# Personal Project

Thomas Schenk

A drawing of a car

Description automatically generated with low confidence

Contents

[Personal Project 1](#_Toc137587404)

[Introduction 3](#_Toc137587405)

[Project description 4](#_Toc137587406)

[Requirements of the final product 5](#_Toc137587407)

[Usecase 5](#_Toc137587408)

[Derived functionality 5](#_Toc137587409)

[Required data 6](#_Toc137587410)

[Realization of the product 7](#_Toc137587411)

[Used hardware 7](#_Toc137587412)

[Software diagrams 8](#_Toc137587413)

[CPP Code 11](#_Toc137587414)

[WiFi Module 12](#_Toc137587415)

[HTTP Module 14](#_Toc137587416)

[Buffer Module 15](#_Toc137587417)

[DateTime Module 17](#_Toc137587418)

[Sensor Module 19](#_Toc137587419)

[HTML/PHP Code 20](#_Toc137587420)

[Index file 20](#_Toc137587421)

[Endpoint file 23](#_Toc137587422)

[Database setup 24](#_Toc137587423)

[Validation of the results 25](#_Toc137587424)

[Valuable insights for next time 27](#_Toc137587425)

## Introduction

This is my personal project for Semester 6. This document will contain everything you need to know about my project, from start till finish. Code that will be written is an exception to this, these files will be put alongside this document as to not clutter. Things like results or graphical images to support the story will be in this document.

## Project description

Being a part of a subgroup which enjoys motorcycles, cars and a lot of other mechanical working things; I gathered quite a bit of knowledge over time about this topic. Knowledge that could be very beneficial to people who don’t have this knowledge. Why? To improve the life of their vehicles.

Lots of people would like to drive older vehicles, but are put off by the seemingly very complicated maintenance which needs to be done on them. While in reality, this maintenance is quite simple. Most of the times, this maintenance however has to be done intuitively, since older vehicles often lack fancy gauges and computers.

My project aims to create such a computer, like a plug and play. While measuring certain diagnostics is a priority; so is saving them. The end user should be able to look back onto their data when they arrive home. That means even though no internet connection is present while driving, the data should be kept and uploaded when there is one.

This product would solve neglect in situations where the owners are not properly informed about maintaining their vehicle.

## Requirements of the final product

This list will serve as requirements for the final product, what is important for the computer to measure? And why? What can we do with the measured data?

### Usecase

First let’s state a use-case for this product. This way we can see that the product user does and we can decide what we would ultimately need.

*Assumed is that the device is mounted in the vehicle; and powered on whenever the vehicle is too.*

The user gets in their vehicle parked in the driveway. The user turns on the keys and starts the engine, takes a little bit to put their coffee in the holder, adjust their seatbelt and mirrors and eventually drive off.

While driving, the user does not interact with the device at all; essentially operating the vehicle like nothing has changed.

After arriving at their home, again, the user would take some time to get everything from their vehicle; undo their seat belt, shut the car off and get out.

After their trip, the user goes online to see the gathered data.

### Derived functionality

Needed functionality:

* Automatic connectivity to backend

The user never interacts with the system outside of the dashboard; therefore, uploading data to this dashboard should be done automatically without user interaction. This includes making and closing connections with one or more network interfaces when prompted.

* Quick one look overview of gathered data

The user should not be overwhelmed with complicated information. The data shown must be concise, clear and most importantly; be all on one page.

* Complete and accurate datapoints

The datapoints gathered must mean something, we could achieve a sense of direction in a sea of data when we apply unique identifiers to the data. Currently the first identifier gravitates towards using time.

* Offline storage of gathered data

Obviously backend connectivity relies on a connection; for web based implementations this is almost always WiFi. To not introduce extra cost by using LTE, offline data generation must be supported.

### Required data

To both limit ourselves to a concise set of data; but simultaneously gathering good quality data; we gather the following data in the current version of the product.

* Rpm

With this we can detect how much the user revs the engine, maybe its excessive or maybe its too little. Excessive or improper revving can damage certain parts of the engine in an accelerated way.

* Oil temperature

Engine oil usually has a very high temperature rating, but if it exceeds this rating. Either the driver is driving in a weird/bad way; or some cooling component is damaged. If the oil exceeds its temperature rating in any way, this can impede the effectiveness of this oil. Thus, accelerating wear on the lubricated parts.

* Acceleration/speed

Speed or acceleration is an indicator of how the user drives its vehicle, speed has so much influence on all factors in a vehicle; that data can easily be derived from it.

All sensory, thus data, will be simulated. Since a heavy focus is put on handling and processing incoming data in a correct manner as described by the requirements, rather than interface hardware which is almost a trivial task. I did this since the challenge would be far greater to focus on a correctly working buffer system with continuous time generation; than interfacing sensors I have worked with before.

## Realization of the product

This chapter will dive deeper in the road to the product currently known. To do this, we will first establish what the current used hardware is. And what the current design is. According to that, explanation about code and functionality will be easier to both convey as a writer; and understand as a reader.

### Used hardware

Currently, only an ESP32 is used to execute all code. I chose for an ESP32 since it already integrates a WiFi chip which is compatible with CPP programming.

For demo-ing purposes, I can attach some potentiometers to incorporate a more user interacted feel of the system. These interfaces directly map on key values provided by the rest of the program. Essentially leaving the door open for any future hardware which will slot into this project.

## Software diagrams

Diagram, schematic

Description automatically generated  
Idea of the system sketch pre development (Currently heavily outdated and superseded by following diagrams, yet gives a good global scope)

C1 Model

A diagram of software system

Description automatically generated with medium confidence

C2 Model

A picture containing text, screenshot, diagram, plan

Description automatically generated

C3 Model

A picture containing text, screenshot, diagram, plan

Description automatically generated

## CPP Code

To provide the ESP32 with functionality, CPP (C++) Code is written to enable it to do certain tasks.

This CPP code can be split into a few different modules, which all intertwine with each other in the main loop. These modules are explored upon and explained in the following sub chapters.

### WiFi Module

To provide the system with a network functionality, the wifi module makes sure to successfully connect to configured wifi networks. The setup of this differs from most conventional methods.

Previously, the code could only have one wifi network; and only connect to that one. However after my first presentation of the project, a usecase arose where the functionality for multiple wifi networks was needed. I adjusted the code to work accordingly.

Firstly, we make a wifi object. This will be our object to redirect all wifi calls to.

WiFiMulti wifiMulti;

Later, we need to make sure our program tells this object what its available networks are. To smooth this over, I made a function which covers this.

void setupWifiCreds()

{

  wifiMulti.addAP("Internet", "");

  wifiMulti.addAP("Test", "Test1234");

  wifiMulti.addAP("ssid\_from\_AP\_3", "your\_password\_for\_AP\_3");

}

Obviously, this function is highly customizable.

To execute wifi connectivity in the main code, I wrote this function to streamline the process. Pay attention to the WiFiMulti command using run, instead of begin.

void connectWiFi() {

  WiFi.mode(WIFI\_OFF);

  delay(1000);

  //This line hides the viewing of ESP as wifi hotspot

  WiFi.mode(WIFI\_STA);

  setupWifiCreds();

  wifiMulti.run();

  if (WiFi.status() != WL\_CONNECTED)

  {

    Serial.println("No usable adresses were found");

  }

}

At the end of our logic loop, I check every 50 reads if we can maybe connect to the internet again. This way we ensure that if there is an internet connection somewhere; and we miss it. We can always reconnect.

retriesConnection++;

  if (retriesConnection >= 50)

  {

    if (WiFi.status() != WL\_CONNECTED)

    {

      connectWiFi();

    }

    retriesConnection = 0;

  }

### HTTP Module

Having internet connection is great, but for this application; I will not be using the internet protocol in its singularity. Instead, its paired with an HTTP overlay; or more commonly known as “The” internet. To achieve this, we need to setup a few things so we can accommodate for the overhead headers which come with this protocol.

In the loop, we only make the http object if we are connected to the internet. If we aren’t, crashes are more prone to happen. But also, who needs http connectivity if you are not connected to the internet at all?

HTTPClient http;

After this, we initiate a connection with a given URL; in our case this is the endpoint page on the webserver.

http.begin(URL);

URL refers to:

String URL = "http://i466372.hera.fhict.nl/connect.php";

After establishing a begin, we immediately initiate and send our date. Without checking if the connection is successful.

http.addHeader("Content-Type", "application/x-www-form-urlencoded");

        int httpCode = http.POST(postData);

        String payload = http.getString();

        Serial.print("URL : "); Serial.println(URL);

        Serial.print("Data: "); Serial.println(postData);

        Serial.print("httpCode: "); Serial.println(httpCode);

        Serial.print("payload : "); Serial.println(payload);

        Serial.println("--------------------------------------------------");

We can do this, since all these overhead returns from the http object, are designed to handle all sorts of miscommunications, weird request, failed endpoints and so on. By omitting unnecessary checks, not only is the program smaller and faster; but we also provide more insights into possible problems by letting the object return its own checks.

### Buffer Module

Like mentioned previously, we only initiate an HTTP connection if we have access to a wifi point. So, what happens if we drive down the road without any connection whatsoever? Will our data be lost?

No, to accommodate for this; there is a buffer. This buffer saves all gathered data, until a stable connection is present and it can empty itself.

To understand how we save data in a buffer, it is important to first understand what we are saving. In this case, a custom buffer with data.

This struct is structured like follows:

#include <string>

#include "time.h"

struct Reading

{

    uint8\_t speed;

    uint8\_t oilTemperature;

    uint16\_t rpm;

    tm timeStamp;

};

*More information on the time/tm object later in this document.*

When we include this in our main, we can use it to declare a buffer of type Reading (in other words: of type custom struct).

RingBuf<Reading, 1000> buffer;

I have tested buffer overflow mechanics to ensure that 1000 entries does not break the memory stack. I did this by letting the program run for as long as the buffer is full and checking if any breakage was happening (most notably a stack overflow).

We only push data into the buffer when we lack a connection, otherwise we just create a Reading object. This is done to minimize latency. Why put something in a buffer first if a data-connection is already present? That would introduce unnecessary extra steps.

It could be argued that data retention in the case of a disconnect would be the reason to put it in a buffer. However, I decided that in the massive amounts of data produced; this doesn’t warrant extra logic.

If we would like to push data to the buffer, we would do it like so.

buffer.push(sensorReading);

The push function fills the buffer until its limit (ours is 1000), if at a 1000; no more data is added to the buffer. Push Overwrite is a function which will overwrite the oldest data. I opted to not do this, since it defies the point of this product.

To retrieve our buffer data, we use a temporary Reading object. We do this to not corrupt the data gathered by other processes which use a global Reading object. We do this until the buffer is empty, or the internet connection disconnects.

while(buffer.isEmpty() == false)

      {

        Reading tempObject;

        buffer.pop(tempObject);

Ofcourse, this object only lives in the loop.

### DateTime Module

To effectively keep track of time, we use a time object. This time object can store the current time. You don’t need an RTC for this, like many implementations of time tracking on Arduino. Instead, it requires the user to first make a call to a” timeserver website”; the object will then store this and count upon it.

This usage perfectly falls into our usecase, where vehicles are started next to buildings and or places with known wifi networks. In the future, this implementation could still prove effective; since LTE could still be used to make a short call to this website. Data uploading would then need to happen at a stable wifi connection.

To start working with time, we need a time object. You need to declare these as structs.

struct tm timeinfo;

We also use a tm struct in our own struct, Reading.

#include <string>

#include "time.h"

struct Reading

{

    uint8\_t speed;

    uint8\_t oilTemperature;

    uint16\_t rpm;

    tm timeStamp;

};

Some configuration is needed, these numbers account for the timezone; and if timesavings are currently used in your country. NtpServer is the server of which the time will be retrieved.

configTime(gmtOffset\_sec, daylightOffset\_sec, ntpServer);

To get the time, a wifi connection is needed like stated before. If this is the case, the following function writes the correct information to the struct.

getLocalTime(&timeinfo);

Then, the following is very important; and it still is semi-unclear to me. However, you NEED to call the printLocalTime function at least once in your working loop. Otherwise, the local time will not update. You cannot statically expect this struct to update itself, which is logical. But still confusing if not understood correctly.

It does this by adding the local ticktime of the program to its already existing ticktime; retrieved from the ntpServer. A commonly used dataformat in embedded systems also uses this, but more rudimentary. It is called the UNIX time. More can be seen here <https://currentmillis.com/> .

Further down in the code, we extract time data from the struct in the following manner

date = ("20" + year + "-" + String((timeinfo.tm\_mon + 1)) + "-" + String(timeinfo.tm\_mday) + "\_" + String(timeinfo.tm\_hour) + ":" + String(timeinfo.tm\_min) + ":" + String(timeinfo.tm\_sec));

We do this in this complicated manner since our SQL statement prefers a string instead of other datatypes.

### Sensor Module

Currently, no sensor module is implemented. Prototype classes are already present; but not fully implemented yet. This was done after the first successful presentation, to try and get some nice changes for the last presentation. However, I got stuck on this. Currently we use hardcoded data which can be altered by potentiometers for a demo.

## HTML/PHP Code

*The code discussed in this chapter will only be of functional matter. No creative aspects will be discussed, since not only is it purely for show; it is also highly trivial.*

Much like the CPP code, the HTML and PHP code also have modules which have a certain responsibility. These modules are split into two files, the index and the endpoint file. The index is used to display the date. While the endpoint is used for the ESP32 to send its data to.

Be aware, that all these files are located on the self-hosting service Fontys gives their students. I did not host it on my own one.

### Index file

The index file is the hub where the user will first land on to view their data. On this page, we really only want to display data. And in the current version, not much else.

#### Database Module

To display this data, we need to first retrieve it. We do this by declaring a few variables needed for the connection. After this, we make it a variable which we can pass around.

$conn = new mysqli($servername, $username, $password, $dbname);

This variable contains the connection to the database.

Immediately after the, we go straight for the data. By executing an SQL query.

$sql = "SELECT id, oilTemperature, rpm, speed, timestamp FROM sensorreadings order by timestamp desc limit 40";

This pulls the corresponding fields, with a limit of 40; all sorted by the timestamp.

After this, we put this in an array. We also explicitly check for the result of the query, as to not populate bad and good data in the same array.

$result = $conn->query($sql);

while ($data = $result->fetch\_assoc()){

    $sensor\_data[] = $data;

}

After this, we check for some JSON numerics; which are needed for the Graphing library. After this, we close off the connection since its no longer needed.

$readings\_time = array\_column($sensor\_data, 'timestamp');

$oil= json\_encode(array\_reverse(array\_column($sensor\_data, 'oilTemperature')), JSON\_NUMERIC\_CHECK);

$rpm= json\_encode(array\_reverse(array\_column($sensor\_data, 'rpm')), JSON\_NUMERIC\_CHECK);

$speed= json\_encode(array\_reverse(array\_column($sensor\_data, 'speed')), JSON\_NUMERIC\_CHECK);

$reading\_time = json\_encode(array\_reverse($readings\_time), JSON\_NUMERIC\_CHECK);

$result->free();

$conn->close();

#### Visualization Module

We are now done fetching the data and we can address a full array by calling one of the above variable names.

We can show our date, by utilizing the following Highcharts script.

<div class="pt-5 mt-5"></div>

    <h2>Engine Monitor</h2>

    <div class="row">

    <div id="chart-temperature" class="col-md-right">

        <script>

            var oilT = <?php echo $oil; ?>;

            var reading\_time = <?php echo $reading\_time; ?>;

            var chartT = new Highcharts.Chart({

              chart:{ renderTo : 'chart-temperature' },

              title: { text: 'Oil Temperature' },

              series: [{

                showInLegend: false,

                data: oilT

              }],

              plotOptions: {

                line: { animation: false,

                  dataLabels: { enabled: true }

                },

                series: { color: '#059e8a' }

              },

              xAxis: {

                type: 'datetime',

                categories: reading\_time

              },

              yAxis: {

                title: { text: 'Temperature (Celsius)' }

                //title: { text: 'Temperature (Fahrenheit)' }

              },

              credits: { enabled: false }

            });

            </script>

    </div>

I have done this for all three previously stated groups of data. Temperature, speed and RPM.

### Endpoint file

To give the visualization file all this data, there needs to be a middleman in all this. Which is a page that gets the data and puts it in the database.

This code also lives on the webserver but is not interacted with by anything but a system or computer. The end user will not even know of the existence of this file.

Firstly, we establish the connection like we have done in the previous file the exact same way. You might notice the word die, the only reason we can use that here is since this, like mentioned, is not a file for user interaction.

$conn = mysqli\_connect($hostname, $username, $password, $database);

if (!$conn) {

    die("Connection failed: " . mysqli\_connect\_error());

}

echo "Database connection is OK<br>";

After this, we just continuously check whether the visitor of the page also does a post request. If so, they need to posses a very specific structure of fields; which obviously corresponds to our database setup. If this is the case, we post them to the database!

if(isset($\_POST["oilTemperature"]) && isset($\_POST["rpm"]) && isset($\_POST["speed"]) && isset($\_POST["timestamp"])) {

    $t = $\_POST["oilTemperature"];

    $r = $\_POST["rpm"];

        $s = $\_POST["speed"];

        $d = $\_POST["timestamp"];

    $sql = "INSERT INTO sensorreadings(oilTemperature, rpm, speed, timestamp) VALUES (".$t.", ".$r.", ".$s.", CAST( '".$d."' AS DATETIME))";

 Echo $sql;

After this, much like we can print HTTP overhead messages, we print the SQL response.

    if (mysqli\_query($conn, $sql)) {

        echo "\nNew record created successfully";

    } else {

        echo "Error: " . $sql . "<br>" . mysqli\_error($conn);

    }

}

## Database setup

I would like to keep this chapter fairly brief since its so trivial, though it must be mentioned. The database setup is also on the Fontys hosting. And is of the type MySql.  
  
I configured it to only have basic field right now. No foreign keys.  
The fields are:

* Rpm
* Speed
* Temperature
* Timestamp
* Id (not used in this application)

## Validation of the results

To validate if our product meets its requirements, we can simulate the expected usecase. I have done this in the form of a video. I will type the script of what happens on the screen below.

*To remind you! This was the usecase we stated before.*

*Assumed is that the device is mounted in the vehicle; and powered on whenever the vehicle is too.*

The user gets in their vehicle parked in the driveway. The user turns on the keys and starts the engine, takes a little bit to put their coffee in the holder, adjust their seatbelt and mirrors and eventually drive off.

While driving, the user does not interact with the device at all; essentially operating the vehicle like nothing has changed.

After arriving at their home, again, the user would take some time to get everything from their vehicle; undo their seat belt, shut the car off and get out.

After their trip, the user goes online to see the gathered data.

Typed out version of validation demo:

The program already lives on the webserver, this server is always live. Therefore, we can navigate to its url. The url is: <http://i466372.hera.fhict.nl/view.php>

When we are on this page, we can see some nice graphs on the screen. Zooming in at the data, we can see that it is outdated. Perhaps from the last time we ran a trip to the store? Who knows. Anyways, lets “Start” our car.

When starting our car, we have a computer screen in the seat beside us; so we can see what the debug terminal says!

When powering up, we see that the wifi takes a short while to connect; but then once connected fairly quickly does the time setup.

The sensors start rolling, and data is being sent!

Since im still on my passenger seat pc, I can go and have a look at my website; see what my values are right now.

Ah, the values are good! We checked if they are updated; and they are! Nice. I am going to drive away from home, losing my connection; but! Ill stop at a nearby parking lot. There I can check my passenger seat pc to see the debug console.

When I stopped, and looked, I could see that the data was stored in a buffer; since I had no internet connection.

Quickly I pull out my phone, curious what my speed was when driving down the parking lot! I might have gone a bit too far.

While checking, I see that the data is still old, from when I was home! Ofcourse, how could I forget; no connection means no data transmission to the server!  
  
I quickly drive home to check what my speed was at the parking lot. Luckily the data updated right away as I pulled in the driveway; I could see this on my passenger pc too.

Luckily, upon checking; I was only going 10 kmph; no worries!

End of usecase.

*I have included a file which shows the video. Inserting it here was very cumbersome.*

## Valuable insights for next time

This is a personal project, so continuation might be very pheasable. But, what would I need to do next time to make this a more extended product?

I find that the biggest obstacle lives in the data generation/gathering mechanism. And I already have some ideas on how to tackle the three data groups currently proposed in the document.

Currently there are some problems which need to be tackled in order to gather the data previously mentioned.

* How are we going to measure RPM?

Most of the time RPM is already integrated in a vehicles dashboard. But I want to tailor this product more towards scenarios where this is not the case; how do we determine engine RPM when this is not integrated?

Usually engine RPM meters are mechanically implemented. This is because to measure the RPM of an engine, you need access to the crankshaft. Mechanically this is way easier to achieve than electrically. However both need to be implemented by design, it is near impossible to implement a mechanical or electrical solution based on the crankshaft.

That is why we are going to implement a system that makes use of a trivial part of the workings of an engine; the spark plugs. By determining the firing pattern, we can approach the number of times the engine fires over; and thus determine how fast the engine is spinning.

* How are we going to measure oil temperature?

In all applications, an oil temperature reading is going to need specialized hardware. Especially if this hardware is missing from the vehicle in the first place.

Most of these oil sensors from OEM manufacturers rely on a thermistor which determines the resistance over the sensor. Knowing this, we can solve our missing sensor problem by making sure we implement a feature where a temperature sensor can be added.

* How are we going to measure speed/acceleration?

Speed measurements often rely on GPS. This is because to measure speed, a current and previous location are needed in order to calculate the speed.

Accelerometers, a group of sensors which are also involved in movement calculations, lack these datapoints. Which would require making a map taking the starting point into account and calculate velocity between the travelled distances. This however would take an enormous amount of calculations and most of all, would require a lot of storage to accommodate for the ever growing map that will eventually be created.

This is a shame, since acceleration isn’t a native measurement of a GPS system. This would mean that for acceleration, we need to add another sensor.   
However, the prognosis is that we can estimate this using the distance and speed gathered by the GPS.