Mastering Embedded System Online Diploma

www.learn-in-depth.com

First Term (Final Project 1)

Pressure Detection System Report

Eng. Mohamed Hazem Yahya Mahrous Ali

Contents:

Introduction

This report presents an analysis of the Pressure Detection System, including its code implementation and a breakdown of the hardware/software partitioning. The Pressure Detection System is designed to monitor pressure values and activate an alarm when the pressure exceeds a certain threshold.

System Overview

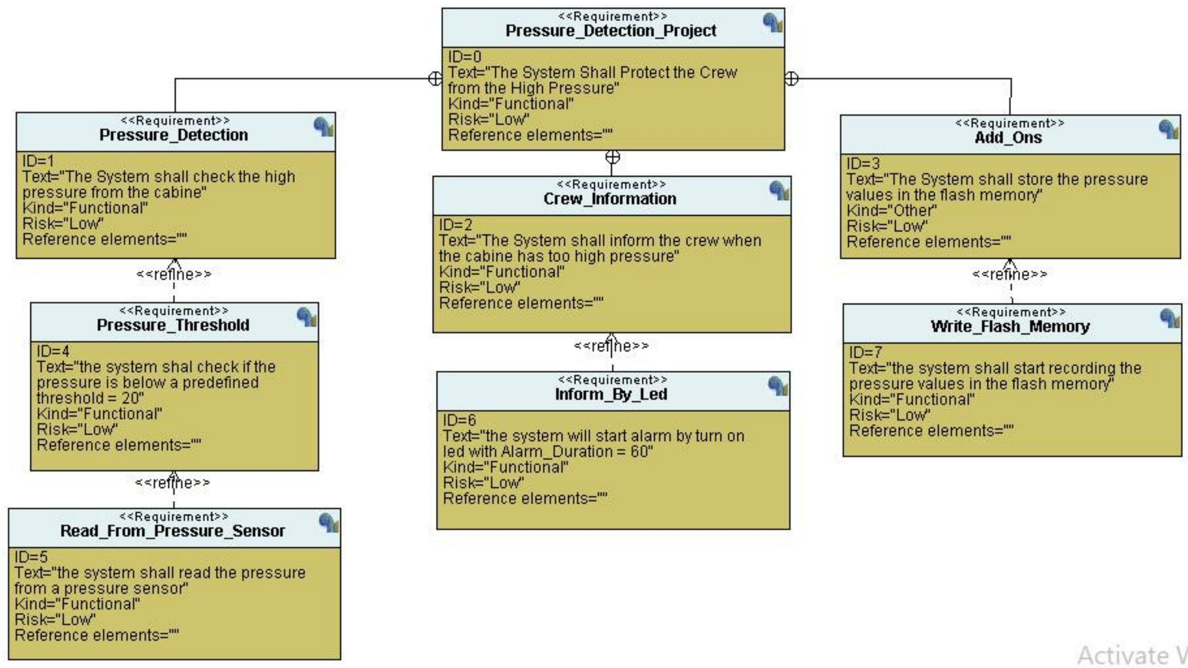
The Pressure Detection System consists of multiple components that interact to achieve its functionality:

1. **Pressure Sensor Driver**: Responsible for reading pressure values from a sensor.
2. **Alarm Monitor**: Monitors pressure values and triggers the alarm if needed.
3. **Alarm Actuator Driver**: Controls the alarm actuator, turning it on or off.

The main goal of the system is to detect pressure values and respond accordingly by triggering the alarm when the pressure exceeds a predefined threshold.

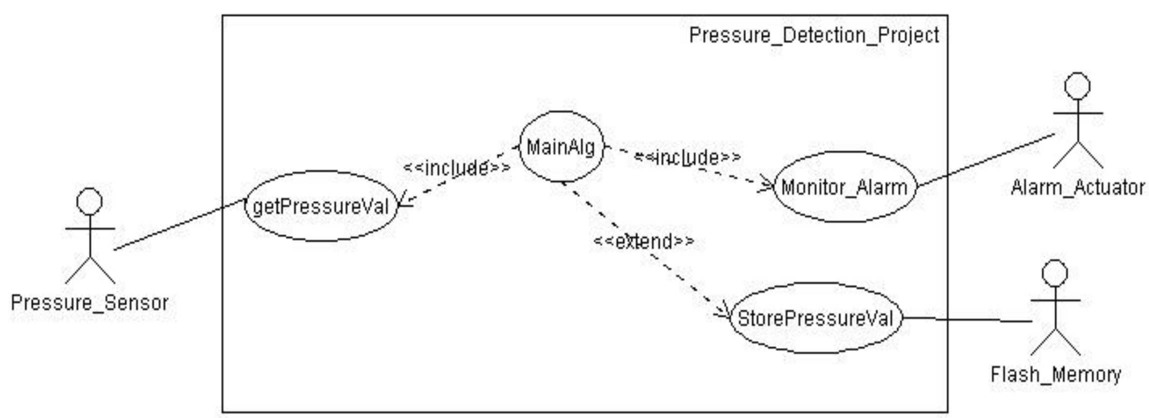
System Analysis

1- Requirements

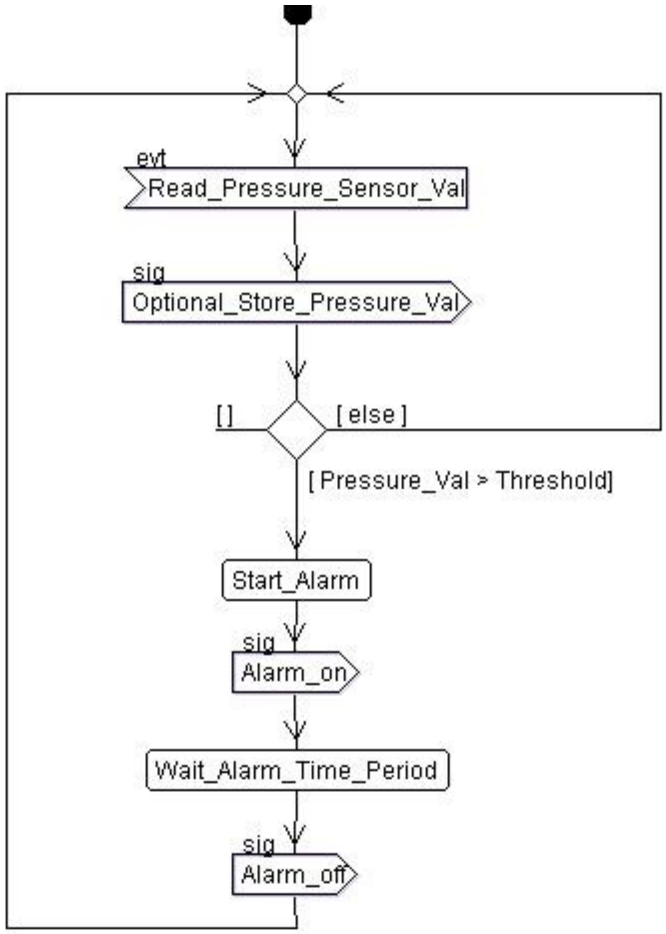


2- Analysis

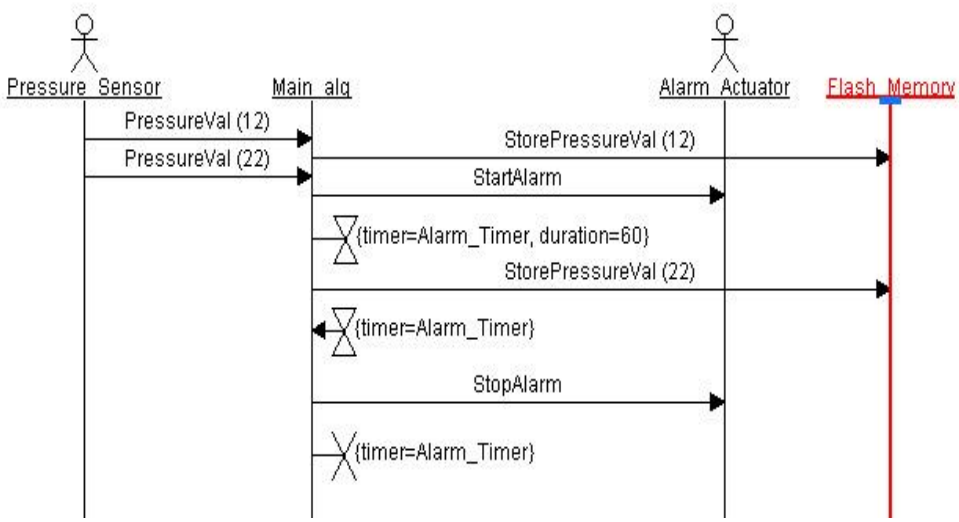
a. Use Case Diagram



b. Activity Diagram

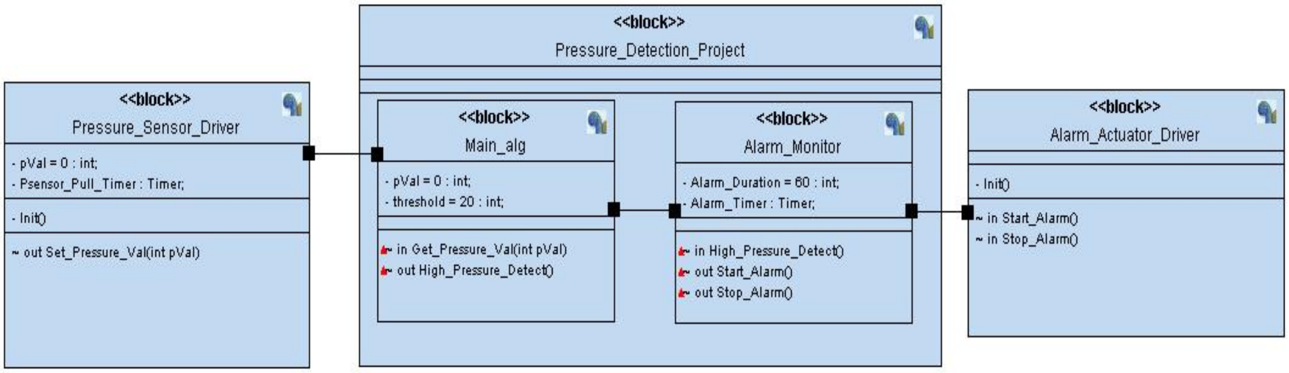


c. Sequence Diagram



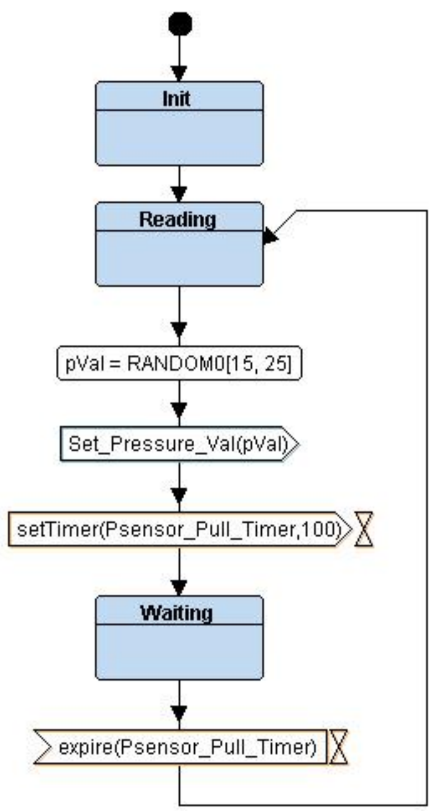
3- Design

a. Block Diagram

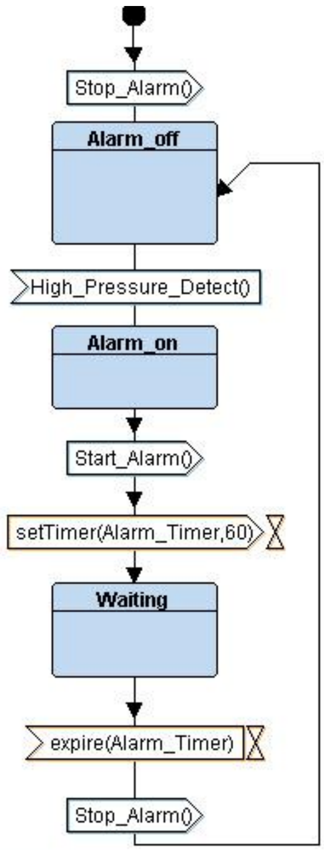


b. State Machine

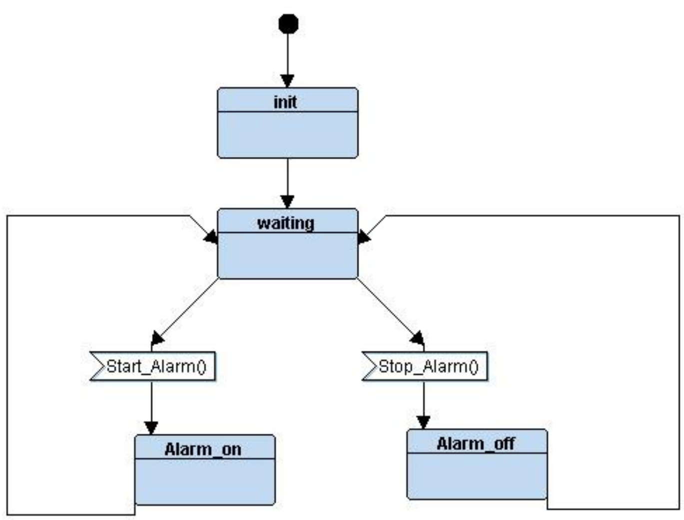
i. Pressure Sensor Driver



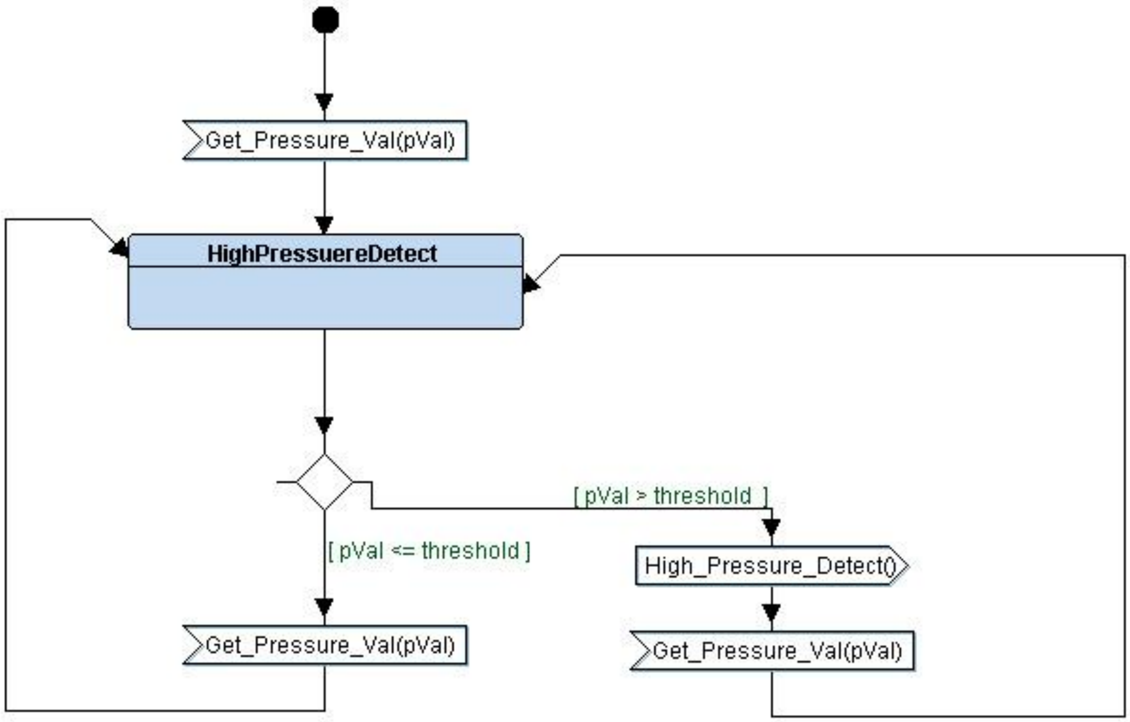
ii. Alarm Monitor



iii. Alarm Actuator Driver



iv. Main Program



Hardware/Software Partitioning

Hardware Components:

1. **Microcontroller Unit (MCU)**:
    - The central hardware component that houses the microprocessor, memory, GPIO ports, and other peripherals.
    - Responsible for executing the software code and interacting with hardware components.
2. **Pressure Sensor**:
    - A hardware component responsible for measuring pressure values.
    - Connects to the microcontroller via GPIO pins to transmit pressure readings.
    - Purely hardware; pressure values are read from the sensor using hardware interactions.
3. **Alarm Actuator**:
    - A hardware component that generates alarm signals (e.g., sound, light) when activated.
    - Controlled by the microcontroller through GPIO pins to turn the alarm on/off.
    - Purely hardware; alarm activation is triggered by hardware interactions.
4. **GPIO Pins**:
    - General-purpose input/output pins on the microcontroller.
    - Used for digital communication between the microcontroller and external components (sensor and actuator).
    - Hardware component, but their manipulation is facilitated by software.

Software Components:

1. **Microcontroller Firmware**:
   - Written in C, the firmware is the main software code running on the microcontroller.
   - Contains the logic for system initialization, state transitions, and interaction with hardware components.
   - Executes the main loop that reads pressure values, monitors alarms, and controls the alarm actuator.
2. **State Management**:
   - A collection of state functions (pSensor_state, AM_state, AA_state) that define the system's behavior.
   - Each state function encapsulates a specific system state and its associated actions.
   - Facilitates state transitions and ensures that the system responds appropriately to pressure changes.
3. **Driver Functions**:
   - Software functions that provide an abstraction layer to interact with hardware registers and GPIO pins.
   - Enable communication with the pressure sensor (reading pressure values) and the alarm actuator (activating/deactivating the alarm).
   - Abstract the low-level hardware interactions, making it easier to control the hardware components.
4. **Main Program**:
   - The main loop of the firmware, which continuously iterates through state functions.
   - Orchestrates the execution of state functions, pressure reading, alarm monitoring, and actuator control.
   - Reads pressure values from the sensor and triggers the alarm actuator based on system conditions.

Code Implementation

The system's functionality is implemented in the provided code. The following sections describe the main components and their roles.

1. Driver Layer (driver.c, driver.h)

The driver layer provides low-level functions for hardware manipulation. It includes functions for setting and resetting individual bits in registers, reading pressure values from the GPIO port, and controlling the alarm actuator.

```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;
}
```

```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 ();
```

2. State Layer (state.h)

The state layer defines the different states of the system. It provides a mechanism to generate state functions automatically and defines state transition functions. In this system, there are states for reading pressure, waiting, and managing the alarm.

```c
/*...


#ifndef STATE_H_
#define STATE_H_

#include <stdio.h>
#include <stdlib.h>

//Automatic State Function Generator
#define STATE_define(_stateFUNC_) void ST_##_stateFUNC_()
#define STATE(_stateFUNC_) ST_##_stateFUNC_


// States Connection
void Set_Pressure_Val(int pVal);
void High_Pressure_Detect();
void Start_Alarm();
void Stop_Alarm();


#endif /* STATE_H_ */

```

3. Pressure Sensor Driver (Pressure_Sensor_Driver.c, Pressure_Sensor_Driver.h)

This component is responsible for reading pressure values from the sensor. It defines two states: "Reading" and "Waiting." The system alternates between these states to periodically read pressure values. If the pressure exceeds a threshold, it triggers the alarm.

```c
/*...

#include "Pressure_Sensor_Driver.h"

// Variables
int pVal = 0;
int pSensor_Pull_Timer = 100;

// State Pointer to function
void (*pSensor_state)();

void pSensor_INIT(){
    // printf("Pressure Sensor Init is done");
}
STATE_define(pSensore_Reading){
    // State Name
    pSensor_State_ID = pSensore_Reading;
    // Get Pressure Value
    pVal = getPressureVal();
    // Set pressure value
    Set_Pressure_Val(pVal);
    // call the waiting state
    pSensor_state = STATE(pSensore_Waiting);
}
STATE_define(pSensore_Waiting){
    // State Name
    pSensor_State_ID = pSensore_Waiting;

    // Delay the sensor before reading again
    Delay(pSensor_Pull_Timer);

    // call the reading state
    pSensor_state = STATE(pSensore_Reading);
}
```

```c
/*...


#ifndef PRESSURE_SENSOR_DRIVER_H_
#define PRESSURE_SENSOR_DRIVER_H_

#include "driver.h"
#include "state.h"

//Define States
enum{
    pSensore_Waiting,
    pSensore_Reading
}pSensor_State_ID;



// Declare States Functions pressure sensor
STATE_define(pSensore_Waiting);
STATE_define(pSensore_Reading);

void pSensor_INIT();

// State Pointer to function
extern void (*pSensor_state)();



#endif /* PRESSURE_SENSOR_DRIVER_H_ */
```

4. Alarm Monitor (Alarm_Monitor.c, Alarm_Monitor.h)

The Alarm Monitor oversees the pressure values and responds by activating or deactivating the alarm as needed. It defines states for the alarm being on, off, and waiting. The system transitions between these states based on pressure conditions.

```c
/*...

#include "Alarm_Monitor.h"
// Variables
int Alarm_Delay_Duration= 60;

// State Pointer to function
void (*AM_state)();

void AM_INIT(){
    // printf("Alarm Monitor Init is done");
}
void High_Pressure_Detect(){
    AM_state = STATE(AM_Alarm_on);
}
STATE_define(AM_Alarm_on){
    // State Name
    AM_State_ID = AM_Alarm_on;

    // Start tha alarm
    Start_Alarm();
    // call the waiting state
    AM_state = STATE(AM_Alarm_Waiting);
}
STATE_define(AM_Alarm_off){
    // State Name
    AM_State_ID = AM_Alarm_off;

    //stop the alarm
    Stop_Alarm();

    // call the waiting state
    AM_state = STATE(AM_Alarm_off);
}
```

```c
/*...


#ifndef ALARM_MONITOR_H_
#define ALARM_MONITOR_H_

#include "driver.h"
#include "state.h"

//Define States
enum{
    AM_Alarm_off,
    AM_Alarm_on,
    AM_Alarm_Waiting
}AM_State_ID;

// Declare States Functions Alarm monitor
STATE_define(AM_Alarm_off);
STATE_define(AM_Alarm_on);
STATE_define(AM_Alarm_Waiting);

void AM_INIT();

// State Pointer to function
extern void (*AM_state)();

#endif /* ALARM_MONITOR_H_ */

```

5. Alarm Actuator Driver (Alarm_Actuator_Driver.c, Alarm_Actuator_Driver.h)

This component controls the alarm actuator. It defines states for the alarm being on and off. The alarm actuator is activated or deactivated based on the system's state.

```c
/*...

#include "Alarm_Actuator_Driver.h"

// State Pointer to function
void (*AA_state)();

void AA_INIT(){
    // printf("Alarm Monitor Init is done");
}

void Start_Alarm(){
    AA_state = STATE(AA_Alarm_on);
}

void Stop_Alarm(){
    AA_state = STATE(AA_Alarm_off);
}

STATE_define(AA_Alarm_on){
    // State name
    AA_State_ID = AA_Alarm_on;

    // alarm on
    Set_Alarm_actuator(0);
}

STATE_define(AA_Alarm_off){
    // State name
    AA_State_ID = AA_Alarm_off;

    // alarm off
    Set_Alarm_actuator(1);
}
```

```c
/*...


#ifndef ALARM_ACTUATOR_DRIVER_H_
#define ALARM_ACTUATOR_DRIVER_H_

#include "driver.h"
#include "state.h"

//Define States
enum{
    AA_Alarm_off,
    AA_Alarm_on
}AA_State_ID;

// Declare States Functions Alarm actuator
STATE_define(AA_Alarm_off);
STATE_define(AA_Alarm_on);

void AA_INIT();

// State Pointer to function
extern void (*AA_state)();

#endif /* ALARM_ACTUATOR_DRIVER_H_ */
```

6. Main Program (main.c)

The main program initializes the system components, sets up the initial states, and enters a loop where it continuously reads pressure values, monitors alarms, and controls the alarm actuator.

```c
#include "driver.h"
#include "Pressure_Sensor_Driver.h"
#include "Alarm_Actuator_Driver.h"
#include "Alarm_Monitor.h"

void setup(){
    GPIO_INITIALIZATION();
    pSensor_INIT();
    AM_INIT();
    AA_INIT();
    // states
    pSensor_state = STATE(pSensore_Reading);
    AM_state = STATE(AM_Alarm_Waiting);
    AA_state = STATE(AA_Alarm_off);
}

int Pressure_Threshold = 20;

void Set_Pressure_Val(int pVal){
    if (pVal > Pressure_Threshold){
        High_Pressure_Detect();
    }
}
int main()
{
    setup();
    while (1)
    {
        pSensor_state();
        AM_state();
        AA_state();
    }
    return 0;
}
```

7. Startup Code (startup.c)

Startup code initializes the microcontroller's hardware, sets up the initial stack and heap, and prepares the environment for running the firmware.

```c
// startup.c
// Eng.Mohamed Hazem Yahya

#include "stdint.h"
extern int main();
extern unsigned int _Stak_Top;

void Reset_Handler ();
void Default_Handler(){
    Reset_Handler();
}
void NMI_Handler () __attribute__((weak, alias ("Default_Handler")));;
void H_Fault_Handler () __attribute__((weak, alias ("Default_Handler")));;
void MM_Fault_Handler () __attribute__((weak, alias ("Default_Handler")));;
void Bus_Fault () __attribute__((weak, alias ("Default_Handler")));;
void Usage_Fault_Handler () __attribute__((weak, alias ("Default_Handler")));;

uint32_t vectors[] __attribute__((section(".vectors"))) = {
    (uint32_t) &_Stak_Top,
    (uint32_t) &Reset_Handler,
    (uint32_t) &NMI_Handler,
    (uint32_t) &H_Fault_Handler,
    (uint32_t) &MM_Fault_Handler,
    (uint32_t) &Bus_Fault,
    (uint32_t) &Usage_Fault_Handler
};


extern unsigned int _E_text;

extern unsigned int _S_Data;
extern unsigned int _E_Data;

extern unsigned int _S_Bss;
extern unsigned int _E_Bss;
```

```c
void Reset_Handler(void){

    // copy data from flash to sram
    unsigned int Data_Size = (unsigned char*)&_E_Data - (unsigned char*)&_S_Data;  // know the size of the data

    unsigned char* P_SRC = (unsigned char*)&_E_text;  //know the source of the data which is end of text at FLASH
    unsigned char* P_DST = (unsigned char*)&_S_Data;  //know the destnation of the data which is start of data at SRAM

    for (int i = 0; i < Data_Size; i++)
    {
        *((unsigned char*)P_DST++) = *((unsigned char*)P_SRC++); //copy data from source to destnation
    }



    // Init BSS section by 0
    unsigned int Bss_Size = (unsigned char*)&_E_Bss - (unsigned char*)&_S_Bss;  // know the size of the bss

    P_DST = (unsigned char*)&_S_Bss;

    for (int i = 0; i < Bss_Size; i++)
    {
        *((unsigned char*)P_DST++) = (unsigned char)0; //make it all 0
    }


    // call main()
    main();
}
```

8. Linker Script (linkerscript.ld)

The linker script defines the memory layout of the microcontroller, specifying where the code, data, stack, and other sections are located in memory.

```
MEMORY
{
FLASH(RX)   : ORIGIN = 0x08000000 , LENGTH = 128K
SRAM(RWX)   : ORIGIN = 0x20000000 , LENGTH = 20K
}


SECTIONS
{
    .text : {
        *(.vectors*)
        *(.text*)
        *(.rodata)
        _E_text = .;
    }> FLASH


    .data : {
        _S_Data = .;
        *(.data)
        _E_Data = .;
        . = ALIGN(4);
    }> SRAM AT> FLASH

    .bss : {
        _S_Bss = .;
        *(.bss)
        _E_Bss = .;

        . = ALIGN(4);
        . = . + 0x1000;
        _Stak_Top = .;
    }> SRAM

```

9. Makefile

The Makefile automates the build process, compiling source files, linking them together, and generating the final binary file that can be flashed onto the microcontroller.

```makefile
CC=arm-none-eabi-
CFLAGS=-mcpu=cortex-m3 -gdwarf-2 -g
# Directories
SRC_DIR=src
OBJ_DIR=object_file
LINKER_DIR=linker_script
OUTPUT_DIR=output_project
# Includes
INCS=-I$(SRC_DIR)
# Libraries
LIBS=
# Collect source files
SRC = $(wildcard $(SRC_DIR)/*.c)
OBJ = $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SRC))

AS = $(wildcard $(SRC_DIR)/*.s)
ASOBJ = $(patsubst $(SRC_DIR)/%.s, $(OBJ_DIR)/%.o, $(AS))

ProjectName = First_Term_Project_1

all: $(OUTPUT_DIR)/$(ProjectName).bin
	@echo "==================== Build is Done ====================="

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.s
	@mkdir -p $(OBJ_DIR)
	$(CC)as.exe $(CFLAGS) $< -o $@
	@echo "====================== "$@" is Done ======================"

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
	@mkdir -p $(OBJ_DIR)
	$(CC)gcc.exe -c $(INCS) $(CFLAGS) $< -o $@
	@echo "====================== "$@" is Done ====================="

$(OUTPUT_DIR)/$(ProjectName).elf: $(OBJ) $(ASOBJ) $(LINKER_DIR)/linker_script.ld
	@mkdir -p $(OUTPUT_DIR)
	$(CC)ld.exe -T $(LINKER_DIR)/linker_script.ld $(LIBS) $(OBJ) $(ASOBJ) -o $@ -Map=$(OUTPUT_DIR)/Map_file.map
	cp $(OUTPUT_DIR)/$(ProjectName).elf $(OUTPUT_DIR)/$(ProjectName).axf
	@echo "===================== "$@" is Done ====================="

$(OUTPUT_DIR)/$(ProjectName).bin: $(OUTPUT_DIR)/$(ProjectName).elf
	$(CC)objcopy.exe -O binary $< $@
	@echo "===================== "$@" is Done ====================="


clean_all:
	rm -rf $(OBJ_DIR) $(OUTPUT_DIR)
	@echo "================== EveryThing is clean =================="


clean:
	rm -rf $(OBJ_DIR)/*.o $(OUTPUT_DIR)/*.bin $(OUTPUT_DIR)/*.elf $(OUTPUT_DIR)/*.map
```

Output Program

1. Memory Map and Symbol Table

The memory map (map file) and symbol table provide insights into how the program is organized in memory and the addresses of various functions and variables.

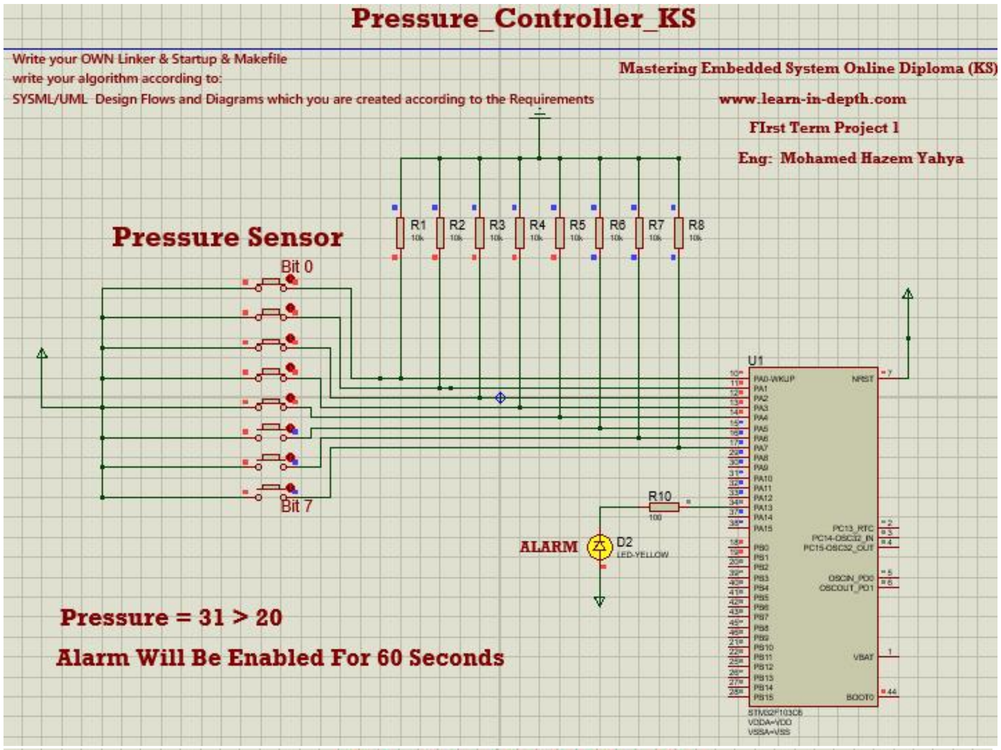
```
Memory Configuration

Name             Origin             Length             Attributes
FLASH            0x0000000008000000 0x0000000000020000 xr
SRAM             0x0000000020000000 0x0000000000005000 xrw
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map


.text           0x0000000008000000      0x388
 *(.vectors*)
 .vectors       0x0000000008000000       0x1c object_file/startup.o
                0x0000000008000000                vectors
 *(.text*)
 .text          0x000000000800001c       0x74 object_file/Alarm_Actuator_Driver.o
                0x000000000800001c                AA_INIT
                0x0000000008000028                Start_Alarm
                0x0000000008000044                Stop_Alarm
                0x0000000008000060                ST_AA_Alarm_on
                0x0000000008000078                ST_AA_Alarm_off
 .text          0x0000000008000090       0xa0 object_file/Alarm_Monitor.o
                0x0000000008000090                AM_INIT
                0x000000000800009c                High_Pressure_Detect
                0x00000000080000b8                ST_AM_Alarm_on
                0x00000000080000dc                ST_AM_Alarm_off
                0x0000000008000100                ST_AM_Alarm_Waiting
 .text          0x0000000008000130       0xc4 object_file/driver.o
                0x0000000008000130                Delay
                0x0000000008000150                getPressureVal
                0x0000000008000168                Set_Alarm_actuator
                0x00000000080001a4                GPIO_INITIALIZATION
 .text          0x00000000080001f4       0x90 object_file/main.o
                0x00000000080001f4                setup
                0x0000000008000238                Set_Pressure_Val
                0x000000000800025c                main
 .text          0x0000000008000284       0x74 object_file/Pressure_Sensor_Driver.o
                0x0000000008000284                pSensor_INIT
                0x0000000008000290                ST_pSensore_Reading
                0x00000000080002c8                ST_pSensore_Waiting
 .text          0x00000000080002f8       0x90 object_file/startup.o
                0x00000000080002f8                H_Fault_Handler
```

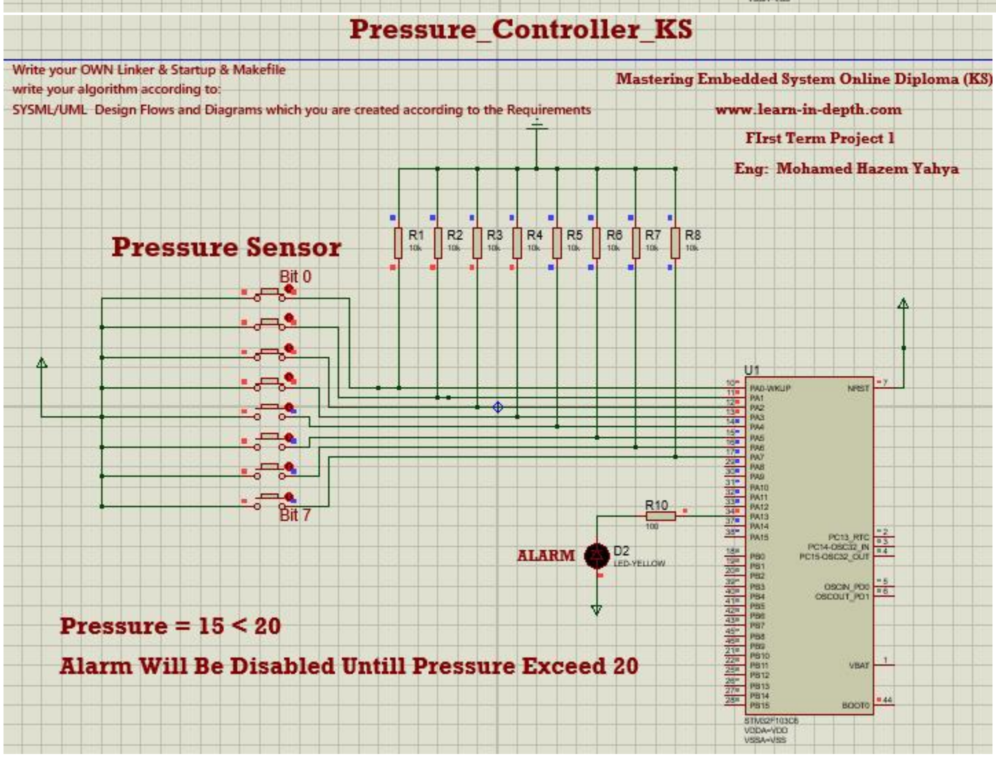
```
 *(.rodata)
                0x0000000008000388                _E_text = .

.glue_7         0x0000000008000388        0x0
 .glue_7        0x0000000008000388        0x0 linker stubs

.glue_7t        0x0000000008000388        0x0
 .glue_7t       0x0000000008000388        0x0 linker stubs

.vfp11_veneer   0x0000000008000388        0x0
 .vfp11_veneer  0x0000000008000388        0x0 linker stubs

.v4_bx          0x0000000008000388        0x0
 .v4_bx         0x0000000008000388        0x0 linker stubs

.iplt           0x0000000008000388        0x0
 .iplt          0x0000000008000388        0x0 object_file/Alarm_Actuator_Driver.o

.rel.dyn        0x0000000008000388        0x0
 .rel.iplt      0x0000000008000388        0x0 object_file/Alarm_Actuator_Driver.o

.data           0x0000000020000000        0xc load address 0x0000000008000388
                0x0000000020000000                _S_Data = .
 *(.data)
 .data          0x0000000020000000        0x0 object_file/Alarm_Actuator_Driver.o
 .data          0x0000000020000000        0x4 object_file/Alarm_Monitor.o
                0x0000000020000000                Alarm_Delay_Duration
 .data          0x0000000020000004        0x0 object_file/driver.o
 .data          0x0000000020000004        0x4 object_file/main.o
                0x0000000020000004                Pressure_Threshold
 .data          0x0000000020000008        0x4 object_file/Pressure_Sensor_Driver.o
                0x0000000020000008                pSensor_Pull_Timer
 .data          0x000000002000000c        0x0 object_file/startup.o
                0x000000002000000c                _E_Data = .
                0x000000002000000c                . = ALIGN (0x4)
```

```
$ arm-none-eabi-nm.exe First_Term_Project_1.elf
20000010 B _E_Bss
2000000c D _E_Data
08000388 T _E_text
2000000c B _S_Bss
20000000 D _S_Data
20001010 B _Stak_Top
0800001c T AA_INIT
20001010 B AA_state
20001014 B AA_State_ID
20000000 D Alarm_Delay_Duration
08000090 T AM_INIT
20001018 B AM_state
2000101c B AM_State_ID
080002f8 W Bus_Fault
080002f8 T Default_Handler
08000130 T Delay
08000150 T getPressureVal
080001a4 T GPIO_INITIALIZATION
080002f8 W H_Fault_Handler
0800009c T High_Pressure_Detect
0800025c T main
080002f8 W MM_Fault_Handler
080002f8 W NMI_Handler
20000004 D Pressure_Threshold
08000284 T pSensor_INIT
20000008 D pSensor_Pull_Timer
20001020 B pSensor_state
2000101d B pSensor_State_ID
2000000c B pVal
08000304 T Reset_Handler
08000168 T Set_Alarm_actuator
08000238 T Set_Pressure_Val
080001f4 T setup
08000078 T ST_AA_Alarm_off
08000060 T ST_AA_Alarm_on
080000dc T ST_AM_Alarm_off
080000b8 T ST_AM_Alarm_on
08000100 T ST_AM_Alarm_Waiting
08000290 T ST_pSensore_Reading
080002c8 T ST_pSensore_Waiting
08000028 T Start_Alarm
08000044 T Stop_Alarm
```

2. Simulation

Simulations allow you to observe how the program executes step by step, aiding in debugging and understanding the program's behavior.





Conclusion

The hardware/software partitioning clearly defines the roles of hardware components and software code in the Pressure Detection System. The microcontroller firmware, state management, and driver functions collaborate to create a functional and responsive system that reads pressure values, monitors alarms, and controls the alarm actuator. This partitioning ensures a clear separation of concerns, enabling efficient development, debugging, and maintenance of the system.