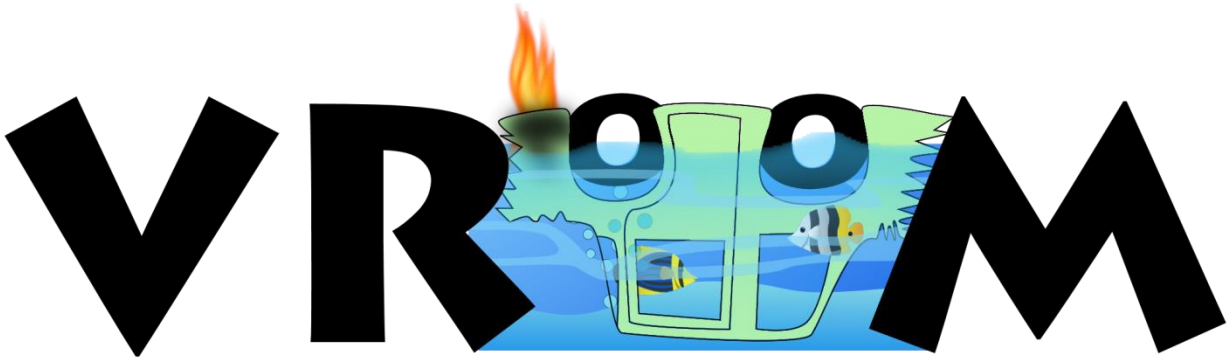
Bachelor Project Report



Vehicle Remote Optimised Observational Management

Bachelor Project

Subject: Project Report

Hand in Date: 2014-12-12

Group #16: Andi Michael Degn 127305@VIA.DK
Kenneth René Jensen 166687@VIA.DK

Supervisor: Stephan Erbs Korsholm SEK@VIA.DK

Version Control			
Version	Date	Main author	Changes
0.1	2015-08-05	Kenneth	Template prepared
0.2	2014-08-14	Kenneth & Andi	Requirements and Accident definition added
0.3	2014-10-28	Kenneth & Andi	Introduction added and problem formulation fine-tuned
0.4	2014-10-29	Kenneth	Analysis and Design section added
0.5	2014-11-04	Kenneth & Andi	Proofing Introduction, Analysis and Design sections
0.6	2014-11-25	Kenneth	Added sensors as actor in UseCase diagram and changed/updated MSD structure format
0.7	2014-12-09	Kenneth & Andi	Implementation section added
0.8	2014-12-10	Kenneth & Andi	Testing and results/discussion sections added
0.9	2014-12-11	Kenneth & Andi	Abstract, conclusion and references added
1.0	2014-12-12	Kenneth & Andi	Proofing all sections

Contents

Figures

Abstract

The purpose of this project is to develop a prototype of an eCall system, as an aftermarket solution. This means that this system works independently of car make and model as it does not utilise any of the cars internal components, except the battery. This includes older cars as well. This system is autonomous and intended to be plug'n'play, easy to use and at an affordable price.

By installing this system in a vehicle, the driver can feel safe and rely on that help will arrive as-fast-as-possible, even in case of a solo accident where it is not possible to call for help. This system brings the driver closer to the help required in a given emergency. It also helps speed up the response time for the emergency personal as the relevant information about the emergency including the position of the car is transmitted directly.

This prototype is not tested in a real car, however a remote control car is used for demonstrating and proof of concept.

1. Introduction

1.1. Background

Nowadays people pay a very steep price for mobility. In Europe alone around 40,000 people die on the roads every year[1].Many of the fatalities are not only caused by the crashes themselves, but by the time it takes for help to arrive.

The first problem is that someone has to notify the authorities. More often than not the involved people are incapacitated and therefore incapable of doing this, so a third party has to do this. Time is wasted.

Next problem arises when the emergency operator needs to collect the required information; it can be challenging for the operator to retrieve the information from a random caller. A protocol is needed. Lastly, the operator needs to dispatch the proper emergency response.

A pan-European project called HeERO[2], addresses an in-vehicle emergency call services based on the common European Emergency number, 112. HeERO is a group of European pilots who develop, implement and tests the infrastructure and standards. This becomes both the service and an IVS (In-Vehicle System) called eCall. At this moment HeERO is in the final stage of phase 2, where Denmark is one out of 15 countries involved in testing and validating the standards in real conditions. eCall addresses standards for third parties as well, but as HeERO project is not yet completed, things might change and some are still undefined.

December 4th 2014 the EU parliament passed a legislation that will ensure that from March 31st 2018 all new passenger cars and light duty vehicles will be fitted with eCall[3].

The IVS in this project will make it possible for people with pre-2018 cars to have the same opportunity and the same safety net as eCall provides with an aftermarket solution.

[1] Commissions white paper: "European transport policy for 2010"
[2] HeERO – Harmonised eCall European Pilot: http://www.heero-pilot.eu/
[3] Source: http://www.europarl.europa.eu/news/en/news-room/content/20141201IPR81901/html/MEPs-back-deal-with-Council-on-automatic-emergency-call-system-for-cars

1.2. Summary

The purpose of this system is to make vehicles on the road able to call for help automatically whenever an accident happens.

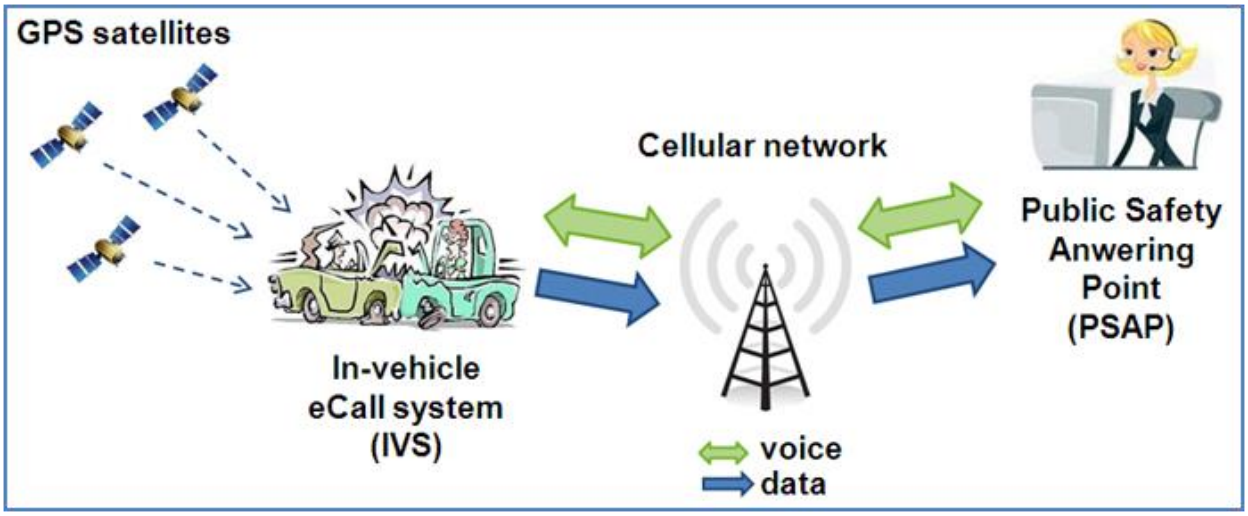


Fig. 1. VROOM communication chain architecture

The basic system architecture is based on the eCall standards and requires both voice and data communication. The data should include GPS coordinates of the vehicle's location and voice communication is essential, not only to provide psychological first aid, but also to ensure an efficient and professional expedition of the accident.

The PSAP (Public Safety Answering Point), known as "Alarm Centralen" in Denmark, is target to the common emergency number in Europe, 112.

The IVS (In-Vehicle System) is the system mounted in the car which actually detects an accident and automatically calls PSAP for help. It also includes a panel for manual activation of the system.

This project revolves around making the IVS. It will be a prototype only and not mounted in a real car. It will be demonstrated on an RC (remote controlled) car and all communication with PSAP is simulated.

1.3. Problem Formulation

Developing a system which can handle incidents solely on feedback from vehicles raises some problems of different nature, such as:

eCall standards:

- Which standards are required to comply with eCall?
- Which type of communication channels should be opened to PSAP?
- Which data is sent to PSAP?

Emergency detection:

- What type of incidents should the IVS handle?
- Should the driver be able to call for help manually?

Handling of data:

- How does the IVS know when an accident occur?
- How does the IVS determine the location of the car?
- How does the IVS record data?
- How does the IVS send data to PSAP?

Human interface:

- How is the driver notified when the emergency call is successful?
- How is the driver informed of the IVS' status?
- How should the driver be able to communicate with PSAP?

Safety:

- How to avoid false alarms?

1.4. Delimitations

Following is delaminated in this project:

- Real communication with PSAP will not be implemented
- Only a rough prototype will be developed as a proof of concept thus:
  - Material of the ECU casing will not be considered
  - Placement of the ECU will not be considered
  - Making the ECU cost effective will not be considered
  - Backup power will not be considered

2. Analysis

2.1. Overall System Architecture

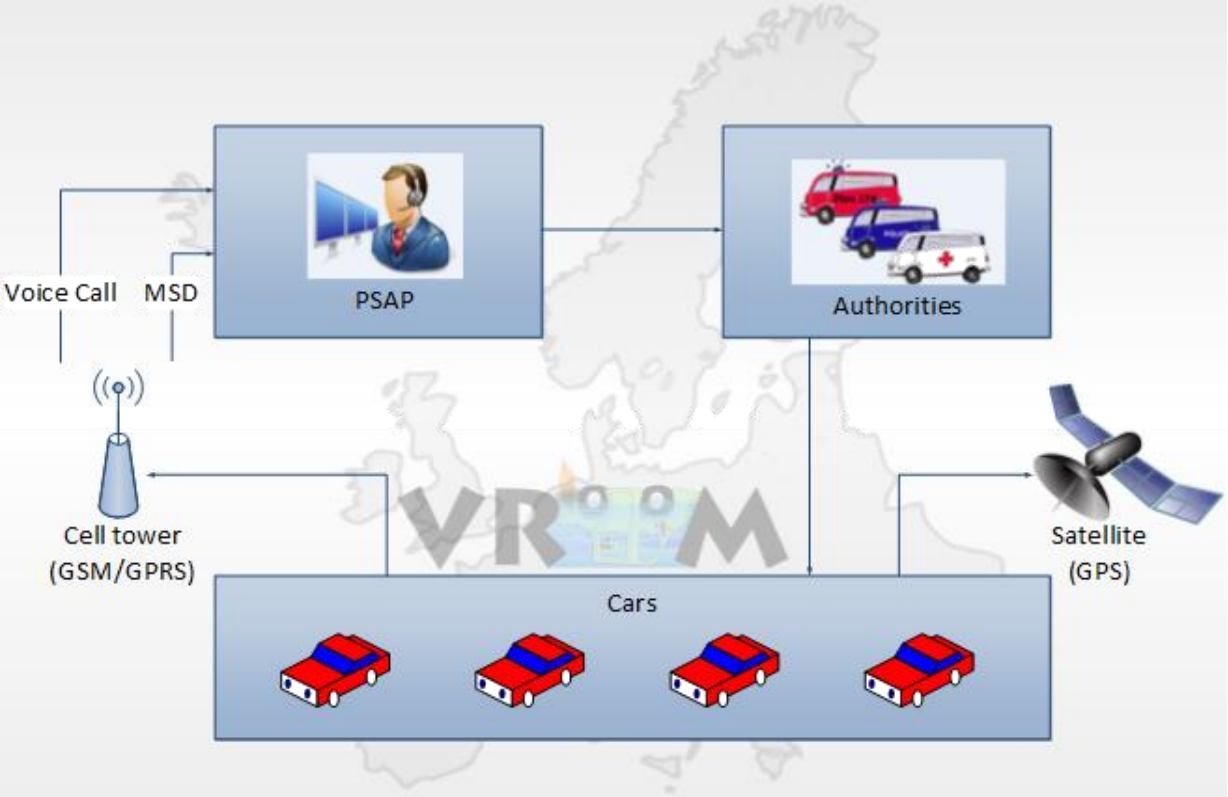


Fig. 2. VROOM system architecture

The VROOM IVS, uses mobile communication technology in order communicate with PSAP. In case of an incident, voice communication is established via GSM network, while an MSD (Minimum Set of Data) packet on 140 bytes is sent to PSAP using GPRS. The MSD includes information about the incident, vehicle location etc. which requires that IVS uses GPS technology. When MSD is successfully received, the PSAP is capable of dispatching the proper help for the involved vehicle.

2.2. UseCase Diagram

The core functionalities of VROOM is to automatically detect when an accident occur and automatically contact PSAP. This means that there are almost no conventional UseCases started by human interactions. However, the few there are, still have very important functionalities. The IVS Sensors are the sensors which automatically trigger when an accident occur.

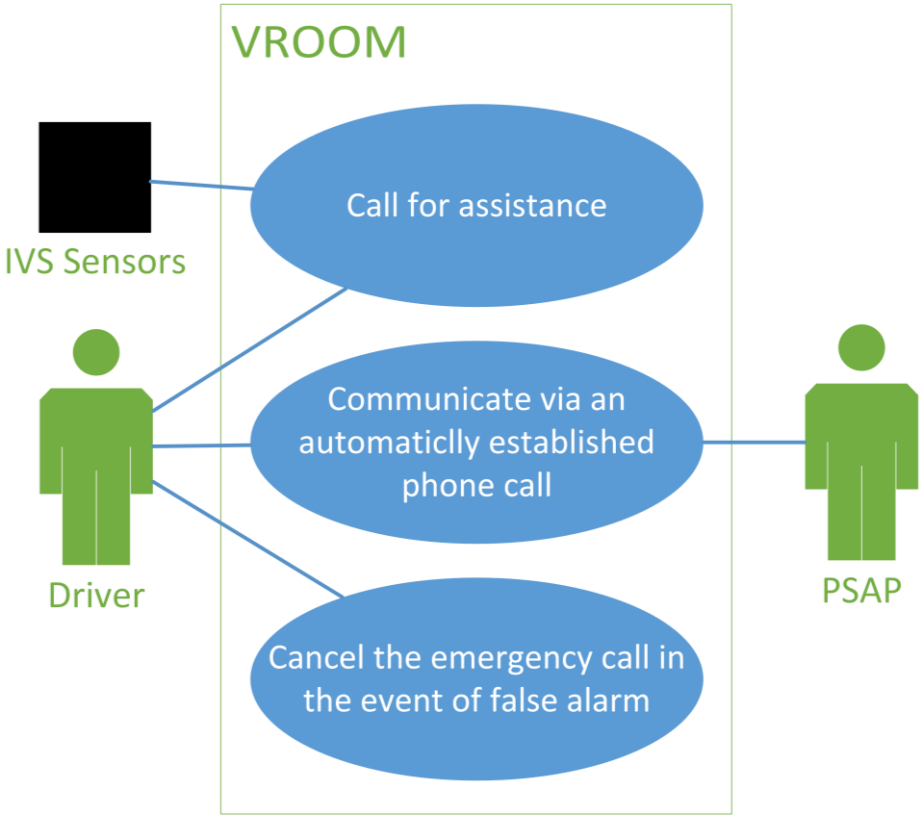


Fig. 3. UseCase Diagram

2.3. Actor Descriptions

These describe in detail the actors, who interacts with VROOM.

Name	Driver
Description	A person driving a vehicle fitted with the WROOM system
Attributes	**Person** / Machine, **Primary** / Secondary, **Active** / **Passive**

Name	PSAP
Description	Personnel at the PSAP
Attributes	**Person** / Machine, Primary / **Secondary**, Active / **Passive**

Name	IVS Sensors
Description	Sensors detecting an accident. Part of the IVS
Attributes	Person / **Machine**, **Primary** / Secondary, **Active** / Passive

2.4. UseCase Descriptions

These describe in detail the conditions, in how and when each UseCase is stared and executed.

Item	Value
UseCase	**Call for assistance**
Actor	Driver and IVS Sensors
Summary	**Driver:** is able to make an emergency call in case of an not-automatically detectable incident or accident, which requires assistance from PSAP **IVS Sensors:** automatically calls for assistance when an accident is detected
Pre-condition	An incident or accident has happened
Post-condition	Automatically establishes voice communication and sends an MSD packet to PSAP
Base Sequence	
Branch Sequence	If the driver cancels the emergency call within the predefined time span for cancelling an emergency
Exception Sequence	

Item	Value
UseCase	**Cancel the emergency call in the event of false alarm**
Actor	Driver
Summary	The driver must be able to cancel an alarm if the alarm was triggered by accident or if there is no need for further assistance. This applies both if the alarm is triggered automatically or manually
Pre-condition	An alarm is triggered while no communication is established yet
Post-condition	The alarm is cancelled. No voice and/or data communication is established to PSAP
Base Sequence	
Branch Sequence	If the driver does not cancel the emergency call within the predefined time span, PSAP is contacted
Exception Sequence	

Item	Value
UseCase	**Communicate via an automatically established voice communication**
Actor	Driver and PSAP
Summary	A voice communication is established when an incident is registered either automatically or manually
Pre-condition	An emergency call has been triggered and not cancelled
Post-condition	Voice channel between the driver and PSAP is established
Base Sequence	
Branch Sequence	
Exception Sequence	If no mobile connection or if driver is unable to communicate

2.5. Requirements

2.5.1. Functional

- The IVS must switch on with the vehicle's ignition
- The IVS must be able to detect a crash involving the vehicle
- The IVS must be able to detect a fire in the vehicle
- In case of an emergency the IVS must establish following communication to PSAP:
  - Voice
  - Data
- The following data must be recorded from sensors:
  - Vehicle location
  - Crash data
  - Cabin temperature
- The driver must be able to call for emergency manually
- The driver must be notified when an emergency call has been performed successfully
- The driver must be able to cancel the alarm in case of false alarm
- The driver must be informed of the IVS' operational state

2.5.2. Non-functional

- MISRA C – 2004 standards[4] for software developing to vehicles must be followed
- eCall standards[5] for pan-European in-vehicle emergency call service must be followed

2.6. Overall System Infrastructure

This activity diagram shows the infrastructure of the IVS. It shows possible branches and the life cycle of the whole system.

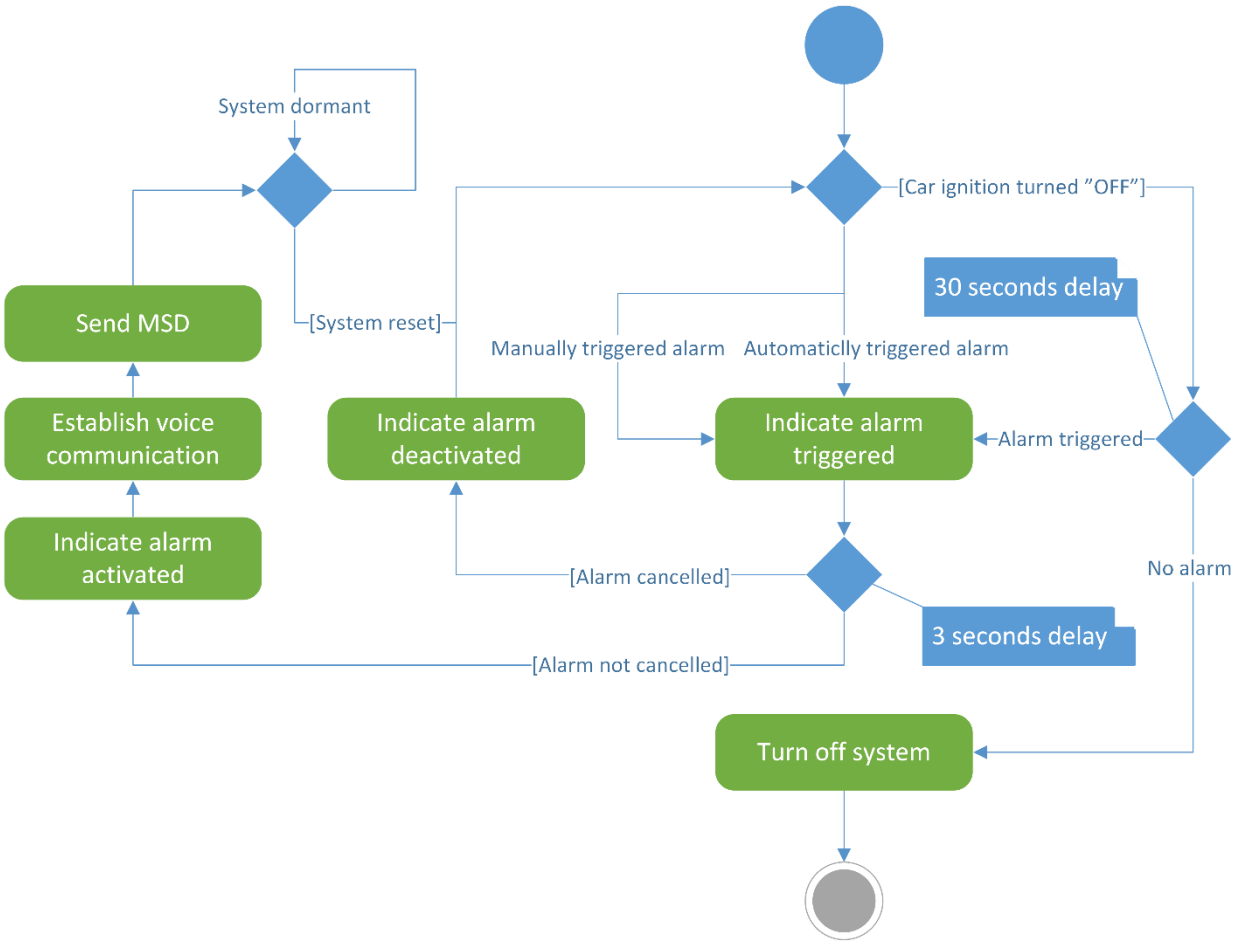


Fig. 4. Activity Diagram – Overall system infrastructure

[4] MISRA C: http://www.misra-c.com/
[5] HeERO Pilot: http://www.heero-pilot.eu/

2.7. Accident Definition

It is important to clarify exactly what constitutes an accident, as this is the core functionality of the system. An accident has happened if:

- the vehicle drives into something that triggers the airbag
- the vehicle is hit by an object with a force in excess of $7G - 20G$ for at least $20ms$[6]
- the temperature in the cabin raises in excess of $1℃$ per 8 seconds[7] (this would be interpreted as fire in the vehicle)

As this system is intended to not interact with the cars internal components, it will not trigger when an airbag deploys. It will instead measure the force of the crash itself.

2.8. Crash Data

To determine whenever an event is considered an accident, accelerometer measurements in different conditions and situations have been done. Crash data is measured in a real car. For this purpose an analysis platform, with a graphical user interface, has been developed (ref. 5.1.1) and the recorded data have been analysed in order to determine a threshold. This threshold determines the difference between a crash and a hard manoeuvre.

By using the accelerometer data and seeing it as a three dimensional vector, it is possible to calculate the total acceleration in an arbitrary direction by using the equation:

$$V_{ACC\_total} = \sqrt{x^2 + y^2 + z^2}$$

This makes it possible to eliminate the complexity of installing the accelerometer in a certain direction as the direction does not influence the total acceleration.

As the gravity is a constant $1G$, it is possible to use that as a baseline for measurements.

The graphs should be read as:

Horizontal axis: time in $9ms$ steps

Vertical axis: acceleration in $1G$ steps

Graph colours:

- Red – x-axis
- Green – y-axis
- Blue – z-axis
- Yellow – Total length of 3D vector

[6] Airbags and Crash sensors: http://www.aa1car.com/library/airbag01.htm
[6] Deflated expectations: http://www.drive.com.au/motor-news/deflated-expectations-20100331-refb.html
[7] Heat detectors: http://en.wikipedia.org/wiki/Heat_detector

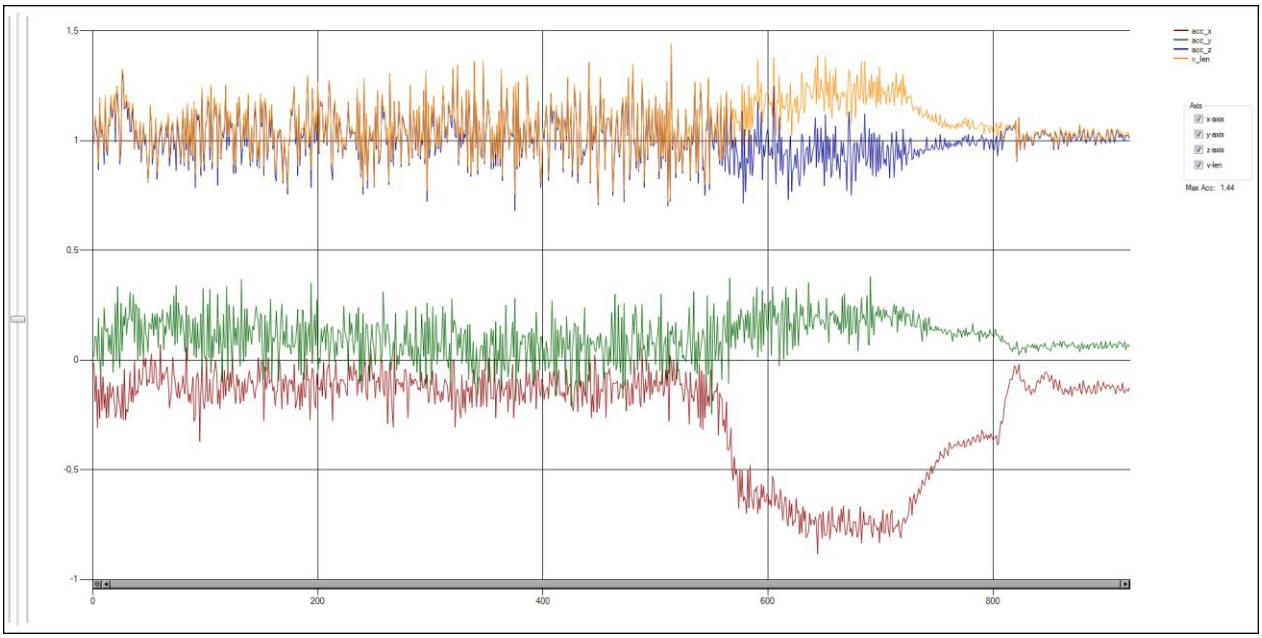


Fig. 5. Acceleration Graph – Emergency breaking 40 - 0 km/h

It is clear that there is almost no change to the total acceleration when breaking from this speed. With an acceleration max of $1.44G$ it is almost within the normal noise of the accelerometer.

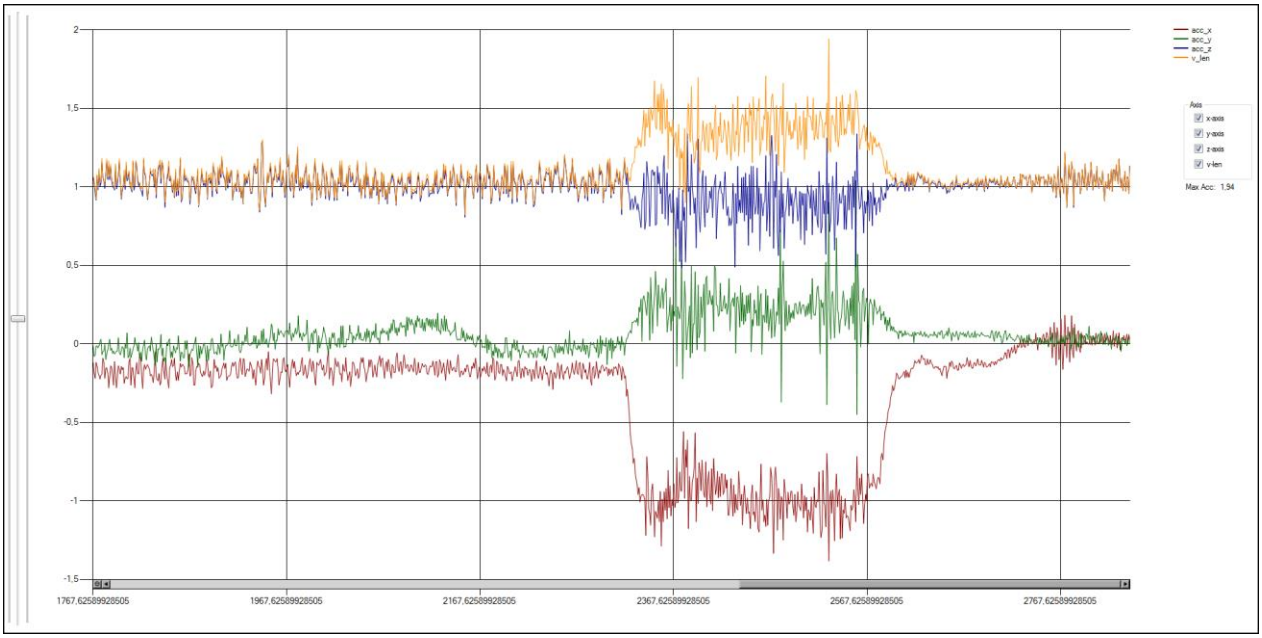


Fig. 6. Acceleration Graph – Emergency breaking 90 - 0 km/h

At 90 km/h there is a much larger difference. There is now a max acceleration of $1.94G$. This is significantly more than gravity alone.

Also other shock factors should be considered, like a door being slammed.

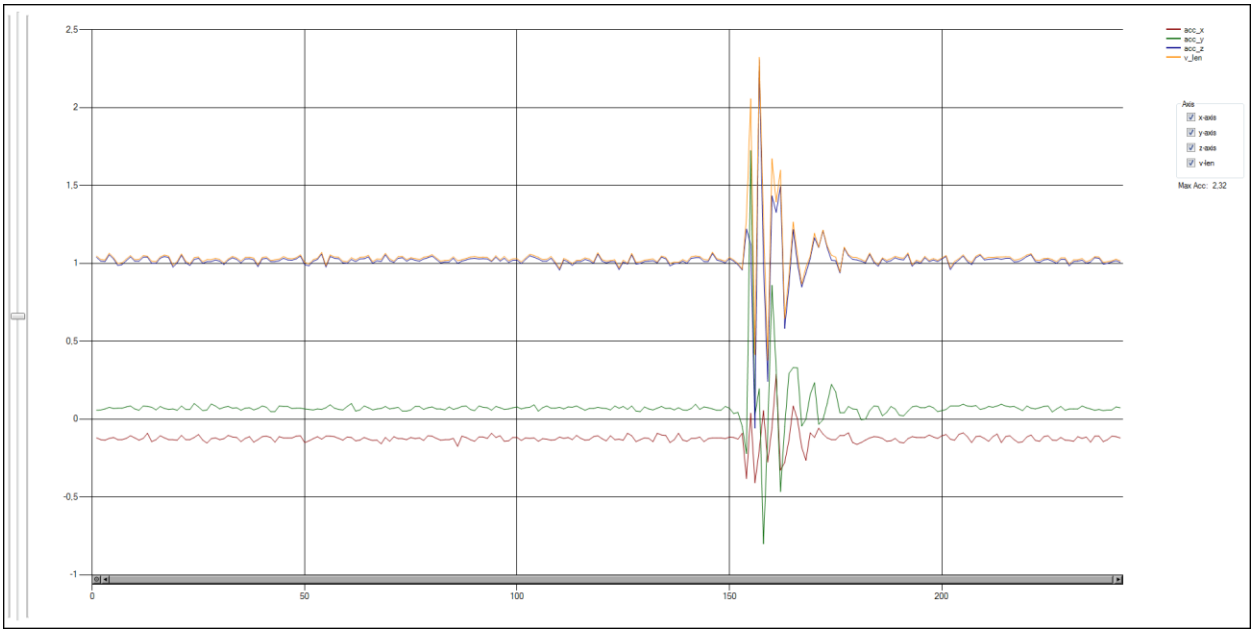


Fig. 7. Acceleration Graph – Slamming car door

Now the max acceleration is peaking at $2.32G$.

As the research showed (ref. 2.7) that most airbags trigger between $7G$ and $20G$, $10G$ is chosen as the trigger value. It is high enough so that it does not trigger at normal to aggressive driving, but low enough to be within the accepted range of a crash. Also there seems to be a consensus that a crash spans over a period of about $20ms$.

With this data in hand, it is decided to use the following parameters to conclude if a crash has happened:

$$Acceleration \geq 10G$$

$$Time\ of\ event \geq 20ms$$

This denotes the parameters to the detection algorithm.

A selection of recorded accelerometer data can be found:

…\Testing and Analysing data\Accelerometer Analysing\...

2.9. Voice Communication

As no test calls is allowed to PSAP, voice communication is simulated using a mobile phone. Although a SIM card is not required to call 112 in Denmark, it is required according eCall standards, as not all countries in Europe supports emergency calls without a SIM card.

2.10. Data Communication

The exact protocol is not available and no PSAP servers are accessible and are therefore out of scope of this project. However, the data structure is prepared so it complies with eCall standards. An MSD (Minimum Set of Data) with a specific structure is sent via GPRS technology.

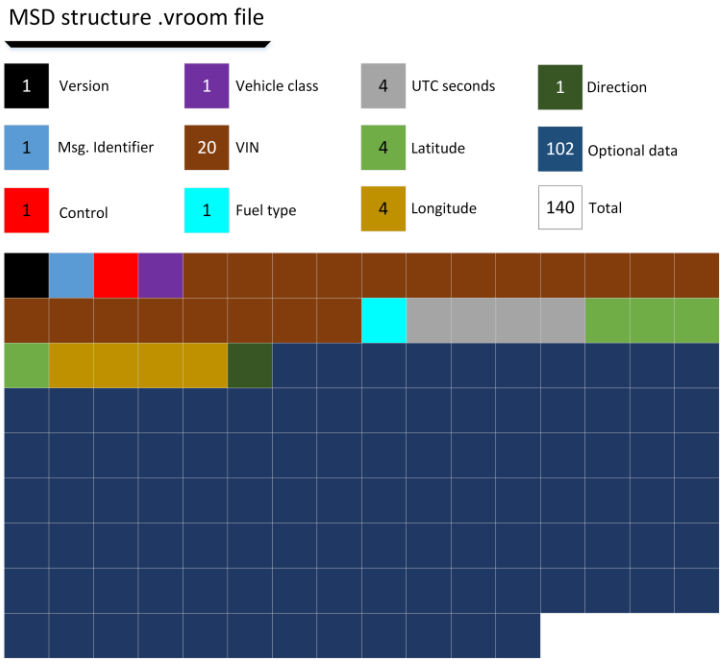


Fig. 8. MSD structure

The size of the structure is fixed 140 bytes. This means that unused bytes is assigned with the value 0x20 (space). The structure in detail are defined in EN 15722.

Name	Size	Type	Description
Version	1	Byte	Encoding optional data reference
Msg. Identifier	1	Byte	Numbers of re-transmissions
Control	1	Byte	Bit representation: 7: Automatic activation; 6: Manual activation; 5: Test call; 4: Confidence in position; 3–0: Reserved
Vehicle class	1	Byte	Bit representation: 7–4: Classification; 3–0: Category
VIN	20	String	Vehicle identification number according ISO 3779

Fuel type	1	Byte	Value representation: 0 Other; 1 Gasoline; 2 Diesel; 3 Naturalgas; 4 Propane; 5 Electric; 6 Hydrogen
Time stamp	4	Integer	UTC seconds (value >= 0)
Latitude	4	Integer	Latitude (WGS-84) in milliarcseconds
Longitude	4	Integer	Longitude (WGS-84) in milliarcseconds
Direction	1	Byte	Direction in degrees. The nearest integer of 255*value/360
Optional data	102	String	Further data (e.g. crash data, number of passengers) or blank field
Total bytes:	140		

3. Design

3.1. In-Vehicle System (IVS)

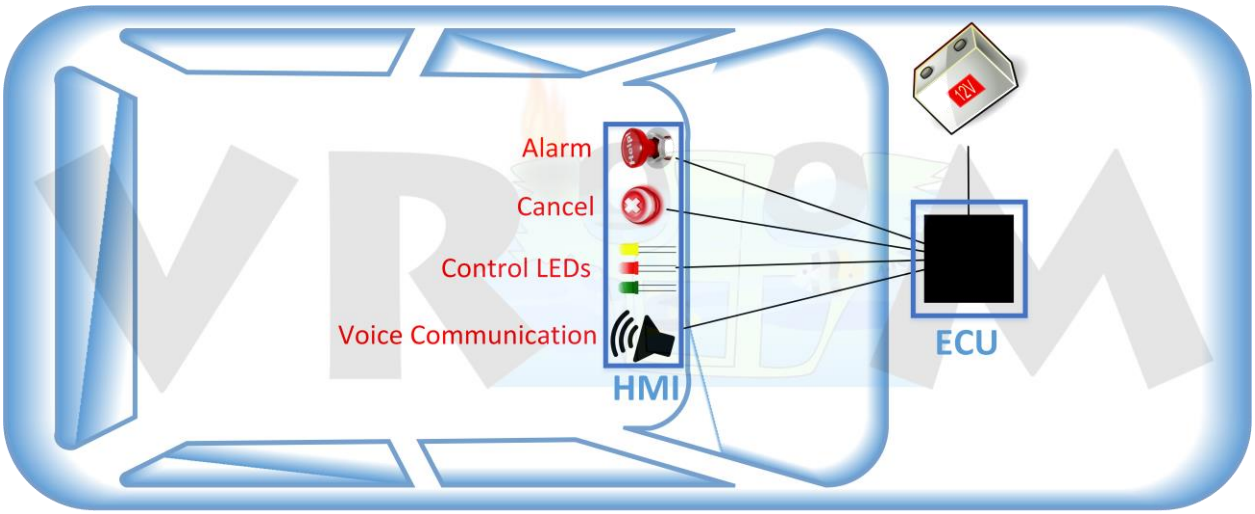


Fig. 9. In-Vehicle System (IVS) – Design

The IVS consists of two parts:

ECU (Electronic Control Unit) which is the core of the system and considered a black-box with no access for unauthorised technicians. The ECU must be installed a secure place with minimal chance of damage in the event of an accident.

HMI (Human Machine Interaction) which is the user interface. The HMI should be installed in the cabin of the vehicle close to the driver.

3.2. Electronic Control Unit (ECU)

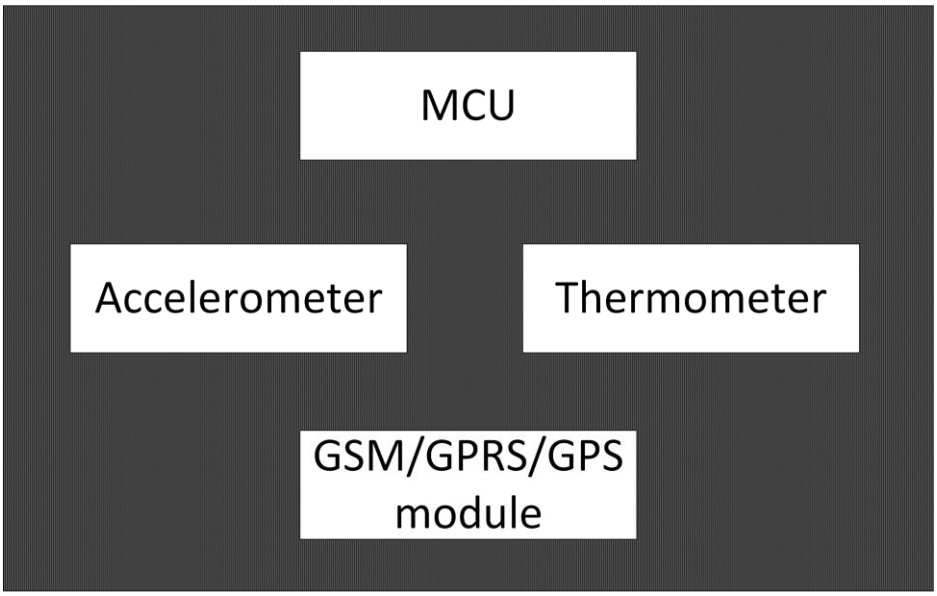


Fig. 10. Electronic Control Unit (ECU) – Design

The ECU consists of different hardware components to accommodate with requirements for this system:

GSM:	Used for voice communication
GPRS:	Used for data communication
GPS:	Used to determine the location of the car
Accelerometer:	Used to measure the acceleration in three dimensions
Thermometer:	Used to determine the cabin temperature
MCU:	Control unit for the hardware

3.3. Human Machine Interaction (HMI)

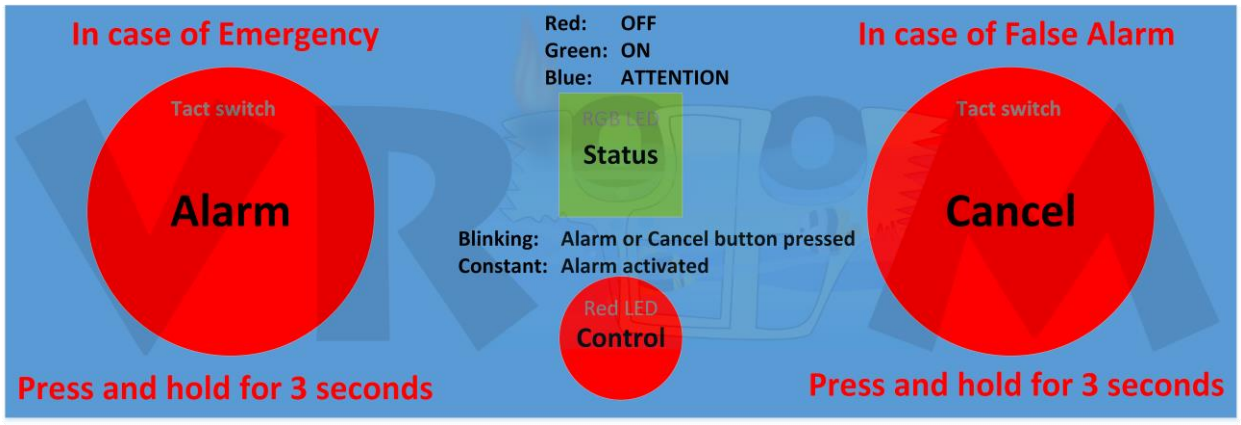


Fig. 11. Human Machine Interaction (HMI) – Overview

The HMI consists of the Car Panel with a three-coloured LED which indicates the operational state of the system and a red control LED to indicate when an emergency call is activated. The user is able to contact PSAP manually by pressing the "Alarm" button for at least 3 seconds and the user has the option to cancel an emergency call in the event of false alarm by pressing the "Cancel" button for at least 3 seconds.

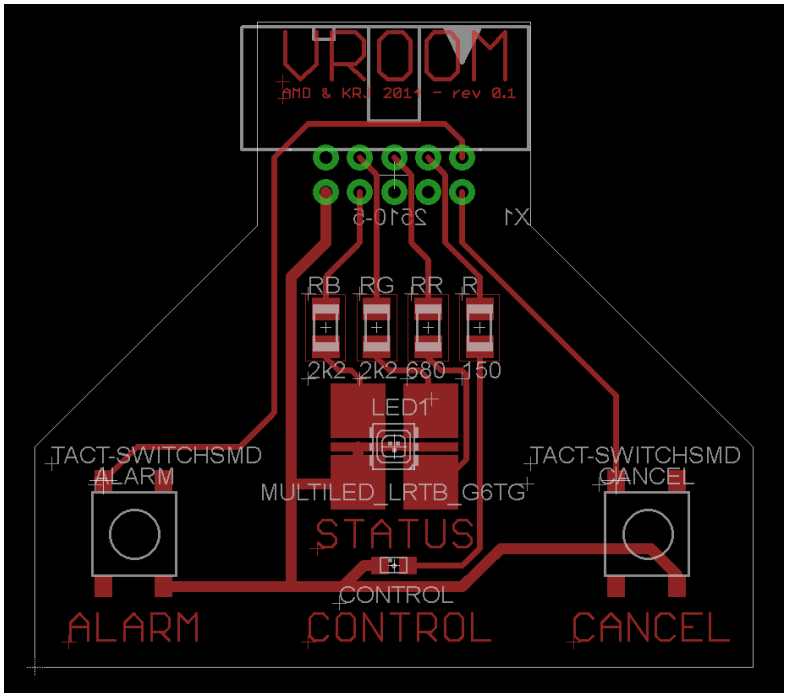


Fig. 12. Human Machine Interaction (HMI) – PCB

A User Guide for VROOM car panel can be found:

…\Documentation\Project Report\Appendices\Appendix D – User Guide VROOM Car Panel.pdf

3.4. Technical Choices

To develop this system some decisions and technical choices have been made both in hardware and software perspective. The decision on hardware is only for prototyping. The ideal use of hardware is to develop a custom board with only the required functionalities. To research and develop a cost efficient module is out of scope for this project.

3.4.1. Hardware decisions

Hardware	Model	Description
MCU	ATMega2560	Low-power 8-bit AVR RISC-based MCU with 256 kb flash memory. Prototype is developed on a STK600 board
GSM GPRS GPS	SIM908	SIM908 provides both GSM, GPRS and GPS. Communication with the chip is done using AT commands via UART. The chip is mounted on a module designed as an Arduino shield and includes on-board audio amplification.
Accelerometer	LIS331HH	3-axis accelerometer
Thermometer	TC72	Digital thermometer
Car Panel HMI	VROOM rev. 0.1	Self-developed prototype with tact-switches and LEDs
LCD Display	HD44780	LCD display used for debugging

It is important that the MCU is low power as it is going to operate in a car, where there is limited power. Here the ATMega 2560 works great, plus it has a vast variety of inputs and interrupts, which makes it a nice platform to develop on.

As this system relies on positioning and mobile communication, both GPS, GSM and mobile data (GPRS, EDGE, 3G or LTE) is needed. The SIM908 has all of it included and seems to be a good module to develop on. It is relative cheap and it is available as a development board.

An accelerometer with a full-scale of at least $\pm 20G$ is needed, the LIS331HH was chosen. It has a full-scale of $\pm 24G$. It uses an SPI interface and is well documented.

As the requirements for the thermometer are relative limited, the TC72 is chosen, as it was already at hand.

The car panel is developed and produced based on the requirements for the HMI.

3.4.2. Software Decisions

VROOM is written in C and complies with most of the rules in MISRA C 2004 coding standards, which is used in the automotive industry.

It is decided not to develop this system using an operating system. This choice is based on the fact that developing the whole framework, gives a better understanding of how things cooperate together.

The way the system is built is based around manual scheduling. Manual scheduling has a number of pros:

- No resource will be starved so all data can be trusted
- The system is highly predictable so it is easy to work around
- There is practically no overhead, so timing is easily calculated

The dangerous thing about using manual scheduling is that the effectiveness is solely dependent on the implementation.

It is a cyclic executive in the way that all tasks are run after a specific premade table.

The scheduler allows all tasks to be interrupted. This is intended, as the only thing that will interrupt the scheduler, is the event of an alarm. In this case, the scheduler is halted and not resumed until the alarm is cleared.

Hence the system runs as a foreground/background system in the way that the scheduling is running in the background and the emergency reporting is interrupt driven and running in the foreground.

4. Implementation

4.1. Introduction

4.1.1. Hardware

The system is built from different modules.

- The MCU is on an STK600 development board as this is easy to build the test platform up around
- The SIM908 module is on an Arduino shield as it has the needed peripherals on the board
- The LIS331HH accelerometer and TC72 thermometer is mounted on an airplane shaped PCB board as it was currently available
- There is an LCD board for debugging
- There is a prototype HMI panel

Detailed description of the port mapping for all the hardware components can be found here.

…\Source Code\MemoryMap.xlsx"

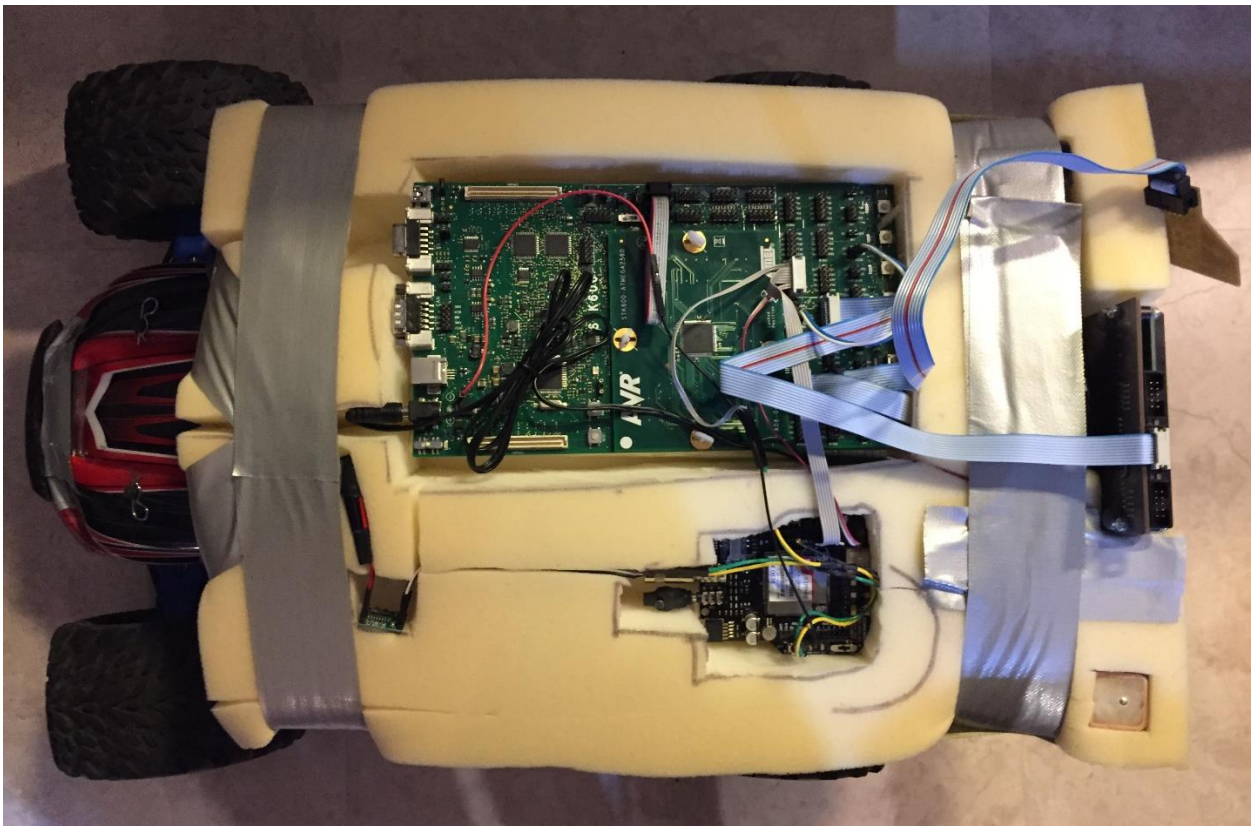


Fig. 13. Test Platform – Overall

It is all mounted on an RC car for testing.

Fig. 14. Test Platform – Accelerometer and thermometer

The accelerometer is mounted firmly on the chassis of the RC car to make sure it measures the impact force of the car.

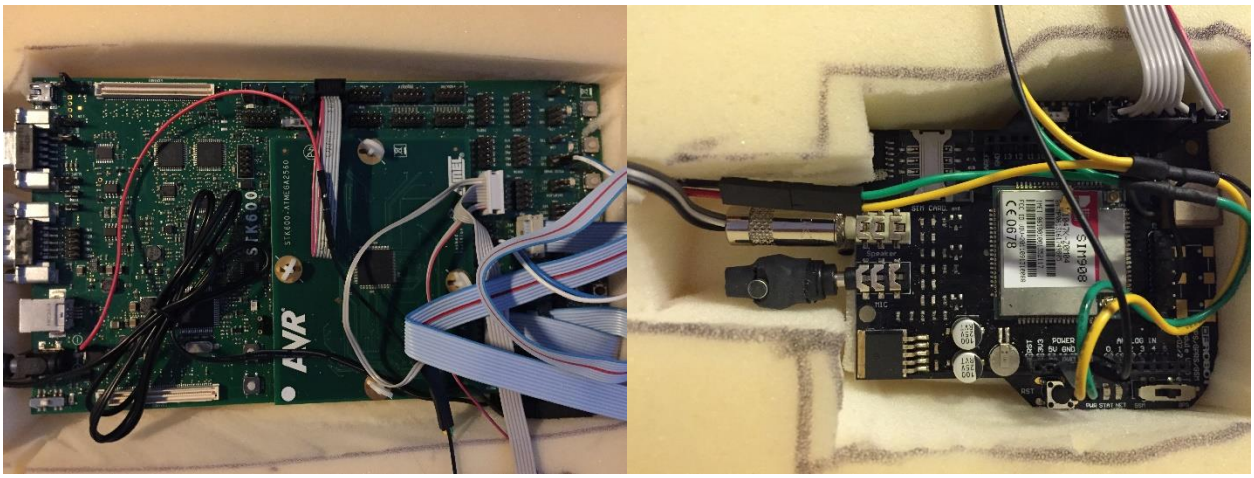


Fig. 15. Test Platform – Modules

It is all packed in foam to protect it from damage when testing.

Fig. 16. Test Platform – Debug and prototype boards

4.1.2. Software

The complete system consists of 6166 lines of code and takes up 14442 bytes of program memory on the MCU.

It has been a priority to keep the code clean and easy to read, modularized, reusable and above all efficient.

This has resulted in separated files for specific purposes, like driver for the SPI and UART but also for hardware elements like the car panel, the accelerometer and the thermometer.

The code is clean in the way that the structure is hierarchical and indented so that it does not clutter. Meaningful naming is used throughout to make sure the code is self-explanatory, to some extent. Finally it is thoroughly documented according to the doxygen standard to make it easy for anyone to dig into and figure out how a specific function or module works.

The reason behind making the code modular is that as this is only a prototype, it is very likely that some of the components need to be replaced with better or cheaper ones. Here the modular design makes it easy to replace components as it is only the component specific file(s) that needs to be re-written. The only downside to this is that it might take up more program memory.

The internal communication devises (UART and SPI) have universal drivers so they are easily reconfigurable in other contexts.

The code is written as efficient as possible, with the available knowledge, to make it possible to run on a cost effective MCU, although this research is not covered in this report, it is still important to take this into account when programming.

4.1.3. File Diagram

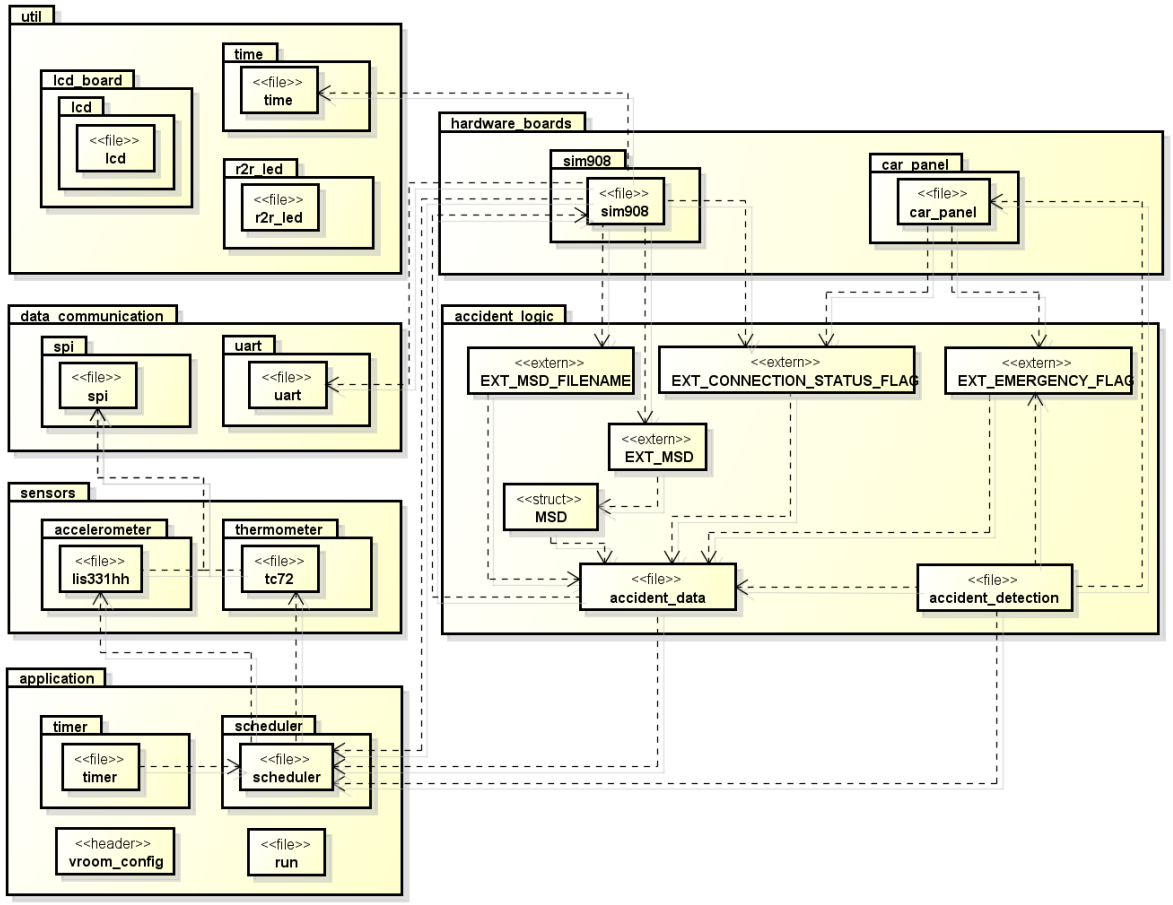


Fig. 17. System File Diagram

application contains the main file and a config header file, which holds all configurations for the system and porting. A scheduler is implemented as a state machine, so that shared recourses are protected. The scheduler is started once by a function call in the initialisation phase and is continues started by a timer.

sensors consist of an accelerometer and a thermometer. Both access the same communication interface, SPI.

data_communication contains SPI and UART modules. SPI is used for sensor communication and UART for SIM908 module communication using AT commands. A general driver for UART0 and UART1 is implemented in order to support loopback to PC using RS232.

hardware_board contains the SIM908 module, which is the communication part of the IVS. It also contains the Car Panel which is the HMI part of the IVS. Car Panel handles the support for manually setting an alarm, cancel the alarm or reset emergency detection, in case of false alarm. It also visualise the connection status of the system.

accident_logic holds the algorithm for automatically crash and fire detection. This module also holds the MSD structure and is responsible for collecting all data needed from the respective modules.

util contains LCD driver used for debugging and an R2R driver for duty time analysing. As the LCD is only used as a tool for debugging, and not a part of this system, it is decided to use an already existing driver written by Peter Fleury.

Detailed description of all functions can be found in doxygen documentation.

…\Documentation\Doxygen\index.html

4.1.4. Code Structure

The following example displays the general code structure used in this project. The example is from a function in accident_data.c, which is called whenever an accident or incident occurs. It inherits information from different modules and assembles it in a global structure `EXT_MSD`.

```
void ad_emergency_alarm(void)
{
   EXT_MSD.version = CONFIG_MSD_FORMAT_VERSION;
   EXT_MSD.vehicle_class = CONFIG_MSD_VEHICLE_CLASS;
   EXT_MSD.fuel_type = CONFIG_MSD_FUEL_TYPE;

   set_MSD_data(&EXT_MSD.time_stamp, &EXT_MSD.latitude, &EXT_MSD.longitude, &EXT_MSD.direction);

   _confidence_in_position = (EXT_MSD.latitude != 0 || EXT_MSD.longitude != 0) ? true : false;
   _set_control_byte( _confidence_in_position,
                      CONFIG_MSD_TEST_CALL,
                      EXT_EMERGENCY_FLAG == EMERGENCY_MANUAL_ALARM,
                      EXT_EMERGENCY_FLAG == EMERGENCY_AUTO_ALARM );
   _set_VIN(CONFIG_MSD_VIN);
   _set_optional_data();

   send_MSD(CONFIG_VROOM_ID);

   call_PSAP();

   EXT_EMERGENCY_FLAG = EMERGENCY_ALARM_SENT;
}
```

The function `set_MSD_data(...);` is called in SIM908 module driver, to get and set the GPS data needed. All other information for MSD are collected from global flags or configuration defines and set locally in this file using static functions.

4.2. Start-up and Main Routine

4.2.1. Overview

The run.c file holds the *main* function of the system, which are where the program starts its execution.

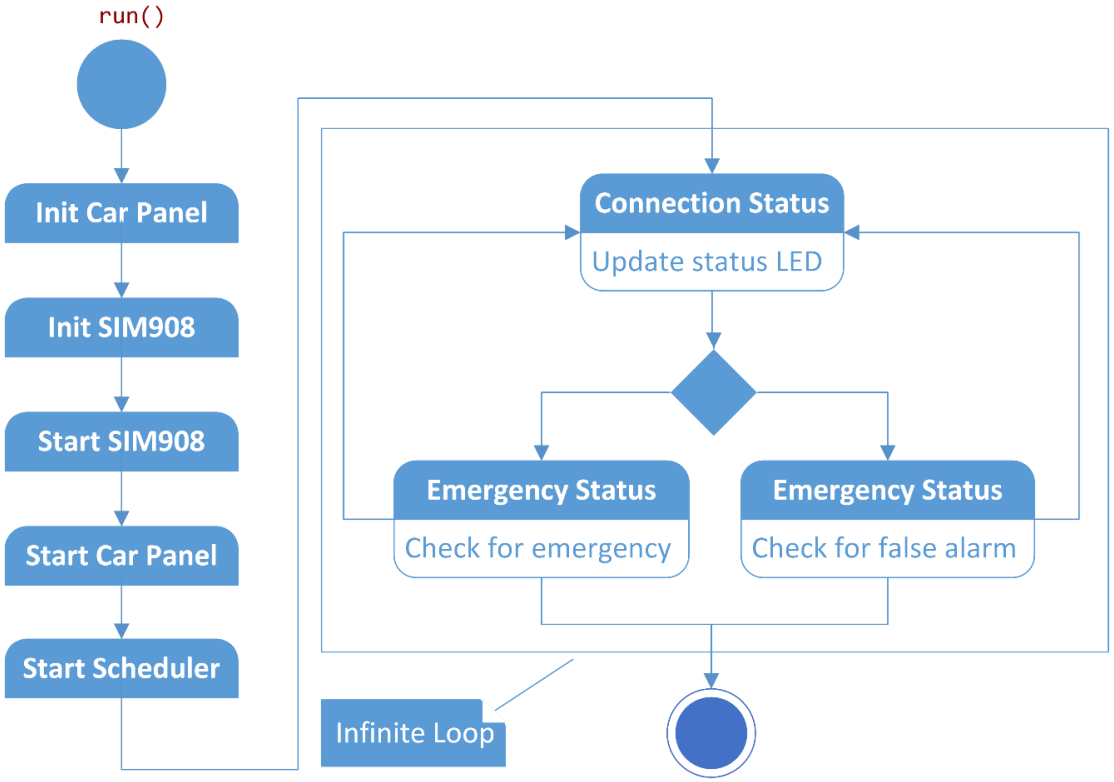


Fig. 18. State Machine Diagram – System initiation and main routine

The *main* function is straight forward and has very little complexity, besides that, it can be configured to execute different contexts of the program. These contexts includes or excludes different tests by setting the macro to the state ON or OFF, in top of the run.c file. This design makes the code well-structured and readable while it makes it easy to test the respectively modules separately.

The actual program is the Integration test, as it integrates all modules in the system. To run this context it must be defined ON while all other contexts must be defined OFF.

```
#define UNIT_TEST                                   OFF

#define MODULE_TEST_SENSORS                         OFF
#define MODULE_TEST_SIM908                          OFF
#define MODULE_TEST_CAR_PANEL                       OFF
#define MODULE_TEST_UART                            OFF

#define INTEGRATION_TEST_SIM908_SENSORS             ON
```

Integration test context is explained in following section.

4.2.2. Code

First the hardware boards are initiated and afterwards started. This is done to ensure nothing is interrupted, as global interrupts are first enabled after the initiation.

The SIM908 module is started and sets up all its configuration using AT commands via UART. (ref. 4.4) Next the Car Panel is started, which simply just enable interrupts for the Alarm button on the Car Panel board. (ref. 0)

Finally the scheduler is started, which is the basic task scheduler of the system. (ref. Basic Task

```
int main (void)
{
   car_panel_init();
   SIM908_init();
   sei();
   SIM908_start();
   car_panel_start();
   scheduler_start(NULL);

   while (1) { . . . }
}
```

Scheduling4.3**Fejl! Henvisningskilde ikke fundet.**)

The infinite loop in main is where actions are performed based on the Status flags in the system.

```
while (1)
{
      /* Sets the status LED on car panel */
   (EXT_CONNECTION_CREG_FLAG == CREG_REGISTERED_HOME_NETWORK ||
    EXT_CONNECTION_CREG_FLAG == CREG_REGISTERED_ROAMING) &&
    (EXT_EMERGENCY_FLAG == EMERGENCY_NO_ALARM ||
     EXT_EMERGENCY_FLAG == EMERGENCY_ALARM_SENT ||
     EXT_EMERGENCY_FLAG == EMERGENCY_FALSE_ALARM)
      ? car_panel_set_status(STATUS_GREEN) : car_panel_set_status(STATUS_RED);

   /* Checks the emergency flags */
   if (EXT_EMERGENCY_FLAG == EMERGENCY_AUTO_ALARM ||
       EXT_EMERGENCY_FLAG == EMERGENCY_MANUAL_ALARM)
   {
      scheduler_halt();
      ad_emergency_alarm();

      /* Enable cancel button in case of false alarm */
      car_panel_set_cancel_button_state(true);
   }

   else if (EXT_EMERGENCY_FLAG == EMERGENCY_FALSE_ALARM)
   {
      scheduler_resume(true);
      EXT_EMERGENCY_FLAG = EMERGENCY_NO_ALARM;
   }
}
```

4.3. Basic Task Scheduling

4.3.1. Overview

The basic task scheduling of the system is where data is gathered and analysed. The data must be gathered consistently and in a controlled fashion so that the data can be trusted.

When reliable data is gathered, it is important to analyse it carefully to determine if an accident has occurred.

As no operating system is used, this is all done using a scheduler. This scheduler is a state machine pattern controlled by a timer.

As the analysis of the accelerometer data and general research showed that a crash spans over $20ms$, the four main tasks run at that speed. The last task run every $8s$, as per the definition of a fire (ref. 2.7). They all have different execution times.

The state machine has 9 states:

1. Initialise thermometer
   - Sends the setup address and parameters to the sensor via the SPI bus
2. Initialise accelerometer sensor
   - Sends the setup address parameters to the sensor via the SPI bus
3. Initialise timer
   - Sets up the registers in the MCU
4. Idle
   - Idle state for when not working
5. Request temperature read
   - Sends address and dummy bytes to the sensor for requesting temperature reading
6. Request accelerometer read
   - Sends address and dummy bytes to the sensor for requesting accelerometer readings
7. Get and store the requested temperature and accelerometer readings
   - Gets the readings from the sensors and store them locally in a variable / buffer
8. Check if a crash has occurred
   - Analyses the accelerometer readings to determine if a crash has occured
9. Check if a fire has errupted
   - Analyses the temperature readings to determine if a fire has errupted

States 1-3 are entered on the startup of the system only. States 3-9 are entered throughout the life cycle of the system, but can be halted.

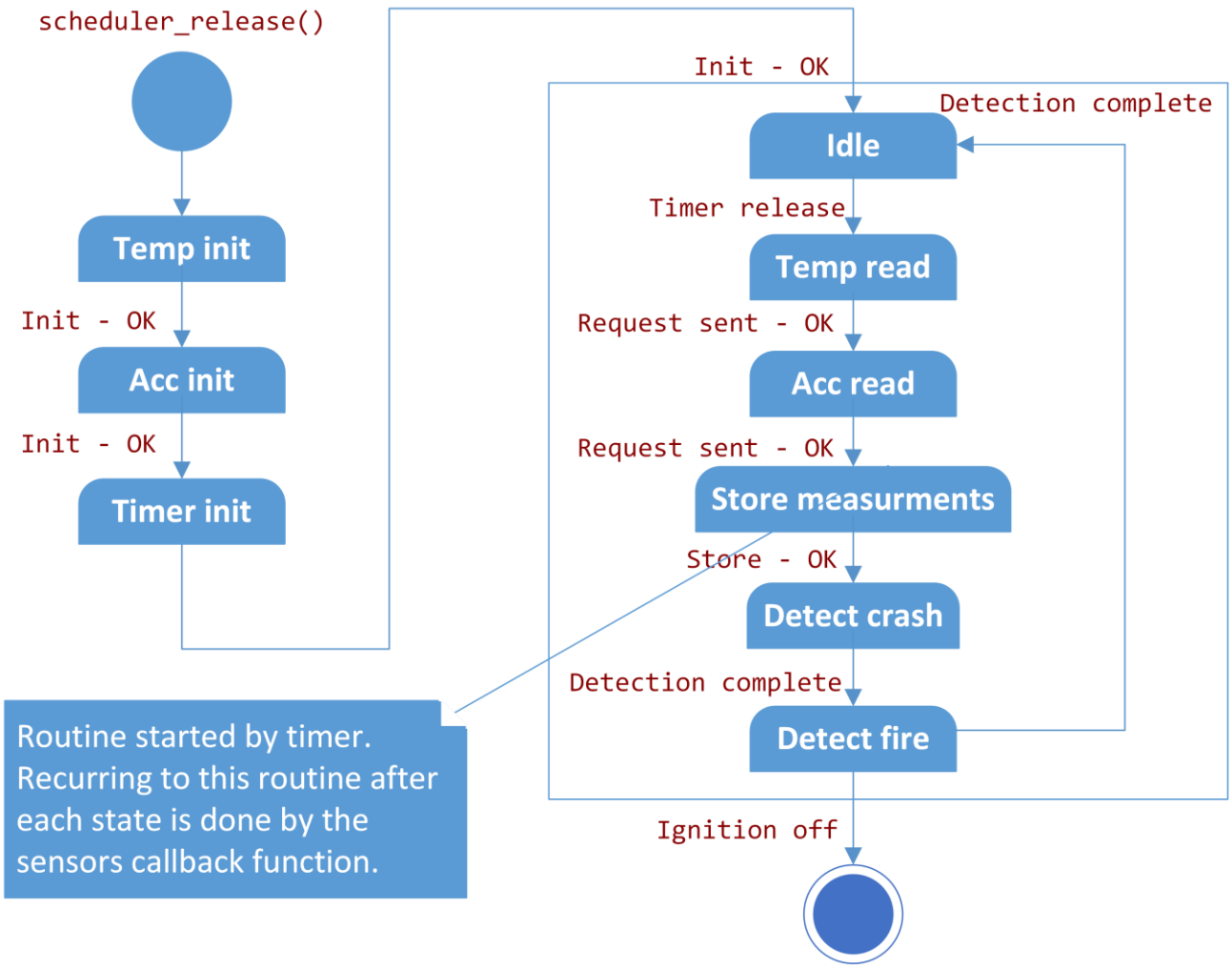


Fig. 19. State Machine Diagram – Task scheduling

4.3.2. Code

The scheduler is very basic. It runs cyclic execution so that everything is completely predictable and has minimal overhead. It supports interruption so that the emergency reporting can happen immediately, when an emergency has been detected.

When started, it sets the state to the first desired state, saves a pointer to a callback function (not used at this time) and releases the scheduler; essentially starting the system core.

```
void scheduler_start(void (*callback_function_ptr)(char __data)) {
        _state = state_tc72_init;
        _callback_function_ptr = callback_function_ptr;
        scheduler_release();
}
```

The scheduler_release function is where the state machine is located.

Here are the first 3 states which is where the initialiasation of the thermometer, accelerometer sensor and timer is done.

```
void scheduler_release(void) {
   switch(_state) {

     /*************************************************************************/
     /* Init state for temperature sensor                                     */
     /*************************************************************************/
     case state_tc72_init :
        _state = state_acc_init;
        init_tc72(PB4);
     break;

     /*************************************************************************/
     /* Init state for accelerometer sensor                                   */
     /*************************************************************************/
     case state_acc_init :
        _state = state_timer_init;
        acc_init(PB0, ACC_NORMAL_MODE, ACC_ODR_400, ACC_24G);
     break;

     /*************************************************************************/
     /* Init state for timer                                                  */
     /*************************************************************************/
     case state_timer_init :
        _state = state_idle;
        timer1_init_CTC(TIMER_PS256, CONFIG_SCHEDULER_FREQUENCY);
     break;
```

The two first states (thermometer init and accelerometer init) are both dependant on completion as they both use the SPI bus. This means that once all of the setup bytes have been successfully sent, the driver releases the scheduler.

```
static void _tc72_callback(uint8_t __data[]) {
    switch (_state)    {
    case INIT:
        _state = RUNNING;
        scheduler_release();
        break;
```

After the timer has been initialised, the scheduler enters the idle state meaning that the system is now running its core routine.

This idle state only sets up the next state.

```
/*************************************************************************/
/* Main routine starts here.                                             */
/*                                                                       */
/* Idle state                                                            */
/*************************************************************************/
     case state_idle :
         _state = state_tc72_read;
     break;
```

When the timer triggers, it releases the scheduler.

```
/*************************************************************************//**
 * @ingroup timer
 * @brief Releases the scheduler when the timer hits the TOP value
 *************************************************************************/
ISR(TIMER1_COMPA_vect)
{
        scheduler_release();
}
```

The system then runs one whole cycle and ends up in the idle state once it is done.

```
/*************************************************************************/
/* Read state for temperature sensor                                     */
/*************************************************************************/
     case state_tc72_read :
#ifdef DEBUG_TASK_MEASURE
        _task_prev_id_temp = r2r_start_task(DEBUG_ID_SENSOR_SCHEDULER_TEMP_REQ);
#endif
        _state = state_acc_read;
        measure_temperature();
#ifdef DEBUG_TASK_MEASURE
        r2r_stop_task(_task_prev_id_temp);
#endif
     break;

/*************************************************************************/
/* Read state for accelerometer sensor                                   */
/*************************************************************************/
     case state_acc_read :
#ifdef DEBUG_TASK_MEASURE
    _task_prev_id_acc = r2r_start_task(DEBUG_TASK_ID_SENSOR_SCHEDULER_ACC_REQ);
#endif
        _state = state_store_in_buffers;
        acc_measure();
#ifdef DEBUG_TASK_MEASURE
        r2r_stop_task(_task_prev_id_acc);
#endif
     break;
```

```
/*************************************************************************/
/* State where measurements are read and stored in variables/buffers     */
/*************************************************************************/
    case state_store_in_buffers :
#ifdef DEBUG_TASK_MEASURE
        _task_prev_id_read = r2r_start_task(DEBUG_ID_SENSOR_SCHEDULER_READ);
#endif
        _state = state_idle;
        _x_axis_buffer[_acc_buffer_tail] = (int)(acc_get_x_axis() * 100);
        _y_axis_buffer[_acc_buffer_tail] = (int)(acc_get_y_axis() * 100);
        _z_axis_buffer[_acc_buffer_tail] = (int)(acc_get_z_axis() * 100);
        _acc_buffer_tail = (_acc_buffer_tail + 1) % CONFIG_ALARM_CRASH_NO_OF_READINGS;

        _temperature = get_temperature();

        _state = state_detect_accident;
#ifdef DEBUG_TASK_MEASURE
        r2r_stop_task(_task_prev_id_read);
#endif
        scheduler_release();
    break;

/*************************************************************************/
/* State where an accident is detected                                   */
/*************************************************************************/
    case state_detect_accident :
#ifdef DEBUG_TASK_MEASURE
        _task_prev_id_crash_det = r2r_start_task(DEBUG_ID_ACCIDENT_CRASH_DETECTION);
#endif
        _state = ++_execution_counter % (CONFIG_ALARM_FIRE_TRIGGER_TIME / (1000 /
CONFIG_SCHEDULER_FREQUENCY)) == 0? state_detect_fire : state_idle;
        check_for_crash();
        scheduler_release();
#ifdef DEBUG_TASK_MEASURE
        r2r_stop_task(_task_prev_id_crash_det);
#endif
    break;

/*************************************************************************/
/* State where fire is detected                                          */
/*************************************************************************/
    case state_detect_fire :
#ifdef DEBUG_TASK_MEASURE
        _task_prev_id_fire_det = r2r_start_task(DEBUG_ID_ACCIDENT_FIRE_DETECTION);
#endif
        _state = state_idle;
        check_for_fire();
        scheduler_release();
#ifdef DEBUG_TASK_MEASURE
        r2r_stop_task(_task_prev_id_fire_det);
#endif
    break;

    default: break;
  }
}
```

Note that as the fire check only happens every 8 seconds, the detect crash state has a counter to determine when it should set the state to the idle state and when to set it to detect fire state.

4.3.3. Execution Time

To ensure that the system can run without any problems the load of the system is measured. This is done by using a DAC (digital to analogue converter) and measure the execution time and period of the different tasks as voltage changes using an oscilloscope. The input of the DAC is 4 bit and has a logic low of $0V$ and a logic high of $5V$. This results in 16 possible voltage levels (V_{out}) and each voltage step is:

$$\frac{5V}{2^4} = 3.125V$$

The formula for calculating the voltage level for a specific function is:

$$V_{out} = \frac{bit0}{16} + \frac{bit1}{8} + \frac{bit2}{4} + \frac{bit3}{2}$$

There are five primary tasks being executed iteratively, when the system is started and running idle. Each have its own ID to identify them when measuring. The ID is the value passed to the DAC which results in the following voltage levels:

Tasks	Task ID	V_{out}
Temperature reading request	10	3.1V
Accelerometer reading request	11	3.4V
Store measurements	12	3.8V
Crash detection	14	4.4V
Fire detection	15	4.7V

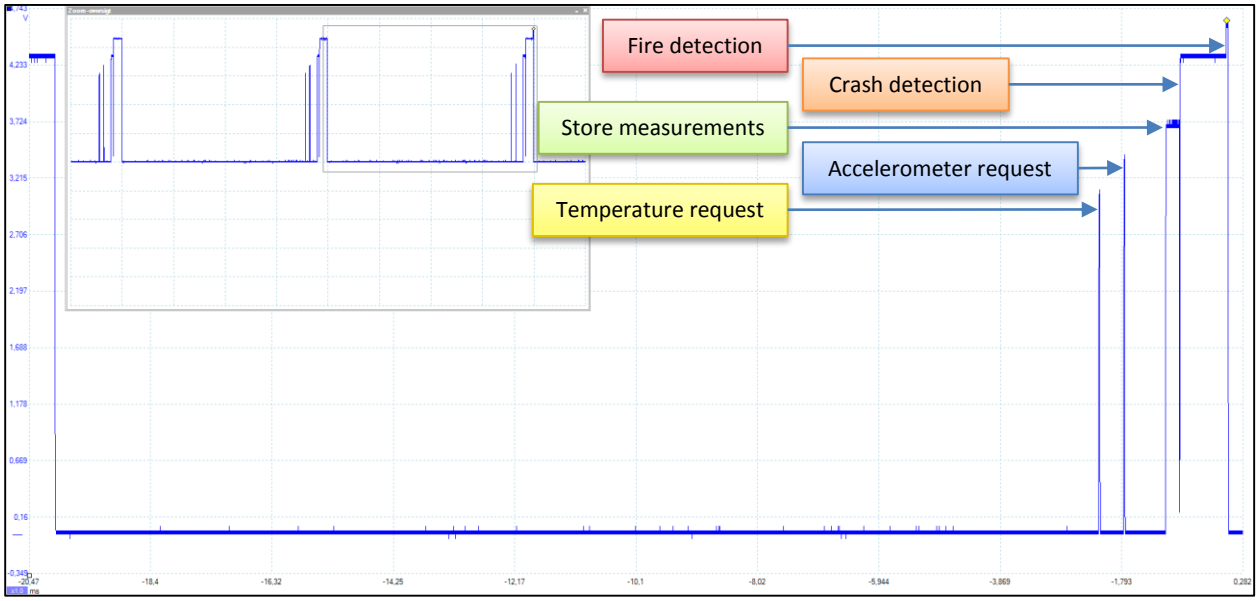


Fig. 20. Timing Graph – Main system

The tasks have the following measurements:

Task	Execution	Period
Temperature reading request	25µs	20ms
Accelerometer reading request	25µs	20ms
Temperature and accelerometer reading	232µs	20ms
Crash detection	771µs	20ms
Fire detection	25µs	8000ms

Although it is obvious from the graph above that the system is not overloaded, it is possible to prove it using the following equation.

$$load = \frac{\sum execution}{period} \cdot 100 \Leftrightarrow load = \frac{(400 \cdot (2 \cdot 25 + 232 + 771) + 25) \cdot 10^{-6}}{8} \cdot 100 = 5.27\%$$

With a 5.27% load on the system, it is safe to assume the system will not suffer from overloading.

4.4. Communication and GPS

4.4.1. Overview

The procedure for initialising the SIM908 module is very simple. The drive pin is toggled, which starts up the module. UART0 is set up to communicate with the module, using AT commands.

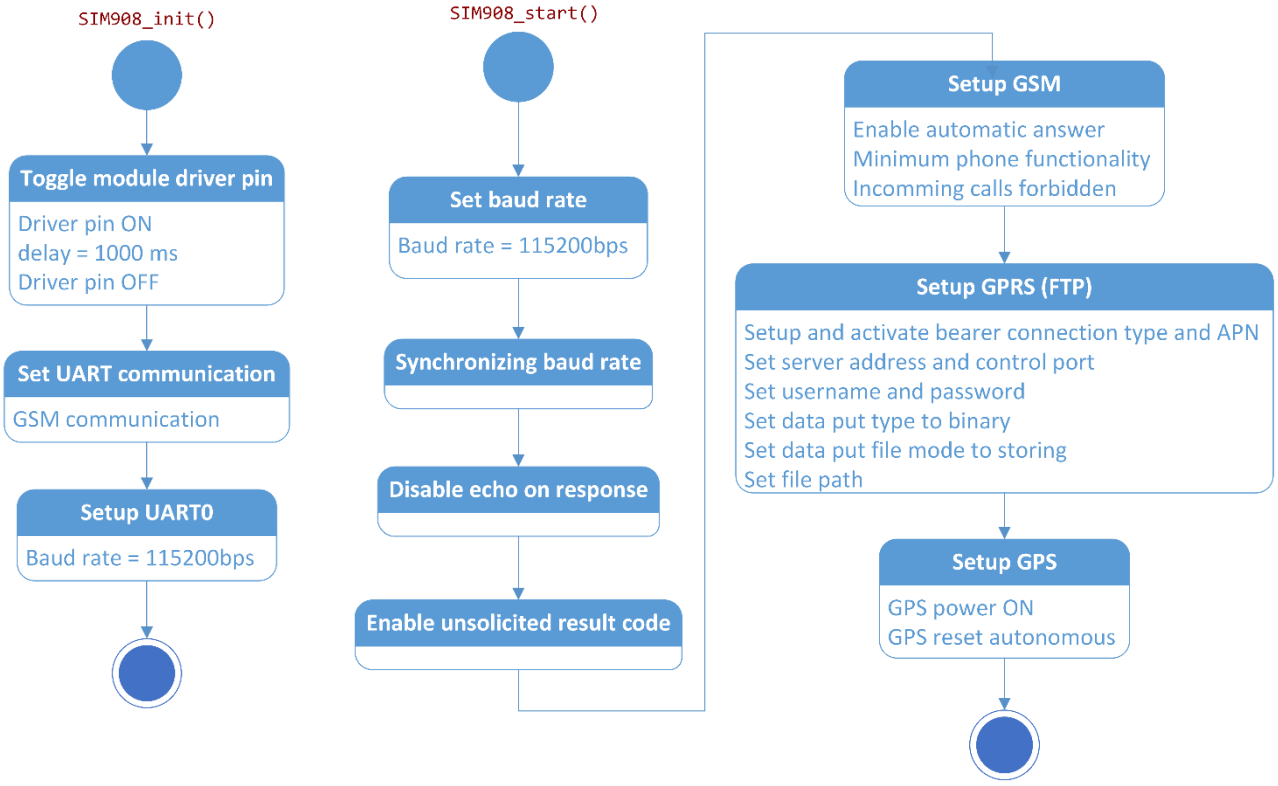


Fig. 21. Activity Diagram – SIM908 initialisation and start-up

In order to setup GSM, GPS and GPRS, global interrupts needs to be enabled, as AT commands is sent through UART, which uses an ISR (Interrupt Service Routine) to handle the reception part of the communication. Responses are captured via a callback function.

Communication with UART is done through two functions: `SIM908_cmd( … , … );` which is the UART tx and `_SIM908_callback( … );` which is the UART rx.

Examples on communication via UART using AT commands:

Set baud rate:
- UART tx: `AT+IPR=115200<CR><LF>`
- UART rx: `<CR><LF>OK<CR><LF>`

Enable CREG unsolicited result code:
- UART tx: `AT+CREG=1<CR><LF>`
- UART rx: `<CR><LF>OK<CR><LF>`

Get GPS data:
- UART tx: `AT+CGPSINF=0<CR><LF>`
- UART rx: `<mode>,<longitude>,<latitude>,<altitude>,<UTCtime>,<TTFF>,<num>,<speed>,<course>`
  `<CR><LF>OK<CR><LF>`

4.4.2. Code

This function sends a stream of characters to the UART. If __wait_for_ok flag is set, it will wait for response before it returns: true if the response is “OK” or false if the response is “ERROR”.

```
bool SIM908_cmd(const char *__cmd, bool __wait_for_ok)
{
   /* Saves the status register and disables global interrupt */
   uint8_t SREG_cpy = SREG;
   cli();

   _ack_response_flag = _ack_ftp_response_flag = _ack_gps_response_flag =
   SIM908_FLAG_WAITING;
   _rx_response_length = _CR_counter = _LF_counter = 0U;

   uart0_send_string(__cmd);
   uart0_send_char(CR);
   uart0_send_char(LF);

   /* Restore interrupt */
   SREG = SREG_cpy;

   return __wait_for_ok ? _wait_response(&_ack_response_flag, SIM908_FLAG_OK) : true;
}
```

The callback routine from UART has become heavier and heavier as the project expanded. Sometimes, but not always, the module needs to wait for a response, before it can execute the next command. This is the reason for the complex function with a wealth of branches. In the following code snippet the logic is left out. It is simply to show all the different branches depending on the type of response.

```
void _SIM908_callback(char data) {
   scheduler_pause();

  /* Mirroring communication from sim908 to uart1 */
   #ifdef DEBUG_UART_ECHO
      uart1_send_char(data);
   #endif

   _rx_response_length++;
   _rx_buffer[_rx_buffer_tail = (_rx_buffer_tail + 1U) % RX_BUFFER_SIZE] = data;

   /* Checking and counting for CR and LF */
   if (data == CR) {
   _CR_counter++;
   } else if (data == LF) {
      _LF_counter++;
   }

   if ((_CR_counter > 0U) && (_LF_counter > 0U)) {
      _CR_counter = _LF_counter = 0U;
      if (_rx_response_length > 2U) { . . .                          /* Skipping empty lines */
         } else
         if ((_gps_pull_flag == SIM908_FLAG_GPS_PULL) &&             /* GPS pull */
            (_check_response(SIM908_RESPONSE_GPS_PULL))) {                  . . .
         } else
         if ((_rx_response_length == 4U) &&                          /* OK */
            (_check_response(SIM908_RESPONSE_OK) == true)) {                . . .
         } else
         if ((_rx_response_length == 7U) &&                          /* Error */
            (_check_response(SIM908_RESPONSE_ERROR) == true)) {             . . .
         } else
         if (((_rx_response_length == 10U) ||                        /* CREG */
            (_rx_response_length == 12U)) &&
            (_check_response(SIM908_RESPONSE_CREG) == true)) {              . . .
         } else
         if ((_rx_response_length == 11U) &&                         /* GPS Ready */
            (_check_response(SIM908_RESPONSE_GPS_READY) == true)) {         . . .
         } else
         if ((_system_running_flag == SIM908_FLAG_WAITING) &&        /* System ready */
            (_rx_response_length == 5U) &&
            (_check_response(SIM908_RESPONSE_RDY) == true)) {               . . .
         }  else
         if ((_rx_response_length == 4U) &&                          /* Sync AT cmd */
            (_check_response(SIM908_RESPONSE_AT) == true)) {                . . .
         } else {                                                           . . .
         }
      }
      _rx_response_length = 0U;
   }
   scheduler_resume(false);
}
```

4.4.3. Execution Time

The setup sequence of the SIM908 module is measured using an oscilloscope to get an idea about how the system behaves in its initial phase. The tasks associated with the setup are:

Task	Task ID	V_{out}
Start of module	3	0.9V
Send command to module	8	2.5V
Wait for response from module	1	0.3V
Callback from the module	5	1.6V

This graph shows the complete setup procedure of the SIM908 module. This module is setup using AT commands via UART.

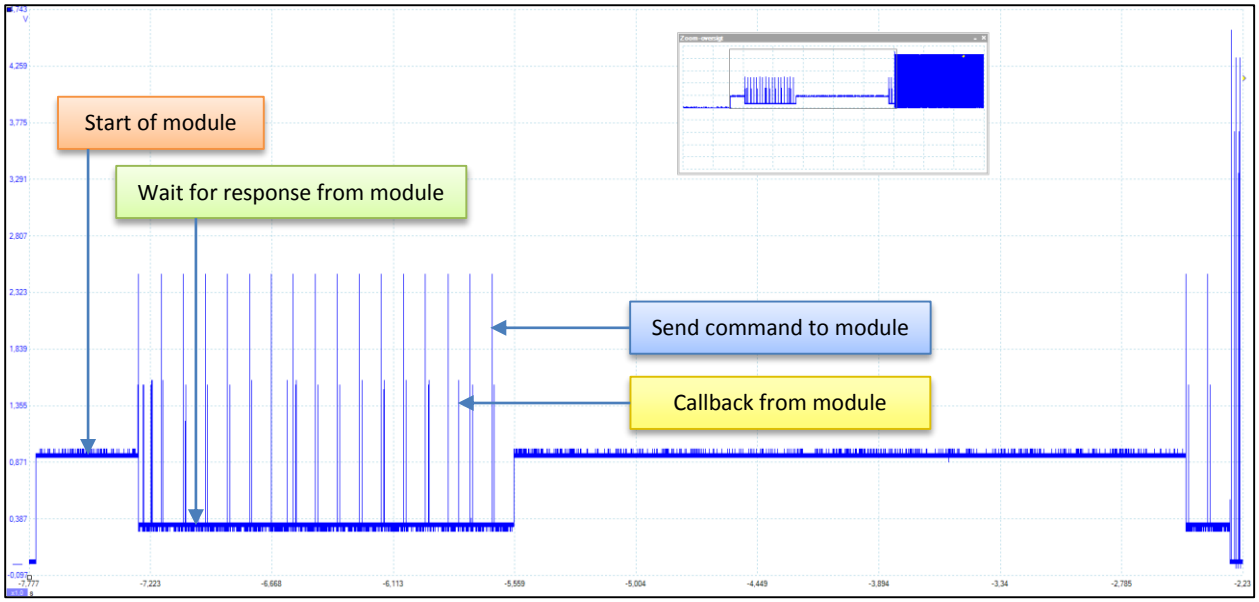


Fig. 22. Timing Graph – SIM908 module setup

As it can be seen in the graph, there are 17 commands executed in the first section of the startup. Those are:

- Init:
  - Set baud rate
  - Sync baud rate
  - Disable ECHO response
  - Enable CREG unsolicited response
- GSM:
  - Enable auto answer
  - Set full functionality
  - Block incoming calls
- GRPS FTP:
  - Set connection type to GPRS
  - Set APN adr.
  - Set user profile
  - Set server adr.
  - Set port
  - Set user name
  - Set password
  - Set data type to binary
  - Set ftp type to storing
  - Set file path

After these there is a waiting period. This is until “GPS Ready” has been received. When this happens the GPS is set up:

- GPS power on
- GPS autonomy mode

This completes the setup of the SIM908 module.

4.5. Car Panel

4.5.1. Overview

Besides visualisation of system status and alarm status, the HMI has 3 functionalities; manually alarm, cancel alarm and reset emergency detection. The cancel alarm, is a routine which is called whenever an alarm is triggered, regardless of whether it is automatically or manually triggered. The reset emergency detection can only be executed, when an emergency call has been done.

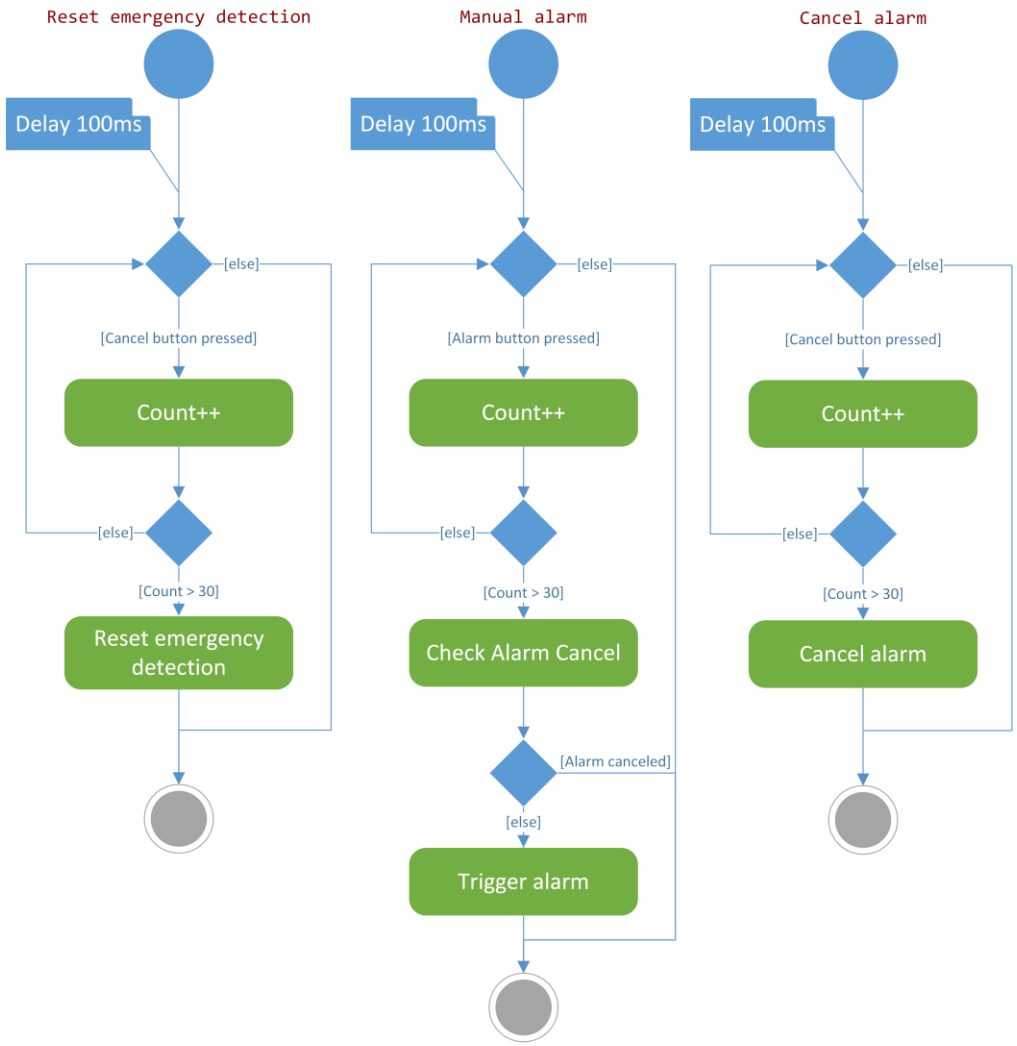


Fig. 23. Activity Diagram – Car panel

4.5.2. Code

Time counting is done in busy waiting and not by a timer, as it was indented to block everything when a button is pushed. Blocking the system led later to a problem in SIM908 communication, as unsolicited result code were missed. The routine is now implemented as a non-block ISR to resolve this problem. Code for the alarm button is shown below.

```
ISR (PCINT1_vect, ISR_NOBLOCK)
{
   /* Check if alarm button is pressed */
   if(!(PIN(PORT) & (1<<BTN_ALARM)))
   {
      _car_panel_counter = 0U;
      while ((_car_panel_counter++ < CONFIG_ALARM_BUTTON_PRESS_TIME) &&
              (!(PIN(PORT) & (1<<BTN_ALARM))))
      {
         _delay_ms(100);
         car_panel_set_control(ALARM_WAITING);
      }

      if (_car_panel_counter >= CONFIG_ALARM_BUTTON_PRESS_TIME )
      {
         if (!car_panel_wait_cancel_emmergency())
         {
            EXT_EMERGENCY_FLAG = EMERGENCY_MANUAL_ALARM;
         }
      }
      else
      {
         car_panel_set_control(ALARM_NOT_ACTIVATED);
      }
   }
   /* Check if cancel button is pressed */
   else if (!(PIN(PORT) & (1 << BTN_CANCEL)))
   {
      . . .
   }
}
```

When the counter exceeds the defined button pressed time (3 seconds), it calls the `car_panel_wait_cancel_emmergency(void);` routine, which is a function implemented using the same principle, by busy waiting and counting. It returns a Boolean whether the alarm is cancelled or not.

4.6. Emergency Reporting

4.6.1. Overview

When an alarm is triggered, the `ad_emergency_alarm(void);` routine is called in order to assembly the information needed for the MSD, into a 140 bytes struct, send it to an FTP server and finally make a voice call.

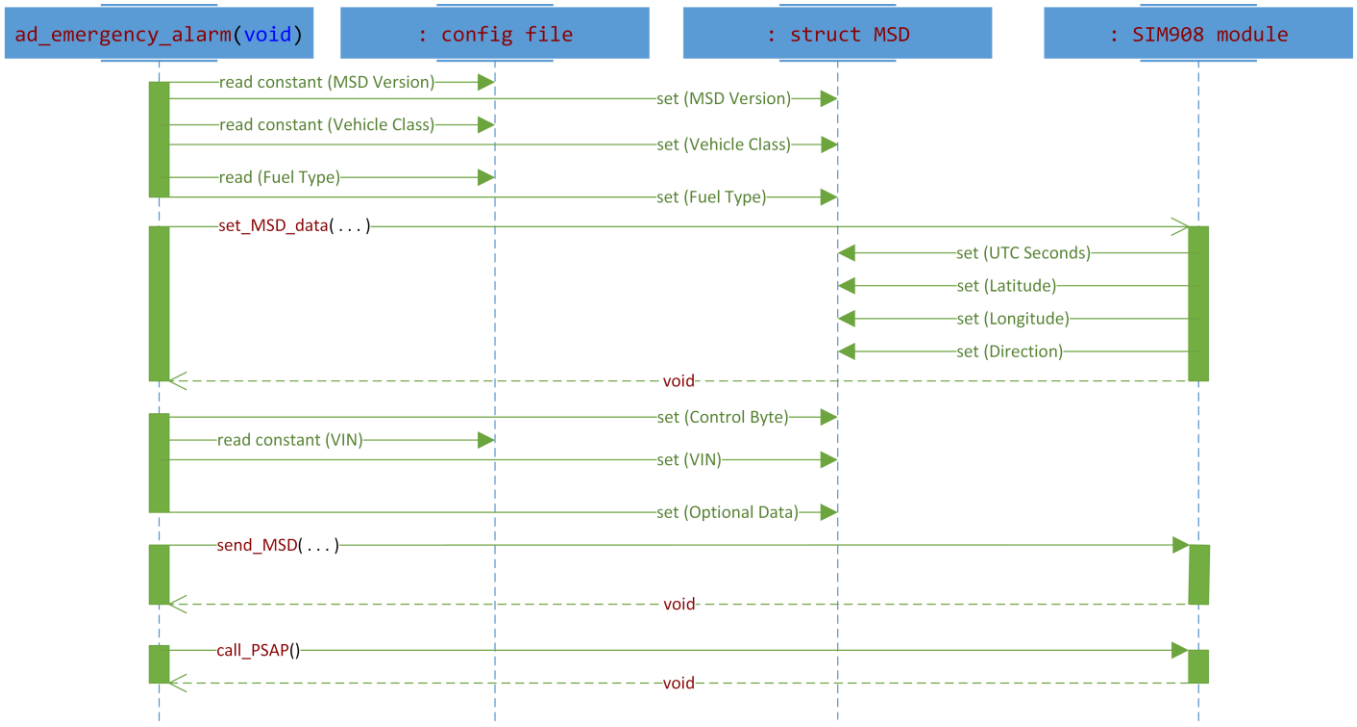


Fig. 24. Sequence Diagram – Emergency alarm

The MSD structure is an external variable in which the parameters and structure are predefined (ref. 2.10). Static data is read from the config file, while dynamic data, as time and location, are read and calculated internal in SIM908 module.

When data is collected and set, the system establishes communication to PSAP. The MSD structure is sent through GPRS to an FTP and subsequently a voice call to the PSAP is established.

4.6.2. Code

```
void send_MSD(const char *__vroom_id)
{
    _wait_for_connection();

    int8_t _retry_ctr = RETRY_ATTEMPTS;
    char *filename = malloc(60U * sizeof(char));

    . . . Filename concatenation has been left out

    while(!SIM908_cmd(filename, true));
    free(filename);

   _ftp_sending_flag = SIM908_FLAG_FTP_SENDING;

    while ((!SIM908_cmd(AT_FTP_OPEN_BEARER1, true)) && (_retry_ctr-- > 0)) {
        _delay_ms(1000);
    }

    do {
       SIM908_cmd(AT_FTP_PUT_OPEN_SESSION, false);
    } while ((!_wait_response(&_ack_ftp_response_flag, SIM908_FLAG_FTP_PUT_OPEN)) &&
             (_retry_ctr-- > 0));

    do {
       SIM908_cmd(AT_FTP_PUT_FILE_SIZE(CONFIG_FTP_FILE_SIZE), false);
    } while ((!_wait_response(&_ack_ftp_response_flag, SIM908_FLAG_FTP_PUT_SUCCESS)) &&
              (_retry_ctr-- > 0));

    do {
       EXT_MSD.msg_identifier = RETRY_ATTEMPTS - _retry_ctr + 1U;
       uart0_send_data((char*)(&EXT_MSD.version), 1U);
       uart0_send_data((char*)(&EXT_MSD.msg_identifier), 1U);
       uart0_send_data((char*)(&EXT_MSD.control), 1U);
       uart0_send_data((char*)(&EXT_MSD.vehicle_class), 1U);
       uart0_send_data(&EXT_MSD.VIN[0], 20U);
       uart0_send_data((char*)(&EXT_MSD.fuel_type), 1U);
       uart0_send_data((char*)(&EXT_MSD.time_stamp), 4U);
       uart0_send_data((char*)(&EXT_MSD.latitude), 4U);
       uart0_send_data((char*)(&EXT_MSD.longitude), 4U);
       uart0_send_data((char*)(&EXT_MSD.direction), 1U);
       uart0_send_data(&EXT_MSD.optional_data[0], 102U);

       uart0_send_char(CR);
       uart0_send_char(LF);
    } while ((!_wait_response(&_ack_ftp_response_flag, SIM908_FLAG_FTP_PUT_OPEN)) &&
             (_retry_ctr-- > 0));

    _delay_ms(100);
    do {
       SIM908_cmd(AT_FTP_PUT_CLOSE_SESSION, true);
    } while ((!_wait_response(&_ack_ftp_response_flag, SIM908_FLAG_FTP_PUT_CLOSE)) &&
             (_retry_ctr-- > 0));

    while(!SIM908_cmd(AT_FTP_CLOSE_BEARER1, true));

   _ftp_sending_flag = SIM908_FLAG_WAITING;
}
```

First a routine is called to ensure that the module has a reliable signal to proceed the file transfer.

Next the filename is created with format: YYYY-DD-MM_HH.MM.SS-(__vroom_id).vroom. This code is left out in the example, but it just simple concatenates the AT command with the time stamp and vroom ID, which in this system, is the phone number.

Finally following AT commands are sent in a sequence using the `SIM908_cmd(..., ...);` function. Some of the commands requires the correct response, before it can be executed.

1. Create filename: AT+FTPPUTNAME="<filename>"
   Required response: OK
2. Open bearer: AT+SAPBR=1,1
   Required response: OK
3. Open FTP PUT session: AT+FTPPUT=1
   Required response: OK
   +FTPPUT:1,1,1260
4. Set write data: AT+FTPPUT=2,140
   Required response: OK
   +AT+FTPPUT=2,140
5. Write data: Data is directly written to UART0 and terminated with <CR><LF> which automatically closes the PUT session
   Required response: OK
   +FTPPUT:1,1,1260
6. End write session: AT+FTPPUT=2,0
   Required response: OK
   +AT+FTPPUT=1,0
7. Close bearer: AT+FTPPUT=0,1
   Required response: OK

4.6.3. Execution Time

This sequence is measured using an oscilloscope to get an idea about how to system behaves when it is started up. The tasks associated with the emergency reporting are:

Task	Task ID	V_{out}
Send command to module	8	2.5V
Wait for response from module	1	0.3V
Callback from the module	5	1.6V

This graph shows the complete emergency reporting procedure of the SIM908 module. This module is setup using AT commands via UART.

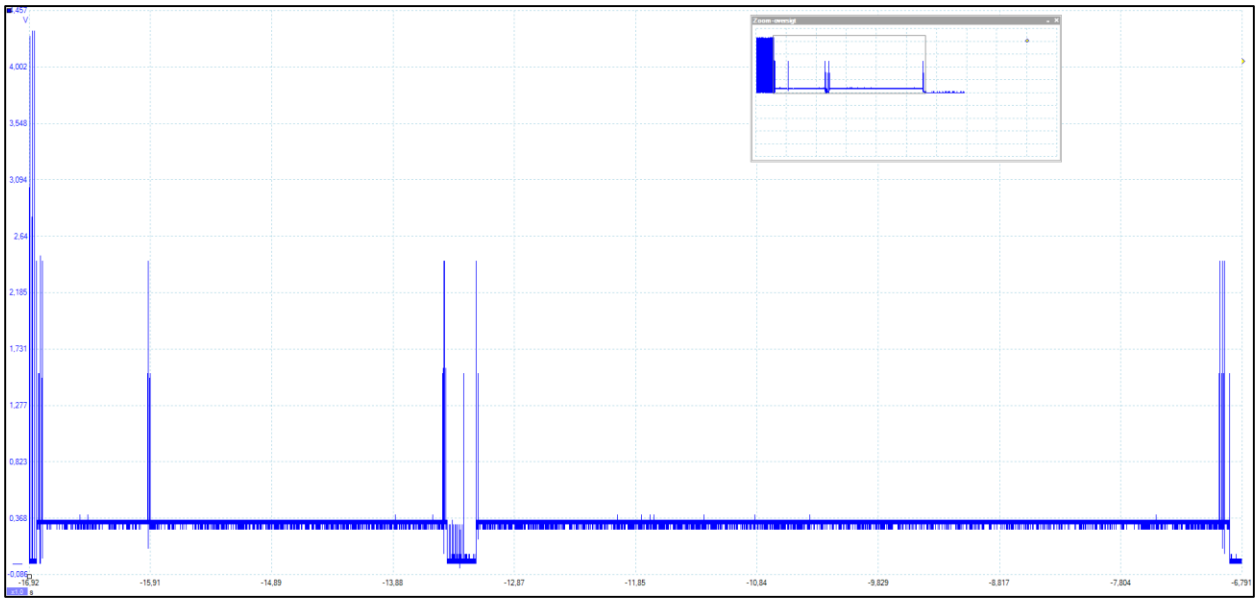


Fig. 25. Timing Graph – Emergency reporting

When an alarm triggers, there are 9 commands sent:

- Requesting GPS information containing position, course and UTC time
- Set the filename for the report
- Open the GPRS connection
- Initialise the session with the FTP server
- Send the data
- Close the session with the FTP server
- Close the GPRS connection
- Enable incoming calls
- Call PSAP

After these commands are sent, the system goes into a dormant state. It remains in this dormant state until the driver has reset emergency detection (ref. 4.5).

5. Testing

5.1. Testing Tools

5.1.1. Accelerometer Analyser

Purpose

As there was a need to be able to see and analyse accelerometer data, the Accelerometer Analyser was developed. This was needed to be able to see how a car is affected by forces in various situations.

Functionalities

This tool has a few, but important functionalities.

- Record live accelerometer data from a serial interface
- Save the recorded data to a file for later analysis
- Display live or recorded data on a graph as acceleration over time
- Display live or recorded data as text
- Display the max acceleration on currently displayed data
- Enable or disable different axis' or vectors
- Zoom on both the time axis and the acceleration axis, both with mouse scroll and with a slider

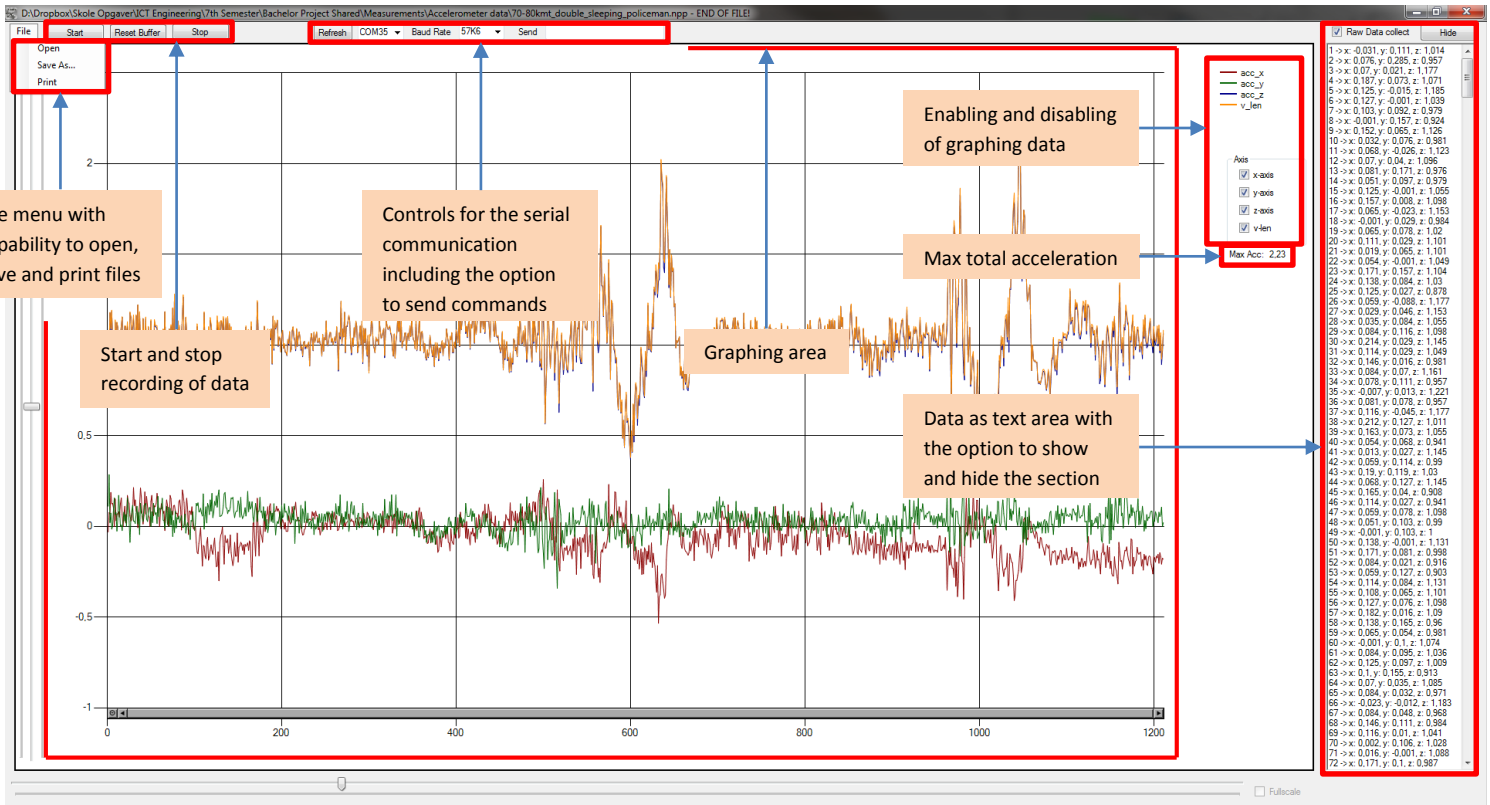


Fig. 26. Accelerometer Analyser – Explained

Implementation

The tool is implemented in C# .NET using Winforms API. The way the tool works is that when data is received from the serial bus, an eventhandler is triggered. This eventhandler stores the data in a BindingList. This BindingList has its own eventhandler. Hence this can initialise the graphing part separate from the eventhandler of the serial bus. This enables the tool to be very efficient and be able to record data at a very high speed, without getting backed up by data. This is due to the fact that the graphing part runs in a separate thread system from the reading part.

Unfortunately the .NET framework has an overhead issue, which means that data received at a very high speed at baud rate 115200 bps and higher, is being backed up in the receive buffer and the system starts to lag behind the actual live data.

Merits

By having this tool, it has been possible to record and analyse own data rather than having to rely solely on other people's findings. It has also made it possible to record the exact scenarios which was deemed necessary for this specific project. However, it has not been possible to use this tool to test actual crash scenarios as no one would lend us their car, so these data is gathered from internet research.

5.1.2. SIM908 AT Terminal

Purpose

As no serial terminal, that was available, satisfied all the needs for developing this system, an alternative was needed.

What was needed was a tool which could display serial ASCII communication with a time stamp.

Functionalities

The functionalities of this tool are very basic. Beside the ASCII communication, the tool also has some fixed features which are specifically developed for this project.

- Signal strength display and logging
- Connection status command button
- GPS ON/OFF, status, reset and location fix
- Display on map feature for checking the accuracy of the GPS signal
- Change colour and font of the text in the terminal
- Option to save the whole communication as a rich text document for logging

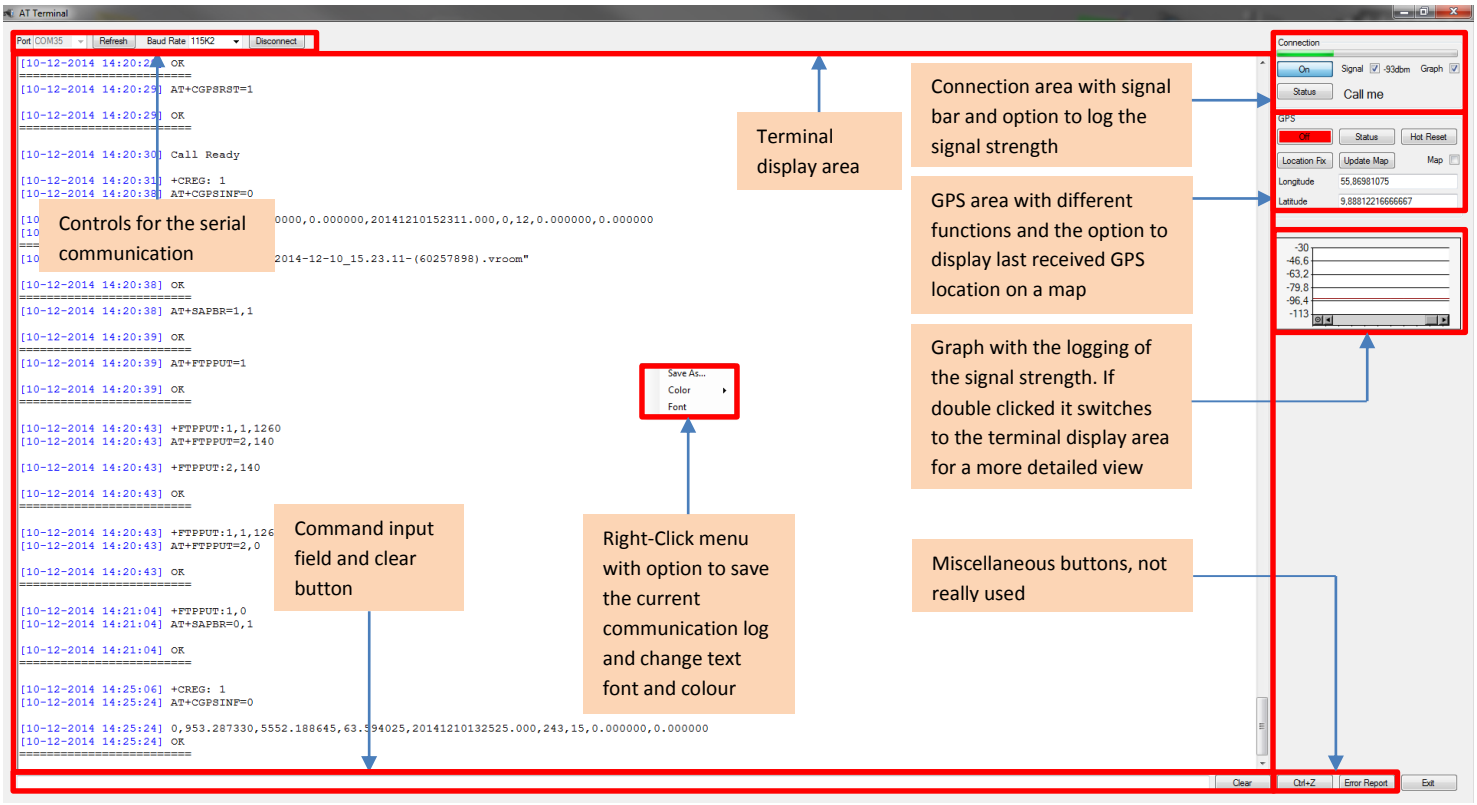


Fig. 27. SIM908 AT Terminal – Explained

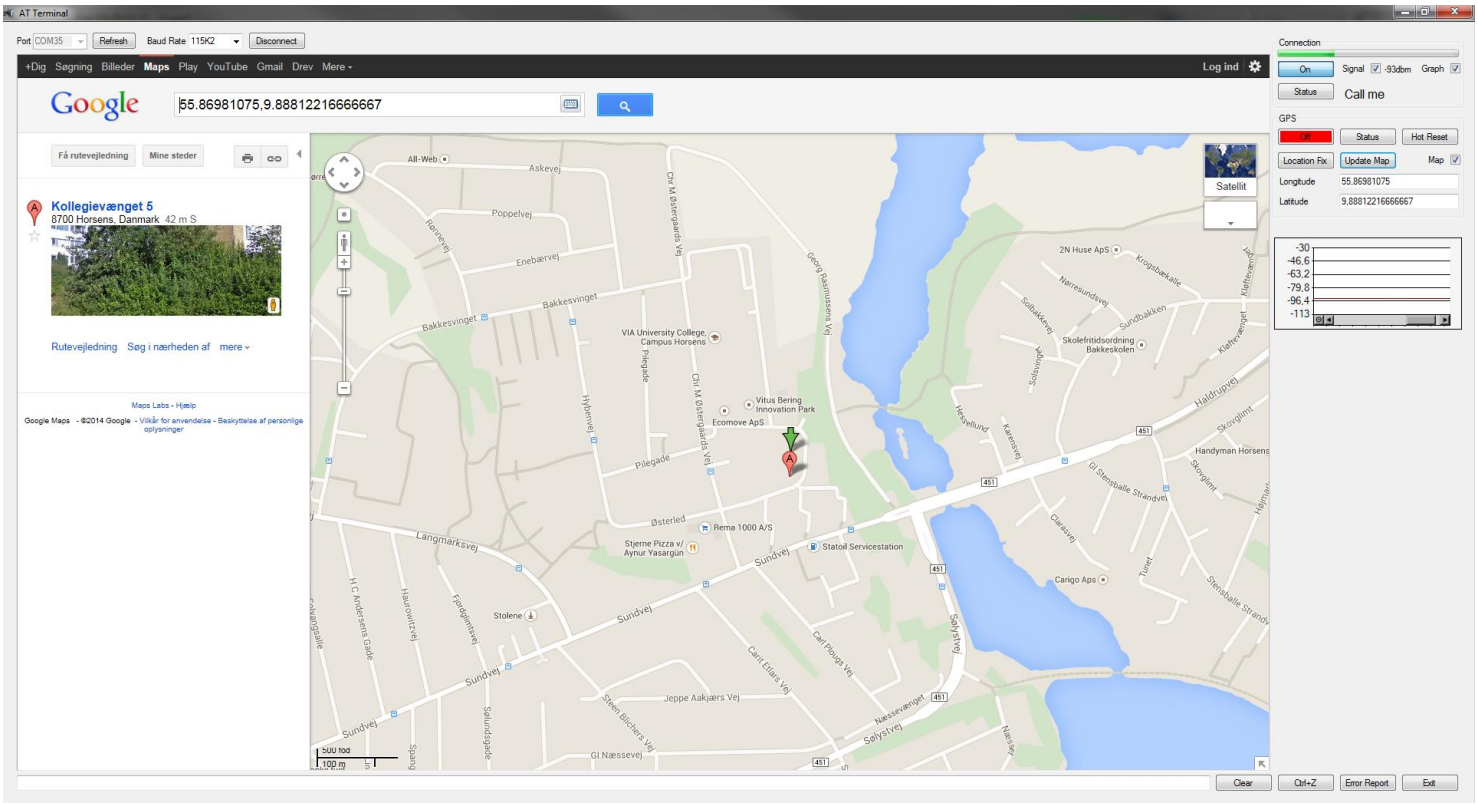


Fig. 28. SIM908 AT Terminal – Map view

Implementation

The tool is implemented in C# .NET using Winforms API. The tool works by interpreting the ASCII communication from the serial connection and then displaying it on the terminal display area. It has a timed task for the signal that updates every few seconds by sending an AT command. If a GPS location fix response is received, the coordinates are converted to decimal coordinates and written in two text boxes for display. The map can then be displayed by clicking on the "Map" check box. The map is displayed using a browser object and the google maps web site.

Merits

By having this tool it has been easy to log and keep track of what was happening with the system, as the system has the option to echo all the UART communication to a serial output. It has also helped a lot in figuring out what commands to run and in which sequence to get the SIM908 module to operate as intended. The map feature has been a good help to make sure GPS accuracy and calculation between different coordinate systems has been done correctly.

5.1.3. PASP Simulator

Purpose

In order to check and analyse the content of the MSD .VROOM file sent through FTP, a PSAP Simulator was needed to decode and analyse the data which constitutes the emergency report.

Functionalities

This tool has following key functionalities:

- Structure and order already existing emergency reports
- Automatically notify and updates when a new emergency report is received
- Display the raw data of the MSD .VROOM file by right clicking the file
- Decode the MSD .VROOM file and visualise it in MSD Details window
- Visualise location of the incidents on a map
- Zoom on map by mouse scrolling or double mouse clicking

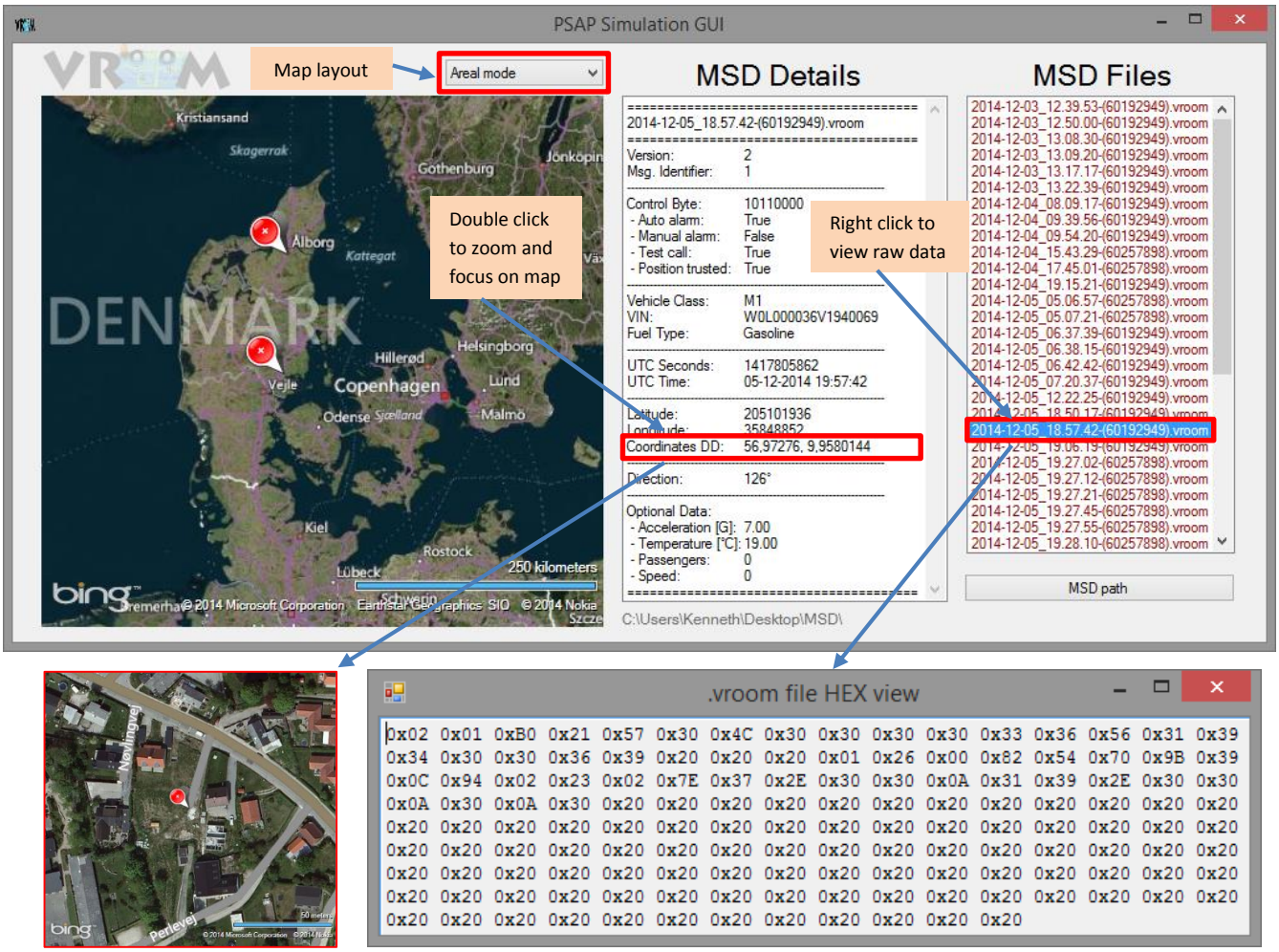


Fig. 29. PSAP Simulator – Explained

Implementation

The tool is implemented in C# .NET using Winforms API. A filesystem watcher is implemented in order to trigger an event when a new file is created. When this eventhandler is triggered the file is read and the raw data is stored in a list. The item is listed in the MSD Files view and an alarm-sound will apply, to notify that an event has occurred.

When a file is selected in the MSD Files view an eventhandler for the listbox will decode the MSD file, display the data in MSD Details view and add the pin to the map, if the position is trusted.

The map is implemented using Bing maps, which is a WPF user control. The component is created in XAML and hosted in a Winform elementhost. A Bing account is created in order to get a credential key which is required in order to use Bing maps.

Merits

Using this tool the MSD files are easily evaluated and analysed. This tool provides a good visual representation and overview of how the MSD is structured and how it could be presented. It also demonstrates the importance of getting the exact information about an emergency and how easy PSAP can determine the necessary response.

5.1.4. Oscilloscope

Purpose

To make sure everything can run on the system without any issue, an oscilloscope was needed to measure the timing of different parts of the system.

Functionality

The functionality of this tool is to measure voltage over time.

Implementation

Along with the oscilloscope, an R2R ladder D/A converter is built and implemented to be able to see the different tasks as voltage changes. This way it is possible to see precisely which task is running at any given time.

Merits

By having this tool it has been possible to confirm and document that the system executes as expected and that the cyclic scheduling runs exactly as intended, without overloading the system.

5.2. Test Plan and Specification

Test plan (Appendix B) and specification (Appendix C) is prepared for this system. Details of testing types and procedures is described in order to document and replicate the test results.

Unit testing, module testing and integration testing are all included in the source code. The respective tests can be executed by defining the context in run.c as ON while other contexts must switched OFF

```
#define UNIT_TEST                                   ON

#define MODULE_TEST_SENSORS                         OFF
#define MODULE_TEST_SIM908                          OFF
#define MODULE_TEST_CAR_PANEL                       OFF
#define MODULE_TEST_UART                            OFF

#define INTEGRATION_TEST_SIM908_SENSORS             OFF
```

5.2.1. Unit Test

MinUnit framework for unit testing is included in this system. Following functions performs unit testing:

- `calculate_temperature(..., ...);` in tc72 driver
- `calc_UTC_seconds(...);` in time.c

The reason for why the unit test platform is not used for more functions, is that the complexity in this system is not in the arithmetic, but in the communication.

5.2.2. Module Test

Module test are performed on each modules. Car Panel and Sensor modules are visually tested and the readings are conducted using the LCD display. Only UART module conforms a validation check for what is sent and received.

5.2.3. Integration Test

Integration test is performed by testing the modules functionalities and interactions with each other, then they are integrated. The integration test ends up integrating all the functionalities of the system and therefor the final program.

5.2.4. Regression Test

Regression testing is conducted whenever a substantial amount of code is changed. This test includes mainly stability testing as it is the only way to test if everything works.

5.2.5. Stability Test

To ensure the system is stable and reliable, different stability tests have been conducted.

Start-up
Over 500 consecutive start-up tests have been conducted with a successful result.

The objective of this test is to ensure that the system has a successful implementation in the start-up procedure and that it starts reliably every time.

The test was conducted by repeating the following steps:

1. Power on
2. Wait (random times between 0 minutes and 10 hours)
3. Trigger alarm (randomly chosen between manual, crash automatic and fire automatic)
4. Power off

The precondition for this test is that the system is located a place where a mobile communication signal is reachable.

Emergency reporting
Two separate tests have been conducted, each sending over 3000 consecutive reports with a 100% success rate.

The objective of this test is to ensure that the system has a stable and reliable implementation of the communication system to ensure that whenever an emergency is registered, an MSD will be sent.

The test was conducted by repeating the following step:

1. Send MSD

The precondition for this test is that the system is located a place where a mobile communication signal is reachable and that one alarm has been triggered.

Long-time run
More than 100 long-time run tests have been conducted. Each varying between 5 minutes and 24 hours, all with successful results.

The objective of this test is to ensure that the system's cyclic scheduler is implemented in a reliable and stable way and will not cease functioning after long periods of time.

The tests were conducted by repeating the following steps:

1. Power on
2. Wait (random chosen of the times specified in the test plan: 5 minutes – 2 hours – 24 hours)
3. Trigger alarm (randomly chosen between crash automatic and fire automatic)
4. Power off

The precondition for this test is that the system is located a place where a mobile communication signal is reachable.

5.2.6. Functionality Test

Functionality tests have been conducted for all aspects for this project.

ECU

This test is conducted alongside the stability testing and has the same success rate (ref. 5.2.5). This test is conducted according to the test plan section 5.2.1:

Trigger on crash and call for assistance		
Type	**Result**	**Note**
Positive	PASS	Call to PSAP is simulated
Negative	PASS	

HMI

This test is conducted according to the test plan section 5.2.1:

Manually activate emergency call		
Type	**Result**	**Note**
Positive	PASS	Call to PSAP is simulated
Negative	PASS	

Cancel a false activation of an emergency call		
Type	**Result**	**Note**
Positive	PASS	
Negative	PASS	

Notification of the system's state		
Type	**Result**	**Note**
Positive	PASS	

5.2.7. Not Conducted Tests

Release testing is not conducted. It is, however, included in the test specification and the test plan.

As this project only focuses on making a prototype of the final product, a release test is not possible. Furthermore, a complete functionality test is not possible as the final HMI design and production is not done.

However, all technical functionalities have been thoroughly tested (ref. 0)

6. Result and Discussion

After careful analysis of the system through testing and timing analysis, it is demonstrated that the system is running as intended. All the required functionalities are implemented and working. However, this does not mean the system is perfect and without flaws.

6.1. Improvements

MISRA C 2004

This standard, which was to be followed, presented some challenges which were too large to be resolved in the given time. This means that a few of the rules specified in the MISRA C 2004 standard are not followed. However, most of the rules are followed and respected.

The rules that are not followed can be found in (Appendix E)

SIM908 Communication

The way the SIM908 communication is implemented turned out to not work as efficiently as it could.

The problem is that as the project grew, more and more responses from the SIM908 module were required to be received and acknowledged. This meant that the callback function for the module became larger and larger. This was a problem as this callback function is called directly from the UART interrupt service routine, so the time it took to check all these different responses meant that part of other responses (especially the unsolicited ones) were missed.

So this one section could be greatly improved by creating a more elaborate system, however, the time to change it was not there when the problem became apparent.

One solution can be to have a framework which can support specific send/response combinations so that whenever a command requires a specific response, a callback function with only that check is passed to the UART.

Another solution could be to pull data from the UART at certain intervals at the cost of larger memory consumption and more processing time.

```
void _SIM908_callback(char data) {
   . . . CODE
if ((_CR_counter > 0U) && (_LF_counter > 0U)) {
     if (_rx_response_length > 2U) {               /* Skipping empty lines */
        if ((_ftp_sending_flag == SIM908_FLAG_FTP_SENDING) &&
           (_check_response(SIM908_RESPONSE_FTP_PUT) == true)) {
                . . . CODE
        } else if ((_gps_pull_flag == SIM908_FLAG_GPS_PULL) &&
           (_check_response(SIM908_RESPONSE_GPS_PULL))) {        /* GPS pull */
                . . . CODE
        } else if ((_rx_response_length == 4U) &&
           (_check_response(SIM908_RESPONSE_OK) == true)) {      /* OK */
                . . . CODE CONTINUES . . .
```

6.2. From Prototype to Market

Although this project only focuses on a prototype, thoughts about how to realise it and get it to the mass market have also been made.

Car Panel

An ideal installation for this interface is intended as a part of the rear-view mirror, as this will minimise the probability of accidently activating an alarm and make for a seamless integration into any car. It will also help keep installation costs to a minimum, as it is not a whole new panel that needs to be installed.

Pricing

A focus group would be a good idea to determine what would be a reasonable price for a product like this, as the cost of this system is a key selling point.

Hardware

Research which hardware will fit the requirements for the system best will be essential to ensure optimal quality/price balance.

Production

Research into where the final product can be produced in a good balance between cost and quality. Even though cost is a selling point, the quality is very important as well, especially in a safety-critical system like this; human lives are at stake.

Code Certification

The code would have to be certified in order to conform to the standards which are needed to realise this type of product.

Materials

The right kind of material is needed for the ECU box as it is very important that it is not damaged in a crash to the extent that it cannot perform the required functionalities. The system need to be functional regardless of the severity of the crash.

Operational Environment

The system will have to be mounted in a secure location and in a way that the accelerometer is detecting the forces experienced by the vehicle. This means that it would have to be researched where to mount it. Furthermore, a backup battery and electronic circuit for delayed off functionality would also be required in order to ensure that the system would still be functional in the event that the vehicles power source is compromised.

7. Conclusion

A prototype for a product that automatically calls for help in the event of a car accident have been presented and documented in this report. The product is suitable for all car makes and models, as it does not interact with any of the cars components except for the battery.

This is achieved by utilising an accelerometer to determine a crash instead of communicating with the car's CANBUS to determine if an airbag is deployed. This is a vast improvement in contrast to other competing products.

The product also includes a thermometer to detect fire, as well as an option to manually trigger an alarm. This can greatly decrease the response time of the emergency response and potentially save lives.

It uses mobile communication, audio as well as data, to notify the emergency services in any given European country.

Although this is only a prototype, there are not many things that needs to be done in order to realise the final product and introduce it to the mass market (ref. 0)

A variety of test tools have also been developed in order to enhance the development of the product.

The product itself is developed as an embedded system written in C. It consists of 6166 lines of code and takes up 14442 bytes of program memory on the MCU.

All of the test tools are written in C# .NET.

8. Appendices

Appendices are located in the Documentation folder.

Project Description:
…\Documentation\Project Report\Appendices\Appendix A – Project Description.pdf

Test Plan:
…\Documentation\Project Report\Appendices\Appendix B – Test Plan.pdf

Test Specification:
…\Documentation\Project Report\Appendices\Appendix C – Test Specification.pdf

User Guide – Car Panel:
…\Documentation\Project Report\Appendices\Appendix D – User Guide VROOM Car Panel.pdf

Disregarded MISRA C 2004 rules:
…\Documentation\Project Report\Appendices\Appendix E – Disregarded MISRA C rules.pdf

Stability testing reports:
…\Testing and Analysing\Data\Stability Testing\...

Accelerometer analysing data:
…\Testing and Analysing\Data\Accelerometer Analysing\...

Datasheets:
…\Documentation\Hardware\Datasheets\...

PCB – Car Panel:
…\Documentation\Hardware\VROOM Car Panel\...

PCB – R2R ladder DAC:
…\Documentation\Hardware\R2R ladder DAC\...

Source Code – Accelerometer Analyser:
…\Testing and Analysing\Tools\Accelerometer Analyser\...

Source Code – SIM908 AT Terminal:
…\Testing and Analysing\Tools\SIM908 AT Terminal\...

Source Code – PSAP Simulator:
…\Testing and Analysing\Tools\PSAP Simulator\...

Source Code – System:

…\Source code\VROOM\...

Memory Map:

…\Source code\VROOM\MemoryMap.xlsx

Doxygen documentation including changelog:

…\Documentation\Doxygen\index.html

9. References

E-Call and HeERO

- ritzau (2014) Om få år ringer bilen selv efter hjælp ved trafikulykke, Available at: http://jyllands-posten.dk/motor/ECE7262441/Om-f%C3%A5-%C3%A5r-ringer-bilen-selv-efter-hj%C3%A6lp-ved-trafikulykke/ (Accessed: 2014-12-10).
- Helen Kearns (2013) eCall: automated emergency call for road accidents mandatory in cars from 2015, Available at: http://europa.eu/rapid/press-release_IP-13-534_en.htm (Accessed: 2014-12-04).
- HeERO (2014) Frontpage, Available at: http://www.heero-pilot.eu/view/da/home.html (Accessed: 2014-12-04).
- HeERO (2014) Publications, Available at: http://www.heero-pilot.eu/view/en/media/publications.html (Accessed: 2014-10-21).
- Jeremy Laukkonen (2014) Advanced Automatic Collision Notification, Available at: http://cartech.about.com/od/Safety/a/Advanced-Automatic-Collision-Notification.htm (Accessed: 2014-12-04).
- righttoride.eu (2014) Europe eCall-ing, Available at: http://www.righttoride.eu/eu-e-call/ (Accessed: 2014-12-10).
- Bilviden.dk (2012) ECALL (Automatisk 112-opkald), Available at: http://www.bilviden.dk/Sikkerhed/Fremtidens-sikkerhedsudstyr/eCall.aspx (Accessed: 2014-12-10).
- Gediminas VILKAS (2014) Automatisk nødopkald fra biler skal redde liv, Available at: http://www.europarl.europa.eu/news/da/news-room/content/20140224IPR36860/html/Automatisk-n%C3%B8dopkald-fra-biler-skal-redde-liv (Accessed: 2014-12-10).
- B. Rosen (2013) Internet Protocol-based In-Vehicle Emergency Call draft-rosen-ecrit-ecall-10.txt, Available at: http://tools.ietf.org/html/draft-rosen-ecrit-ecall-10#ref-eCall-MSD (Accessed: 2014-12-10).

OnStar

- OnStar (2014) Frontpage, Available at: https://www.onstar.com/us/en/home.html?source=ct (Accessed: 2014-12-04).
- Erik Morsing (2014) Opel får OnStar og 4G forbindelse i bilerne, Available at: http://media.gm.com/media/dk/da/opel/news.detail.html/content/Pages/news/dk/da/2014/opel/03-04-on-star.html (Accessed: 2014-12-04).
- Sean Gornstein (2002) OnStar gør det lettere at være bilist, Available at: http://www.talefod.dk/hvorfor/vis_artikel/onstar-gor-det-lettere-at-vare-bilist/ (Accessed: 2014-12-04).

- Jeremy Laukkonen (2014) GM's OnStar Service: How Does It Work?, Available at: http://cartech.about.com/od/Safety/a/Gms-Onstar-Service-How-Does-It-Work.htm (Accessed: 2014-12-04).

Airbag

- AA1Car (2014) Air Bags & Crash Sensors, Available at: http://www.aa1car.com/library/airbag01.htm (Accessed: 2014-12-04).
- CarsDirect (2013) How Does Your Car's Airbag System Work, Available at: http://www.carsdirect.com/car-safety/how-does-your-cars-airbag-system-work (Accessed: 2014-12-04).
- Sally Dominguez (2010) Deflated expectations, Available at: http://www.drive.com.au/motor-news/deflated-expectations-20100331-refb.html (Accessed: 2014-12-04).

10. List of Acronyms

VROOM:	Vehicle Remote Observing Organizational Management
VIN:	Vehicle Identification Number
GPS:	Global Positioning System
GSM:	Global System for Mobile Communications
GPRS:	General Packet Radio Service
IVS:	In-Vehicle System
ECU:	Electronic Control Unit
HMI:	Human Machine Interaction
MSD:	Minimum Set of Data
PSAP:	Public-safety answering point
EENA:	European emergency number association (Standard)
CEN:	The European Committee (Comité Européen de Normalisation)
ISO:	International Organization for Standardization
ISR:	Interrupt Service Routine
RC:	Remote Control
DAC:	Digital to Analogue Converter