



E-Kart

15.04.2018

Alex Lardschneider

TFO "Max Valier" Bolzano
5IA Information Technology

Tutor: Wild Michael
In collaboration with: Galliani Alex & Moroder Johannes

Table of contents

Table of contents	1
Project Overview	4
Goals	4
Motivation	4
Project timeline	6
06.11.2017 - Abstract submission	6
Middle of November	6
29.01.2018 - 02.02.2018 - Project week	6
Late February	6
Early March	6
Late March	6
Late April	6
Early May	6
02.05.2018 - Project submission	6
11.05.2018 - 12.05.2018 - Project day	6
Independent Software Components	7
Microcontroller 1 (Teensy 3.5)	7
Microcontroller 2 (ESP8266 based Wemos D1)	7
iOS Application	8
macOS Application	8
Theoretical background	9
Arduino	9
Hardware Design	9
Software Design	9
Basic Arduino Sketch	10
Teensy	10
ESP8266	10
iOS	11
MFi Program	11
macOS	11
Swift	12
I2C	12

SPI	12
Serial Communication	13
CAN-BUS	13
PlatformIO	13
GPS, GLONASS, and NMEA	14
Used Hardware	14
Hardware choices	14
Teensy 3.5	15
Wemos D1	15
SparkFun Venus GPS plus SMA Antenna	15
DS18B20 Temperature Sensors	15
SparkFun MPU6050 Accelerometer and Gyro	16
TMP006 Infrared Temperature Sensor	16
SparkFun BlueSMiRF Bluetooth	16
SparkFun CAN-BUS shield	16
Cable connections	17
Prototyping and PCB	17
Main Microcontroller Software Design (Teensy 3.5)	18
Tools	18
Dependencies	18
Structure	20
Message Relay Software Design (Wemos D1)	21
Tools	21
Dependencies	21
Structure	21
iOS App Design	22
Tools	22
Structure	22
Dependencies	23
Navigation	25
App layout	25
Connection Flow	26
Data logging	27
File format	28
Data transmission	29
Data format	29

Checksum	30
Data evaluation	31
macOS data converter	31
PostgreSQL database	32
Testing and validation	32
Testing on the Arduino platform	32
iOS Testing	33
Tweaks and improvements	34
Experience and closing	34
Where to go from here	35
Sources	36

Project Overview

E-Kart is an attempt to build a fully electric race go-kart, using latest battery, engine, and powertrain components. This go-kart should be noise- and emissionless and still be more powerful than a traditional internal combustion engine (ICE) powered equivalent. My part of the project is to build the engine control system (ECU) to control and monitor mission-critical components and an iOS app to display a few parameters and track info on an iPhone, which will be put into the steering wheel.

The iOS app should be able to display important parameters like current speed, engine, brake, and drivetrain temperatures and log these parameters for further evaluation later on.

Goals

1. Build a fully electrical go-kart using the latest available technology
2. Make it as fast, powerful and lightweight as possible
3. Make it technologically advanced using many computers
4. Prove that electric vehicles can and will beat gasoline powered ones.
5. Try to beat the acceleration record for electric go-karts

Motivation

Traditional ICE powered go-karts contain a lot of parts, and in many cases wear is going to be an issue. In fact, a traditional go-kart needs to have its pistons and transmission changed every 40 hours. That is going to get expensive very fast. An electric go-kart doesn't need a transmission, and the electric motor is very simple, thus we can overcome the 40-hour limit. From a technological standpoint, many go-karts are also kept very simple. Using many microcontrollers and a smartphone we want to build the most advanced go-kart possible, and if possible beat the current 0-100 km/h acceleration record.

My motivation for this projects comes from my passion to work with a combination of hardware and software components. I also really like fast and powerful cars and a go-kart is the perfect fit for this since it is equipped with a 130kW engine, normally found on way bigger cars.

Since joining this school, I started working with Arduino clones bought from eBay in my free time and started some smaller projects like a weather station on my mountain hut or a

small IoT infrastructure at home, consisting of a mesh network of ESP8266 microcontrollers. I like discovering solutions to difficult problems and finding the most efficient way to solve them and this project strengthened this quality. Building this go-kart has been a real challenge for me and I acquired a lot of valuable knowledge working with so many different components.

Below you can find some pictures of the finished project presented at the “Projekttag” on 11.05.2018 and 12.05.2018.



Project timeline

- I. 06.11.2017 - Abstract submission
The abstract needs to be submitted to the tutor.
- II. Middle of November
Arrival of go-kart chassis and electrical components for the powertrain.
- III. 29.01.2018 - 02.02.2018 - Project week
A full week available to work on the project.
- IV. Late February
Delivery of electrical components.
- V. Early March
First working prototype of the iOS app.
- VI. Late March
First working prototype of the go-kart software
- VII. Late April
Project description submission (this document).
- VIII. Early May
Arrival of the chassis cover made out of compound materials.
- IX. 02.05.2018 - Project submission
Submission of the project.
- X. 11.05.2018 - 12.05.2018 - Project day
Two days available to present the project to outside spectators.

Independent Software Components

The different software parts powering this project will be split into 4 different completely independent parts. This is done mostly because of technical limitations like different platforms and programming languages, and for logical reasons.

The microcontrollers on the go-kart are powered by two separate software components, one iOS app powers the user interface mounted on the steering wheel and a macOS application evaluates telemetry data collected by the microcontroller.

Microcontroller 1 (Teensy 3.5)

This microcontroller is the brain of the project. It interfaces with the various sensors mounted on the go-kart, transforms and processes this data, sends it to the relay microcontroller and saves it to the SD card for eventual evaluation later on.

A Teensy 3.5 microcontroller, based on the Arduino platform, powers this software. It is a breadboard-friendly development board with loads of features. Powered by a 120 MHz ARM-Cortex based CPU with floating point unit, it features sufficient digital inputs needed to realize this project (I2C, SPI, Serial).

Microcontroller 2 (ESP8266 based Wemos D1)

The first iterations of this project featured a Bluetooth connection between the Teensy 3.5 and the iOS application. Further research revealed that this is not feasible without a special partnership with Apple, the so-called Apple MFi program and the ordered Bluetooth shield, which does not feature Bluetooth Low Energy.

Ordering a compatible Bluetooth shield was not possible without exceeding the deadline, so I decided on using a Wemos D1 microcontroller I had lying around at home.

The ESP8266 based Wemos D1 is a fully fledged ARM microcontroller, also compatible with the Arduino platform. It features a 2.4 GHz Wifi antenna and a fully implemented TCP/IP stack. Current iterations of this project feature this microcontroller as a message relay, forwarding data packets from the main microcontroller to the client and vice-versa via WiFi. The relay automatically advertises a WiFi access point to remain connected during driving.

iOS Application

The iOS application, written in Swift and Objective C (for low-level networking and sockets), displays transmitted telemetry data on a pager based layout.

The main screen, visible during driving, consists of three different page based views. The first view displays a map with the current vehicle position centered on the map. It automatically follows vehicle movement when a new data set arrives.

The second view, which is visible by default when you connect to the go-kart, displays important telemetry data and the vehicle speed at the top. Parameters are highlighted in different colors, to sign warning or critical data with yellow or red colors respectively.

The third and last view displays more telemetry data, which is not crucial during driving. It also allows for some basic vehicle configuration.

Two buttons mounted on the steering wheel allow the driver to quickly switch between the three pages.

When first opening the application, the user is presented with a welcome screen, where the user can connect to the WiFi access point and the vehicle. Errors during the connection are displayed with dialogs.

macOS Application

A small macOS application, written in Swift, converts the telemetry data stored by the main microcontroller and stores it in a PostgreSQL database (or any other database depending on the user's availability or preference). This allows for data statistics using conventional SQL function aggregators like SUM, MIN or MAX.

Theoretical background

Arduino

The Arduino platform is a open source hardware and software platform. It designs popular single-chip microcontrollers and kits (DIY) to allow cheap robotics prototyping. The board features various analog and digital pins and supports the most common protocols used in robotics, I2C, SPI and one or more hardware serial interfaces. Almost all boards feature a USB interface which is used to load the bootloader and software.

The Arduino platform also provides its own IDE, based on the popular Processing language project.

Originally founded in Ivrea, Italy in the year 2003, Arduino still provides hardware schematics and software under the GNU General Public License.

The most common Arduino board is the Arduino Uno, originally released as Arduino RS232.

Hardware Design

Most Arduino boards consist of a single CPU, based on an Atmel 8-bit AVR microcontroller (most common one used is the ATmega328). In 2012, a 32-bit based Arduino, the Arduino Due was released. Atmel designs and licenses the ARM CPU architecture used by AVR.

All Arduino boards come preloaded with a custom bootloader that simplifies uploading of programs to the built-in flash memory. Current Arduino models are programmed using the USB interface, which is using RS232 communication. Original Arduino models feature an FTDI USB-to-serial adapter, the cheaper China copies, including the one I own, are fitted with the cheaper CH340G USB-to-serial converter.

Most of the ATmegas CPU pins are exposed via female 0.1-inch pin headers for external development. Several plug-in application shields are also available.

All Arduino boards feature a built-in LED, most commonly available on the digital I/O pin 13. It can be referenced in software using the macro `LED_BUILTIN`.

Software Design

Arduino can be programmed using every available programming language, which produces binary code compatible to the AVR architecture. Atmel provides its own IDE and compiler suite for AVR programming.

Arduino programs are called sketch and are saved as .ino files. A minimal sketch consists of two main methods, called `setup` and `loop`. Like in Processing, the `setup` method is called

once at boot. The loop method, however, runs, as the name indicates, in a loop until the microcontroller loses power or experiences a software crash.

Basic Arduino Sketch

Below you can find a minimal Arduino sketch, which continuously flashes the built-in LED in one-second intervals.

```
void setup() {  
    pinMode(LED_PIN, OUTPUT);    // Configure pin 13 to be a digital output.  
}  
  
void loop() {  
    digitalWrite(LED_PIN, HIGH); // Turn on the LED.  
    delay(1000);                 // Wait 1 second (1000 milliseconds).  
    digitalWrite(LED_PIN, LOW);  // Turn off the LED.  
    delay(1000);                 // Wait 1 second.  
}
```

Teensy

The Teensy development board is a third party Arduino compatible development board. It features more powerful CPUs and I/O pins. Compared to the Arduino boards, Teensy fully supports all USB hardware types, allowing it to emulate keyboards (USB-HID) or MIDI devices. Next to I2C, Teensy also features an I2S interface.

Teensy 3.5, the microcontroller is based on a 120 Mhz 32-bit ARM Cortex CPU design with 512 kB of flash memory. It is a downscaled version of the Teensy 3.6 offered at a cheaper price, but features 5V logic level resistant I/O ports, whereas the Teensy 3.6 is only 3.3V compatible.

ESP8266

ESP8266 is a low cost, WiFi-equipped Arduino based microcontroller. It features a full TCP/IP stack, making it an attractive option for IoT devices. The ESP8266 features a RISC based 32-bit ARM CPU with up to 16 MiB of flash memory.

The Wemos D1 is a breakout board which makes connecting the ESP8266 to standard 0.1 inch (2.54 mm) easy.

iOS

iOS (originally called iPhone OS) is a mobile operating system developed and distributed by Apple exclusively for its own mobile phone lineup. It was originally released in 2007 and is based on the 32-bit ARM architecture. Newer models (devices equipped with Apple A7 or later) are fully 64-bit based. The latest iteration of iOS - iOS 11 - is only available on 64-bit hardware. It is written in C, C++, Objective C and later iterations partly in Swift.

iOS software can be developed with every programming language which compiles to ARM bytecode. Deployment to devices is only allowed through Xcode and its proprietary code signing software, although unofficial sideload methods exist (Cydia Impactor).

The original unveiling of iOS also featured the first iteration of the iOS SDK (Software Development Kit) without support for third-party applications. The SDK is free to use on Apple hardware without support for Windows or Linux, although Apple officially supports Swift (Apple's own iOS and macOS programming language) development on Linux.

The latest available release is iOS 11.

MFi Program

Apple's MFi Program is a licensing program for certain Apple hardware and software. It covers various hardware interfaces including its corresponding software interfaces, like the headphone jack, the lightning connector or the Bluetooth interface. Apple currently charges 100\$ a year to be MFi partner.

macOS

macOS, originally called OS X, is a desktop operating system developed and distributed by Apple exclusively for its Mac Desktop lineup. It is based on technologies originally developed at NeXT, a company that Steve Jobs created after being fired from Apple. NeXT was later acquired by Apple, promoting Steve Jobs to CEO.

First releases of macOS, back then called OS 9 and OS X (the X referring to the roman numeral for 10) only ran on PowerPC based systems. In 2006, Apple announced that they were switching to Intel CPUs and the x86-64 architecture. OS X 10.4 Tiger was the first release to only run on x86.

At the 2016 WWDC OS X was officially renamed to macOS to be more consistent with the other operating system names (iOS, tvOS, and watchOS). The currently latest available version is macOS 10.13 High Sierra.

Swift

Swift is a general-purpose programming language developed by Apple for its platforms macOS, iOS, tvOS, and watchOS. It is based on the LLVM project, also developed by Apple and is designed to seamlessly interface with current Cocoa-based interfaces. Swift was first released at the 2013 WWDC (Worldwide Developer Conference) bundled with Xcode 6 in San Francisco and its latest available version is Swift 4.1.

It employs modern programming paradigms (Optionals and chaining) and compared to Objective-C a simpler syntax. It is commonly referred to as “Objective-C without the C”.

I2C

I2C is a multi-master multi-slave bus system developed by Phillips to connect multiple “slave” ICs to one or more “master” systems. It is only intended for short distance communication with a focus on reliability instead of speed. It only needs two wires, one for SDA (data signal) and SCL (clock signal).

I2C is an asynchronous interface, meaning that no clock data is transmitted and both devices need to agree on the same clock (or similar) beforehand. This makes the hardware requirements at both sides relatively complex. An I2C data message contains 10 bits, one start and end bit and 8 data bits. Each device connected to the I2C bus needs its own address (for example 0x680).

While there is no theoretical speed limitation, specification limits the fastest data transfer mode (High-Speed Mode) to 3.4Mbps.

SPI

SPI (Serial Peripheral Interface) is a data bus commonly used to send data to various hardware devices such as SD card readers, shift registers or sensors. It uses a separate data and clock line together with one device select line (Arduino’s `CHIP_SELECT`) to choose the device you wish to talk to.

Compared to I2C, SPI is a synchronous interface, which keeps the clock on both sides in sync and thus eliminates the need for a start and stop bit. The clock signal tells the device exactly when to sample bits on the data line. This also eliminates the need for complex hardware at the master and slave side.

Because the clock is sent to each device, there is no theoretical speed limit at which SPI can operate.

Serial Communication

Serial communication is a communication where the data bits are sent one bit at a time, sequentially over a wire. This contrasts with parallel communication where bits can be sent in parallel with interlinked wires.

Serial communication can be used for long-range data transfer, where parallel communication cannot be used due to timing constraints and increased cable cost. Like I2C, serial communication is asynchronous, where both sides need to agree on the same speed (baud rate) in order to get correct data.

The serial interface consists of two data wires, TX and RX. TX is used for data transmission, RX for receiving data. To send data between two devices, you need to cross over the connections, so that RX1 goes to TX2 and TX1 to RX2. The Teensy 3.5 used in this project features 5 completely independent hardware UARTs. Theoretically, every I/O pin on a microcontroller can be used as a serial interface using software at the cost of reliability.

CAN-BUS

A CAN bus (Controller Area Network) is a robust interface designed to allow microcontrollers to talk with each other without a host computer. It was originally developed for automobiles to interconnect many components but has since been used in many other contexts.

A CAN bus consists of two wires, CANH (CAN High) and CANL (CAN Low). Multiple devices can be connected to the same wires. Since the Arduino can not directly communicate via the CAN bus, a special IC has to be added in between to convert CAN data to SPI.

PlatformIO

PlatformIO is an open source ecosystem for IoT development. It is suited for embedded device development and has configuration files and toolchains available for the most popular microcontrollers. PlatformIO is written in Python and is cross-platform compatible and can be installed via Python's own package manager (pip). It can be invoked from the command line to create a project, build the code and deploy it to the microcontroller. Since the official Arduino IDE is very basic and lacks many (in my opinion) important features, I decided to use PlatformIO together with JetBrains' excellent CLion IDE.

GPS, GLONASS, and NMEA

GPS (Global Positioning System) is a satellite-based navigation system originally developed for the US military. After a plane crash, which could have been prevented had GPS been public at the time, the US government decided to open GPS for private use. GPS does not require user hardware to send data, but only receives signals broadcast from satellites. GPS needs a minimum of 4 satellites for trilateration to work (3 satellites would give us a maximum of 2 possible points on a sphere in the worst case, so sometimes 3 satellites are enough for basic navigation).

GPS provides location, direction, velocity and time information with an accuracy of down to 30 cm with high-end devices.

Since GPS is solely owned by the US military, Russia, China, and the EU decided to build its own global satellite-based positioning system, with the most famous one being Russia's GLONASS. The receiver used in this go-kart can use both GPS and GLONASS.

The receiver outputs the data in the NMEA format (National Marine Electronics Association). This format consists of multiple different sentences, where each sentence has its own name which is sent at the beginning of the message. Each message contains a different type of information for location, accuracy, satellite count or bearing. TinyGPSPlus is used on the microcontrollers to parse the NMEA data.

Used Hardware

Hardware choices

I chose most of the hardware based on experience with the various components. I used various Arduino microcontrollers for IoT projects at home and already knew how Arduino programming works. I had experience using the DS18B20 temperature sensors for an ongoing weather station project and they still work very reliably. A weather balloon project we built in the fourth term in school made me familiar with the GPS antenna and its interface. The rest of the components were chosen based on reliable manufacturers and use cases. Below you can find the hardware components I used for this project.

Teensy 3.5

Main microcontroller to which all sensors and analog inputs are attached. Logs telemetry data and sends it to the message relay. It also houses the microSD card, where all sensor data is logged.

Wemos D1

Message relay which acts as a WiFi Access Point the iOS Application can connect to. Receives messages from the main microcontroller via a serial interface and transmits it via a socket connection to the client.

SparkFun Venus GPS plus SMA Antenna

The latest version of the Sparkfun Venus GPS lineup features improved reliability in a smaller package. It is fitted with a standard SMA connector where we can attach an SMA antenna. The one we chose is a normal antenna with a 5m cable attached, so we can position it anywhere on the go-kart, furthest away from external interference like high current sources. The GPS connects data about latitude and longitude of the go-kart, and in case of CAN-BUS failure also speed and acceleration data (CAN-BUS is a software/hardware interface we use to read engine RPM data from the electric inverter).

The GPS sensor is connected to the serial nr. 5 interface on the main microcontroller.

DS18B20 Temperature Sensors

The go-kart is fitted with a total of 7 temperature sensor. I chose the DS18B20 lineup since I had experience working with it and it features a waterproof version. Two waterproof temperature sensors are mounted in the liquid cooling system, to measure coolant temperature before and after the radiator. Two sensors are mounted in between the battery packs on both sides to measure battery temperatures. Another sensor is mounted behind the radiator exhaust and one measures general outside air temperature. The last temperature sensor measures the temperature inside the electrical box that houses all hardware components. This sensor can measure temperatures from -25°C up to 125°C.

These sensors are connected via the Dallas (the manufacturer of the sensors) OneWire interface, meaning that we can connect all 7 sensors on only 3 wires (GND, Vcc, and Signal).

SparkFun MPU6050 Accelerometer and Gyro

This sensor measures acceleration and gyro data from up to 9 different axes. We use it to measure lateral acceleration in turns and acceleration or deceleration. The sensor is connected via I2C to the main microcontroller and is advertised on the address 0x68.

TMP006 Infrared Temperature Sensor

The TMP006 is an infrared temperature sensor to measure brake pad temperature. We need infrared sensors in these areas since we can't physically mount conventional sensors on rotating parts. Each sensor is advertised on its own I2C address (from 0x40 to 0x43) and has a measuring range from -25°C up to 125°C.

Two sensors are mounted on the brake pads in the front and one is mounted on the drivetrain brake pad in the back. A fourth one would have been used to measure asphalt temperature, but since it was DOA that plan was scrapped.

It has been retired by SparkFun shortly after I ordered it. No replacement model is available as of middle of April 2018.

SparkFun BlueSMiRF Bluetooth

This Bluetooth 4.0 sensor was originally planned to be used for data transmission between the main microcontroller and the iOS application. Since Apple's MFi program does not allow interfacing with non-BLE sensors, that plan had to be scrapped and this sensor has been replaced with the Wemos D1 WiFi chip. The sensor can be interfaced using a serial interface and supports data transmission rates up to 115200 baud.

SparkFun CAN-BUS shield

This CAN-BUS shield allows for interfacing with CAN connectors. It features a D-SUB 9-pin connector and can be connected to the electrical inverter using a custom DB9 to OBD-II cable. This shield allows us to read important inverter parameters, like engine RPM, coolant temperature, battery voltage, inverter temperature, engine temperature and power output.

This shield is originally planned to be mounted on an Arduino Uno and features pins in the correct location. It also contains connectors for a serial monitor, a GPS antenna and an SPI microSD card slot, but since we have better external hardware we will not use built-in connectors. We only need the SPI pins to interface with the MCP2515 CAN-to-SPI ASIC so we do not need to mount the board like a shield.

This shield can read CAN data up to 1 Mb/s at an SPI speed of 10 MHz.

Cable connections

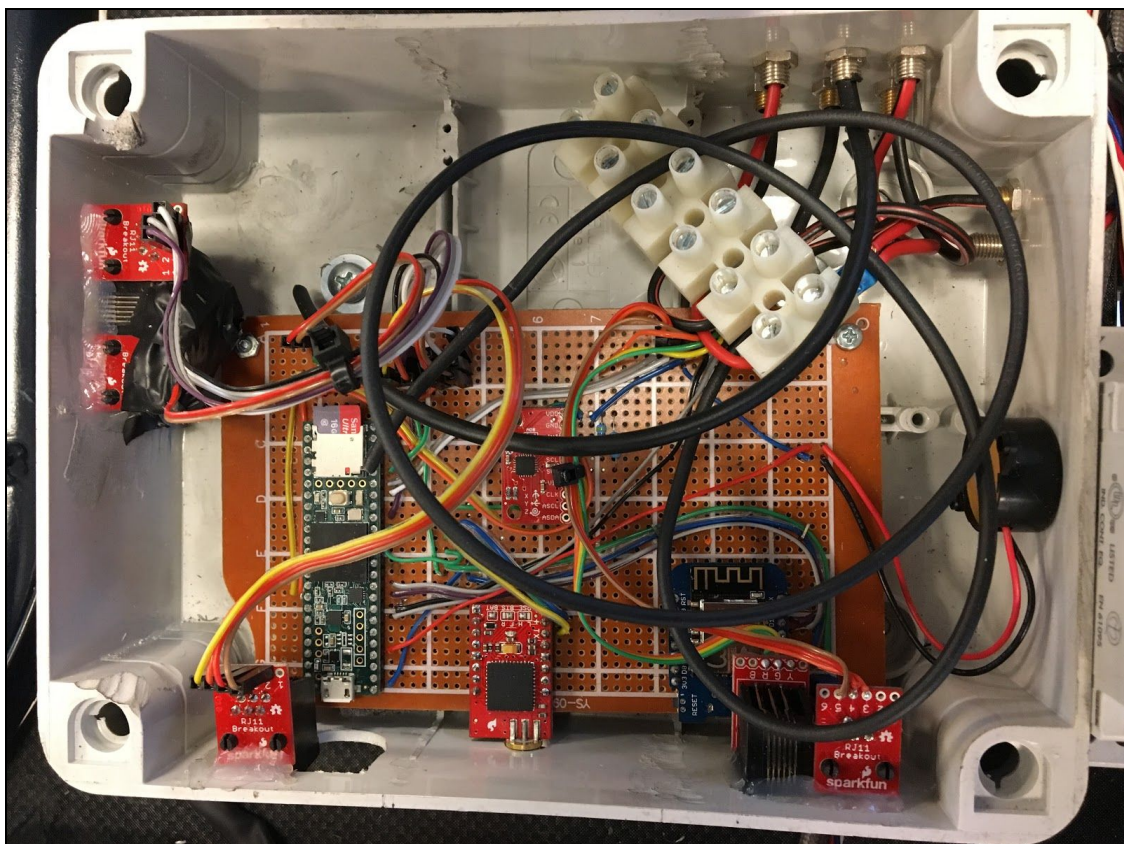
Since sensors need to be able to be connected and disconnected without effort at any time, all external sensors are attached to an RJ11 male connector. It features up to 6 wires making it the perfect choice for all the sensors I use. A female counterpart is soldered to the breadboard the main interface is attached to. RJ11 connectors have been ordered off SparkFun.

Prototyping and PCB

All the prototyping (testing and developing of hardware) is done on breadboards, which are a type of PCB where all rows (not columns) are interconnected. Components can be pressed into the board which allows for fast development since no soldering is required.

For actual use inside the go-kart, a breadboard is not an optimal choice since the solderless connections can become loose during driving due to vibrations. To prevent that from happening, all components will be soldered to a standard 0.1 inch PCB.

Below is a picture of the finished PCB, mounted together with all the electrical components in a waterproof plastic box.



Main Microcontroller Software Design (Teensy 3.5)

Tools

The software for the main microcontroller is programmed using the Arduino SDK. Since the official toolchain is limited to Arduino's own IDE, I decided to use the PlatformIO toolchain so that I can use the JetBrains CLion IDE, which I prefer and it offers much more advanced features and code smell checks. The PlatformIO toolchain can be installed on macOS, Windows or Linux and is built on python. Installation on macOS works using brew, pip or easy_install. It can be invoked from the command line to automatically create a compatible project structure for the required IDE:

```
platformio init --board teensy35 --ide clion
```

This command creates the project structure compatible with JetBrains IDE and includes the required toolchain. Deploying works as follows:

```
platformio run -t upload
```

This command automatically compiles the user-written code, any additional libraries, and the Arduino SDK into a platform compatible executable binary. PlatformIO then discovers the microcontrollers port (/dev/wchusb14*** on macOS) and uploads the code. At the end, the microcontroller is restarted and the program starts running. A serial monitor can be attached if needed.

Dependencies

This project depends on some external software libraries to help with interfacing with sensors. The dependencies I used were mostly recommended by the vendor of the sensors. Adding external code is easy. PlatformIO automatically creates a `libs` folder, where libraries in the Arduino library format can be placed. They are automatically included during compilation.

Currently, the following libraries are used for the main microcontroller:

- **TinyGPSPlus**

Handles encoding and converting the GPS NMEA sentences coming from the GPS receiver via the serial interface. It allows accessing properties via C++ structs and properties. This library is a fork from the original TinyGPS library, which did not expose properties via nested C structs and didn't properly read the HDOP property.

- **SdFat**

Handles access to the SD card and file reads/writes. The version I use is a special fork from the original SdFat project, since Teensy 3.5 connects its SD slot to a special SPI interface, without the need of the CHIP_SELECT pin and command. The forked version automatically uses the newer SPI interface. Newer versions of this library also allow the opening of multiple files at once, which is needed for this project since different sensor data is logged in separate files.

- **TMP006**

Interfaces with the TMP006 infrared temperature sensors. The library allows for querying temperature data from a remote surface using the IR sensor and the local sensor temperature. The library uses the Wire interface internally, which is Arduino's way to interface with I2C sensors. Normally each sensor has the same I2C address, but since we are using multiple of the same sensors we needed to change the addresses by pulling ADDR0 and ADDR1 on the sensor to GND or to Vcc

- **DS18B20**

Interfaces with the DS18B20 temperature sensors. Supports Fahrenheit, Kelvin, and Celsius. Each sensor is addressed using 10 hexadecimal values and the library supports multiple sensors.

It depends on the OneWire library for communication with the sensors. Usually, it is included with the Arduino SDK but it needs to be included manually if any other toolchain is used.

- **CAN-BUS library**

The SparkFun CAN-BUS shield ships with its own Arduino compatible library. Without modification, the library supports reading of engine RPM, engine coolant temperature, vehicle speed, MAF sensor (mass flow sensor, not used in our case since the engine is electrical) and O2 sensor (also not used for the same reason).

Custom values provided by our engine controller can be read by supplying the library with a hexadecimal address value (for example 0x05).

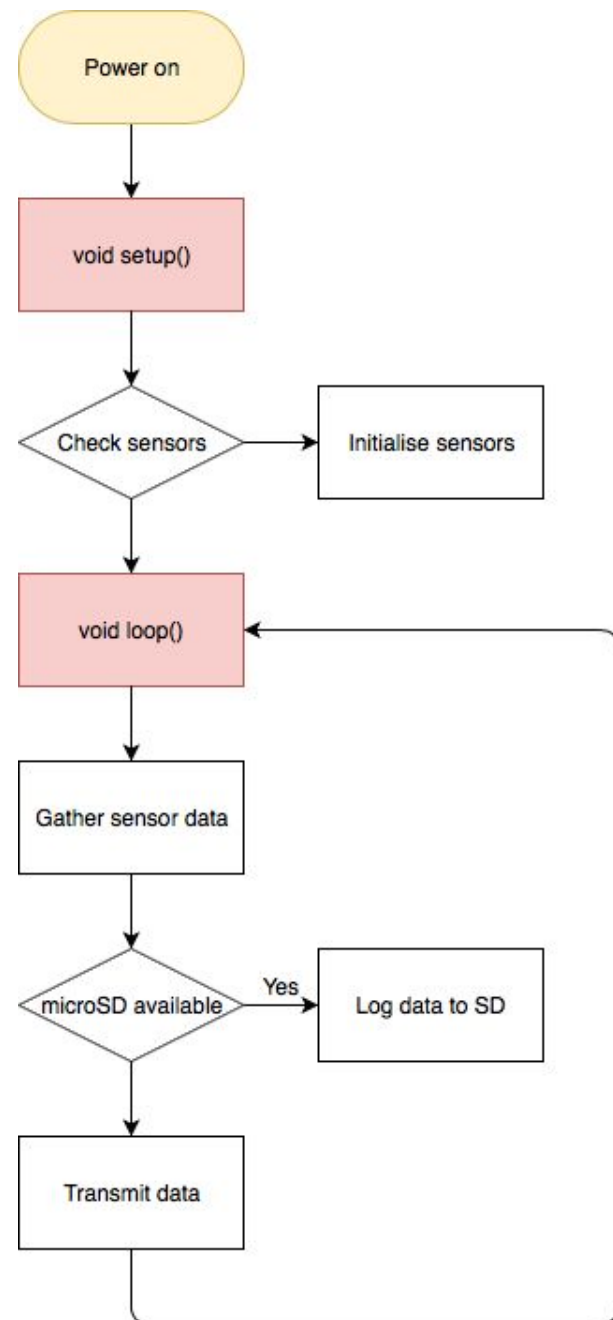
Structure

The project is structured like a normal Arduino project, with the difference of using .cpp and .h file extension instead of the Arduino IDE .ino extension. A main.cpp class handles the setup and void methods. The code for each different sensor is located in a separate directory inside the src folder. Each sensor type has its own source and header file.

Each sensor file has an init method, where code that needs to run at boot gets executed (for example the OneWire interface is initialized, the existence of the microSD card is checked, baud rate for the serial interfaces is set and so on).

Another method reads the data from the sensor and saves it into a global GoKart struct, where all data is collected. This struct persists over the lifetime of the program, so it always contains the latest collected data, which the message sender code can then transmit.

The nearby flow charts shows how the microcontroller software works. On boot, the Teensy 3.5 first starts the CPU and sets relevant registers and executes the setup method. The setup method calls all init methods of sensors and other hardware. This method only runs once at boot. The loop method is executed immediately after and runs in a loop until the microcontroller crashes or loses power. It continuously collects sensor data, writes it to the microSD card if available and transmits it to the message relay.



Message Relay Software Design (Wemos D1)

Tools

For the message relay, the same PlatformIO toolchain is used, with the only difference being another board definition when using the init command:

```
platformio init --board d1_mini --ide clion
```

This command initializes the project as already described in section Microcontroller Software Design # Tools above.

Dependencies

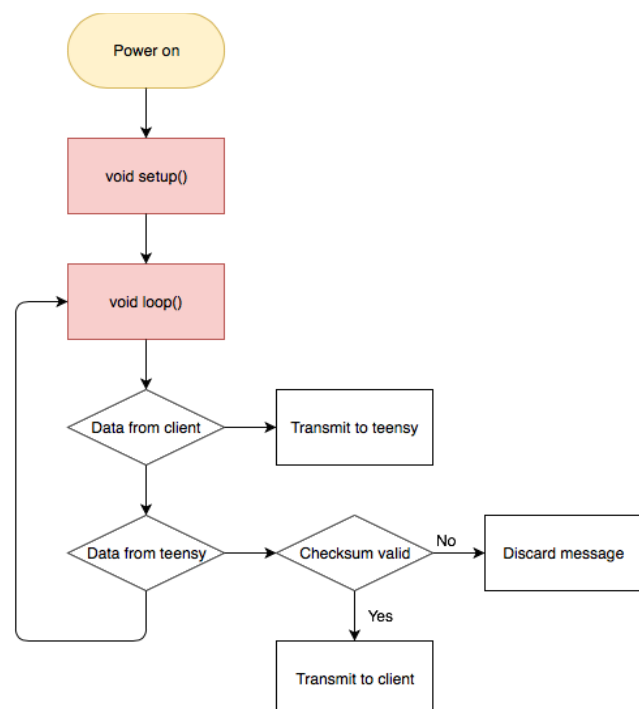
Since the only purpose of the Wemos D1 microcontroller is to relay serial messages to the WiFi socket and vice versa, no external dependencies are needed. The Espressif SDK included in the Arduino toolchain contains built-in libraries to create WiFi access points and TCP socket servers.

Structure

The program for this microcontroller is structured in one single file to relay messages and a utils file for generating and verifying checksums for the messages (as described below).

On microcontroller startup, the setup method starts the WiFi access point with a previously defined SSID and passphrase and initializes the serial interface with the correct baud rates (9600 baud in my case).

The loop method continuously checks if a new client tries to connect to the socket server and accepts the connection. Only one concurrent client is supported. If a client is connected, the microcontroller



checks for incoming messages from the serial and socket connections and forwards it to the counterpart. Since the serial interface does not guarantee message integrity, a checksum needs to be added to each message. If the checksum does not match the computed one, the message is discarded.

iOS App Design

Tools

The iOS app is programmed using Swift and Objective-C. Apple provides an own IDE called Xcode to develop, build and deploy applications for Apple devices. Xcode handles project creation, package identifier registration, app capability handling (you need to request special permission to use certain software or hardware features, like maps or Bluetooth), code signing and deploying.

Up until now, Xcode is and remains the only official IDE to develop iOS applications. JetBrains sells its own IDE for iOS development called AppCode, but it cannot keep up with the rapid changing aspect of Swift development and lacks important features like the Interface Builder with drag and drop support for view binding.

External tools like Cydia Impactor can be used to install applications from Windows or Linux but they are not officially supported and often used maliciously. Using such tools often ends in a revocation of the developer account.

The app is built to target iOS 11 so that new APIs can be used, like the `NEHotspotConfiguration` API, which is required to connect to WiFi APs programmatically without leaving the app.

Structure

iOS applications do not need to follow a special structure like Java programs do (using packages and source dirs). Program files can be placed anywhere in the application module. They can also be grouped using logical folders (these folders do not represent file system folders). In my case, the UI and logic are separated, where each view controller and its relevant UI classes are grouped together.

Application assets (images, backdrops,...) need to be placed in a folder called `"Assets.xcassets"` so they can be loaded in Xcode.

All the storyboards are placed in a folder called `"Base.lproj"` which represents the base language of the application, in my case English.

Dependencies

No official dependency management solution exists for iOS applications (like Gradle for Android). Apple's solution would be to download the library from a `git` repo and bundle it as an external framework. This makes it impossible to check for updates.

Apple does bundle the Swift Package Manager (SPM) to Swift, but it lacks many features and compilation takes longer since the frameworks are linked differently.

A few open source package managers exist, however, the two most popular ones being CocoaPods and Carthage. Since I already had experience using CocoaPods, I decided to use it for this project.

CocoaPods works by including a `Podfile` in each library `git` repo, which CocoaPods then scans and adds to its index. The whole index is then downloaded locally on the developer's computer when installing CocoaPods, and by creating a `Podfile` in the project folder you can reference these libraries. To create a `Podfile` and install dependencies you can use the following basic commands:

```
pod init
```

This command automatically searches for an Xcode Project and transforms it into a Workspace (the difference being the possibility to bundle frameworks used by CocoaPods). It also creates a `Podfile` where you can add libraries. A basic `Podfile` looks like this:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'GoKart' do
  # Comment the next line if you're not using Swift and don't want to use dynamic frameworks
  use_frameworks!

  # Pods for GoKart

  pod 'BEMSimpleLineGraph'

  pod 'SwiftSocket'
end
```

This file would install the dependencies `BEMSimpleLineGraph` and `SwiftSocket`, link them to the Project `GoKart` and build them for iOS 9.0 and newer.

To install the dependencies listed in the Podfile, you can run two different commands, depending if you want to freshly install new dependencies or just update changed ones.

```
pod install
```

This command downloads the library, copies it into the Pods folder inside the project, compiles it and links its framework to the project.

```
pod update
```

This command only updates changed dependencies (like a newer library version).

Theoretically, after running the pod command, Xcode should reload the changed components and detect the new frameworks on its own, but I found that that does not happen every time and the build fails and imports are not resolved. A solution to this problem is to close the workspace and reopen it so that all components are reloaded.

I decided to use the following libraries in this app:

- **BEMSimpleLineGraph**

This library provides a line graph, which I will use to display power data in relation to time on the drive screen. It allows animation of the line and bezier curves to smooth out corners. The view can be integrated into the project by creating a normal view inside the Interface Builder and setting its class to `BEMSimpleLineGraph`.

- **SwiftSocket**

Swift does not provide a way to connect to normal TCP sockets without using native C code. `SwiftSocket` provides an abstraction layer over these C functions and allows the use of modern language features to make socket handling easier. This library is used to connect to the socket server provided by the Wemos D1 message relay.

- **SwiftyBeaver**

A small logging library, which allows logging after levels (ERROR, WARNING, INFO, and DEBUG).

Navigation

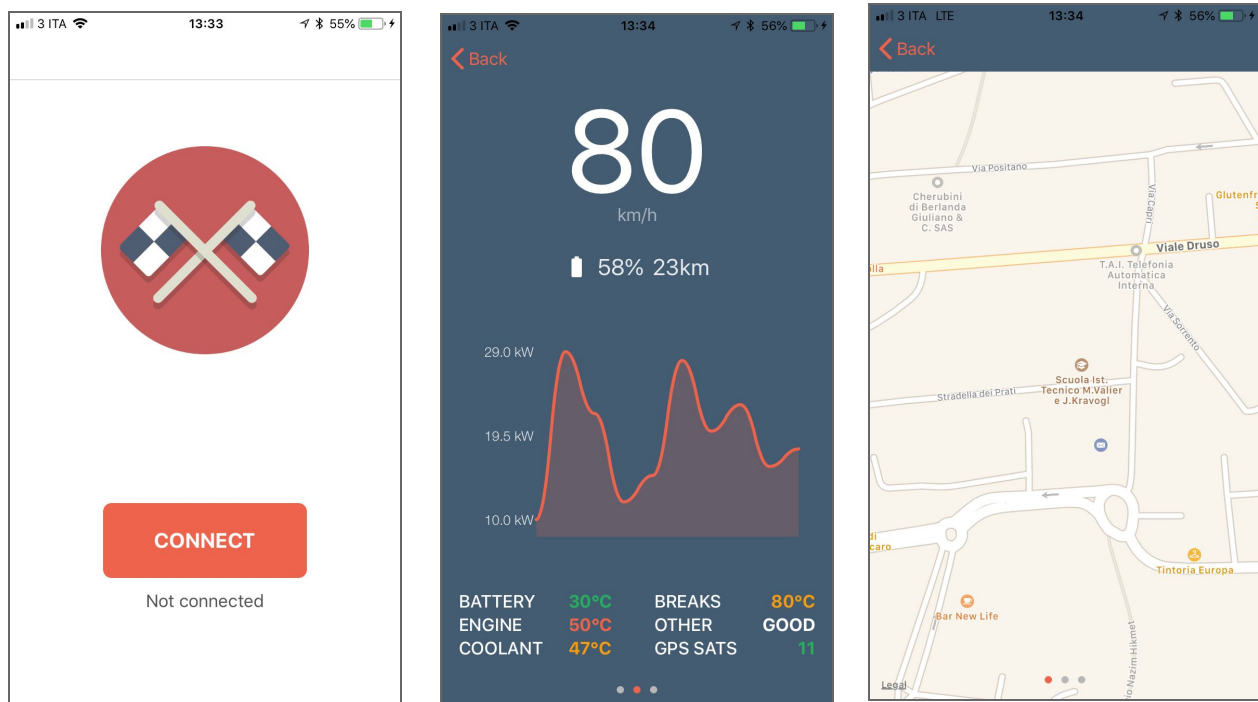
When launching the app, the user is presented with a welcome screen. It features a small image at the top and a red connect button at the bottom with additional progress text (see screenshots below). When pressing on connect and the connection to the message relay is established, the layout changes into driving mode, where all driving data is displayed. This screen consists of a pager based layout with three pages, each displaying different data (like described in Independent Software Components # iOS Application).

App layout

There are many different ways to build app layouts in Xcode. The most common way is to use the Interface Builder, where you can add views and create layout constraints using drag and drop in a live preview.

I decided to use a single interface storyboard (Main.storyboard) and place all view controllers (the activity equivalent on Android) in it. The view controllers are linked using push segues.

A separate storyboard represents the screen when the app is launched and no code is loaded yet (LaunchScreen.storyboard).



Connection Flow

The connection flow is not really easy since we need to check a lot of things in order to acquire a reliable connection.

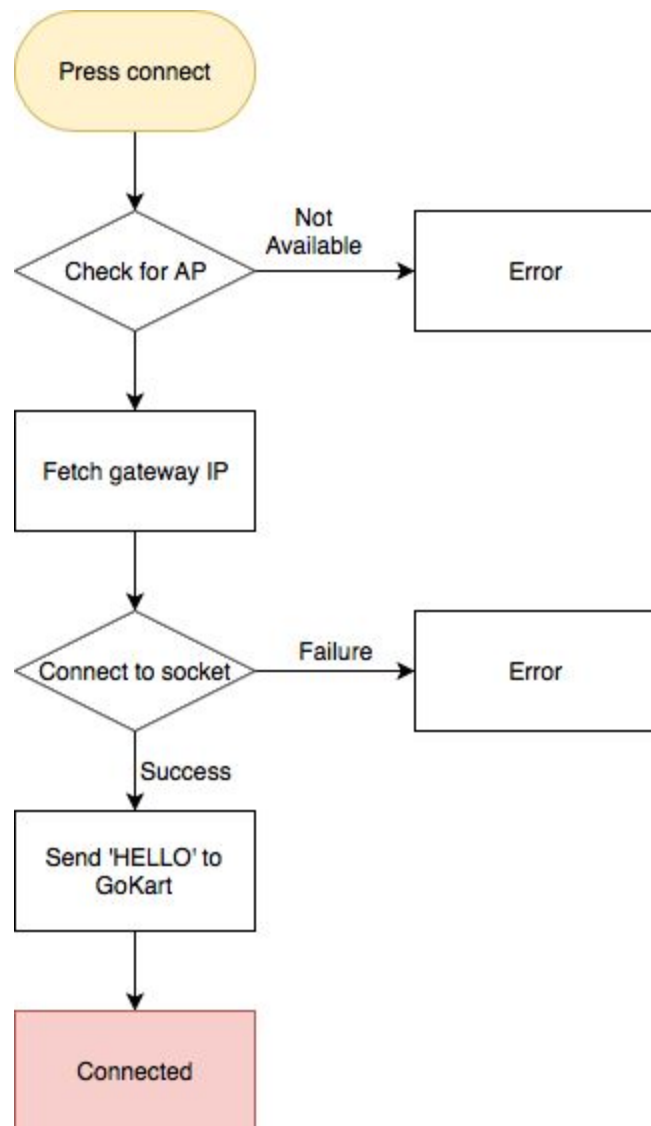
First of all, we need to connect to the WiFi AP broadcast by the message relay. This can be done by using Apple's `NEHotspotConfiguration`, which keeps the WiFi connection alive until the user exits the app. Unfortunately, there is no other way to do this programmatically except manually connect to the WiFi AP from the Settings app. If the WiFi AP is not available the app displays an error message.

To connect to the socket, we first need to get the gateway IP address, which in our case is the message relay where the socket server listens. We could hardcode this IP address, but I noticed it changing at random intervals, so better fetch it on every connect.

Next, we try to connect to the socket at the gateway IP address. A timeout of 10 seconds ensures that we are not stuck in an endless loop, which can happen with raw C sockets (which `SwiftSocket` is based on). If the connection fails, an error message is displayed.

If the connection is successful, a 'HELLO' welcome text is sent to the message relay (which is then forwarded to the main microcontroller) to signal that a new client is connected and is ready to accept telemetry data.

Unfortunately, this type of socket does not provide us with disconnect information, so we need to rely on time-based pings to check if the connection is still alive. If no new data arrives in more than 30 seconds, we consider the connection dead and disconnect.



Data logging

Since not all available telemetry data is transmitted to the client (way too much data to for serial connection running at 9600 bauds), we decided to log all available data for later evaluation. All data is collected in the GoKart struct, which persists over the lifetime of the program. This data is then periodically written to the microSD card (a boolean flag keeps track if data has been updated and it is only written if true).

If the SD card is not available the data is simply not logged and the program continues to work without this feature.

Every category of data is written to a separate file (GPS, temperature, acceleration, CAN, system, ...) so that writes can be reduced and data can be written in smaller packets. This makes it also easier to collect certain data without needing to evaluate all categories.

Since Teensy doesn't offer UTC timekeeping (like using NTP or other network time coordination protocols), we cannot use a timestamp for the different files. If we would use the same filename every time, all telemetry data would be overwritten on reboot, which can happen. Because of that, we use a random base folder for each file, generated by reading noise from the floating analog input pins and converting this noise to a 16 char hash. A sample filename with base folder would look like this:

```
Wkep49ank3J30enrp/gps.csv
```

Wkep49ank3J30enrp represents the base folder name lasting throughout the program lifetime (it will be different on next boot).

gps.csv is the telemetry file containing the decoded GPS data.

The properties currently logged are:

- GPS data: **gps.csv**
- Temperature data: **temperatures.csv**
- Acceleration data: **acceleration.csv**
- CAN data: **can.csv**
- System parameters (uptime, free memory, cpu usage): **system.csv**

File format

The telemetry data is saved as a plain CSV file, with the data timestamp (a simple `millis()` function keeping track of the time since microcontroller boot) as the first data point. All other data is attached to the string after the timestamp.

A sample dataset for the GPS data would look like this:

```
10830|46.546|11.594|260|7|1.24|18.9|18-04-2018_09:24:13|54839|124
```

The individual data entries represent the following categories:

10830: The expired milliseconds after microcontroller boot.

46.546: Current GPS latitude in WGS84.

11.594: Current GPS longitude WGS84.

260: Altitude in meters.

7: Amount of satellites currently visible (needs to be at least 4).

1.24: Horizontal dilution (position accuracy).

18.9: Horizontal speed in km/h.

18-04-2018_09:24:13: Current time provided by the GPS antenna.

54839: Number of NMEA sentence chars processed.

124: Number of sentences with a checksum error.

This format can vary depending on the different categories, but the first data point will always be the `millis()` timestamp. The pipe symbol ("`|`") is used as a separator.

Data transmission

The most important data gets transmitted to the message relay via serial communication, to then be transmitted via a socket connection to the iOS app. Data transmission intervals work like with data logging, so that a boolean flag is used to keep track of new data. The serial communication line from the main microcontroller to the message relay works with a speed of 9600 baud, which is a tradeoff between speed and reliability.

Data format

The data is sent to the message relay in different categories like with logging so that relevant data is bundled. Each message starts with a dollar sign ("\$\$"), followed by the type of the message, a colon (":") and then the data. The data is succeeded by another dollar sign and then the checksum. The message relay checks the integrity of the message and only forwards it to the client if the message is valid.

A sample message would look like follows:

```
$GPS:46.546,11.594,260,7,1.24,9,24,13$e
```

The individual data entries represent the following categories:

\$GPS: GPS message identifier, depends on the data category

46.546: Current GPS latitude in WGS84.

11.594: Current GPS longitude WGS84.

260: Altitude in meters.

7: Amount of satellites currently visible (needs to be at least 4).

9: Current time hours

24: Current time minutes

13: Current time seconds

54839: Number of NMEA sentence chars processed.

124: Number of sentences with a checksum error.

\$e: Message end separator with checksum character ("e")

This message format was chosen because using dollar signs (or any other uncommon char) as separator allows us to easily identify a valid message and the start and end.

Various messages are available and identified using a 3 character long string. The following data identifiers are currently used:

GPS: Identifies GPS data

TMP: Temperature data

ACC: Accelerometer data

CAN: CAN bus data

Checksum

Compared to a TCP socket connection, serial does not guarantee message order and integrity. This becomes obvious especially at high baud rates, where chars get flipped or lost. To combat this problem, an XOR checksum is added to each message. This checksum gets compared to a computed checksum on the message relay, and the message is only forwarded to the iOS app if the transmitted and computed checksums match.

The checksum is computed by XOR-ing each character in the message, including the dollar signs and identifier. The XOR function looks like this:

```
unsigned char getChecksum(char *string) {  
    unsigned char XOR = 0;  
  
    for (int i = 0; i < strlen(string); i++) {  
        XOR = XOR ^ string[i]; // NOLINT  
    }  
  
    return XOR;  
}
```

Both sides need to use the same checksum function for this to work. The checksum is not used on the iOS app but the message relay still forwards it, since string manipulation uses precious CPU cycles on our Wemos D1 microcontroller, which is not suited for such operations.

Data evaluation

We log all data so that we can evaluate it later on, should there be a failure of a component or should we decide to tweak some software parameters. For that to work, we need to be able to collect a lot of data really fast and save it in a database to be able to analyze or graph it.

macOS data converter

A simple macOS command line converter application written in Swift reads all CSV files, converts them into SQL format and inserts it into the database. The program can be launched using the following command:

```
./converter import <base folder> <database connection string>
```

A sample usage with the telemetry data used throughout this document and a PostgreSQL database would look like follows:

```
./converter import /Volumes/GoKart/Wkep49ank3J30enrp psql://localhost@user:pass/telemetry
```

This command reads all available telemetry files on the microSD card called GoKart, with the base data path Wkep49ank3J30enrp. It then connects to a PostgreSQL database listening on localhost with the user/password combination user:pass and inserts it into the database called telemetry. This command requires that the schema in the specified database is already available.

Should the schema be missing, it can be created by executing the following command:

```
./converter schema psql://localhost@user:pass/telemetry
```

PostgreSQL database

For my testing purposes, I used a PostgreSQL database since I had the most experience with it and it offers useful features like triggers and functions.

Instead of installing the database locally on my computer, I decided to install it in a Docker container so I can destroy and rebuild it if I want to clear all data and restart from scratch.

The PostgreSQL database can be launched using the following command, provided that Docker is installed:

```
docker run -it --rm --link network:postgres postgres psql -h postgres -U postgres
```

This command creates a throwaway PostgreSQL container (meaning that all data will be lost when the container is stopped) in a network called “network”, with hostname “postgres”, username “postgres” and database “postgres”. It also automatically connects to the databases psql command line interface, where you can execute SQL commands.

If you want to keep the data you added to the database, you can mount an external volume and put the PostgreSQL files on it. Volumes persist even after the container is stopped.

Testing and validation

Testing on the Arduino platform

Unfortunately, no proper first-party solution for running unit tests is available for the Arduino platform. Some third-party solution exist, but none are compatible with Teensy 3.5 microcontrollers.

A third party testing suite for the ESP8266 platform is available, but since the code is really simple and only contains default the `setup()` and `loop()` methods, unit tests were left out (since those methods cannot be mocked).

Another challenge when developing for Arduino was the lack of a proper debugging or stack trace solution. This made developing for the Teensy 3.5 microcontroller especially frustrating since I often encountered OOM (Out Of Memory) errors caused by a third party library, where the microcontroller would simply hang without any message or kernel dump. A physical reboot by unplugging the power was necessary to restart it.

Fortunately, the ESP8266 microcontroller displays a kernel dump when a segmentation fault or some other race condition occurs. This makes debugging relatively easy since the kernel dump can be converted back to a readable method call stack when building the code with symbols enabled (enabled by default). A raw kernel dump looks like this:

```
Exception (0): epc1=0x402103f4 epc2=0x00000000 epc3=0x00000000 excvaddr=0x00000000
depc=0x00000000

ctx: sys
sp: 3ffffc10 end: 3fffffb0 offset: 01a0

>>>stack>>>
3ffffdb0: 40223e00 3fff6f50 00000010 60000600
3ffffdc0: 00000001 4021f774 3fffc250 4000050c
3ffffdd0: 400043d5 00000030 00000016 ffffffff
...
3fffff80: 4021c0b6 3fffdab0 00000000 3fffdcb0
3fffff90: 3ffecf40 3fffdab0 00000000 3fffdcc0
3fffffa0: 40000f49 40000f49 3fffdab0 40000f49
<<<stack<<<
```

The number after the “Exception” keyword indicates the type of exception (0), in this case, a `IllegalInstructionCause`. This kernel dump can be symbolized using the built-in exception decoder in the Arduino IDE, making it readable for humans. It will then show the method where the exception happened.


ESP8266 offers an experimental debugger built into the Arduino IDE, but at the time of writing this document, I was unable to get it to work.

iOS Testing

Xcode and Swift offer support for writing both unit and UI tests. Test integration can be added automatically when creating a project or retrofitted later on if needed.

Tests need to be written in an own class, which needs to subclass the Cocoa method `XCTestCase`, which marks it as a test class. The test class can contain however many test methods as needed. A basic testing method, comparing 1 to 1 can be written as follows:

```
func testOneIsOne() {
    XCTAssertEqual(1, 1, "1 should always equal 1")
}
```



Theoretically, this test should never fail. If it does (using different values), Xcode automatically creates a test exception breakpoint, which halts the debugger at the place the assertion failure occurs, so the failing code can be debugged.

At the time of writing this document, the iOS app contains unit tests for the socket data receiver and the connection handler. No UI tests were written due to time constraints.

Tweaks and improvements

Testing the go-kart resulted in a couple of proposed improvements to make collecting the various data even more reliable. Since we were time-constrained, these improvements have not been implemented and are purely speculative.

The proposed improvements include:

- Replace the reed-switch based RPM meter with an IR-based probe, since the reed switch requires a magnet to be mounted on the rear driving axle. This magnet causes an imbalance. Using physical switches also requires software bouncing, which limits the measurable revolution at 1200 RPM with a 50 ms debounce time.

Using IR based RPM measuring removes axle imbalance and the need of debouncing.

- Replace the Wi-Fi-based data transmission with a physical one. Since Apple requires being registered to the MFi-program, this solution is only practical in theory.

A Wi-Fi data transmission adds latency (around 100 ms in our case) and adds potential points of failure. A physical connection is both faster and more reliable. A potential drawback is a physical connection requiring the phone to be mounted on the steering wheel at all times since it needs to be connected to a cable.

Experience and closing

Planning and implementing this project has improved my knowledge of both the software and the hardware side.

Working with so many different sensors and data interfaces was a real challenge and made me more familiar with technologies used when working with microcontrollers or other low-level devices. I already had some experience working with the Arduino platform, but nothing came close to the complexity of this project. Building the data communications with a relay device in between was first thought to be a small feat, but resulted in hundreds of code lines to get the reliability to a point where we could make it through a simulation race without disconnects.

Where to go from here

Since this go-kart was paid by AutoTest GmbH, it will ultimately be handed over to their respective owner as a personal vehicle after all development and testing are done.

The app and hardware design will also be handed over to Autotest GmbH, where they can install it on their personal phone by flashing the GoKart .ipa package, which is the built application.

Since I won't be available to improve and maintain this project in the future, the code will be available as Open Source Project.

We plan to complete our first full race mid-July at the Safety Park in Laives, IT, where this project will finally, after hundreds of hours of hard work, come to an end.

Sources

These are the sources used to compile this project description. All information from these sources was taken between 10. of April and 25. of April.

<https://www.sparkfun.com/products/14055>

<https://en.wikipedia.org/wiki/Arduino>

<https://en.wikipedia.org/wiki/IOS>

<https://en.wikipedia.org/wiki/MacOS>

<https://developer.apple.com/programs/mfi/>

<https://www.pjrc.com/store/teensy35.html>

[https://en.wikipedia.org/wiki/Swift_\(programming_language\)](https://en.wikipedia.org/wiki/Swift_(programming_language))

<https://en.wikipedia.org/wiki/ESP8266>

<https://www.espressif.com/en/products/hardware/esp8266ex/overview>

<https://www.sparkfun.com/products/12577>

<https://www.sparkfun.com/products/retired/11859>

<https://www.sparkfun.com/products/11028>

<https://www.sparkfun.com/products/11050>

<https://www.sparkfun.com/products/11058>

<https://www.sparkfun.com/products/132>

<https://www.sparkfun.com/products/13262>

<https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi>

https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

<https://en.wikipedia.org/wiki/I²C>

<https://learn.sparkfun.com/tutorials/i2c>

<https://learn.sparkfun.com/tutorials/serial-communication>

https://en.wikipedia.org/wiki/Serial_communication

https://en.wikipedia.org/wiki/CAN_bus

<https://platformio.org>



https://en.wikipedia.org/wiki/Global_Positioning_System

<http://www.gpsinformation.org/dale/nmea.htm>

<https://gps.gov>

<https://draw.io> (Flowcharts)