Telemetry Module for DTU Roadrunners Solar Car

Victor Alexander Hansen s194027, Steffan Martin Kunoy s194006, Tjark Petersen, s186083
Department of Electrical Engineering
Technical University of Denmark
Kgs. Lyngby, Denmark
31015 Introductory Project - Electrical Engineering (Group 7)
s194027@student.dtu.dk, s194006@student.dtu.dk, s186083@student.dtu.dk

***Abstract*—This paper presents the telemetry project for the DTU ROAST solar car. The software and hardware developed for the telemetry system provides a two-way communication between the solar car and a support vehicle. The solar car module can read data from the CAN bus, store it locally in a black box and stream it to the support vehicle. In return the support vehicle can send commands to the solar car. All messages are secured with an RSA encryption. A graphical user interface enables the user to stream the CAN data and send commands. In addition, support for a simple UDP network stream was also implemented which can for instance be accessed in Matlab.**

A provisional hardware setup of the solar car and support vehicle modules was implemented using two Teensy microcontrollers on perfboards. A range test of the two modules had a successful and reliable transmission distance of up to around 160 m. While this does not satisfy the original goal of 400 m - 1000 m, the final solution is concluded to have a good communication chain design between the solar car and support vehicle modules, and can be readily improved upon to support a longer transmission distance.

***Index Terms*—Telemetry, CAN, RSA encryption, Matlab**

I. Introduction

In 2023, the DTU Roadrunners Solar Team (abbreviated ROAST) is set to participate in the Bridgestone World Solar Challenge, a race spanning over 3000 km across Australia from Darwin in the Northern Territory to Adelaide in South Australia [1]. During the race the car will be mainly powered by its solar panels which emphasizes the need to create an energy-efficient car.

Throughout the race the solar car will be accompanied by a support vehicle, which can monitor the car's condition and should be able to issue commands to the solar car's internal systems. This is a crucial function as the solar car will be navigating Australia's busy highways, where for instance the heat and tire pressure may cause a detriment to the vehicle. The support vehicle will therefore be able to analyze and react to the data stream being transmitted from the solar car, even when the driver is preoccupied driving the car.

Additionally, valuable data can be collected during the race which can help to identify possible areas of improvement of the solar car. Therefore a ”black box” system is needed to capture all system activity in a local storage.

These key functionalities require a system which is able to read and write data from the solar car's on-board Controller Area Network (CAN) bus and transmit it via a radio frequency (RF) transceiver to the support vehicle, while simultaneously logging data locally. The support vehicle is equipped with a similar module such that the support crew can react to the data manually or automatically by sending commands back to the CAN bus via RF. Thus, the capabilities of such a telemetry module play a vital role in the communication between both vehicles, considering that the distance between them may be up to 1 km depending on traffic conditions.

This project aims at implementing a functioning prototype of a telemetry system while ensuring that the system meets the specifications with a solid solution.

A. Problem Statement

Remote sensing of data from the solar car will give the ROAST team a competitive advantage as the support vehicle will be able to take on a more active role in managing and optimizing the car's performance. This will ease the burden on the driver and leverage the expertise of the supporting crew. For these reasons we have chosen to focus our project on the following guiding questions:

- How can we design a module that can read and write sensor data and commands from a CAN bus network?
- How can CAN data be transmitted securely over a distance of up to 1 km?
- How does the unit sitting in the support vehicle process data upon receiving it from the solar car?
- What is the best method for storing data locally (black box) while being accessible at a later time?
- How can the CAN data be made accessible in the support vehicle for further processing e.g. in Matlab?

II. Methodology

The project was executed under a series of design and implementation sprints, each resulting in an increasingly finished product. At first a detailed specification was drafted which then served as a reevaluation tool for future sprint procedures. Subsequently, we created our first proof of concept (POC) model with the hardware setup on two breadboards. Our aim was to test the basic functionality of the system, such as receiving CAN messages and sending data over the RF channels.

Fig. 1. Bridgestone World Solar Challenge logo [1].

During the following design sprints we focused mostly on software development objectives until we had achieved a minimum viable product (MVP). At this point our system worked as intended, albeit with reliability issues and software bugs. Hence, the code was validated thoroughly with unit tests that determine the behavior of our code against its intended behavior. The final hardware sprint resulted in soldering of our entire setup on a perfboard for better reliability. Real-life tests were then conducted to prove the efficacy of our product. Upon applying some final touches, the final product was completed and handed over to DTU ROAST.

III. Design

The overall system consists of three separate modules. The solar car unit (SCU) is connected to the CAN bus and communicates with the support vehicle unit (SVU) via a wireless RF connection. The SVU in turn interacts with a laptop in the support vehicle through a serial connection. The TelemetryUI program running on the laptop broadcasts the CAN bus message stream on the local network. A block diagram for the system is outlined in Fig. 2.

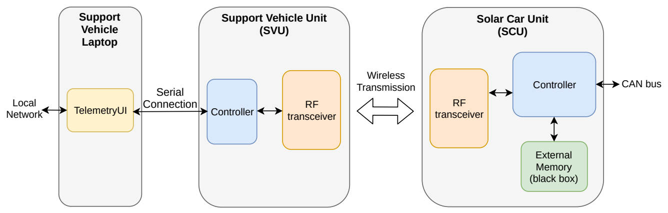


Fig. 2. Overview of the system's components in block diagram form.

IV. Components

A. Microcontrollers

We use Teensy 3.6 and 4.0 microcontrollers as connection hubs in the SCU and SVU respectively. The Teensy line-up has been used as the main microcontroller platform in most DTU Roadrunners projects and we decided to use them as well after considering the performance of newer Teensy boards. Being based on the Arduino framework, both Teensys offer support for numerous protocols and software libraries to ease the implementation of our design.

B. CAN bus

The CAN network in the SCU enables all the car's electronic control units (ECU) to send and receive data without a host computer. This is possible thanks to the special structure of the CAN message.

Fig. 3 shows a detailed structure of the CAN frame. The fields of importance for our project is the message identifier, RTR, control field and data field. The rest of the fields serve error checking and coordination of communication on the bus.

Each ECU has a unique ID, which the message identifier uses to determine where the CAN message should go to. When two messages try to access the bus, the one with the highest priority (lowest ID) will be granted access to the bus, while the other messages have to wait. After the ID comes the RTR bit which determines whether the frame remotely requests or actually sends data to a ECU. The control field, in other literature referred to as "Data Length Code" (DLC), holds the number of bytes used in the following frame field containing the actual data[3][4].

The Teensy 3.6 used in the solar car features a CAN controller receiving data from the CAN bus as well as handling message assembly. However, in order to interface with the physical CAN bus, a transceiver is needed. Here the MCP2551 was chosen as it has been used extensively in other DTU Roadrunners projects. The component maintains one input and one output serial stream to the CAN controller and drives the CAN bus when a message is transmitted.

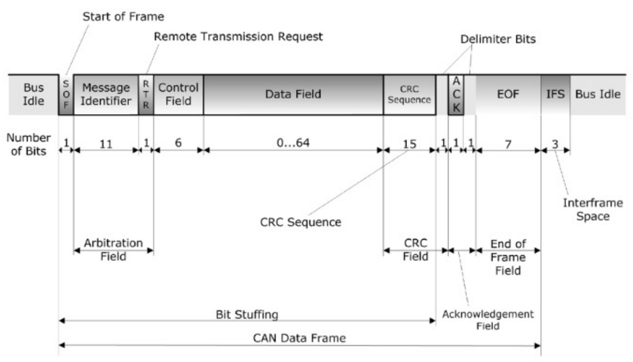


Fig. 3. Structure of a CAN frame, non-extended [2].

C. RF network

Two RF transceivers are required to provide a wireless connection between the SCU and SVU. There are many options for radio modules supporting long range communication at the expense of achievable data rate. We chose a trade-off between both by selecting the nRF24L01+ operating on the 2.4 GHz ISM band. The module features a detachable antenna, power amplifier and low-noise amplifier to provide a theoretical range in excess of 1000 m and data transfer rates of up to 2 Mbit/s.

An advantage of using the off-the-shelf RF module is the automated message protocol, known as Enhanced Shockburst© [5]. Messages are transmitted in packages, structured in flexible byte segments, as illustrated in Fig. 4. Only the data payload and address needs to be specified prior to sending a package; the preamble, packet and cyclic redundancy check (CRC) segments are generated automatically by the RF module.

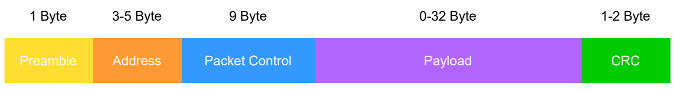


Fig. 4. The Enhanced Shockburst© packet structure.

D. SD card

We chose an SD card as a permanent local storage solution for the SCU's "black box" due to its ease of use and accessibility through the Teensy 3.6 integrated SD card slot. Furthermore, this solution scales easily in terms of storage capacity, with new SD card sizes exceeding 1 TB.

V. Software Stack

Several key functionalities of the telemetry module are implemented in software. This, together with the aim of designing a system to be expanded upon in the future, makes the choice of software frameworks very important. Thus, an emphasis has been placed on the reusability and maintainability of the developed code.

A. Build system

As a build system, PlatformIO [6] was chosen for the project. It is a version control friendly, cross-platform embedded build tool which includes library management and works by defining *environments* to allow development with different embedded platforms on the same code base. It comes with a VScode plugin for ideal IDE support. Other project groups working on the solar car in parallel integrated their software into the PlatformIO ecosystem as well, easing extensive code reuse in the future.

B. Real Time Operating System

We chose to build the software for the telemetry system on top of a real time operating system. The small kernel running on the microcontroller behind the scenes when using a RTOS allows a step up in abstraction from bare metal programming. Different tasks can be spawned as threads running concurrently. These threads get delegated time slots to run on the processor by the real time kernel based on their priority.

This comes at the cost of introducing overhead when switching between threads, but the great flexibility in code organization possible through the use of an RTOS was determined to make the overhead a worthwhile trade-off.

As a specific RTOS, we chose ChibiOS [7], since it is supported on both the Teensy 3.6 and 4.0 and because it attempts to keep the memory footprint of the real time kernel as small as possible.

C. Graphical User Interface

The graphical user interface (GUI) included in the telemetry system serves a proof-of-concept purpose. Therefore, scala-swing [8] was chosen as a framework, due to the effectiveness of the scala language [9] and the high level of abstraction used when describing a user interface in scala-swing. Furthermore, a GUI implementation in scala can run on any system with a Java virtual machine and is therefore cross-platform compatible.

D. Data Processing

Due to the wishes of DTU ROAST to use the telemetry data in applications of predictive modelling and performance optimization, we needed a way to retain the data being sent on the CAN bus, both locally via logging and remotely in real time via streaming.

We log all the messages received on the CAN bus to the external "black box" memory as `.csv` files. Each line in the file corresponds to a message with date and time stamps, ID, RTR, DLC and 64-bit data fields as displayed in Listing 1.

```
"time","id","rtr","len","data"
"18/06/2021 07-56-23",851,0,4,4266193389
"18/06/2021 07-56-24",55,0,6,24023809739244
"18/06/2021 07-56-24",205,0,4,3791082139
"18/06/2021 07-56-25",4,0,6,251964400399878
"18/06/2021 07-56-26",1279,0,5,65365801097
```

Listing 1. Example of a `.csv` log file with random data.

For remote access to the data stream we decided to implement a simple UDP network stream being hosted by the TelemetryUI. This allows all devices on the local network to monitor the data being sent over the serial connection for use in further data processing. For a POC case we created a simple Matlab script to sample CAN messages from the UDP port and print them formatted to the terminal.

E. Other Libraries

In order to interface with the components presented in the previous section, a set of libraries were included in the project. These include a CAN network driver to support use of the Teensy 3.6's integrated CAN controller [10], an RF OSI layer providing seamless integration with the nRF24L01+ modules [11], and an SD card driver to control data logging in the black box memory [12].

VI. Internal Communication

The main function of the telemetry system is to bridge data streams between different end points. As such, a flexible way to pass commands and associated data payloads both ways through the system is needed.

The majority of messages passed between the two telemetry subsystems will contain CAN messages as part of the stream from the solar car to the support vehicle. Therefore, the message protocol is designed around 16 byte messages, which are large enough to comfortably fit a CAN message and a time stamp from when the message was received. A diagram of the byte layout of a message is shown in Fig. 5.

The system is built around *commands* and *message types*. All message types begin with a command byte. The remaining 15 bytes are interpreted based on the message type. All message types can be worked with in an uninterpreted state as a `BaseTelemetryMsg`. As an example, a `CanTelemetryMsg` message type is shown in Fig. 5.

One message type can be connected to different commands. For instance, the `CanTelemetryMsg` message type can be used to send data from the SCU to the SVU as part of the data stream setting the command to `RECEIVED_CAN = 0x00` or it can be used to inject a CAN message from the SVU into the solar car's CAN bus setting the command to `BROADCAST_CAN = 0x01`. Even though such message type and command combinations are only sent one direction

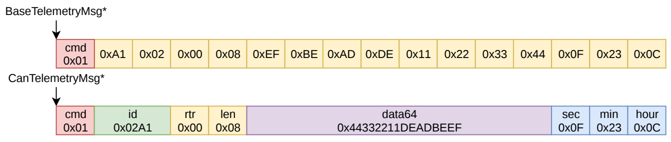


Fig. 5. The different message types used in the telemetry system.

between the telemetry modules, a unified message system for both directions was chosen due to the inherent simplicity.

When decoding messages, only the first byte needs to be considered and it will unambiguously decide how the rest of the message should be interpreted. This eases the decoding of messages on the receiving end and makes for a expandable framework.

As an extra feature not included in our original specification, we have made an encryption protocol for our telemetry module. The feature can be activated by transmitting an `ENABLE_ENCRYPTION` command to the SCU. The encryption is based on the RSA algorithm [13], which essentially generates an encryption key based on the modulus and totient function of two prime numbers. The pairings of prime numbers suited for the encryption protocol can be seen in a table uploaded to the DTU ROAST BitBucket [14].

When a message is encrypted, each of the 16 bytes is extended to a 16-bit unsigned integer to avoid overflow problems increasing the total message size to 32 bytes. Once transmitted to the receiving node, the message is then decrypted and the overflow protection bytes can be removed leaving us with 16 bytes again. This essentially doubles the size of the data payloads being sent in our RF transmission, which is the downside of our encryption solution.

The CAN bus used in the solar car is driven at $125\,\mathrm{kbit/s}$. With the shortest possible CAN message being 47 bits and the longest being 111 bits, a theoretical throughput of $1126\,\mathrm{s}^{-1}$ to $2659\,\mathrm{s}^{-1}$ messages can be achieved at full usage of the bandwidth.

The RF connection transmits Shockburst© messages of worst case 392 bits length at $1\,\mathrm{Mbit/s}$ and is thus able to achieve a throughput of $2551\,\mathrm{s}^{-1}$ messages which is slightly less than the peak CAN bus throughput. This is unlikely to occur in reality, since all CAN messages need to have an empty data field to achieve the peak throughput.

A serial connection of up to $6\,\mathrm{Mbit/s}$ can be achieved using the Teensy 4.0. When sending from the SVU to the TelemetryUI, the message types are converted into JSON [15] strings, as shown in Listing 2, which are 117 bytes long in the worst case, giving a throughput of $6410\,\mathrm{s}^{-1}$ messages. The JSON string can be used directly to set the state of an object inside scala which then in turn can be used for further processing.

```
{"cmd":0,"can":{"id":1018,"rtr":false,"len":5,
"data":831599620407,
"stamp":{"hour":8,"minute":58,"second":27}}}
```

Listing 2. Example of a JSON string.

VII. Implementation

A. Solar Car Unit

The SCU is built around the Teensy 3.6 and includes the aforementioned CAN transceiver and RF module. In the solar car, all modules receive power through a 4-pin molex connector at $12\,\mathrm{V}$. The molex connector also includes the differential pair of wires of the CAN bus.

Different voltages are required to operate the hardware setup correctly. Therefore, a linear voltage regulator is chosen to step down the external voltage from $12\,\mathrm{V}$ to $5\,\mathrm{V}$ to power the

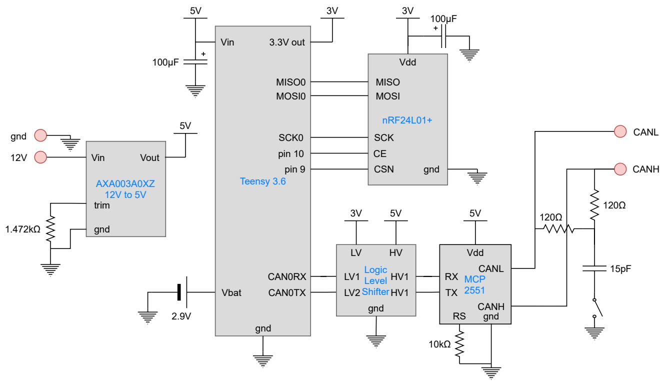


Fig. 6. The circuit diagram for the SCU.

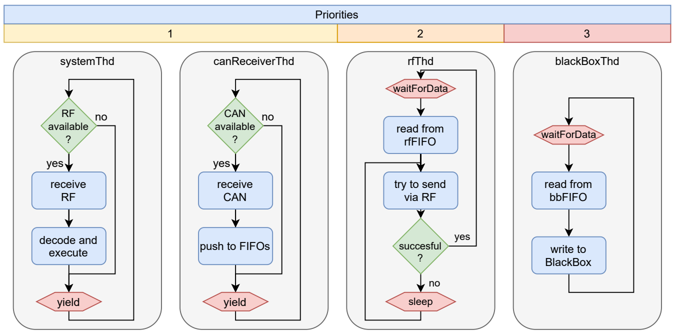


Fig. 7. The execution flow of the 4 threads running in the SCU.

Teensy and the CAN transceiver. The RF module is powered from the Teensy 3.6's integrated $3.3\,\mathrm{V}$ regulator.

A real time clock (RTC) is included on the Teensy 3.6 which can keep the time while the Teensy is powered off when a coin cell battery is connected to its V_{bat} terminal. This ensures that the generated time stamps associated with a received CAN message are always valid, provided the RTC is synchronized.

A complete circuit diagram of the SCU is shown in Fig. 6. The final circuit contains decoupling capacitors for the Teensy as well as the RF module and an optional CAN bus termination which can be activated by a switch.

On the software side, the SCU is executing four threads to handle its tasks. A diagram of the execution flow of the threads is shown in Fig. 7.

There are two threads running at the lowest priority. Since the right to execute is based on priority, these threads use a technique called *cooperative multithreading* to solve the problem of arbitrating execution slots between them. Cooperative multithreading is based on the two threads voluntarily giving up their execution right by calling `yield` in reasonable time intervals, thereby giving the other thread at the same priority the right to execute [16].

In the system thread, messages are received via RF, decrypted, decoded and the action connected to the message is executed. The CAN receiver thread tries to receive new messages from the CAN bus. These messages then need to be passed to the two high priority threads, which process them.

This inter-thread communication is performed using two First-In-First-Out (FIFO) buffers, one for each consumer thread, which decouple the producer and the consumer. The two high priority threads only need to execute when new data is available. To achieve this, a counting semaphore is used which allows one thread to hand out execution rights by incrementing the semaphore counter and another thread to wait for execution rights [17]. This technique is used to let the producing CAN receiver thread activate the consumer threads when new data is available in the FIFO.

The black box thread takes a CAN message, converts it into a `.csv` file line and writes it into a buffer, which is emptied to the actual SD card when filled. When a predefined number of CAN messages have been logged to one file, a new file is opened. The RF thread encrypts the CAN messages and tries to send them to the SVU. If unsuccessful the RF thread goes to sleep for a predefined amount of time and wakes up periodically to reattempt transmission.

The system thread always runs in order to receive commands from the SVU. All other threads can be paused from the system thread making it easy to stop logging or streaming of the CAN data.

B. *Support Vehicle Unit*

The SVU is anchored around a Teensy 4.0 microcontroller connected to an RF module. The unit receives power and serial data through a micro USB cable connected to a laptop. Furthermore, the RF module is bridged via the 3.3 V supply and SPI pins, as per the circuit diagram in Fig. 8.

On the software side, the SVU core executes a *receiver* and a *transmitter* thread on the same priority level with thread shifting enabled by the `yield` method [16]. The execution flow of both threads is outlined in Fig. 9.

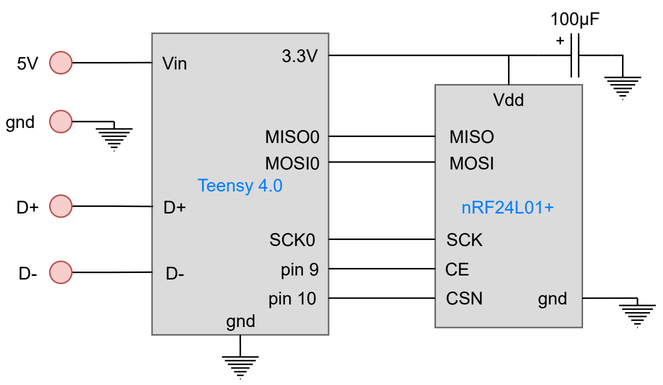


Fig. 8. The circuit diagram for the SVU.

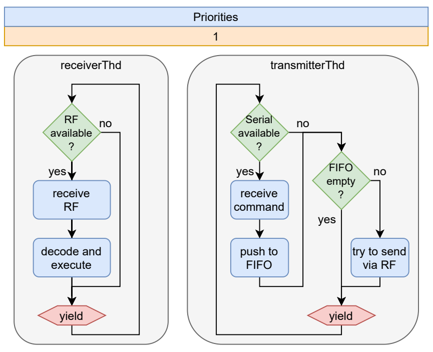


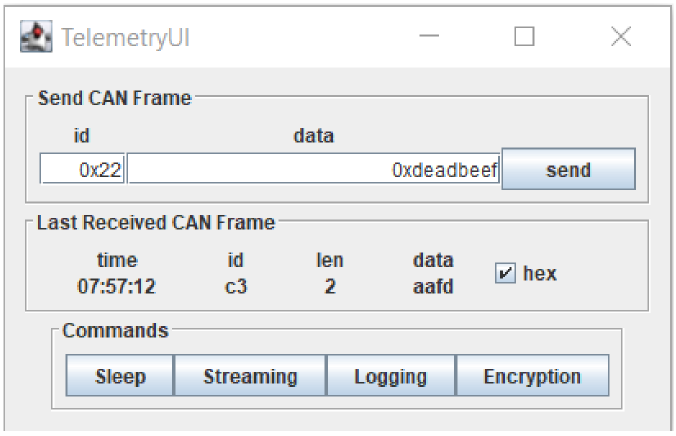
Fig. 9. The execution flow of the 2 threads running in the SVU.

During execution, the receiver thread listens for incoming messages on the RF channel, decodes and relays the message to the TelemetryUI. Meanwhile, the transmitter thread waits for serial input from the TelemetryUI to transmit via the RF channel. The incoming bytes are interpreted as telemetry messages with a `cmd` field specifying the command being sent to or received from the SCU. Messages are buffered in a FIFO queue to ensure proper transmission.

C. *TelemetryUI*

The user interface written in scala-swing is divided into three interface components: one presenting received CAN messages, one allowing the user to inject CAN messages into the solar car CAN bus, and finally one containing command buttons controlling the state of the SCU. A screenshot of the interface is shown in Fig. 10.

In the background of the program, multithreading is used to catch new messages received via the serial connection to the SVU, which then are broadcasted to UI elements and spawn a separate thread streaming data to the local network via a UDP port.


TelemetryUI

Send CAN Frame

id	data
0x22	0xdeadbeef

Last Received CAN Frame

time	id	len	data
07:57:12	c3	2	aafd

✓ hex

Fig. 10. A screenshot of the GUI.

VIII. Testing and Validation

As mentioned in the methodology section, we spent time thoroughly testing and validating our code design. Software validation was performed concurrently with writing our code to make sure that the actual behavior of the program matched the expected behavior. We used this to validate our code design with unit tests, testing both edge cases and random cases.

Doing this provided us a test bench for our project, which greatly improved the credibility of our work, proving that our design works in general cases and not just the case we represent in our report.

Once all the subsystems' functionalities had been validated, testing of the final product began. Initially, we were interested in testing the efficacy of our transmission protocol by sending and receiving asynchronous messages between the SCU and SVU. Once this was demonstrated, a range test was performed in a suitable outdoor environment. In the end, we achieved a transmission range of around 160 m before the connection became too unreliable [18].

IX. Discussion

A. Range limitation

The measured operating range of the two RF modules turned out to be well short of the 400 m that was targeted, let alone the 1000 m claimed by the manufacturer. While this was disappointing, there are several indicators for potential range improvement.

First of all, the hardware setup could be significantly upgraded to ensure a stable power supply to both RF modules, which seemed to be the main error source. A PCB implementation would certainly have been a possibility, given its energy-efficiency, but due to the project time frame this was not achievable.

Secondly, time constraints prevented us in testing all of the RF module's many software configurations to enhance the range. To this end, we have explored some configurations, such as the power-amplifier level, but our RF modules mainly use the default settings.

B. Throughput limitation

When testing the throughput capabilities of the system, it was found that the library code used to interface with the RF modules took up to $1200\,\mu s$ to send one message, taking around $200\,000$ clock cycles on the Teensy 3.6.

Combining the execution time spent on receiving CAN messages and writing to the black box, the SCU spends approximately $2\,\mathrm{ms}$ on each received CAN message, giving the system a throughput of $500\,\mathrm{s}^{-1}$ messages, thus falling short of the theoretical $2659\,\mathrm{s}^{-1}$ messages the CAN bus can handle at peak usage by a large margin.

This shows that there is a lot of performance potential to be gained in optimizing the library code by narrowing down its functionality and directly working with interrupts. Additionally, the message protocol used internally can be optimized to reduce the number of bits sent per message, at the cost of increasing the processing workload.

One goal was to allow for automated control from the support vehicle. As for now, the CAN data stream is only broadcasted via UDP, but a UDP client can not send CAN messages the other way. Since a two way communication is already used for very basic server-client handshaking, this functionality can be integrated into the TelemetryUI relatively easily.

X. Conclusion

The project set out to create a telemetry module for the ROAST solar car, and is successful in most of the points listed in the problem statement. The solar car module can read and write data or commands to and from the CAN bus. The solution for the processing of data in the support vehicle uses both a GUI and streaming of data directly to Matlab. The issue of storing the data locally was solved using a SD card in the Teensy 3.6 as a black box. In addition, a feature for securely encrypting and decrypting the message was implemented.

The only features for which the final product does not meet the specification are the maximum reliable transmission distance and the maximum achievable system throughput. These issues need further enhancement in the hardware and software, which can be the basis for future projects.

However, considering all of the above, the final product still stands as a satisfactory solution to the issues listed in the problem statement.

Acknowledgments

The authors would like to thank Martin Schoeberl and Christian Kampp Kruse for their mentoring and assistance throughout the project. The authors would also like to thank Claus Suldrup Nielsen and the rest of the DTU Roadrunners team for their outstanding cooperation throughout the project.

References

[1] *Bridgestone world solar challenge*. [Online]. Available: https://worldsolarchallenge.org/.

[2] *Can message frame*. [Online]. Available: https://copperhilltech.com/blog/controller-area-network-can-bus-message-frame-architecture/.

[3] "Ds/iso 11898-1:2015, road vehicles - controller area network (can) - part 1: Data link layer and physical signalling," in. Dansk Standard, 2016, vol. 1. edition, ch. 10.8.

[4] *Can bus explained - a simple intro 2021*. [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en.

[5] *Nrf24 transmission protocol*. [Online]. Available: https://lastminuteengineers.com/nrf24l01-arduino-wireless-communication/.

[6] *Platformio homepage*. [Online]. Available: https://platformio.org/.

[7] *Chibios homepage*. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php.

[8] *Scala-swing github repository*. [Online]. Available: https://github.com/scala/scala-swing.

[9] *Scala language homepage*. [Online]. Available: https://www.scala-lang.org/.

[10] *Acan library documentation*. [Online]. Available: https://github.com/pierremolinaro/acan.

[11] *Nrf24 osi library repository*. [Online]. Available: https://github.com/nRF24/RF24.

[12] *Sdfat repository*. [Online]. Available: https://github.com/greiman/SdFat.

[13] *Rsa algorithm implementation*. [Online]. Available: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/.

[14] *Tabel for prime number combinations for encryption*. [Online]. Available: https://bitbucket.org/dtucar/ecocar-solar/src/master/Telemetry/prime_tabel.png.

[15] *Json homepage*. [Online]. Available: https://www.json.org/json-en.html.

[16] *Chibios priority management*. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:books:rt:kernel_threading#priority_management_api.

[17] *Chibios counting semaphores*. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:books:rt:kernel_counter_semaphores.

[18] *Short video with system range test*. [Online]. Available: https://youtu.be/-iwST3REn40.

DTU Electrical Engineering
Department of Electrical Engineering

31015 Introductory project

s186083 Tjark Petersen
s194006 Steffan Martin Kunoy
s194027 Victor Alexander Hansen

DTU Electro	Spring 2021	Group: 7, ID: MEK2
Course 31015	Title	Group members
Introductory project - Electrotechnology	ROAST Telemetry	s186083 Tjark Petersen s194006 Steffan Martin Kunoy s194027 Victor Alexander Hansen
Document:	Project plan	4 pages
Version/Status:	2. edition	June 21, 2021

1 Problem introduction

This project concerns a telemetry module for use by the DTU Roadrunners Solar Team (ROAST). In 2023, ROAST is set to participate in the Bridgestone World Solar Challenge, a race spanning over 3000 km across Australia from Darwin in Northern Territory to Adelaide in South Australia. The solar car have limited opportunities for recharging during the race, so the main source of energy during the race, is from solar energy. To this end, solar panels will cover the car, however, the solar car is only allowed to have a maximum of 4 square meters of solar panels. This emphasizes the need to create an energy-efficient solar car to win the race [1].

Throughout the race the solar car will be monitored by a support car, which can monitor the car's condition and issue commands. This is a crucial function as the solar car will be navigating Australia's busy highways, where the heat and tire pressure may cause a detriment to the vehicle. The support car will therefore be able to analyze and react to the data stream being transmitted from the solar car, even when the driver is preoccupied by driving the car and navigating traffic.

The main function of the telemetry module is to read data from the solar car's on-board CAN (Controller Area Network) bus and transmit it via an RF-transceiver to the support car, while simultaneously logging data locally in a ”black box” memory chip. The support car is equipped with a similar module such that the support crew can react to the data manually or automatically by sending commands back to the CAN bus. As a consequence, the telemetry module plays a vital role in the communication between both vehicles, considering the distance between them may be up to 1 km depending on traffic conditions.

Therefore, the telemetry module must both serve the purpose of fulfilling the necessary specifications for communication, while also being as energy-efficient as possible. The project will mainly focus on the former that is, ensuring that the module meets the specifications with a solid solution.

2 Problem description

Remote sensing of data from the solar car will give the ROAST team a competitive advantage as the support vehicle will take on a more active role in controlling and optimising the car's performance. This will ease the burden on the driver and engage a large part of the supporting crew. For these reasons we have chosen to focus our project on the following points: Remote sensing of data from the solar car will give the ROAST team a competitive advantage as the support vehicle will be able to take on a more active role in managing and optimizing the car's performance. This will ease the burden on the driver and leverage the expertise of the supporting crew. For these reasons we have chosen to focus our project on the following guiding questions:

- How can we design a module that can read and write sensor data and commands from a CAN bus network?
- How can CAN data be transmitted securely over a distance of up to one kilometer?
- How does the unit sitting in the support vehicle process data upon receiving it from the solar car?
- What is the best method for storing data locally (black box) while being accessible at a later time?

- How can the CAN data be made accessible in the support vehicle for further processing e.g. in Matlab?

3 Problem scope

As mentioned in the introduction, the car will be using the CAN standard for distributing data, which scopes how we gather data and interact with the car significantly. A solution therefore requires an interface between the CAN bus and the microcontroller.

There are multiple solutions to the problem at hand that were discussed, among them implementing a whole solution on a PCB or more modular solutions involving prebuild microcontrollers like the Teensy[3]. We chose to consider prebuild microcontroller based solutions and to not focus on implementing the final product on a PCB. The decision was based on us wanting to design a product which is modular and flexible and thus can be altered and improved during its life time.

A PCB implementation is expected to achieve a higher efficiency due to reduced current draw compared to the implementations we are considering. We prioritized a more modular and flexible product over the potential power savings though since the difference in current draw is expected to be of the order of some tenths of milli-Ampère.

4 Target audience

The target audience for our project is the DTU ROAST team, who will integrate the telemetry module in the solar car and support vehicle. Subsequently, the car is set to enter the 2023 Bridgestone World Solar Challenge.

5 Problem solution

1. Preliminary design of telemetry module in block diagram form. Establish specifications and requirements for the final product.
2. Select components for preliminary design. Ordering components not available from DTU.
3. Prototyping telemetry module on two breadboards. Write code to test limited functionality.
4. Improve on prototype. Order new components if needed. Optimize code and write more extensive tests.
5. Improved prototype of telemetry module on breadboard. Repeat Steps 3. and 4. until fundamental requirements in Step 1. are satisfied.
6. Develop more permanent solution, i.e. compact, soldered and configurable. Complete tests in Step 5.
7. Design GUI for interaction with PC. Test with USB connection to support-car module. Remote control via PC.
8. Present solution to DTU ROAST team. Discuss improvements and additional features.
9. Integrate telemetry module with solar car. Ongoing testing of solution. Feedback and evaluate project.

6 Resources

- 1x Teensy 4.0 development board

- 1x Teensy 3.6 development board
- 2x nRF24L01 + PA/LNA Wireless Transceiver with antenna
- 1x Micro SD memory card (32GB)
- 2x MCP2551 CAN Transceiver w/ breakout board

At this stage, our telemetry modules, one for both the solar and support car, will consist of the components above. The size of the memory SD card is uncertain at this point, but we will use 32GB for testing purposes.

In addition to the hardware ressources needed for this project, the DTU ROAST team provides a great source of insight for the project. Reports from different teams and last years telemetry project lies available, which helps laying the groundwork for our project [2].

7 Activity Plan

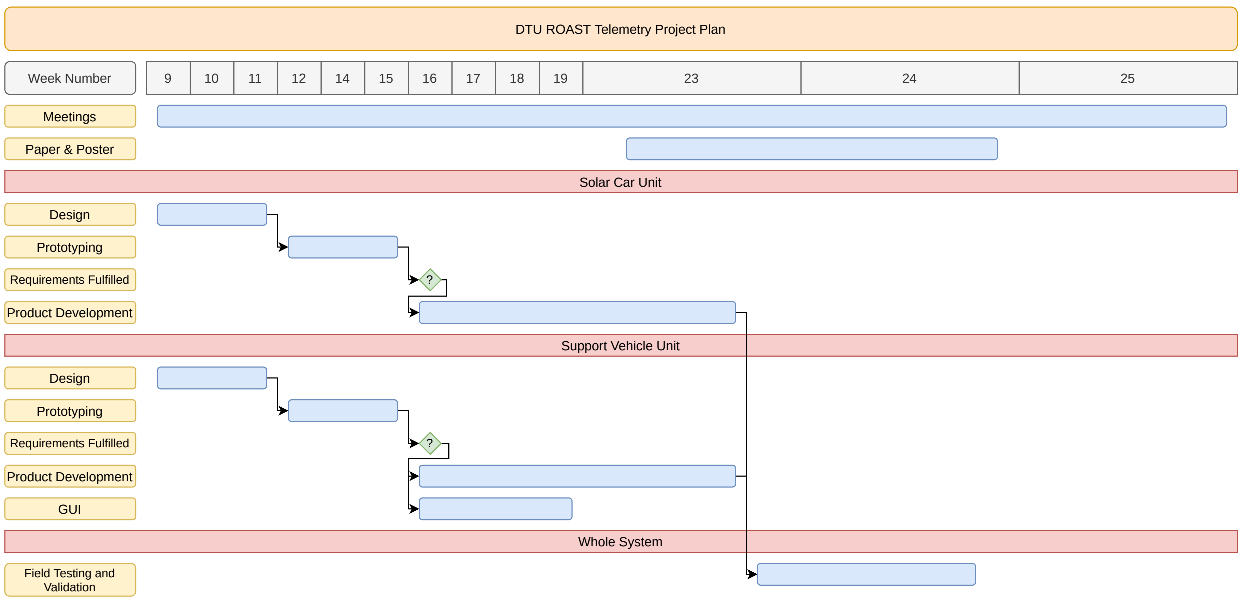


Figure 1: The project activity plan

Meetings

We will have a weekly meeting with both the project coordinator Christian Kampp Kruuse and our supervisor Martin Schoeberl on Fridays. In addition, meetings will be arranged with our second supervisor Vitaliy Zhurbenko if necessary.

Paper writing

We will begin drafting a project paper once we have achieved an MVP (Minimum Viable Product). A Github repository will be updated continuously as a full documentation of the project.

Design

The design phase of the project will include designing prototypes to be implemented once the hardware components become available. Furthermore, basic software programs may be written and debugged during the design phase. We intend for all design, prototyping and testing of the Solar Car Unit (SC) and Support Vehicle Unit (SV) to run in parallel.

Prototyping

During prototyping we want to test the basic functionality of our product. The setup should be configurable, which is why we plan to use breadboards and jumper wires. Basic tests will be written in software to make sure the product meets the basic requirements.

Requirements fulfilled

Once the basic requirements have been fulfilled by the prototype we will begin rigorously testing to find ways to further optimise our design. This is to ensure that the product will not have any fundamental flaws.

Product Development

This stage will concern the design of an MVP as well as later design iterations.

GUI

As a final touch we intend to create a basic GUI allowing PC users to interact with the telemetry module, as well as sending and receiving data on the CAN bus.

Field Testing and Validation

Finally, our product will be integrated with the ROAST solar car and support vehicle, allowing us to test it in operation. We will be able to gain feedback from the DTU ROAST team and evaluate the project.

8 System Specification

In the following three subsections feature lists for each of the three systems are presented.

8.1 Solar Car Unit (SC)

Feature ID	**Description**
SC.SD	The system uses a SD card as a permanent storage of log files
SC.SD.write	Write a data structure to a file on the SD card
SC.SD.read	Read a file into a data structure
SC.RF	The system needs to communicate via a RF module with the support vehicle
SC.RF.rxCmd	Receive a system configuration command from the support vehicle
SC.RF.rxCanCmd	Receive a message to be broadcasted via the solar car CAN bus
SC.RF.txResp	Transmit response to a received command
SC.RF.txLogUnit	Transmit a log unit to the support vehicle
SC.RF.txLogFile	Transmit a log file to the support vehicle
SC.CAN	The system needs to communicate via the solar car CAN bus
SC.CAN.rx	Receive all messages broadcasted on the CAN bus
SC.CAN.tx	Transmit a message on the CAN bus
SC.BUF	The system needs to buffer logged data in local flash memory until enough data to fill a file is collected
SC.BUF.add	Add a log unit to the buffer
SC.BUF.rd	Read a logging unit from the buffer into RAM
SC.BUF.rdToFile	Read all logged data from the buffer into a file
SC.BUF.clr	Empty the buffer

8.2 Support Vehicle Unit (SV)

Feature ID	**Description**
SV.RF	The system needs to communicate via RF with the solar car
SV.RF.txCmd	Transmit a system configuration command to the solar car
SV.RF.txCanCmd	Transmit a CAN message to the solar car
SV.RF.rxResp	Receive a response from the solar car unit
SV.RF.rxLogUnit	Receive a log unit from the solar car unit
SV.RF.rxLogFile	Receive a log file from the solar car unit
SV.UART	The system needs to communicate with a computer via UART
SV.UART.rxCmd	Receive a command via UART
SV.UART.rxCanCmd	Receive a CAN bus message via UART
SV.UART.txLogUnit	Transmit a log unit to the computer
SV.UART.txLogFile	Transmit a log file to the computer

8.3 Support Vehicle Computer Program (PR)

Feature ID	**Description**
PR.UART	The program needs to communicate with the support vehicle unit via UART
PR.UART.txCmd	Transmit a system configuration command via UART
PR.UART.txCanCmd	Transmit a CAN message via UART
PR.UART.rxLogUnit	Receive a log unit via UART
PR.UART.rxLogFile	Receive a log file via UART
PR.FILE	The program needs to output received log files
PR.FILE.write	Write a file containing the last received log file
PR.GUI	The program is interface via a GUI
PR.GUI.getConf	Get configuration parameters from GUI
PR.GUI.getCanCmd	Get a CAN message from the GUI
PR.GUI.monitor	Show incoming log data stream
PR.GUI.saveLog	Pass log file to file writer

9 References

[1] Bridgestone World Solar Challenge, URL: https://worldsolarchallenge.org/

[2] Reports and source files provided by DTU ROAST

[3] Teensy microcontroller, URL: https://www.pjrc.com/teensy/