acknowledge bit at the time 171µs. Because of this, it is not strictly necessary to use a receiving Zybo, because the other node will confirm the CRC of the message.

18.5 Conclusion

It has been shown that the CAN hardware does work. It is possible to transmit a message from one Zybo to another, without error. The substantial overhead does limit the potential data bandwidth. Additionally it was possible to measure the time it takes to construct a CAN frame, although this might vary a great deal from one controller to the next.

Parameter	XCanPs controller
Frame time	87.5µs
Build time	4.43µs
Data bandwidth	563kb/s

Table 7: Results obtained from the test of the CAN bus.

19 WiFi

A WiFi connection is required between the go-kart and a computer. This connection is required to have a range of at least 80m. Verifying that this is possible is done throughout this section.

19.1 Method

A Zybo with the TP-LINK TL-WN722N connected is placed with 80m, unobstructed, distance to a laptop. The ad-hoc network is initiated using the method described in section 14. `ping` is used to verify that the connection is made.
A small program was written to verify a stable connection. This program increments a counter every milliseconds and sends the value via WiFi PC to the Zybo. Allowing this program to run for 10 seconds should yield 10000 data points on the receiving end. This was repeated for 80m and 110m. In both cases the complete dataset was received. It was decided not to continue the test since the requirement was met at that point.

20 Node Software

When addressing the node software it is important to remember the distinction between sensor node software and WiFi node software. The software for the WiFi node has not been implemented therefore this section will only address the verification of the functionalities of the sensor node software implemented on a GPS sensor node.

20.1 GPS Sensor Node

The functionality of the developed GPS sensor node software was tested as shown in figure 20.1. The GPS service virtualization was run to input data as a GPS module does. The GPS node software then processed the data and output it to standard output as the CAN bus connection was not realized.
When running the software is run it outputs only a string of booleans. This is not very readable by humans, so to show the functionality the program was set to print in a humanly understandable way. A segment of the output from this can be seen in code 20.1. The first frame should be a timestamp frame. It can be seen that the the first frame has node ID 14, 4 data bytes, message type 1 and data 2374, meaning that it is from the GPS node with a timestamp of 2374 milliseconds. The next frame should then be a data frame containing latitude and longitude information. It can be seen that it has node ID 14, 8 data bytes and message type 9, which is a latitude longitude message. To

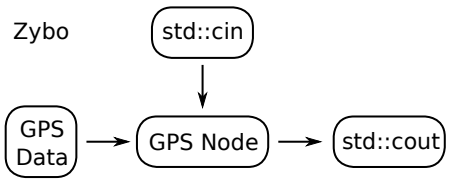


Figure 20.1: Testing the node software. The node software receives data from a list of GPS data and inputs from `std::cin`. It outputs to `std::cout`.

extract the latitude and longitude from the data field an unpacking algorithm is needed. Using that it is found that latitude is 55.3673346 degrees North and longitude is 10.4318456 degrees East.

Code 20.1: Struct for data packet.

```
1                                   ⋮
2  Start of frame: 1
3  Node id: 1110
4  Number of data bytes: 0100
5  Message type: 000001
6  Data: 00000000000000000000100101000110
7  Full frame: 11110000001010000000000000000000000100101000110
8
9  Start of frame: 1
10 Node id: 1110
11 Number of data bytes: 1000
12 Message type: 001001
13 Data:
       00100001000000000110001010000010000001100011011111000101
       11111000
14 Full frame:
       11110001001100000100001000000000110001010000010000
       00110001101111100010111111000
15                                  ⋮
```

It should be possible to send a sync command to the node by giving it the message ID `10001111000`. When typing this into the shell it can be seen that the nodes timestamp is reset. By sending the message ID `10001111010` the node stops outputting data. The node starts sending data again if the message ID `10001111001` is sent.

The link to the CAN program is not made and therefore the tests are made with standard I/O. By using service virtualization it is verified that the GPS sensor node software has implemented the wanted functionalities.

21 Front End

According to the use cases a user should be able to monitor data created on the network. This section describes the test done to verify that this functionality is implemented correctly.

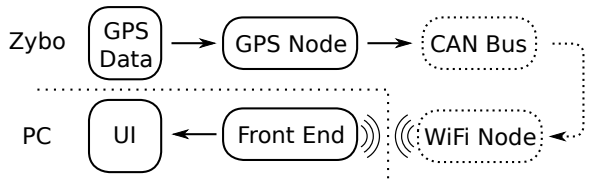


Figure 21.1: The setup used to verify the functionality of the front end. The dotted boxes are virtualised but the correct connection is maintained between the Zybo and the PC.

21.1 Method

Figure 21.1 depicts an overview of the setup used in the verification. An amount of GPS data is recorded and is presented to the GPS node using service virtualisation.

The two parts, CAN bus and WiFi node were not finished, see sections 13.6 and 15.2 respectively, as such it was necessary to create service virtualisation for this link. A small utility was written to serve this purpose, it virtualises the dotted boxes seen in figure 21.1. This utility reads the messages sent by the GPS node, extracts the timestamp and inserts it into the data frame in place in the DLC nibble and then outputs it to stdio. The message is then on the form seen in figure 17.3, the form expected by the front end. This means that, seen from the perspective of the front end the underlying structure is correct.
In order to most accurately reproduce the actual function of the system, the test was done using a wireless connection between a Zybo and a PC. The connection was established using the method described in section 14. On the PC, the front end was started, taking its input from a socat listener.

```
>> ./frontend | socat - tcp-listen:2049
```

Then, on the Zybo, the GPS and WiFi nodes were started, piping their output to socat:

```
>> ./sensornode | ./verification | socat - tcp
    :196.178.10.10:2049
```

Here, the first and last data points from the data set are shown:

```
>> N 55.36734 E 10.43185
>> N 55.36707 E 10.43109
```

This corresponds with the data created on the GPS node.

Part V
Conclusion

Throughout this report has been described the creation of a system for gathering and monitoring data from the SDU go-kart. Here will be presented a consollidation of the outcome of the different parts of the project, as well as some of the observations with respect to the working methods employed by the authors.

Initially, an attempt was made to exemplify the requirements of a system such as the one being developed. Each part of the system was analysed to this effect. While the analysis does touch on every part of the project, it was not until late in the project that the value of the analysis was clear to the authors. Some mistakes were made to this effect. For instance, on making the connection between the PC and the Zybo, three different methods were tried before finally settling on socat. A process that might have been avoided with proper analysis of the problem at hand. Even with the detours, a collection of requirements were set. Many of which were verified in part IV.

22 State of the Requirements

The results of the verification is consolidated here to provide an overview of the state of the project. Each requirement is listed with an explanation as to whether the requirement was met. Requirements marked with ✓were fully realised, – partially and ✗ are not realised.

22.1 Functional Requirements

- ✓ **Read data from sensors/data producers:** Data was successfully read from a GPS module. The data was gathered using the real GPS module but service virtualisation was utilised in verifying the functionality of the node responsible of handling the GPS. This is explained in section 20.
- ✓ **Timestamp all data:** A system was devised such that a packet contain the timestamp of the data is sent prior to sending the data. The timestamp was left out of the actual data packet to limit the bandwidth spent on transmitting timestamps. This is explained in section 20.
- ✓ **Transmit data wirelessly between Zybo and monitoring station:** This was succesfully done using a WiFi connection. An ad-hoc network was created using a WiFi/USB dongle, which communicates wirelessly to a PC. Verification of this feature can be seen in section 19.
- ✗ **Log data to SD card:** Due to time constraints, it was not possible to implement this feature.
- ✗ **Transfer logs to monitoring station:** Since a log of the data is not kept, this requirement could not be fulfilled.

✓ **Start/stop transmission from nodes:** This feature is part of Go-CAN, the protocol developed in the report. Commands can be issued to a node which will stop start and stop the transmission. This is shown in section 20.

✓ **Present data to user:** A basic program reading the GPS data was written to act as a placeholder GUI. The output of this program is shown in section 21.

✓ **Provide an API for data access:** A class was written to provide an API which acts as a front end to the underlying system. A function was written to interpret the GPS data and present it in a human readable format. This API was utilised in verifiying the previous requirement.

– **Must continue to function on node failure:** This requirement is fulfilled only partially. The network is agnostic to the number of nodes present at any one time and the failure of any sensor node will not bring the system down. The WiFi node does, however, provide the link between the CAN network and the monitoring station. Due to this, if the WiFi node fails, the system can no longer be used in monitoring data.

22.2 Operational Requirements

✓ **A developer can add new nodes:** A framework has been created which allows the developer to add the necessary code to support nodes. Guidelines on how to implement new nodes are provided in appendix C.

✓ **A developer can modify existing nodes:** Since the code for the framework mentioned in the previous requirement is available to the developer, it will also be possible to modify the code for any existing node.

✓ **A developer can add API for specific sensor:** The framework provides a front end class in which functions for accessing data from a particular node can be written. An outline of the process of adding to the API can be seen in appendix C.

✓ **A developer can add custom (G)UI:** Since the API is readily available to the developers, the parameters can be accessed and utilised for any purpose, including creating a GUI.

✗ **A user can request a data log:** Since a log of the data is not kept, this feature was not implemented.

– **A user can start/stop data transmission from nodes:** While toggling of transmission is implemented on a per node basis, there is no way for a user to issue this command through the API.

22.3 Quality of Service Requirements

- ✓ **Can support up to 16 nodes:** GoCAN has a 4 bit node ID. This can uniquely identify up to 16 different node, of which one has to be the WiFi node. The distribution of the node ID's currently in use can be seen in section 13.3.
- ✓ **Has wireless range greater than 80m:** A test was done to verify the range. It was found to be functional at a range of 110m. The test is described in section 19.
- ✗ **CAN network has at least 1Mb/s data bandwidth:** With the CAN controller used, it was not possible to get the required data bandwidth. The actual data bandwidth was found to be 720 Kb/s. This calculation is presented in section 18.3.
- ✗ **Timestamps with a precision of 1 ms:** While timestamping is done, it is currently not possible to guarantee 1 ms precision. This is due to the node being run on an ordinary Linux system. If the data gathered on the network is to be useful in any data processing application, it is necessary to port the system to a real time platform.

22.4 Design Requirements

- ✓ **Must allow for integration of new nodes:** The network is made such that it is agnostic with respect to the type and number of nodes on the network. This allows adding new nodes to the network without breaking existing nodes. Guidelines on how to implement new nodes are provided in appendix C.
- ✓ **Software must be modular to allow for simple integration of nodes:** The software was created in such a way that as few files as possible need intervention from developers adding new nodes. This will help simplify the process and allow integration of new nodes with minimal knowledge of the underlying network. Guidelines on how to implement new nodes are provided in appendix C.

23 Future Work

Unfortunately this project did not amount to a finished working product, but given more time, it would be possible to make the CAN controller work on all nodes, independently of whether it runs Linux or bare-metal code. In addition to this, there are some features which would be highly beneficial to include. All of this is discussed in this section.

Design CAN Controller on FPGA

As the final implementation might be done in PL, it is possible to utilize the 8 Mb/s transfer rate of the CAN transceivers. This does not conform with the ISO11898-2 standard, though, but a custom controller in FPGA can easily produce and interpret signals up to or beyond 8 Mb/s. First objective of

future work would be to implement the one missing link to make the entire project work. This would be implemented on FPGA for two reasons; the XCanPs controllers built into the Zybo can operate up to 1 Mb/s, but the transceivers can operate up to 8 Mb/s, so better to use the faster option. Another reason is modularity. An IP core could be included both on a bare-metal coded Zybo, and one running Linux. So instead of working on two different methods for implementing a CAN controller in the PS, one method would work for all Zybos.

Startup Security

As described in section 13.4, the WiFi node sends out a sync signal when the whole network starts up. The WiFi node will send this signal after a fixed time, but there is a risk, that not all nodes are ready to receive yet. An added security measure would be to add an "Ready for Sync" message, that all nodes will send out at a fixed interval. The WiFi node will then need to know which to expect, and will not send a synchronize message to the other nodes until all of them have reported ready.

Timestamping

As described in section 15.3 the time stamping on the sensor nodes is done in userspace in Linux, which does not produce accurate timestamps. The reason for using Linux on the nodes is mainly that USB and especially WiFi requires complex drivers that are already implemented in Linux. As the frames are timestamped when they reach the WiFi node there is no requirement for it to be real-time. This leaves a solution to the imprecise timestamping to let the WiFi node run Linux and the rest of the nodes to run on bare-metal code. That of course requires a USB driver that can run on bare-metal code.

Extended Command Structure

As described in section 13.3, the WiFi node can only transmit 4 different message types, because 4 bits are used to address the command. According to the OD in 14, three of these are already in use; Sync, Start and Stop. This leaves only one more command to transmit.

A more prudent approach would be to restructure the commands, and possibly use the data field to convey more information. Four command IDs will be used: sync, command, set parameter, and get parameter. Sync will be as it is now, but command would be any commands to control specific nodes, such as start, stop or change mode. The data field would then contain one byte of data, determining which command is being sent. The getting and setting of parameters, would require one or two bytes of the data field to determine which node is being manipulated. Additionally, changing parameters on a go-kart while it is driving can be dangerous, so it will be important to include safety measures on the target node.

Internode Communication

One of the advantages with a bus, is that all nodes can receive all of the communication. At this stage though, the nodes cannot interpret what the other nodes are sending out, but there might be situations where one node would want to get data from another. One example could be velocity information from the front wheels being used by the motor controller to determine if the wheels are slipping, and reduce the torque accordingly. To reduce noise, it would be better to put a node closer to the front of the car, than to use long dedicated cables for each sensor in the front of the car, and then transmit the data over the CAN bus.

At this point, this is not possible, because one node has no knowledge of other nodes than itself and the WiFi node. This is to make the system more modular. Although implementing internode communication would likely make the network more rigid, it would likely be possible to ensure that the front node in the above example could be changed, without having to make changes to the motor controller node.

Utilizing Asymmetric Multiprocessing

A proposed solution for the problem of accessing the CAN controllers from Linux is implementing asymmetric multiprocessing. It is supported by the Zynq-7000 AP SoC since there are two cores on the same processor which share common memory as well as peripherals.

The idea of this is to run Linux OS on one core, while on the other one a bare-metal system, which both of them can communicate with each other through shared memory. The core running Linux is needed to be set up as the master CPU since it is the one that starts the bare-metal system. Also, the operating system needs to be configured as symmetric multiprocessing with a maximum number of one CPUs. It is a good approach because it will ensure that Linux configures the interrupt control distributor and the snoop control unit appropriately for multi-CPU environment, but only running on one of the two CPUs. Lastly, the shared memory should be handled properly to avoid conflicts.

It is a promising solution, especially suitable for multi-core embedded platforms where different systems need to run simultaneously. In the case of this report, it could have solved the communication difficulty between the CAN controller and Linux, but due to lack of time, it was researched only at a theoretical level. For more details about the instructions on implementing this mechanism, the reader may refer to the Xilinx document [12].

References

[1] Wikipedia, https://en.wikipedia.org/wiki/Earth's_magnetic_field

[2] University of Southern Denmark, 2014, http://www.sdu-vikings.dk/

[3] Blackman, Jason and Monroe, Scott, Texas Instruments, January 2013, Overview of 3.3V CAN (Controller Area Network) Transceivers.

[4] Xillybus LTD, 2013-2016 http://xillybus.com/xillinux

[5] Open source, June 2015, https://wireless.wiki.kernel.org/en/users /drivers/ath9k_htc/devices

[6] National Instruments, 2013, http://www.ni.com/white-paper/14162/en/

[7] Xilinx, 2015, https://github.com/Xilinx/embeddedsw/blob/master /XilinxProcessorIPLib/drivers/canps/examples /xcanps_polled_example.c

[8] Xilinx, 2016, http://www.wiki.xilinx.com/Linux+CAN+driver

[9] Hall, Brian, June 2016, Beej's Guide to Network Programming - Using Internet Sockets.

[10] www.boost.org, Boost C++ Libraries, http://www.boost.org/doc /libs/1_62_0/doc/html/boost_asio/examples/cpp11_examples.html

[11] VectorNav, vn-100 library, http://www.vectornav.com/support/downloads

[12] John McDougall, February 14, 2013, XAPP1078 v1.0, Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors, https://www.xilinx.com/support/documentation/application_notes /xapp1078-amp-linux-bare-metal.pdf

[13] Xilinx wiki Asymmetric Multiprocessing, http://www.wiki.xilinx.com/Multi-OS+Support+(AMP+%26+Hypervisor)

[14] Wikipedia, Bluetooth, https://en.wikipedia.org/wiki/Bluetooth

[15] Wikipedia, IEEE_802.11 , https://en.wikipedia.org/wiki/IEEE_802.11 #Protocol

[16] The CONFIG_PREEMPT_RT community, Real-Time Linux Wiki, https://rt.wiki.kernel.org/index.php/Main_Page

[17] The CONFIG_PREEMPT_RT community, Real-Time Linux Wiki, https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch

A Use Case Narratives

Table 8: Usecase narrative for monitor live data from go-kart.

Use case:	Monitor live data
Actors:	User
Purpose:	Monitor live data from go-kart while driving
Overview:	The engineer starts the system. The system will begin collecting data on the go-kart and transfer them to a stationary computer. The computer will present data to the engineer on a UI.
Type:	Essential
Preconditions:	Go-kart microcontroller is paired with stationary computer
Postconditions:	System transfers data from go-kart sensors to a stationary computer, showing data in a UI.
Special requirements:	-
Actor action	**System response**
1. Start system	2. Start collecting data 3. Transfer data to stationary computer 4. Present data in UI
Alternative flow of events	
Any line: User can stop the system at any point in time.	

Table 9: Usecase narrative for log go-kart data.

Use case:	Log data
Actors:	User
Purpose:	Log go-kart data
Overview:	After startup the system will log the collected data locally on the go-kart
Type:	Essential
Preconditions:	System is running and collecting go-kart data
Postconditions:	System logs collected go-kart data.
Special requirements:	-
Actor action	**System response**
	1. Log collected data locally on go-kart
Alternative flow of events	
Any line: User can stop the logging at any point in time.	

Table 10: Usecase narrative for read logged data.

Use case:	Read logged data
Actors:	User
Purpose:	To read data that is logged locally on the go-kart
Overview:	The engineer will ask the system for the logged data. The system will transfer the logged data to a stationary computer
Type:	Essential
Preconditions:	Data is logged in logfile on the go-kart
Postconditions:	The log file is on the stationary computer
Special requirements:	-
Actor action	**System response**
1. Ask for log file	2. Transfer logged data to stationary computer 3. Save data to log file on stationary computer
Alternative flow of events	
Any line: If connection is lost the system should detect it and mark the transferred log file as invalid	

Table 11: Usecase narrative for start and stop data collection.

Use case:	Toggle data collection
Actors:	User
Purpose:	To start and stop data collection from specified sensors
Overview:	The engineer will ask the system to start or stop collecting data from a specific sensor. The system will then start or stop the system respectively.
Type:	Important
Preconditions:	System is running
Postconditions:	Data collecting is started or stopped for specific sensor.
Special requirements:	-
Actor action	**System response**
1. Ask system to start or stop data collecting for specific sensor.	2. System starts or stops data collecting for specific
Alternative flow of events	
-	

Table 12: Usecase narrative for add/modify data producers.

Use case:	Add/modify data producers
Actors:	Developer
Purpose:	To add new nodes to the network, or modify the existing ones
Overview:	Developer adds a node to the network without having to modify existing nodes.
Type:	Essential
Preconditions:	System developed
Postconditions:	A node is modified or added, and the system can be operated the same as before.
Special requirements:	-
Actor action	**System response**
1. A new node is added by a developer.	
Alternative flow of events	
-	

Table 13: Usecase narrative for add custom GUI.

Use case:	Add custom GUI
Actors:	Developer
Purpose:	Add a User Interface, to neatly display data from go kart
Overview:	An engineer develops a program to read relevant data from the front end program on the stationary computer, and presents it graphically
Type:	Important
Preconditions:	System is not running
Postconditions:	System runs as usual, but with a new beautiful GUI
Special requirements:	-
Actor action	**System response**
1. Develop a GUI for the system.	2. System runs as usual, but with the new GUI
Alternative flow of events	
-	

1	**Wifi 0001**	**Data Length**	**Data Types**	**Fields**	**Scaling**	**Units**
2	00	0	-	Sync	-	-
3	01	0	-	Start	-	
4	10	0	-	Stop	-	
5	**IMU 0010**	**Data Length**	**Data Types**	**Fields**	**Scaling**	**Units**
6	000001	4	u32	Timestamp	1	ms
7	001000	8	float	Pressure, Temperature	1, 1	kPa, °C
8	001001	6	s16	A_x, A_y, A_z	0.00478	$\frac{m}{s^2}$
9	001010	6	s16	G_x, G_y, G_z	0.000266	rad/s
10	001011	6	s16	M_x, M_y, M_z	0.15	µT
11	001100	24	s16	A_x, A_y, A_z, G_x	0.00478, 0.000266	$\frac{m}{s^2}$, rad/s
12				G_y, G_z, M_x, M_y	0.000266, 0.15	rad/s, µT
13				M_z, Yaw, Pitch, Roll	0.15, 1e-4	µT, °
14	**Sevcon 0100**	**Data Length**	**Data Types**	**Fields**	**Scaling**	**Units**
15	000001	4	u32	Timestamp	1	ms
16	001001	8	s16	I_d, I_q, V_d, V_q	0.0625	A,V
17	001100	8	s16	I_a, I_b, I_c, Ω_e	0.0625, 1e-4	A, rad
18	001101	8	s16	V_a, V_b, V_c, Ω_e	0.0625, 1e-4	V, rad
19	001010	4	s16	Speed, Torque	0.0625	$\frac{km}{h}$, Nm
20	**GPS 1110**	**Data Length**	**Data Types**	**Fields**	**Scaling**	**Units**
21	000001	4	u32	Timestamp	1	ms
22	001001	8	u32	Lattitude, Longitude	1,00E-07	°
23	001010	4	s32	Altitude	0.1	m

Table 14: The data length is given in bytes. Lines 8, 9 and 10 are compensated input directly from the respective sensors, i.e. no data fusion is used. In lines with different types of sensor data, the Scaling and Units may refer to more than one data field. For instance on line 11, 0.00478 and $\frac{m}{s^2}$ refer to $A_x, A_y \text{and} A_z$, while 0.000266 and rad/s refer to G_x.

C Adding a New Node to the System

In adding a new node to the system there are two steps that need three be done:

- Prepare the sensor node software.
- Add appropriate functions to the API.
- Update Object Dictionary.

This appendix will explain those steps to enable a user to add their custom nodes to the network.

C.1 Preparing sensor node software

The general node software is described in section 15.1 and the class diagram in figure 15.3. When developing a new node with another sensor the generic classes CAN_link, Protocol and Node should be used. New classes should be developed to extract data from sensor, pack data according to protocol and put the data in a data_packet. It is advantageous to implement this functionality into a Sensor_Packer and a Sensor class to keep a similar software design through the nodes. The interface to the generic Node class is a call to its function `put_data_packet(data_packet)`.

Sensor class

A Sensor class should be implemented to interface the sensor. This class will be very sensor specific and not generic. It should update its own variables with all relevant from the sensor. This functionality should be implemented in a loop function that runs in a thread and calls a function in the Sensor_Packer class when data is updated.

Sensor_Packer class

The Sensor_Packer class should also be implemented. It is sensor specific, because it needs to pack sensor data according to the specific sensors messagetypes. A Sensor_Packer class should inherit from the base Packer class shown in code snippet C.1. In the constructor of the Sensor_Packer all messagetypes for the sensor should be put in `messagetypes`.

Code C.1: Packer class.

```
1 class Packer {
2 protected:
3    std::vector<std::bitset<6> > messagetypes;
4 public:
5    Node* node;
6    void set_node(Node* node_in);
7 };
```

The class needs to pack data into the `data_packet` shown in code snippet 15.2. It is responsible to populate the variables `messagetype` and `data`. To do that all data that is to be sent needs to be converted into `vector<bool>`.

When `data_packet` is populated correctly it needs to be passed to the generic Node class using its member function `put_data_packet(data_packet)`.

C.2 Creating the API

The front end is comprised of a list of functions which enable the user to access the data collected on the network. The data, up until the front end is not in human readable format. Each function in the API reads the appropriate data and converts it to a more useful format. A template of an API function is given in code C.2.

Code C.2: Function template for accesing data

```
1  #define <MESSAGE_ID> "11110001001"
2
3  <PARAMETER_TYPE> frontend::get_<PARAMETER>()
4  {
5     std::vector<data_t> v;
6
7     while(!fifo_o->get(<MESSAGE_ID>, v));
8
9     if (!v.empty())
10    {
11       double t = std::stoi(v.back().millis);
12       std::string d = v.back().data;
13
14       /*Decipher d*/
15    }
16
17    return <PARAMETER_TYPE>;
18 }
```

The definition of the message id of the frame being decoded should be placed in `frontend.hpp`. The return type of functions is optional. The `get_coordinate()` function implemented, for example, returns a struct containing the relevant data. All data is kept in a vector with 0-1024 elements where the last element of the vector is always the newest. At line 5 a container for this vector, `v` is instantiated. The while loop will continue to run until `v` is populated. This will not happen until the `get` function succesfully locks the mutex. Each entry in `v` is a struct as seen in code C.3

Code C.3: Struct used to store each datapoint

```
1  struct data_t
2  {
3     std::string millis;
4     std::string data;
5  };
```

In this struct, `millis` is the number of milliseconds since startup in binary. `data` is the binary form of the associated data given by `MESSAGE_ID`. Once `v` is populated the data can be accessed as shown in lines 11 and 12.