HIGH PRESSURE DETECTION

Mastering Embedded System Online Diploma

WWW.LEARN-IN-DEPTH.COM

FIRST TERM (FINAL PROJECT 1)

ENG. Nouran S.

My Profile

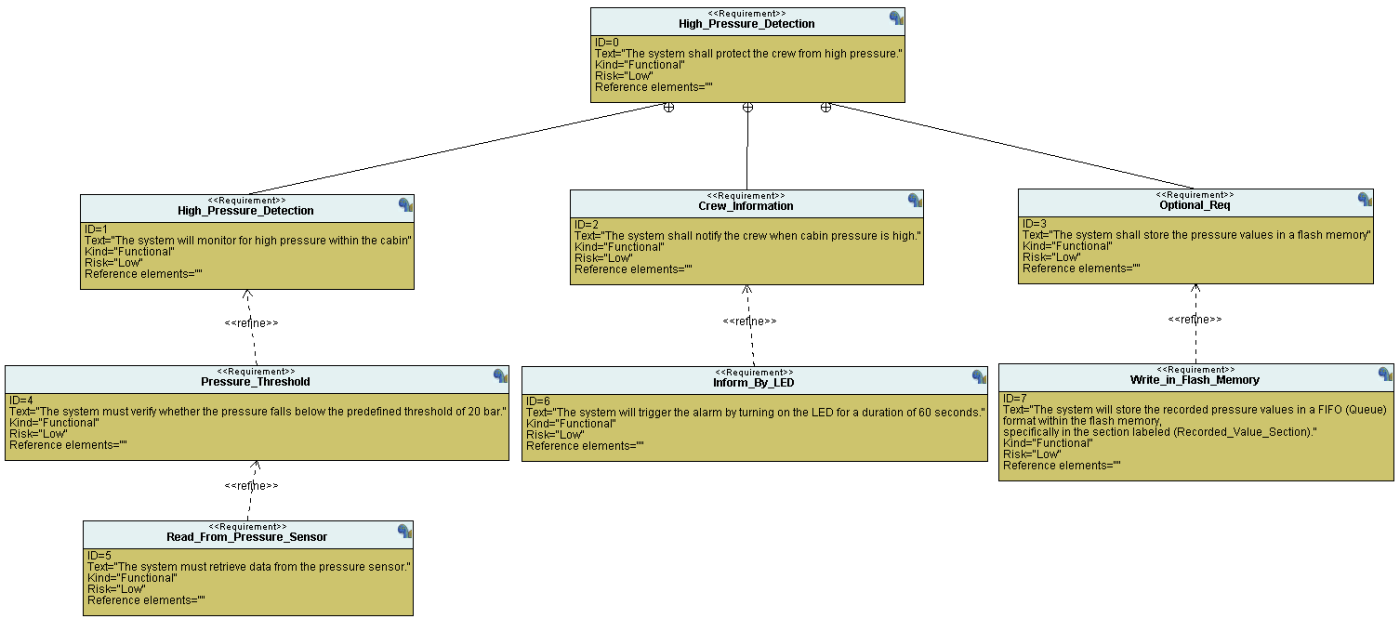
1. Project Overview:

- This embedded system project involves designing and implementing a high-pressure detection system that triggers an alarm when the pressure inside a cabin exceeds a certain threshold. The system is designed using state machines to manage the interactions between various components: the pressure sensor, alarm system, and controller. The system is developed in C and runs on an STM32 microcontroller, utilizing GPIO for hardware interaction.

2. System Specifications:

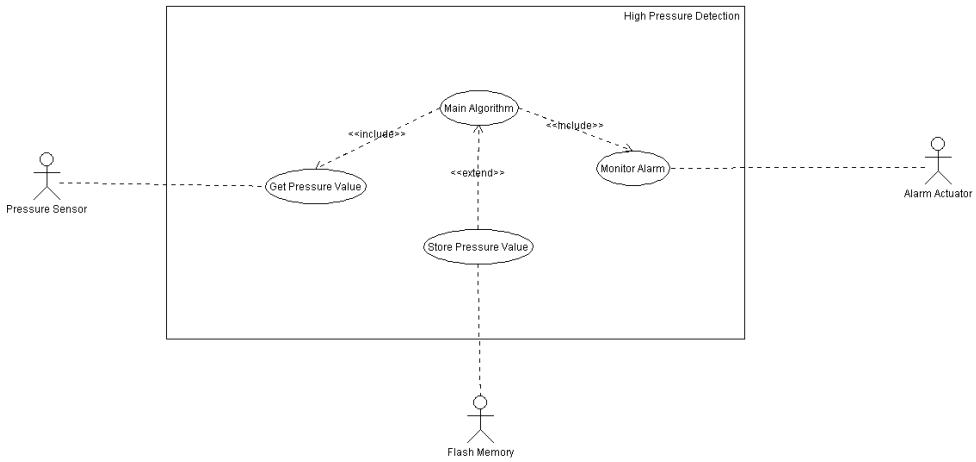
- The project is based on the following specifications:
  - A pressure controller must monitor the cabin pressure and trigger an alarm when the pressure exceeds 20 bars.
  - The alarm system should remain active for 60 seconds before resetting.

3. Requirements Diagram:

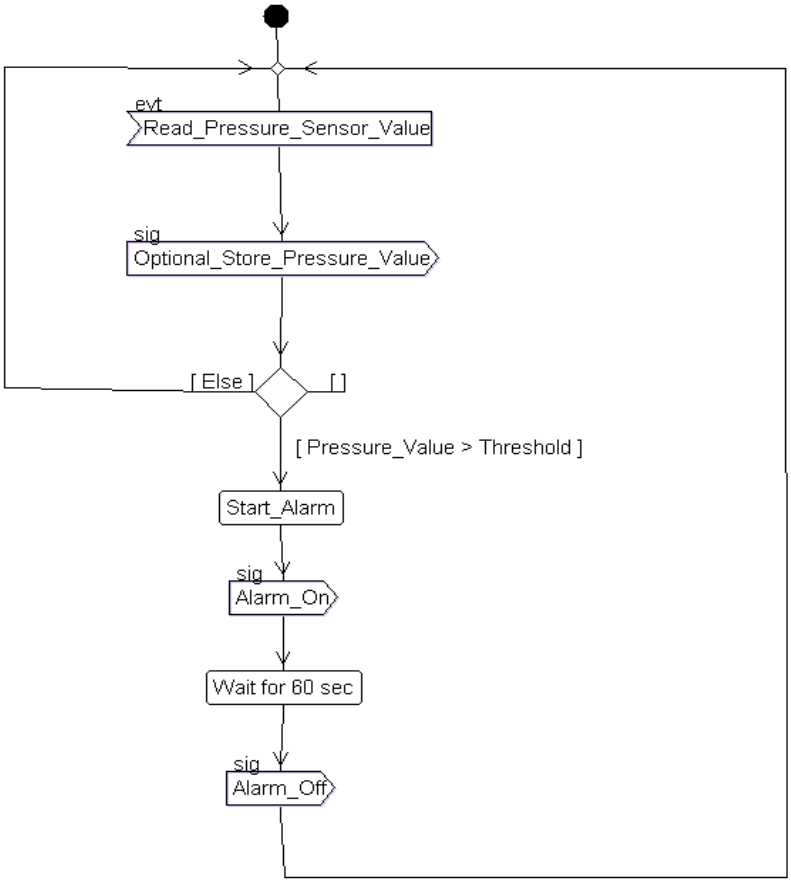


4. System Analysis:

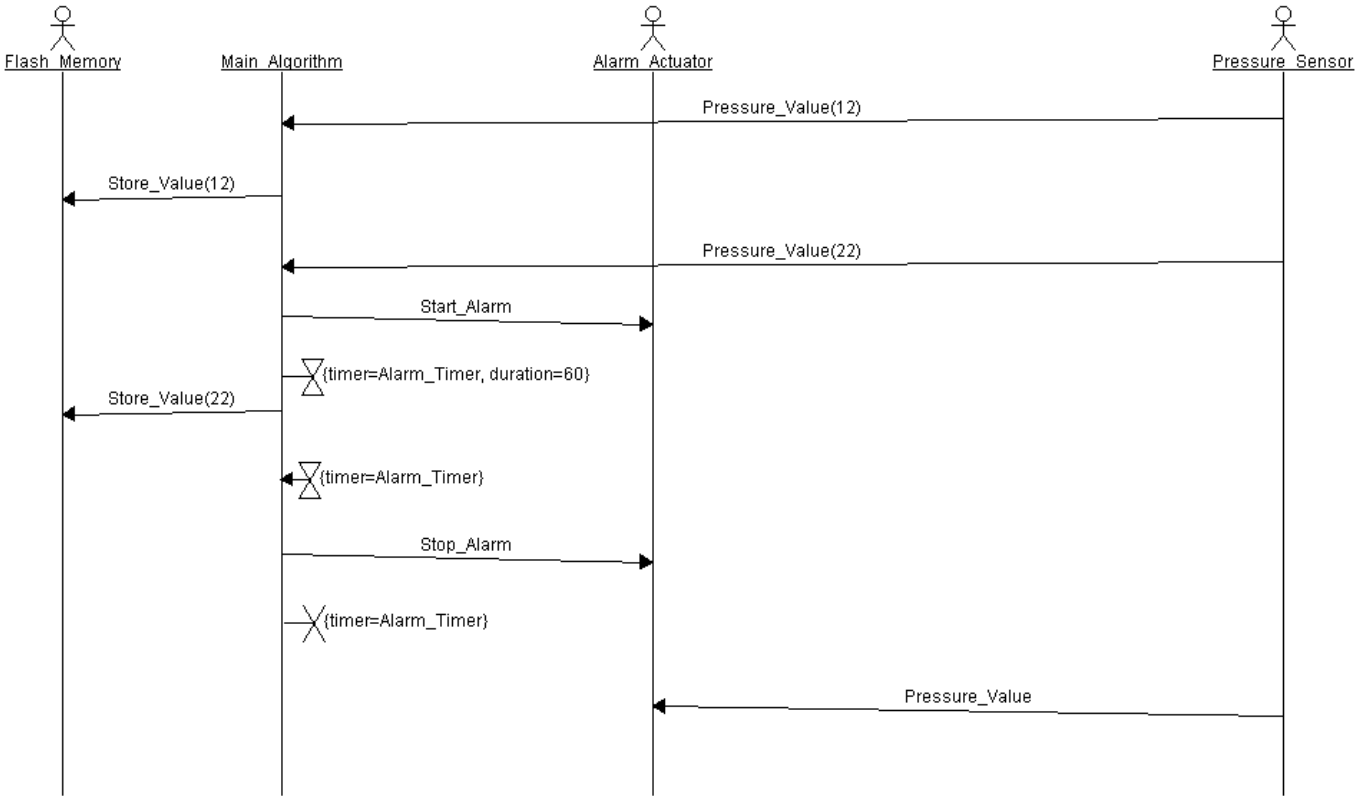
- Use Case Diagram



- Activity Diagram



- Sequence Diagram



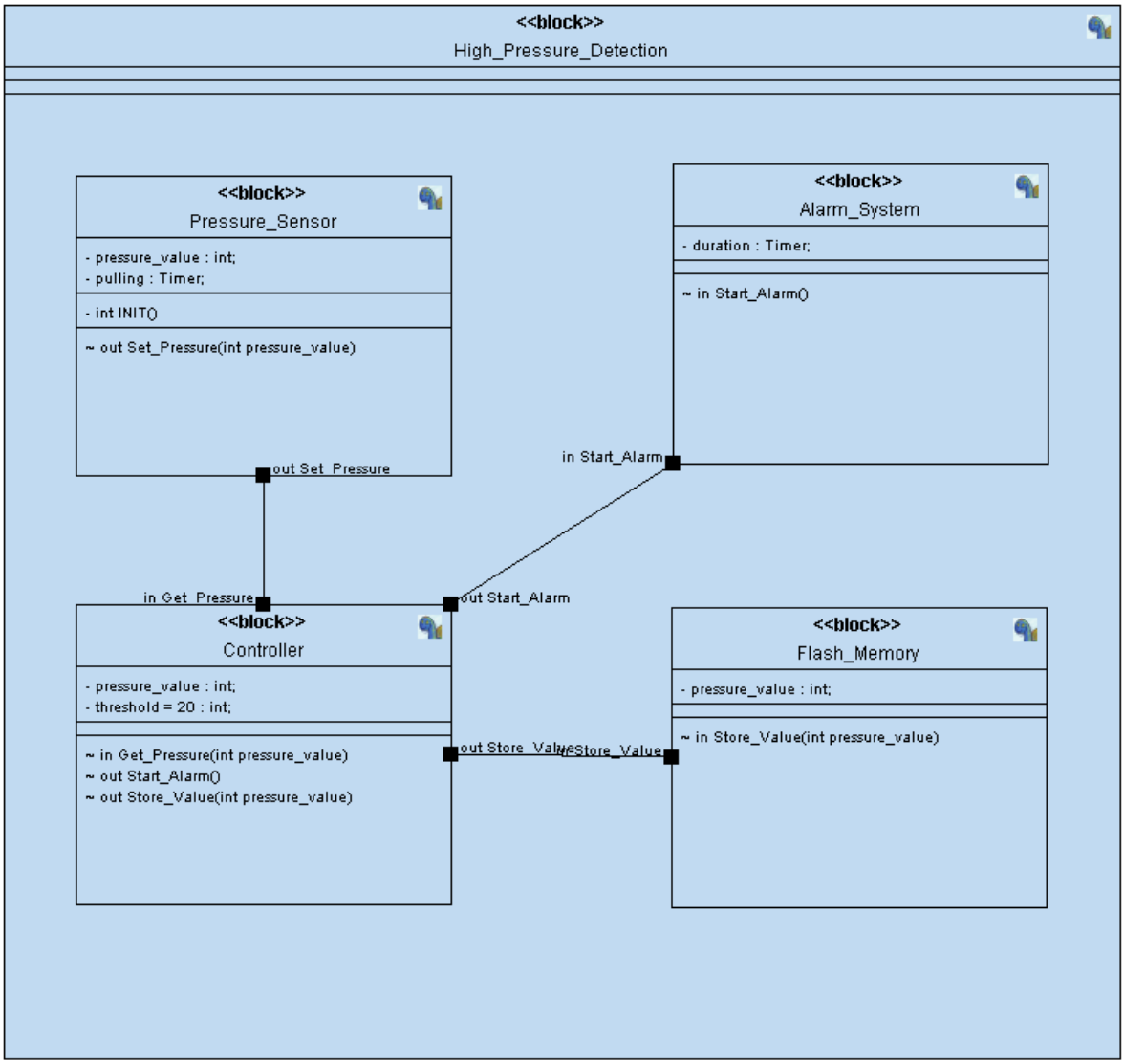
5. System Design:

State Machines

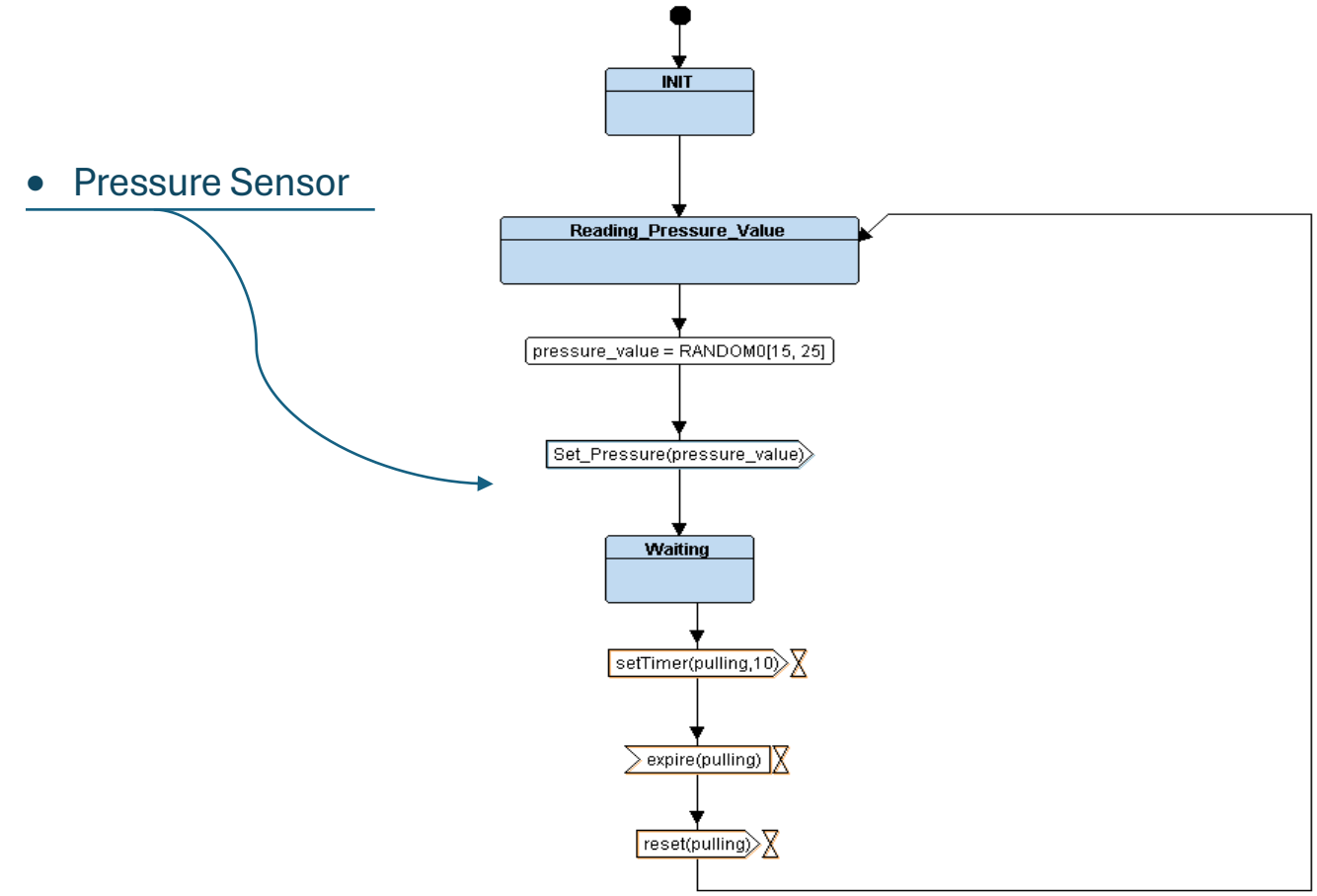
The system is divided into multiple states, each representing different phases of operation:

1. **Pressure Sensor**:
    - Sensor_Reading: Reads the pressure value.
    - Sensor_Waiting: Waits for a certain duration before reading the pressure again.
2. **Alarm System**:
    - Alarm_Waiting: Idle state where the alarm is off.
    - Alarm_Start: Activates the alarm.
    - Alarm_Stop: Deactivates the alarm after the set duration.

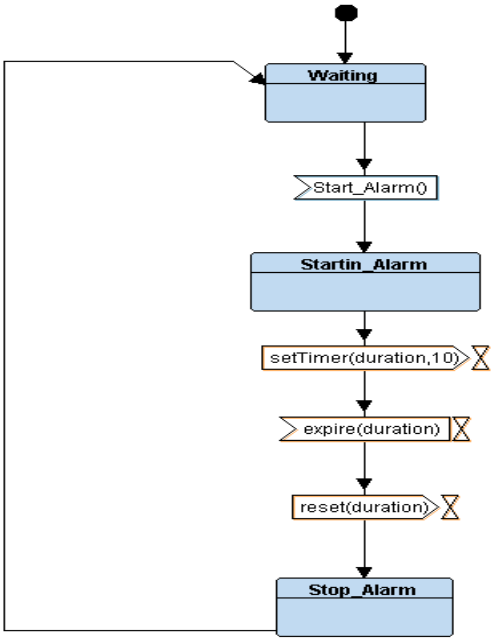
3. **Controller**:
    - Controller_Waiting: Idle state waiting for the pressure value.
    - Controller_AlarmOn: Activates the alarm if the pressure exceeds the threshold.
    - Controller_Storing: Stores the pressure value in memory if it's below the threshold.



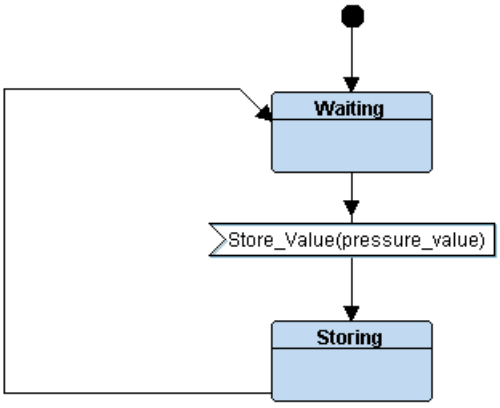
- Pressure Sensor



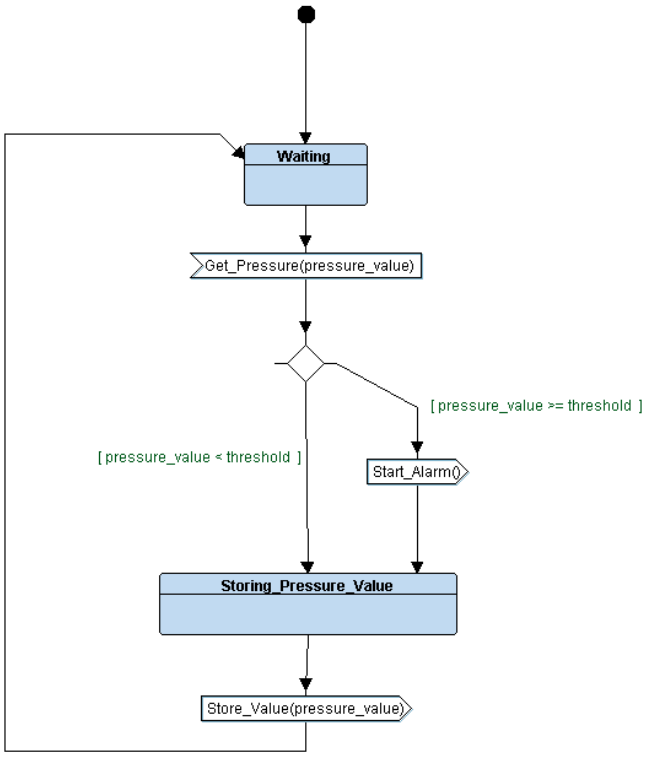
- Alarm System



- Flash Memory



- Controller



]

6. Code Implementation:

The project follows a modular approach, with separate C files for each component:

- **Pressure_Sensor.c:** Contains code for the pressure sensor logic and state transitions.
- **Alarm_System.c:** Handles the alarm system functionality.
- **Controller.c:** Implements the controller logic for decision-making based on the pressure readings.
- **driver.c:** Provides low-level GPIO and delay functions for interacting with hardware components.

➢ Linker Script

```
/* Linker Script for STM32F103C6 (Cortex-M3) */

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x8000000, LENGTH = 32K
    SRAM  (rwx) : ORIGIN = 0x20000000, LENGTH = 10K
}

SECTIONS
{
    /* Interrupt Vector Table */
    .isr_vector :
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector))   /* Keep the vector table in the binary
*/      . = ALIGN(4);
    } > FLASH

    /* Code section */
    .text :
    {
        . = ALIGN(4);
        *(.text)
        *(.text*)
        *(.rodata*)
        . = ALIGN(4);
        _E_text = .;
    } > FLASH

    /* Initialized data section */
    .data :
    {
        . = ALIGN(4);
        _S_data = .;
        *(.data)
        *(.data*)
        . = ALIGN(4);
        _E_data = .;
    } > SRAM AT > FLASH

    . = ALIGN(4);

    /* Uninitialized data (BSS) */
    .bss :
    {
        _S_bss = .;
        *(.bss)
        *(.bss*)
        . = ALIGN(4);
        _E_bss = .;
    } > SRAM
}
```

➢ Startup

```c
#include <stadint.h>

extern int main(void);

void Reset_Handler(void);
void Default_Handler(void) { Reset_Handler(); }

void NMI_Handler(void)        __attribute__((weak, alias("Default_Handler")));
void HardFault_Handler(void)   __attribute__((weak, alias("Default_Handler")));
void MemManage_Handler(void)   __attribute__((weak, alias("Default_Handler")));
void BusFault_Handler(void)    __attribute__((weak, alias("Default_Handler")));
void UsageFault_Handler(void)  __attribute__((weak, alias("Default_Handler")));
void DebugMon_Handler(void)    __attribute__((weak, alias("Default_Handler")));
void SVC_Handler(void)         __attribute__((weak, alias("Default_Handler")));
void PendSV_Handler(void)      __attribute__((weak, alias("Default_Handler")));
void SysTick_Handler(void)     __attribute__((weak, alias("Default_Handler")));

// Stack size: booking 1024 bytes
static unsigned long stack_top[256];

// Vector Table
__attribute__((section(".vectors")))
void (* const g_pfnVectors[])() = {
    (void (*)()) ((unsigned long) stack_top + sizeof(stack_top)),
    &Reset_Handler,
    &NMI_Handler,
    &HardFault_Handler,
    &MemManage_Handler,
    &BusFault_Handler,
    &UsageFault_Handler,
    0,
    0,
    0,
    0,
    &SVC_Handler,
    &DebugMon_Handler,
    0,
    &PendSV_Handler,
    &SysTick_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;

void Reset_Handler(void) {
    unsigned int i;

    // Copy data section from FLASH to SRAM
    unsigned int data_size = (unsigned char *)&_E_data - (unsigned char *)&_S_data;
    unsigned char *P_src = (unsigned char *)&_E_text;
    unsigned char *P_dst = (unsigned char *)&_S_data;

    for (i = 0; i < data_size; i++)
        *((unsigned char *)P_dst++) = *((unsigned char *)P_src++);

    // Initialize BSS section to 0
    unsigned int bss_size = (unsigned char *)&_E_bss - (unsigned char *)&_S_bss;
    P_dst = (unsigned char *)&_S_bss;
    for (i = 0; i < bss_size; i++)
        *((unsigned char *)P_dst++) = 0;

    // Jump to main
    main();
}
```

➢ Makefile

```
# Compiler and flags
CC = arm-none-eabi-
CFLAGS = -mcpu=cortex-m3 -mthumb -g -gdwarf-2
LDFLAGS = -T STM32F103C6TX_FLASH.ld -nostartfiles

# Include directories
INCS = -I .

# Source files
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
AS = $(wildcard *.s)
AsOBJ = $(AS:.s=.o)

# Project name
ProjectName = High_Pressure_Detection

# Targets
all: $(ProjectName).hex
    @echo "*************************"
    @echo "***   Build Complete   ***"
    @echo "*************************"

# Assembly files to object files
$(AsOBJ): $(AS)
    $(CC)as $(CFLAGS) $< -o $@
    @echo "*** Finished building assembly: $@ ***"
    @echo " "

# C files to object files
%.o: %.c
    $(CC)gcc -c $(CFLAGS) $(INCS) $< -o $@
    @echo "*** Finished building object: $@ ***"
    @echo " "

# Linker stage
$(ProjectName).elf: $(AsOBJ) $(OBJ)
    $(CC)ld $(LDFLAGS) $(OBJ) $(AsOBJ) -o $@ -Map=map_file.map
    @echo "*** Finished linking: $@ ***"
    @echo " "

# Create binary and hex files
$(ProjectName).bin: $(ProjectName).elf
    $(CC)objcopy -O binary $< $@
    @echo "*** Finished creating binary: $@ ***"
    @echo " "

$(ProjectName).hex: $(ProjectName).elf
    $(CC)objcopy -O ihex $< $@
    @echo "*** Finished creating hex: $@ ***"
    @echo " "

# Clean all generated files
clean_all:
    rm -f *.o *.bin *.elf *.hex *.map *.asm
    @echo "*** Cleaned all generated files ***"
    @echo " "

clean_o:
    rm -f *.o
    @echo "*** Cleaned object files ***"
    @echo " "
```

7. Main Program Flow:

- The system initializes the GPIO and state machines for the sensor, alarm, and controller in the SetUp() function. The main loop then continuously monitors the pressure, checks the alarm status, and updates the state machines accordingly.

```c
#include "driver.h"
#include "state.h"
#include "Alarm_System.h"
#include "Pressure_Sensor.h"
#include "Controller.h"

void SetUp()
{
    Sensor_INIT();
    Sensor_State_Ptr = STATE(Sensor_Waiting);
    Alarm_State_Ptr = STATE(Alarm_Waiting);
    Controller_State_Ptr = STATE(Controller_Waiting);
}

int main(void)
{
    SetUp();
    while(1)
    {
        Sensor_State_Ptr();
        Controller_State_Ptr();
        Alarm_State_Ptr();
    }
}
```