Learn In Depth

First Term Project: Pressure Controller

Ali Emad Abdo

September 27, 2023

LaTeX

Contents

1. Project Overview

1.1 Design Sequence

In this project we intend to provide the full explanation of the project steps, simulating a real life deal about a pressure control system.

The six following steps forms the structure of the system architect, and all are a pre-coding steps.

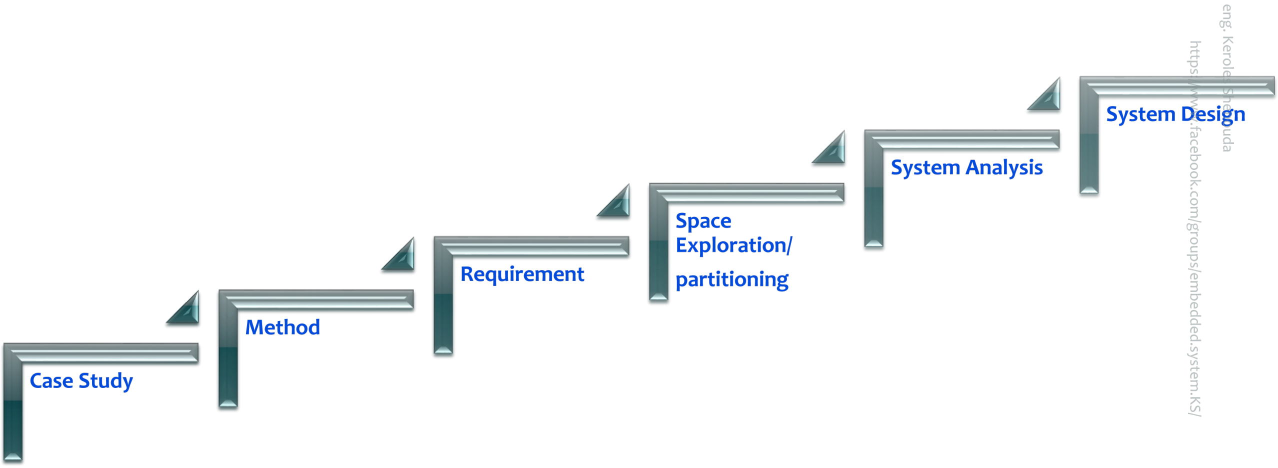

Figure 1: Embedded Systems Design Sequence.

1.2 Coding Sequence

For the Coding Part we are not implementing only the app or the main algorithm but we build every thing from scratch, starting from startup code, linker script and even the makefile.

1.3 perquisite tools of this project:

- Arm cross tool chain
- Text editor (optional)
- Protues 8 for hardware simulation
- TTools for Diagrams

2. case study

Last week, we had a new customer who inquired about our services, he asked for an embedded system to monitor the pressure level in his high hills climbing car and if the pressure exceeded 20 bars the system starts an alarm.

So we want to start our deal but first we must be clear in every thing, so started analysing every thing and show it in diagrams.

the basic use case components:

- Specifications: - pressure detection - pressure monitoring - alarm activation - stay on for specific duration - no maintenance after 1 month

- Assumptions: - The sensor will not fail - The alarm will not fail - the sound level is constant

- Versioning:

  - V1: the current release

  - V2: Add 2 features [Adaptive alarm - circuit protection]

  - V3: environmental analysis [temperature - magnetic field - rough waves]

3. Method

As Embedded systems engineer who worked around a multitude levels of projects, the agile scrum methodology has been our number one choice for the Software Development Life Cycle (SDLC).

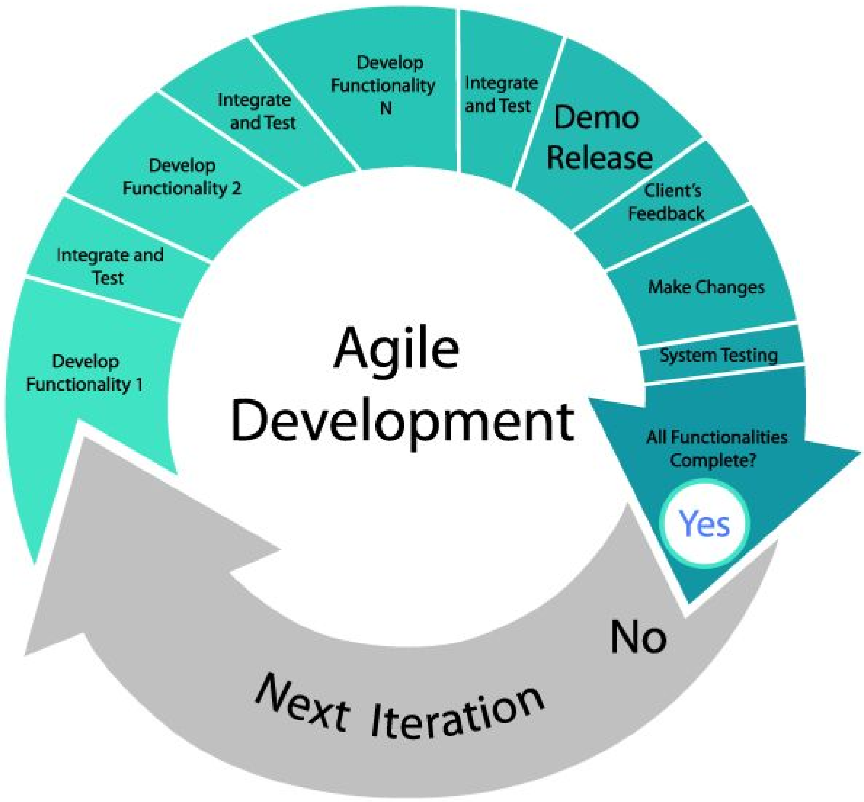

Figure 2: Agile Software Development Life Cycle

The scrum methodology validate the unclear vision and a enhance the version-based products, making the addational changes suitable to applay at any phase of the project.

It also provides the future improvements required by the customer.

4. Requirements

As the customer changes has no end, we applied our requirements diagram on the full system level that the user has already in his high hills climbing car, our phase one requirements provided in the blocks of IDs number 6, 8, 2, and 10

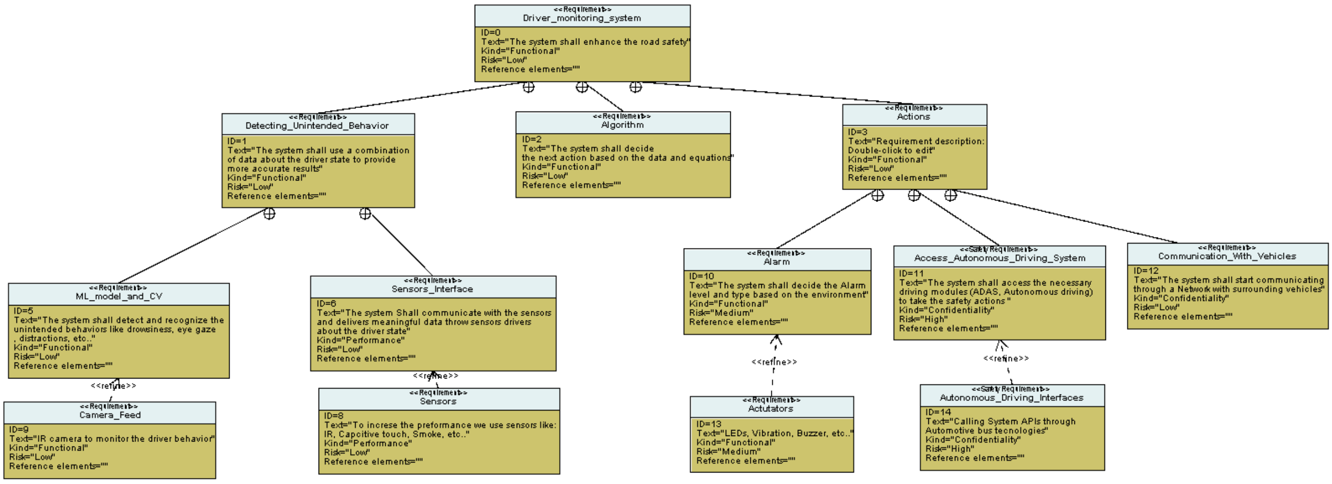

Figure 3: Requirements Diagram

5. Space Exploration or Partitioning

Hardware/software partitioning, also known as hardware/software co-design, is an important stage in the design of embedded systems and, if well executed, can result in a significant improvement in system performance. The process of hardware/software partitioning, as the name implies, involves deciding which system components should be implemented in hardware and which should be implemented in software.

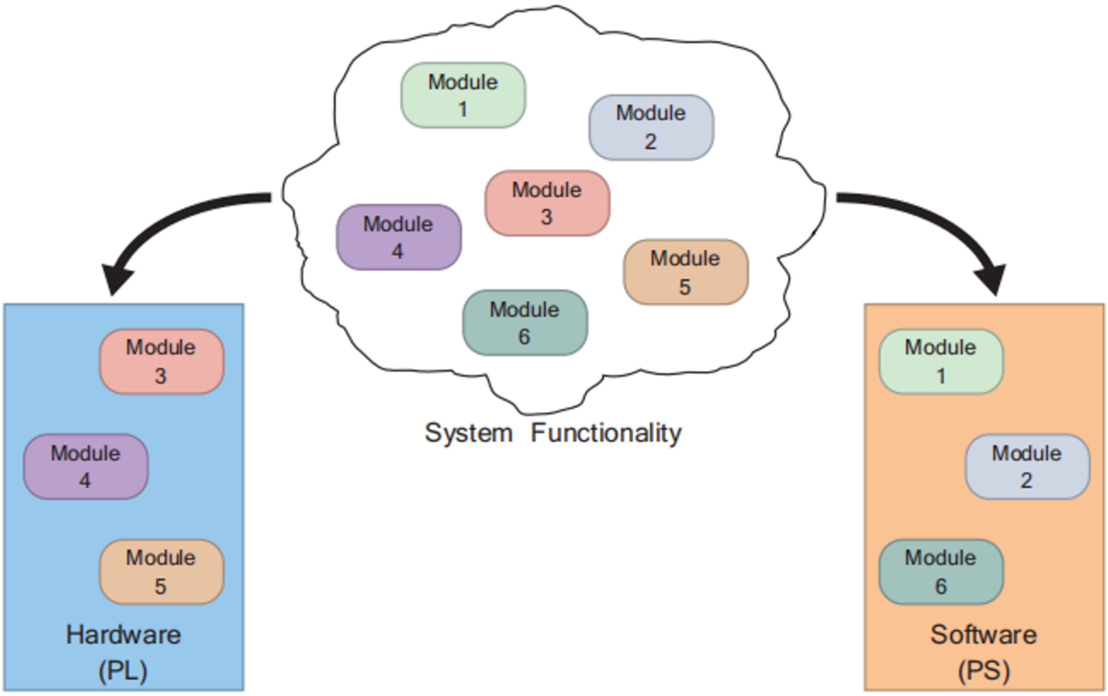

Figure 4: Hardware/software modules partitioning

5.1 Hardware Partitioning

Here we can assign some tasks from the requirements like the pressure sensor and the LED or the buzzer

5.2 Software Partitioning

On the other hand, the rest of requirements are software tasks like pressure monitoring, main algorithm and calculations

6. System Analysis

System analysis is the process of identifying and understanding the problem or need, gathering detailed requirements, and creating models to represent the system. It aims to define what the system should do and assess its feasibility, ultimately laying the foundation for system design. which can be afforded by our three detailed diagrams: Use case, activity and sequence diagrams

NOTE: For the following diagrams the pressure monitoring system is embedded within the car system [sensors - algorithm - actuators].

6.1 Use case diagram

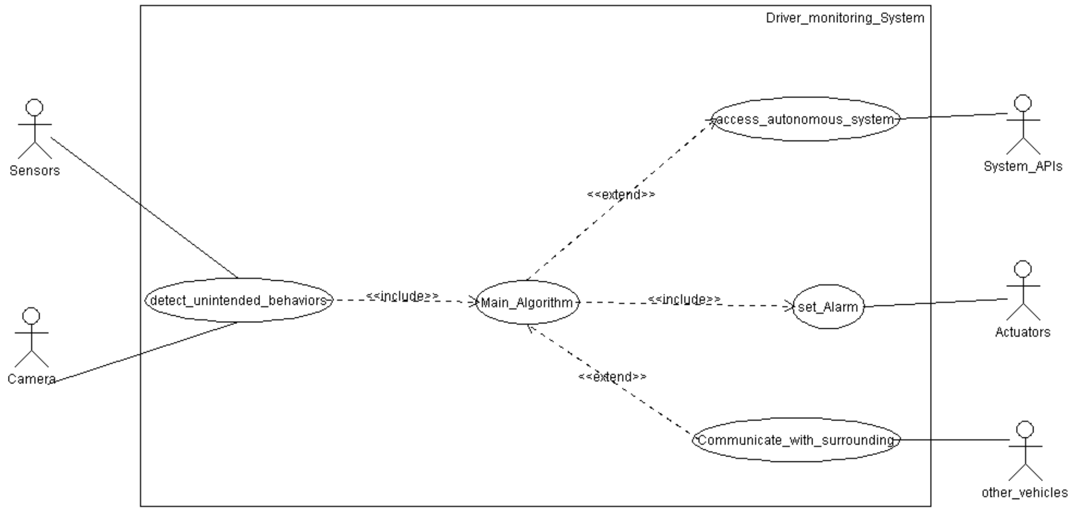

Figure 5: Use case diagram for the complete system of the car

6.2 Activity diagram

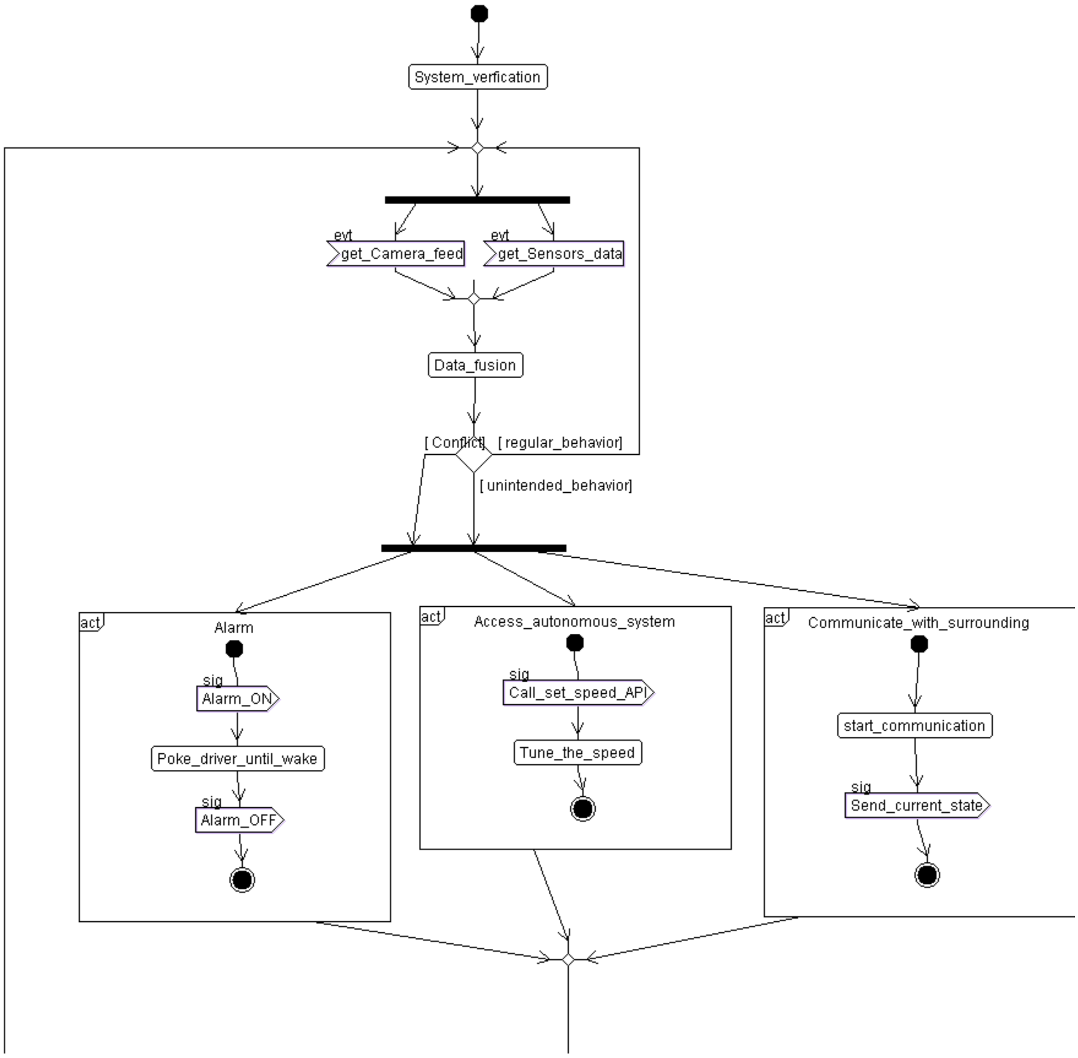

Figure 6: Activity diagram for the complete system of the car

6.3 Sequence diagram

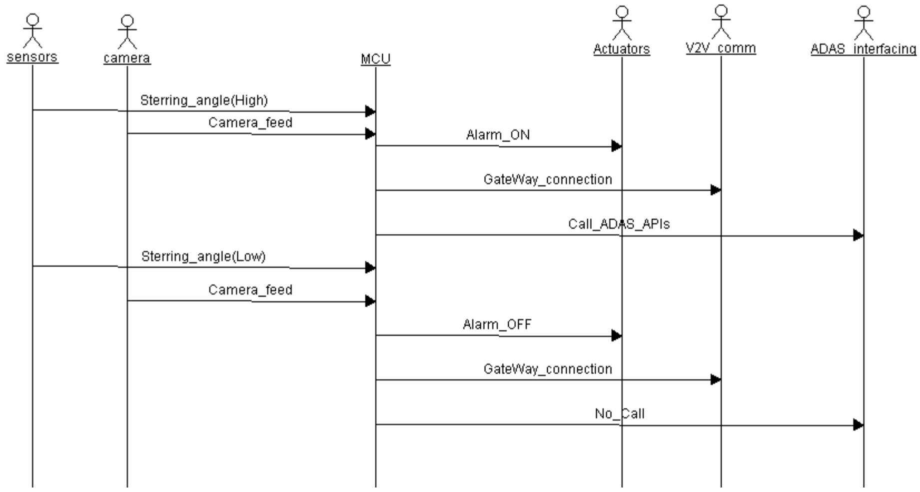

Figure 7: Sequence diagram for the complete system of the car

7. System Design

System design is the phase where the overall structure and components of the system are planned and detailed. It involves architectural, high-level, and detailed design, database and user interface design, coding, testing, deployment, and ongoing maintenance. The goal is to transform the requirements into a working and efficient system utilizing the state diagrams.

NOTE: For the following diagrams the pressure monitoring system is embedded within the car system [sensors - algorithm - actuators].

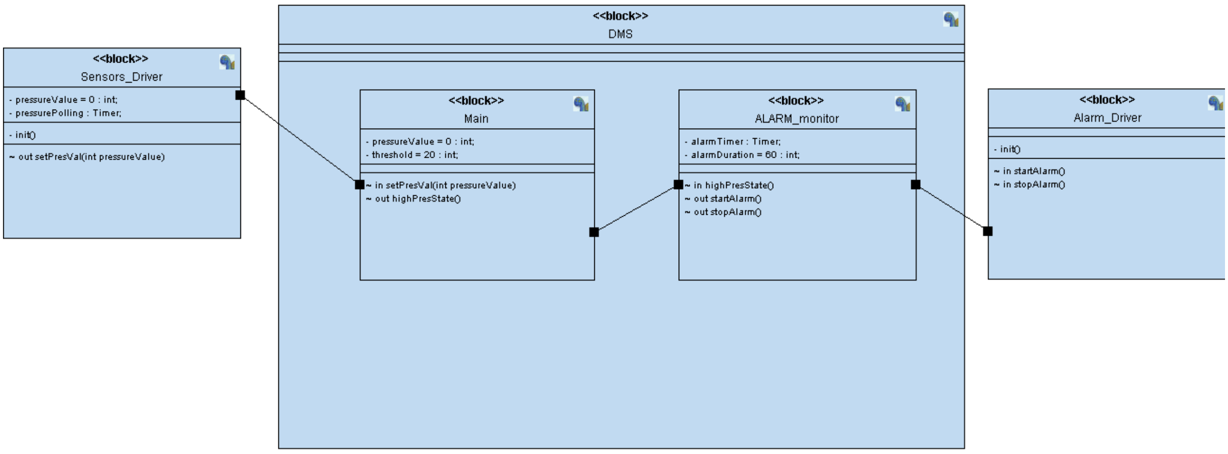

Figure 8: System modules diagram

7.1 Main state diagram

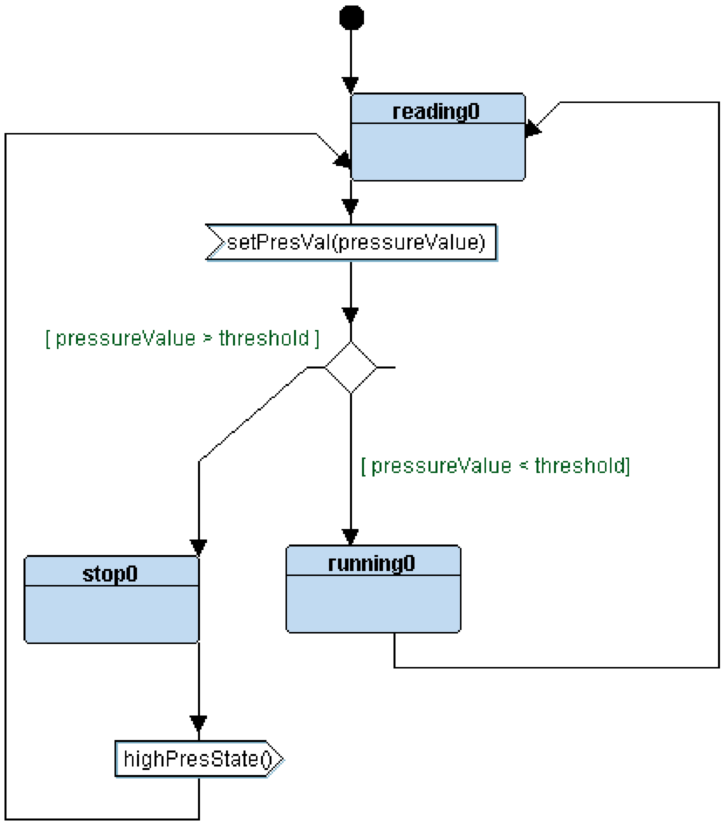

Figure 9: Main algorithm state diagram for the complete system of the car

7.2 Sensors state diagram

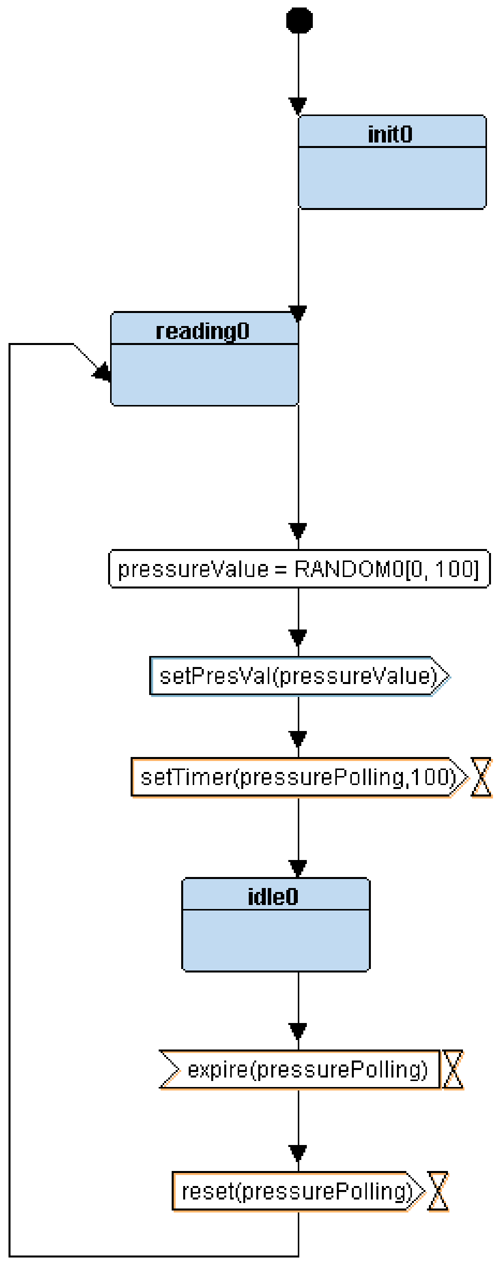

Figure 10: Sensors state diagram for the complete system of the car

7.3 Actuators state diagram

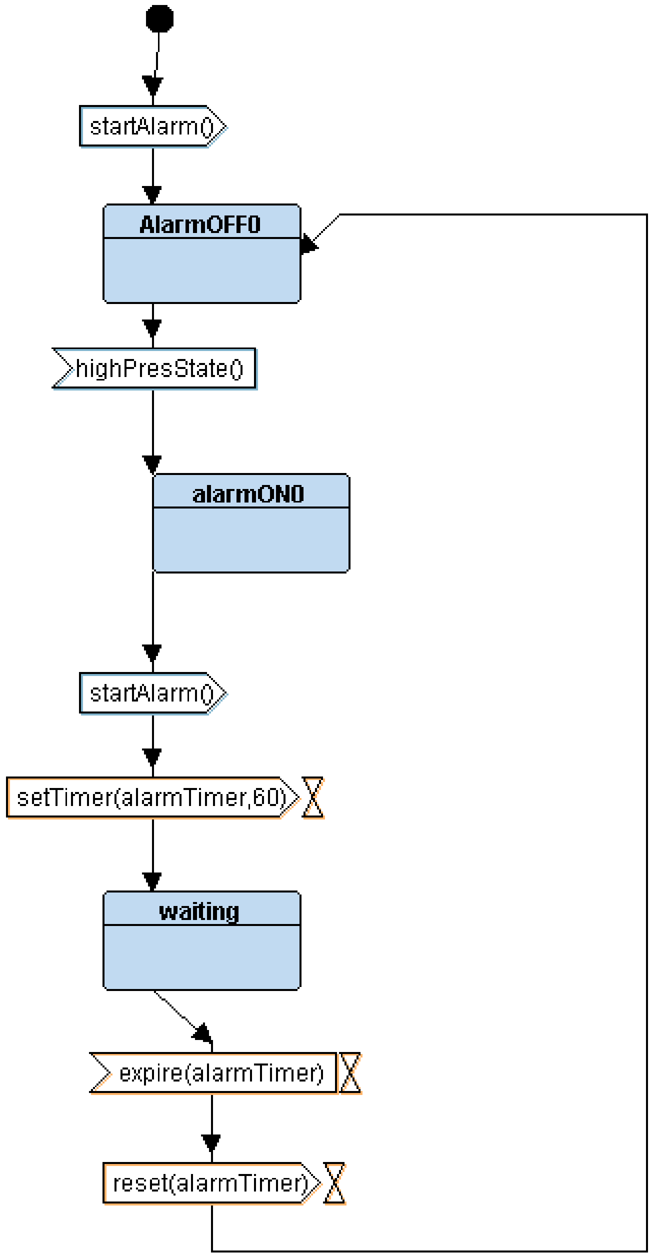

Figure 11: Alarm monitoring state diagram for the complete system of the car

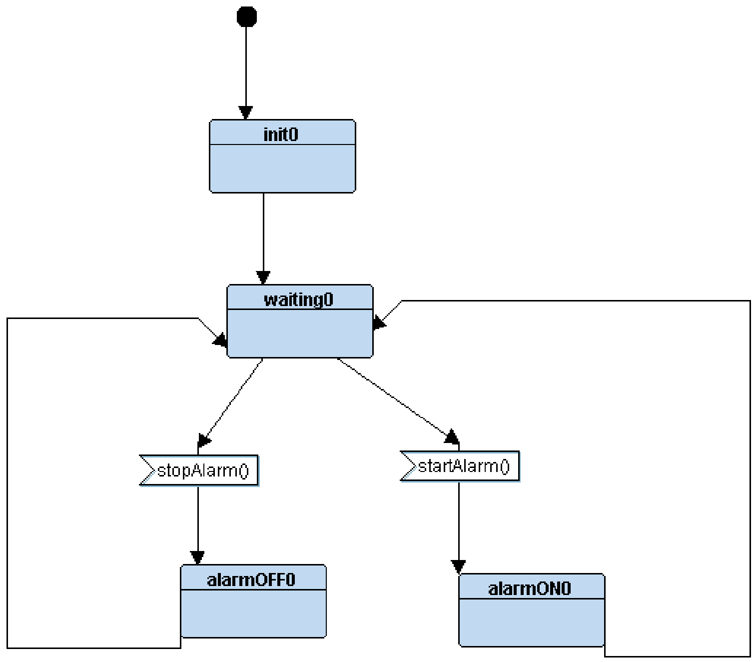

Figure 12: Alarm state diagram for the complete system of the car

8. Startup

```c
/* included for data types */
#include <stdint.h>

/* extern Linker variables */
extern uint32_t _E_text;
extern uint32_t _S_data;
extern uint32_t _E_data;
extern uint32_t _S_bss;
extern uint32_t _E_bss;
extern uint32_t _stack_pointer;

/* extern main function */
extern int main();

/* Handlers */
void reset_Handler();
void Default_Handler();


void NMI_Handler()__attribute__((weak,alias("Default_Handler")));
void H_fault_Handler()__attribute__((weak, alias("Default_Handler")));

/* Vectors Table */
uint32_t vectors[] __attribute__((section(".vectors"))) = {
    (uint32_t) &_stack_pointer, /* & as it's not a function */
    (uint32_t) reset_Handler, /* for fn & is optional */
    (uint32_t) NMI_Handler,
    (uint32_t) H_fault_Handler
};

/* Reset Handler implementation */
void reset_Handler(){

    /*source and destination pointers*/
    uint8_t* srcPtr = (uint8_t*)_E_text;
    uint8_t* dstPtr = (uint8_t*)_S_data;

    /* Sections sizes */
    uint32_t data_size = (uint8_t*)_E_data - (uint8_t*)_S_data;
    uint32_t bss_size  = (uint8_t*)_E_bss  - (uint8_t*)_S_bss;

    /*copying data section from flash to memory*/
    for (uint32_t i=0; i<data_size; i++){
        *dstPtr = *srcPtr;
        dstPtr = dstPtr + (uint8_t)1;
        srcPtr = srcPtr + (uint8_t)1;
    }

    /*initialize bss section in ram with 0*/
    for (uint32_t i=0; i<bss_size; i++){
        *dstPtr = (uint8_t)0;
        dstPtr = dstPtr + (uint8_t)1;
    }

    main();
}

/* the rest of Handlers Implementation */
void Default_Handler(){
    reset_Handler();
}
```

Figure 13: Startup C code for STM32f103c6

9. Linker Script

```
MEMORY
{
flash(RX): ORIGIN = 0x08000000, LENGTH = 128K
sram(RWX): ORIGIN = 0x02000000, LENGTH = 20K
}

SECTIONS
{
    .text : { /* _S_text = flash start so no need to store it*/
        *(.vectors*) /* The Startup vector table*/
        *(.text)
        *(.rodata)
        _E_text = . ;
    }> flash

    .data : {
        _S_data = . ;
        *(.data)
        _E_data = . ;
    }> sram AT>flash

    .bss : {
        _S_bss = . ;
        *(.bss)
        _E_bss = . ;
    }> sram

    . = . + 0x1000;
    _stack_pointer = . ;
}
```

Figure 14: Linker script for STM32f103c6

10. MakeFile

```makefile
CC=arm-none-eabi-
CFLAGS=-c -mcpu=cortex-m3 -gdwarf-2
INCS=-I .
LIBS=

SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

AS=$(wildcard *.s)
AsOBJ=$(AS:.s=.o)

project_name=pressureController

all : pressureController.bin
	@echo "Done Building"

%.o: %.c
	$(CC)gcc.exe $(INCS) $(CFLAGS) $< -o $@

$(project_name).elf : $(OBJ)
	$(CC)ld.exe -T Linker_script.ld $(LIBS) $(OBJ) -o $@ -Map mapFile.map

$(project_name).bin: $(project_name).elf
	$(CC)objcopy -O binary $< $@

clean_all:
	rm *.o *.elf *.bin

clean:
	rm *.elf *.bin
```

Figure 15: makefile to automate the building process

11. Application

11.1 Main algorithm

```c
#include <stdint.h>
#include <stdio.h>

#include "driver.h"

int main (){
    GPIO_INITIALIZATION();
    Set_Alarm_actuator(0);

    while (1)
    {
        if (getPressureVal() > 20){
            Set_Alarm_actuator(1);
            Delay(6000);
            Set_Alarm_actuator(0);
        }
    }
}
```

Figure 16: Main code logic

11.2 Drivers

```c
#include <stdint.h>
#include <stdio.h>

#define SET_BIT(ADDRESS,BIT)   ADDRESS |=  (1<<BIT)
#define RESET_BIT(ADDRESS,BIT) ADDRESS &= ~(1<<BIT)
#define TOGGLE_BIT(ADDRESS,BIT)  ADDRESS ^=  (1<<BIT)
#define READ_BIT(ADDRESS,BIT) ((ADDRESS) &   (1<<(BIT)))


#define GPIO_PORTA 0x40010800
#define BASE_RCC   0x40021000

#define APB2ENR   *(volatile uint32_t *)(BASE_RCC + 0x18)

#define GPIOA_CRL *(volatile uint32_t *)(GPIO_PORTA + 0x00)
#define GPIOA_CRH *(volatile uint32_t *)(GPIO_PORTA + 0X04)
#define GPIOA_IDR *(volatile uint32_t *)(GPIO_PORTA + 0x08)
#define GPIOA_ODR *(volatile uint32_t *)(GPIO_PORTA + 0x0C)


void Delay(int nCount);
int getPressureVal();
void Set_Alarm_actuator(int i);
void GPIO_INITIALIZATION ();

```

Figure 17: Pressure controller driver header

```c
#include "driver.h"
#include <stdint.h>
#include <stdio.h>
void Delay(int nCount)
{
    for(; nCount != 0; nCount--);
}

int getPressureVal(){
    return (GPIOA_IDR & 0xFF);
}

void Set_Alarm_actuator(int i){
    if (i == 1){
        SET_BIT(GPIOA_ODR,13);
    }
    else if (i == 0){
        RESET_BIT(GPIOA_ODR,13);
    }
}

void GPIO_INITIALIZATION (){
    SET_BIT(APB2ENR, 2);
    GPIOA_CRL &= 0xFF0FFFFF;
    GPIOA_CRL |= 0x00000000;
    GPIOA_CRH &= 0xFF0FFFFF;
    GPIOA_CRH |= 0x22222222;
}

```

Figure 18: Pressure controller driver implementation

12. Binary Utilities

Sections, symbols, architecture, disassemble and other valuable data can be given from the binary utilities provided by the arm-none-eabi- cross tool chain like:

1. `nm`: This utility displays symbol names and their values in object files, making it useful for inspecting the symbols and functions in a binary.

2. `objdump`: It provides detailed information about the object file, including disassembled code, sections, headers, and more, aiding in binary analysis and debugging.

3. `objcopy`: This tool is used for copying and converting object files, allowing you to manipulate and transform binary files as needed, such as extracting sections or changing formats.

4. `readelf`: It provides comprehensive information about the structure and attributes of ELF (Executable and Linkable Format) files, including headers, sections, and program headers, aiding in the analysis and debugging of executables and shared libraries.

This data is considered the X-ray of our embedded software solution as we can see every memory section, symbols states, symbol table, the assemble code, and any data from the executable (relocatable) image.

mainly we can check most of this data through the linker map file which is provided within the GitHub repository.

12.1 Sections

Notice here the difference after adding integer uninitialized global variable (4 bytes in bss) and integer initialized global variable (4 bytes in data). also you can see that these 4 bytes are allocated in the Virtual Memory Address (VMA) or the flash and in the Load Memory Address (LMA) or the RAM

```
$ arm-none-eabi-objdump.exe -h pressureController.elf

pressureController.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001a8  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .debug_info   00001538  00000000  00000000  000101a8  2**0
                  CONTENTS, READONLY, DEBUGGING
  2 .debug_abbrev 0000042a  00000000  00000000  000116e0  2**0
                  CONTENTS, READONLY, DEBUGGING
  3 .debug_loc    000001e8  00000000  00000000  00011b0a  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_aranges 00000060  00000000  00000000  00011cf2  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_line   00000428  00000000  00000000  00011d52  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_str    000005d3  00000000  00000000  0001217a  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .comment      0000007e  00000000  00000000  0001274d  2**0
                  CONTENTS, READONLY
  8 .ARM.attributes 00000033  00000000  00000000  000127cb  2**0
                  CONTENTS, READONLY
  9 .debug_frame  0000011c  00000000  00000000  00012800  2**2
                  CONTENTS, READONLY, DEBUGGING
```

Figure 19: The elf image sections before using global variables

```
$ arm-none-eabi-objdump.exe -h pressureController.elf

pressureController.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001bc  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  02000000  080001bc  00020000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  02000004  080001c0  00020004  2**2
                  ALLOC
  3 .debug_info   00001579  00000000  00000000  00020004  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_abbrev 00000457  00000000  00000000  0002157d  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_loc    000001e8  00000000  00000000  000219d4  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_aranges 00000060  00000000  00000000  00021bbc  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_line   00000429  00000000  00000000  00021c1c  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_str    000005f2  00000000  00000000  00022045  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .comment      0000007e  00000000  00000000  00022637  2**0
                  CONTENTS, READONLY
 10 .ARM.attributes 00000033  00000000  00000000  000226b5  2**0
                  CONTENTS, READONLY
 11 .debug_frame  0000011c  00000000  00000000  000226e8  2**2
                  CONTENTS, READONLY, DEBUGGING
```

Figure 20: The elf image section after using global variables

12.2 Symbols

Notice here the difference before adding the bss and data sections all symbols were assigned to the text section as it the only available section.

```
$ arm-none-eabi-nm pressureController.elf
02000000 T _E_bss
02000000 T _E_data
080001a8 T _E_text
02000000 T _S_bss
02000000 T _S_data
02001000 T _stack_pointer
0800019c T Default_Handler
08000010 T Delay
08000030 T getPressureVal
08000084 T GPIO_INITIALIZATION
0800019c W H_fault_Handler
080000d4 T main
0800019c W NMI_Handler
08000104 T reset_Handler
08000048 T Set_Alarm_actuator
08000000 T vectors
```

Figure 21: The elf image symbols before using global variables

```
$ arm-none-eabi-nm.exe pressureController.elf
02000008 B _E_bss
02000004 D _E_data
080001bc T _E_text
02000004 B _S_bss
02000000 D _S_data
02001008 B _stack_pointer
080001b0 T Default_Handler
08000010 T Delay
08000030 T getPressureVal
08000084 T GPIO_INITIALIZATION
080001b0 W H_fault_Handler
080000d4 T main
080001b0 W NMI_Handler
02000004 B pressure
08000118 T reset_Handler
08000048 T Set_Alarm_actuator
02000000 D threshold
08000000 T vectors
```

Figure 22: The elf image symbols after using global variables

12.3 Shell Script

Another powerful tool we can use is shell scripting language, to get all of this data with writing the commands only once and then can be executed any number of times by only one writing the shell name in the terminal. We created our custom script to generate all required data like Sections, symbols, architecture and the disassemble and write then to some text files. The used commands are as illustrated in 23, and you can reach the script it self through the GitHub repository.

```
echo "############Startup.c symbols###########" >> symbols.txt
arm-none-eabi-nm startup.o >> symbols.txt
echo "############main.c symbols###########" >> symbols.txt
arm-none-eabi-nm main.o >> symbols.txt
echo "############Driver.c Symbols###########" >> symbols.txt
arm-none-eabi-nm driver.o >> symbols.txt
echo "############all symbols###########" >> symbols.txt
arm-none-eabi-nm pressureController.elf >> symbols.txt
echo "############Reading the .elf image (-a to execute them all)###########" >> symbols.txt
echo "############The symbols table###########" >> symbols.txt
arm-none-eabi-readelf.exe -s pressureController.elf >> symbols.txt
echo "############Sections headers###########" >> sections.txt
arm-none-eabi-readelf.exe -S pressureController.elf >> sections.txt
echo "############Sections###########" >> sections.txt
arm-none-eabi-objdump.exe -h pressureController.elf >> sections.txt
arm-none-eabi-readelf.exe -A pressureController.elf >> architecture.txt
arm-none-eabi-objdump.exe -D pressureController.elf >> Disassembly.txt
```

Figure 23: Shell script to automate data generation

13. Error Log

- Here i missed understanding of the Symbol concept, should i use the address of operator '&' with it or not and as a symbol that representing an address i should have used it.

  Trying to use the value of `stackpointer` directly will lead to unpredictable errors like the one in 24

```
$ arm-none-eabi-gcc.exe -c startup.c -o startup.o
startup.c:86:2: error: initializer element is not constant
  (uint32_t) _stack_pointer,
  ^
startup.c:86:2: note: (near initialization for 'vectors[0]')
```

Figure 24: Stack pointer error

- Incrementing the pointer value is just by adding 1 and cast it to **unsigned char** no need to cast it as a pointer

```
$ arm-none-eabi-gcc.exe -c startup.c -o startup.o
startup.c:86:2: error: initializer element is not constant
  (uint32_t) _stack_pointer,
  ^
startup.c:86:2: note: (near initialization for 'vectors[0]')
startup.c: In function 'reset_handler':
startup.c:109:19: error: invalid operands to binary + (have 'uint8
_t * {aka unsigned char *}' and 'uint8_t * {aka unsigned char *}')

   dstPtr = dstPtr + (uint8_t*)1;
                   ^
startup.c:109:10: warning: assignment from incompatible pointer ty
pe [-Wincompatible-pointer-types]
   dstPtr = dstPtr + (uint8_t*)1;
          ^
startup.c:110:19: error: invalid operands to binary + (have 'uint8
_t * {aka unsigned char *}' and 'uint8_t * {aka unsigned char *}')

   srcPtr = srcPtr + (uint8_t*)1;
                   ^
startup.c:110:10: warning: assignment from incompatible pointer ty
pe [-Wincompatible-pointer-types]
   srcPtr = srcPtr + (uint8_t*)1;
          ^
```

Figure 25: incremental address error

- we can not use **nm** utility to show the symbols from a .bin file
- use -Map without '=' to avoid writing the map file in the terminal
- there is a problem with proteus version of the project [UNSOLVED]

14. Conclusion

In this project report, we have meticulously detailed the design and development of a pressure control system for a high-hill climbing car. The project commenced with a comprehensive project overview, emphasizing both design and coding sequences. Notably, we began from scratch, creating startup code, linker scripts, and makefiles to ensure a well-structured foundation.

A critical phase of this endeavor was the case study, which involved understanding the customer's requirements and versioning needs. We applied Agile Scrum methodology to enable adaptability and iterative refinement, ensuring that the evolving customer requirements were accommodated seamlessly. Through detailed requirements analysis, hardware/software partitioning, and system modeling, we laid the groundwork for a robust pressure monitoring system embedded within the car.

This project report also offers valuable insights into startup code, linker scripts, makefiles, algorithm design, and driver implementation, making it a comprehensive resource for those interested in embedded system development. Moreover, it highlights challenges faced and their resolutions, providing a holistic view of the project's journey.