

SafePressure Alert

Mastering Embedded System Online Diploma

[www.learn-in-depth.com](http://www.learn-in-depth.com)

Implementation Document

**First Term (Final Project 1)**

Eng. Abdallah Shabaan Ghazy

email:[abdallah.shabaan.ghazy@gmail.com](mailto:abdallah.shabaan.ghazy@gmail.com)

**My Profile:**

<https://www.learn-in-depth-store.com/certificate/abdallah.shabaan.ghazy%40gmail.com>

### Design for **Pressure** Control System (PCS)

**Version 1.0 approved**

**Prepared by:** Abdallah Shabaan Ghazy

**Organization:** learn in depth

**Date Created:** 8/13/2024

### **Implementation Document**

#### 1. ****Introduction****

* **Objective:** This document describes the implementation of the Alarm Management System for pressure control. It aims to detail how the system was implemented and the specifics of each part to ensure compliance with the requirements.
* **Overview:** The system is designed to monitor pressure levels and alert users if thresholds are exceeded. It consists of multiple components that interact to ensure efficiency and accuracy.
* **Constraints:** Assumptions and constraints considered during implementation.

#### 2. ****System Description****

* **System Overview:** The system uses pressure sensors to read values and a control system to alert users based on these readings.
* **Key Components:**
  + Pressure sensors
  + Control unit
  + User interface
* **Technologies Used:**
  + Programming Language: C
  + Development Environment: GCC

#### 3. ****Implementation Requirements****

* **System Requirements:**
  + Read values from pressure sensors
  + Alert the user when thresholds are exceeded
  + Display values on the user interface
* **Performance Criteria:**
  + System response time less than 1 second
  + Pressure reading accuracy of 0.1 bar
* **Configuration Details:**
  + The system is configured to read pressure values every 0.5 seconds and update the user interface periodically.

#### 4. ****System Design****

* **Overall Structure:**
  + The system includes a control unit and a set of connected pressure sensors.
* **Module Design:**
  + Control Unit: Implements core logic and manages communications.
  + Pressure Sensors: Measure pressure and send data to the control unit.
* **Interfaces:**
  + Interface between the control unit and pressure sensors
  + Interface between the control unit and the user interface

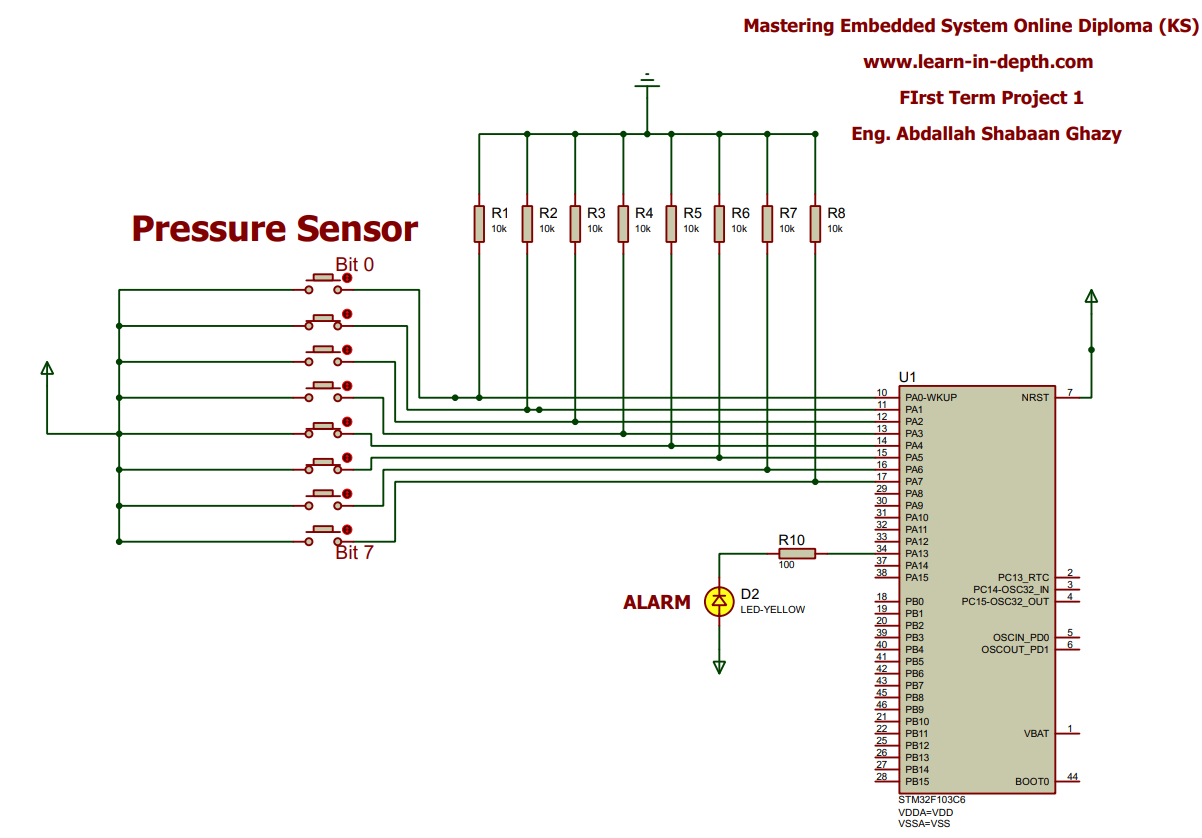
#### 5. ****System Implementation****

* **Implementation Details:**
  + The code was developed in C and all modules were successfully integrated.
* **Procedures:**
  + Code development
  + Module testing
  + System integration
* **Issues and Solutions:**
  + A problem with sensor data reading was discovered and resolved by adjusting the code.

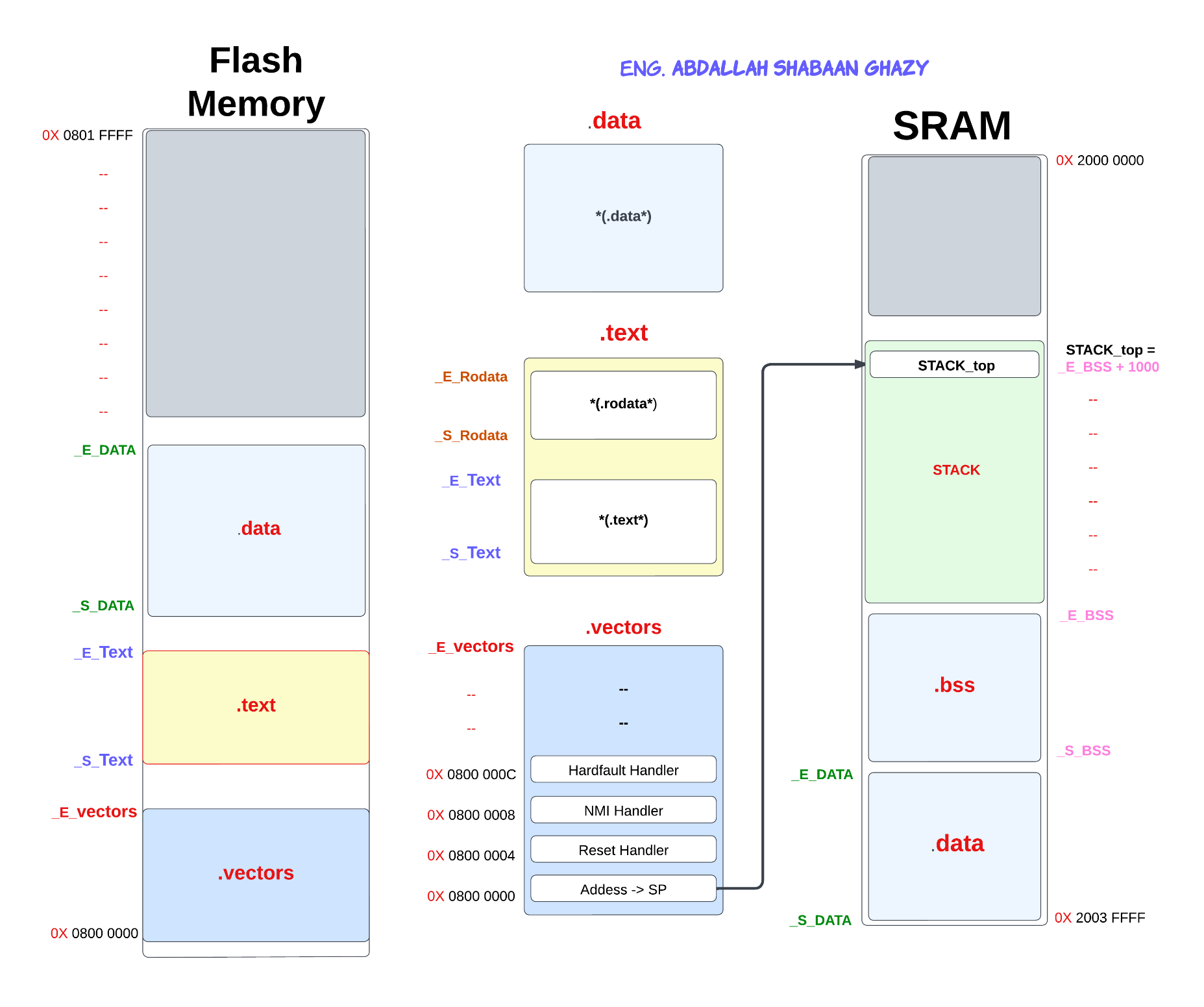
#### 6. ****System Testing****

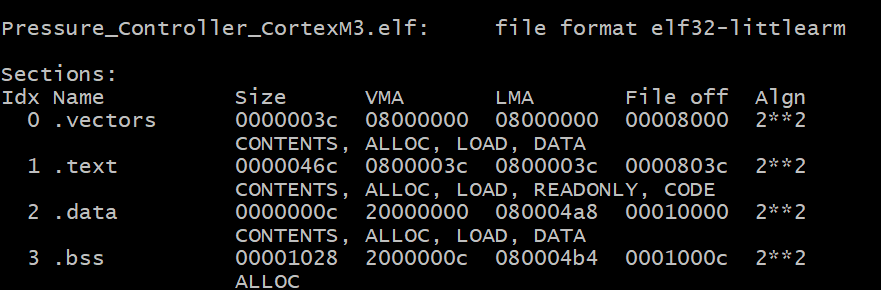
* **Test Plans:**
  + Test reading values from pressure sensors
  + Test the alert system
* **Test Results:**
  + All tests passed successfully, with minor improvements to system performance.
* **Issues and Tests:**
  + A slight delay in system response was detected and improved.

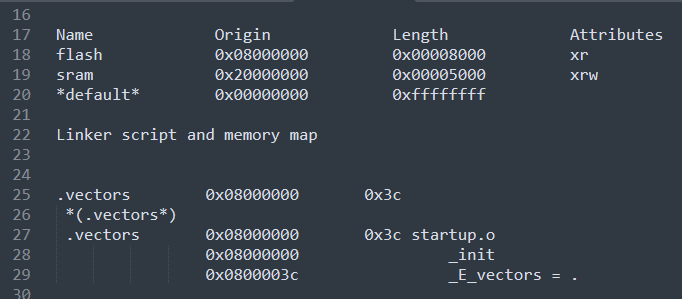
**7 . Appendices**

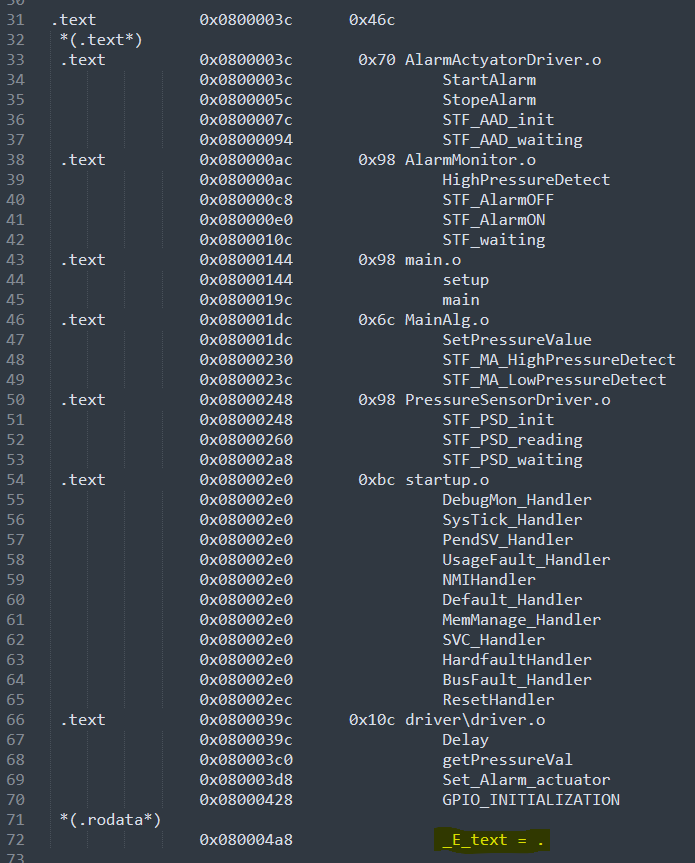
* **Diagrams:**

**Memory Map Description**



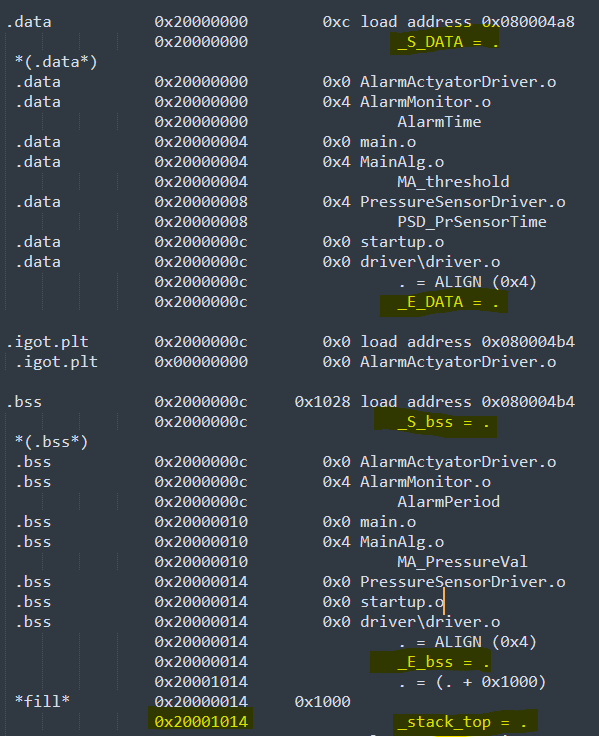




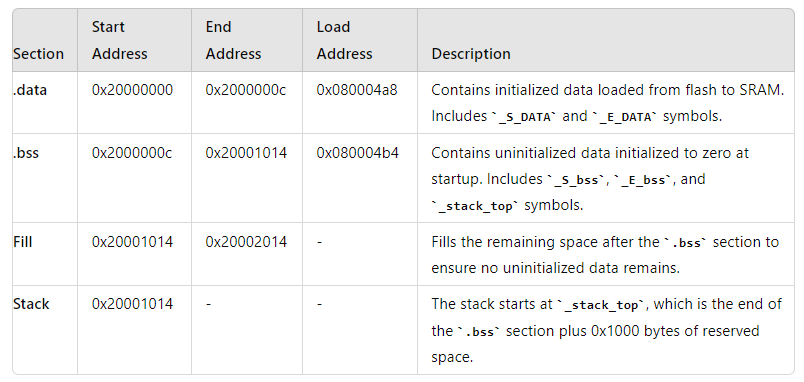


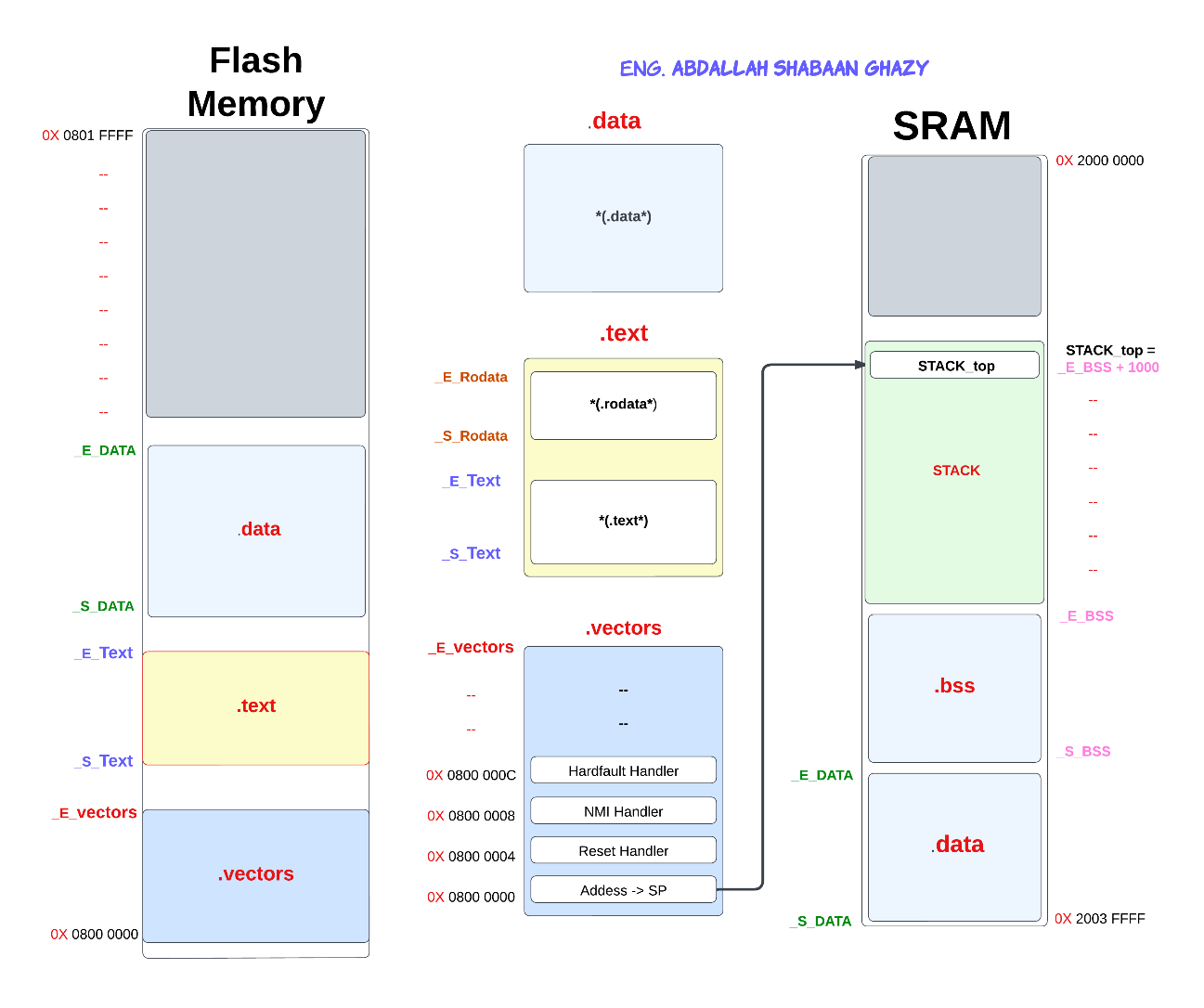
**Flash memory**

* **- - - -**
* **- - - -**



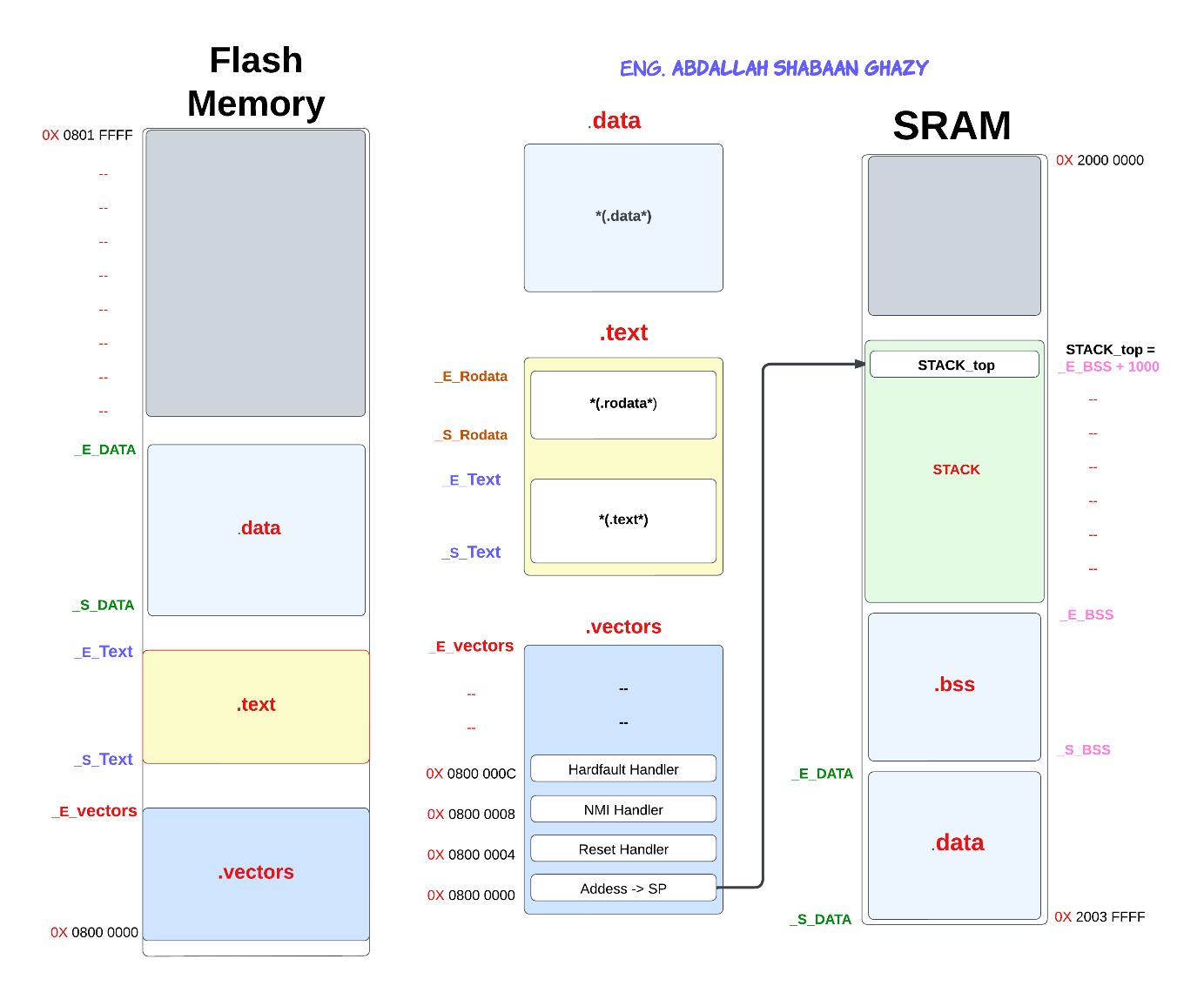
**SRAM**



**Flash Memory Layout**

**Description:** The Flash memory is divided into several sections that store different types of data. Below is a description of each section and its contents:

1. **Section .vectors**
   * **Address:** Starts at the beginning of Flash memory.
   * **Contents:**
     + **SP (Stack Pointer):** The initial stack pointer address.
     + **Reset Handler:** The address of the reset handler, which is the entry point of the program.
     + **Vector Table:** Contains pointers to various interrupt handlers.
2. **Section .text**
   * **Address:** Follows the .vectors section.
   * **Contents:**
     + **.text**: Contains the executable code of the application.
     + **.rodata**: Contains read-only data such as constants and static strings.
3. **Section .data**
   * **Address:** Located after the .text section.
   * **Contents:**
     + **.data**: Initialized global and static variables. This section is copied from Flash to SRAM during startup.

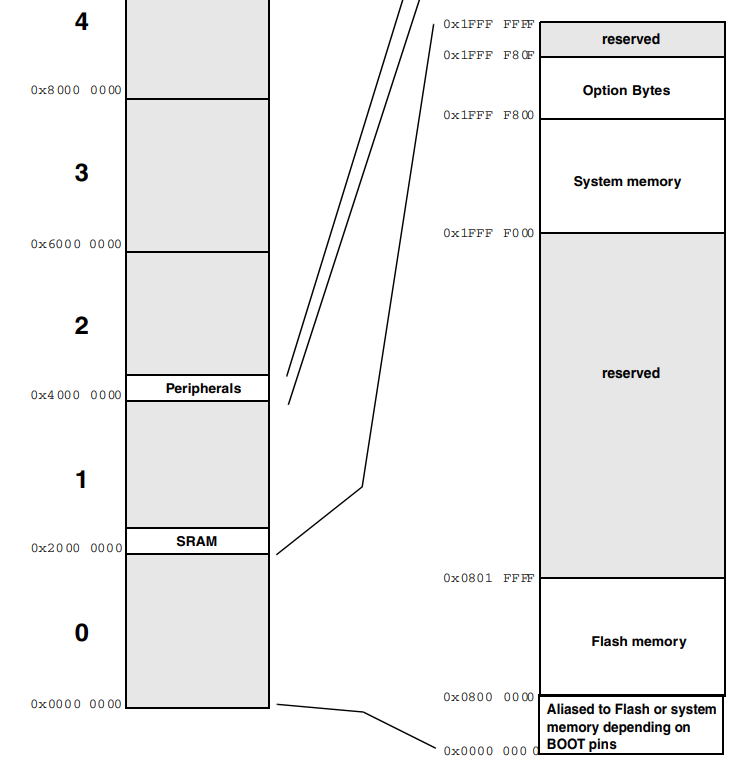


**SRAM Initialization and Layout**

**Description:** After the microcontroller starts, the following initializations occur in SRAM:

1. **Relocation of .data Section**
   * **From Address:** \_S\_DATA
   * **To Address:** \_E\_DATA
   * **Purpose:** Initializes .data section in SRAM with values from Flash.
2. **Initialization of .bss Section**
   * **From Address:** \_S\_BSS
   * **To Address:** \_E\_BSS
   * **Purpose:** Clears and initializes .bss section in SRAM to zero.
3. **Stack Initialization**
   * **Address:** STACK\_top
   * **Calculated as:** \_E\_BSS + 1000 (or another size based on your stack requirements)
   * **Purpose:** Sets up the stack pointer.

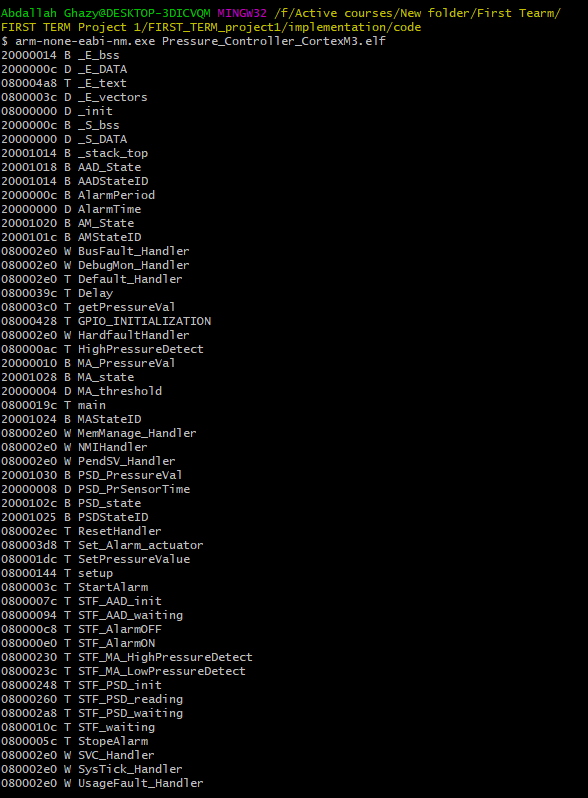
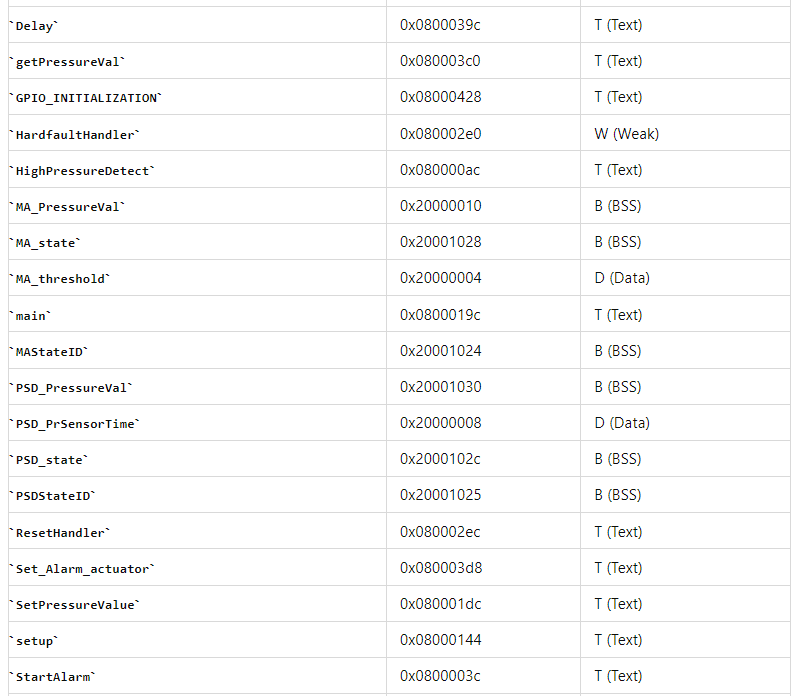
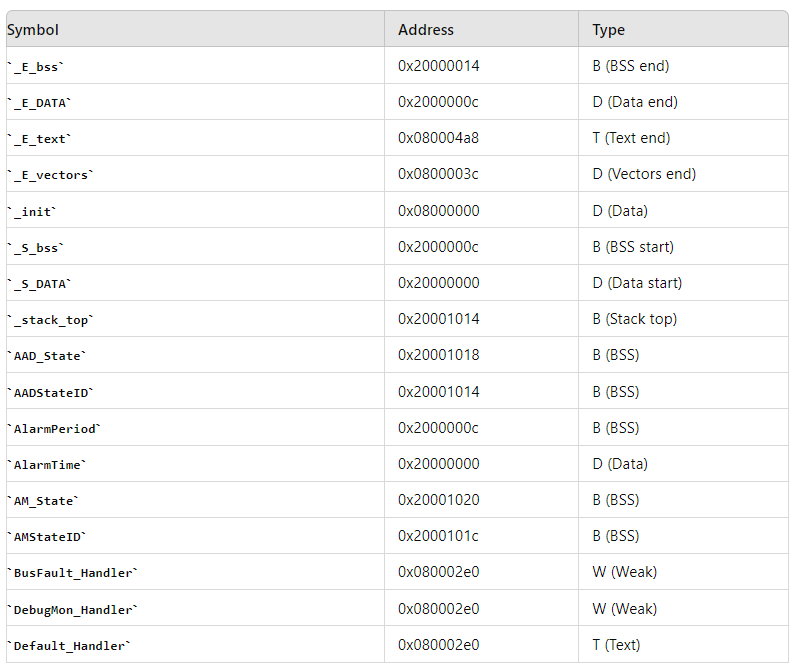
**Memory mapping**





**The Vector tables**

**Symbols Table**

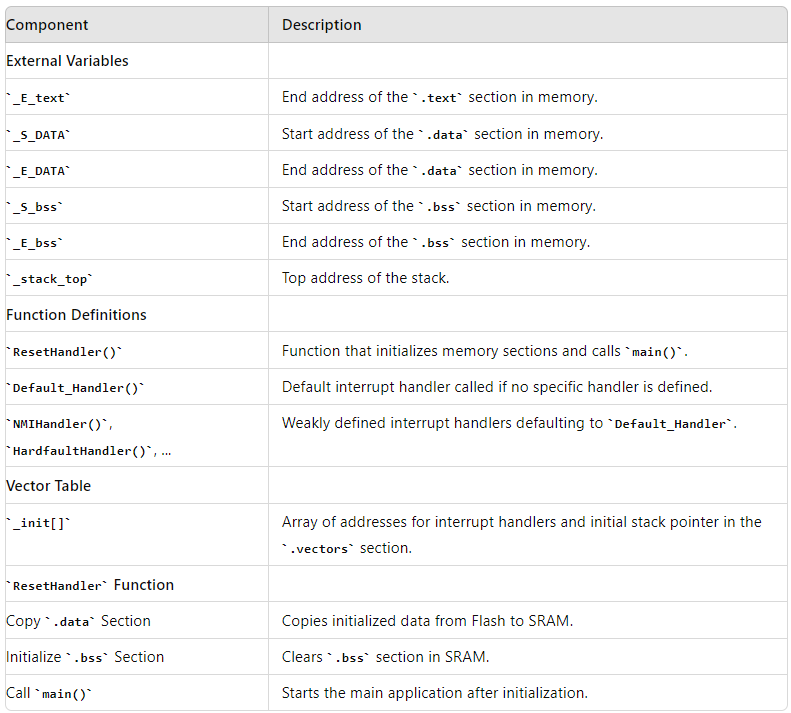
****

**Startup Code**

### **Code Description**

This code is a basic startup code for an embedded system. It includes setting up initial memory configuration, defining interrupt handlers, and copying data sections from Flash to SRAM. Below is a description of the code components and a table summarizing the details.

### **Table Summary**



**#include** <stdint.h>

**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;

**extern** **unsigned** **int** \_stack\_top;

**extern** **int** **main**();

**void** **ResetHandler**(**void**);

**void** **Default\_Handler**(**void**);

**void** **Default\_Handler**(**void**)

{

**ResetHandler**();

}

// Weak aliases for interrupt handlers

**void** **NMIHandler**(**void**) **\_\_attribute\_\_**((weak, alias("Default\_Handler")));

**void** **HardfaultHandler**(**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** **SVC\_Handler**(**void**) **\_\_attribute\_\_**((weak, alias("Default\_Handler")));

**void** **DebugMon\_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")));

uint32\_t \_init[] **\_\_attribute\_\_**((section(".vectors"))) = {

(uint32\_t) &\_stack\_top, // Initial stack pointer

(uint32\_t) &**ResetHandler**, // Reset Handler

(uint32\_t) &**NMIHandler**, // NMI Handler

(uint32\_t) &**HardfaultHandler**, // HardFault Handler

(uint32\_t) &**MemManage\_Handler**, // MemManage Handler

(uint32\_t) &**BusFault\_Handler**, // BusFault Handler

(uint32\_t) &**UsageFault\_Handler**, // UsageFault Handler

(uint32\_t) 0, // Reserved

(uint32\_t) 0, // Reserved

(uint32\_t) 0, // Reserved

(uint32\_t) 0, // Reserved

(uint32\_t) &**SVC\_Handler**, // SVC Handler

(uint32\_t) &**DebugMon\_Handler**, // DebugMon Handler

(uint32\_t) &**PendSV\_Handler**, // PendSV Handler

(uint32\_t) &**SysTick\_Handler** // SysTick Handler

};

**void** **ResetHandler**(**void**)

{

**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** (**unsigned** **int** i = 0; i < DATA\_size; i++) {

\*((uint8\_t\*)P\_dst++) = \*((uint8\_t\*)P\_src++);

}

**unsigned** **int** bss\_size = (**unsigned** **char**\*)&\_E\_bss - (**unsigned** **char**\*)&\_S\_bss;

P\_dst = (**unsigned** **char**\*)&\_S\_bss;

**for** (**unsigned** **int** i = 0; i < bss\_size; i++) {

\*((uint8\_t\*)P\_dst++) = (uint8\_t)0;

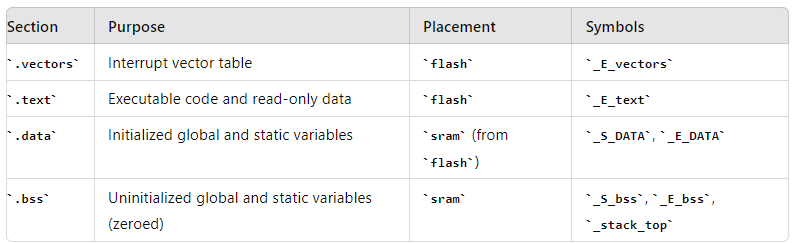
}

**main**();

}

### **Linker Script Description**

This linker script defines the memory layout and section placement for an embedded system. It specifies the memory regions, the sections within these regions, and the addresses for different sections and symbols. Below is a detailed description of each part of the script and a table summarizing the layout.



MEMORY

{

flash (rx) : ORIGIN = 0x08000000, LENGTH = 32K

sram (rwx) : ORIGIN = 0x20000000, LENGTH = 20K

}

SECTIONS

{

.vectors : {

\*(.vectors\*)

\_E\_vectors = .;

} >flash

.text : {

\*(.text\*)

\*(.rodata\*)

\_E\_text = .;

} >flash

.data :

{

\_S\_DATA = .;

\*(.data\*)

. = ALIGN(4);

\_E\_DATA = .;

} >sram AT> flash

.bss :

{

\_S\_bss = .;

\*(.bss\*)

. = ALIGN(4);

\_E\_bss = .;

. = . + 0x1000;

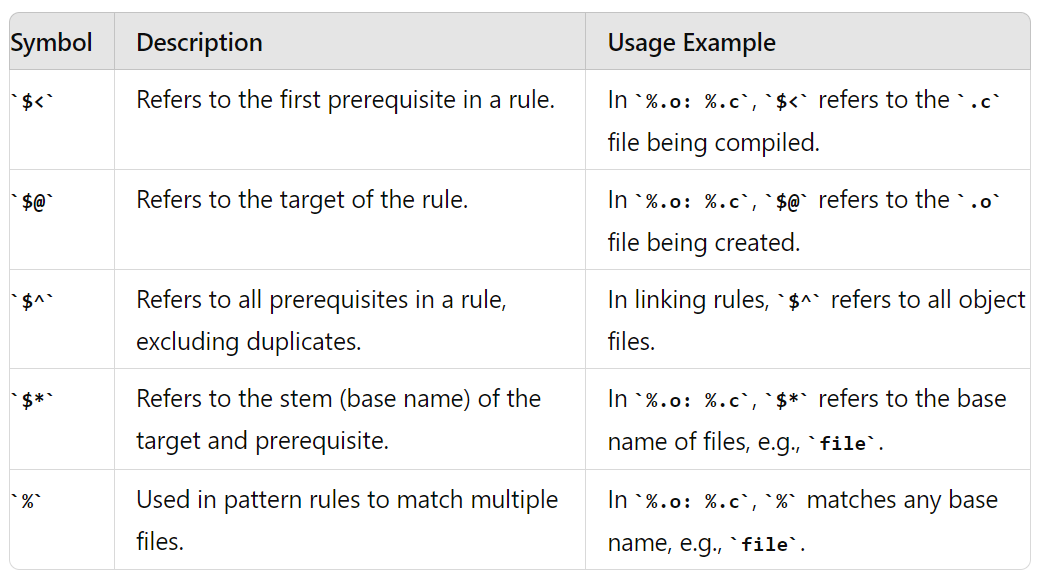
\_stack\_top = .;

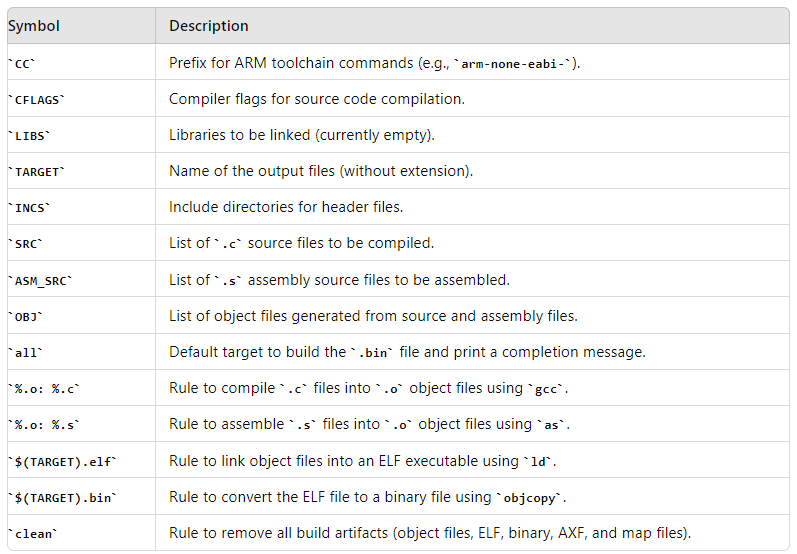
} >sram

}

**Makefile Explanation**

This Makefile is used to compile and link a project for an ARM Cortex-M3 microcontroller. It specifies the build process, including compiling source files, linking them, and generating output files.

**Symbols Summary Table**



# Toolchain variables

CC=arm-none-eabi-

CFLAGS= -mcpu=cortex-m3 -mthumb -std=c99 -gdwarf-2

LIBS=

TARGET=Pressure\_Controller\_CortexM3

INCS=-I .

# Source and object file lists

SRC = $(wildcard \*.c) driver\\driver.c

ASM\_SRC = $(wildcard \*.s)

OBJ = $(SRC:.c=.o) $(ASM\_SRC:.s=.o)

# Default target

all: $(TARGET).bin

@echo "Build is done"

# Rule to build object files from C source

%.o: %.c

$(CC)gcc -c $(CFLAGS) $(INCS) $< -o $@

# Rule to build object files from assembly source

%.o: %.s

$(CC)as $(CFLAGS) $(INCS) $< -o $@

# Rule to build the ELF file

$(TARGET).elf: $(OBJ)

$(CC)ld -T linker\_script.ld $(LIBS) $(OBJ) -o $@ -Map=Map\_file.map

cp $(TARGET).elf $(TARGET).axf

# Rule to build the binary file

$(TARGET).bin: $(TARGET).elf

$(CC)objcopy -O binary $< $@

# Clean rule to remove build artifacts

clean:

rm -f \*.o \*.elf \*.bin \*.axf \*.map

**Main.c**

**#include** <stdint.h>

**#include** <stdio.h>

**#include** "AlarmActyatorDriver.h"

**#include** "AlarmMonitor.h"

**#include** "MainAlg.h"

**#include** "PressureSensorDriver.h"

**#include** "driver/driver.h"

**#include** "state.h"

**extern** **void** **GPIO\_INITIALIZATION**();

**void** **setup**() {

**STATE\_INIT**(PSD\_init);

**STATE\_INIT**(AAD\_init);

PSD\_state = **STATE**(PSD\_waiting);

MA\_state = **STATE**(MA\_LowPressureDetect);

AM\_State = **STATE**(AlarmOFF);

AAD\_State = **STATE**(AAD\_init);

}

**int** **main**() {

**GPIO\_INITIALIZATION**();

**setup**();

**while** (1) {

**PSD\_state**();

**MA\_state**();

**AAD\_State**();

**AM\_State**();

}

}

**MainAlg.c**

**#include** "MainAlg.h"

**#include** "state.h"

**#include** <stdio.h>

**#include** "driver/driver.h"

**int** MA\_PressureVal = 0;

**int** MA\_threshold = 20;

**void** (\*MA\_state)();

**void** **SetPressureValue**(**int** p){

MA\_PressureVal = p;

MA\_state = (MA\_PressureVal <= MA\_threshold) ? **STATE**(MA\_LowPressureDetect) : **STATE**(MA\_HighPressureDetect);

// printf("-------------- Pressure Value = %d -------------------------- \n",MA\_PressureVal);

}

**STATE\_DEFINE**(MA\_HighPressureDetect){

// printf("The CPU in MA\_HighPressureDetect State ............\n");

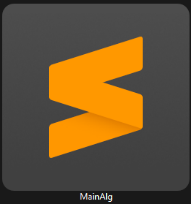
**HighPressureDetect**();

}

**STATE\_DEFINE**(MA\_LowPressureDetect) {

// printf("The CPU is in MA\_LowPressureDetect State ............\n");

}

**MainAlg.h**

**#ifndef** MA\_H\_

**#define** **MA\_H\_**

**#include** "state.h"

**enum** {

*MA\_HighPressureDetect*,

*MA\_LowPressureDetect*

} MAStateID;

**STATE\_DEFINE**(MA\_HighPressureDetect);

**STATE\_DEFINE**(MA\_LowPressureDetect);

**extern** **void** (\*MA\_state)();

**#endif**

**PressureSensorDriver.c**

**#include** "PressureSensorDriver.h"

**#include** "state.h"

**#include** <stdio.h>

**#include** "driver/driver.h"

**int** PSD\_PressureVal ;

**int** PSD\_PrSensorTime = 10000;

**void** (\*PSD\_state)();

// int PSD\_Get\_Pressure\_random(int l, int r, int count) {

// return (rand() % (r - l + 1)) + l;

// }

**STATE\_DEFINE**(PSD\_init){

PSDStateID = PSD\_init;

// printf("The Sensor in init State ............\n");

}

**STATE\_DEFINE**(PSD\_reading){

PSDStateID = PSD\_reading;

// printf("The Sensor in reading State ............\n");

// PSD\_PressureVal = PSD\_Get\_Pressure\_random(15, 30, 1);

// printf("PSD\_Reading : PressureVal = %d\n", PSD\_PressureVal);

PSD\_PressureVal = **getPressureVal**();

**SetPressureValue**(PSD\_PressureVal);

PSD\_state = **STATE**(PSD\_waiting);

}

**STATE\_DEFINE**(PSD\_waiting){

PSDStateID = PSD\_waiting;

// delay

**Delay**(PSD\_PrSensorTime);

PSD\_state = **STATE**(PSD\_reading);

}

**PressureSensorDriver.h**

**#ifndef** PSD\_H\_

**#define** **PSD\_H\_**

**#include** "state.h"

**enum** {

*PSD\_init*,

*PSD\_reading*,

*PSD\_waiting*

} PSDStateID;

**STATE\_DEFINE**(PSD\_init);

**STATE\_DEFINE**(PSD\_reading);

**STATE\_DEFINE**(PSD\_waiting);

**extern** **void** (\*PSD\_state)();

**#endif**

**AlarmActyatorDriver.c**

**#include** <stdio.h>

**#include** "state.h"

**#include** "AlarmActyatorDriver.h"

**void** (\*AAD\_State)();

**void** **StartAlarm**(**void**) {

// printf("The Alarm in Start State ............\n");

**Set\_Alarm\_actuator**(0);

AAD\_State = **STATE**(AAD\_waiting);

}

**void** **StopeAlarm**(**void**) {

//printf("The Alarm in Stope State ............\n");

**Set\_Alarm\_actuator**(1);

AAD\_State = **STATE**(AAD\_waiting);

}

**STATE\_DEFINE**(AAD\_init) {

AADStateID = AAD\_init;

// printf("The Sensor in init State ............\n");

}

**STATE\_DEFINE**(AAD\_waiting) {

AADStateID = AAD\_waiting;

// printf("The Alarm in waiting State ............\n");

}

**AlarmActyatorDriver.h**

**#ifndef** AAD\_H\_

**#define** **AAD\_H\_**

**#include** "state.h"

**enum** {

*AAD\_init*,

*AAD\_waiting*,

*AAD\_AlarmON*,

*AAD\_AlarmOFF*

} AADStateID;

**STATE\_DEFINE**(AAD\_init);

**STATE\_DEFINE**(AAD\_waiting);

**extern** **void** (\*AAD\_State)();

**#endif**

**AlarmMonitor.c**

**#include** <stdio.h>

**#include** "state.h"

**#include** "AlarmMonitor.h"

**int** AlarmTime = 10000;

**int** AlarmPeriod = 0;

**void** (\*AM\_State)();

**void** **HighPressureDetect**(**void**) {

// printf("The CPU in High Pressure Detect ............\n");

AM\_State = **STATE**(AlarmON);

}

**STATE\_DEFINE**(AlarmOFF) {

AMStateID = AlarmOFF;

// printf("The CPU in Alarm OFF State ............\n");

**StopeAlarm**();

}

**STATE\_DEFINE**(AlarmON) {

AMStateID = AlarmON;

// printf("The CPU in Alarm ON State ............\n");

**StartAlarm**();

AM\_State = **STATE**(waiting);

}

**STATE\_DEFINE**(waiting) {

AMStateID = waiting;

// printf("The CPU in waiting State ............\n");

// delay

**Delay**(AlarmTime);

AM\_State = **STATE**(AlarmOFF);

}

**AlarmMonitor.h**

**#ifndef** AM\_H\_

**#define** **AM\_H\_**

**#include** "state.h"

**enum** {

*AlarmOFF*,

*AlarmON*,

*waiting*

} AMStateID;

**STATE\_DEFINE**(AlarmOFF);

**STATE\_DEFINE**(AlarmON);

**STATE\_DEFINE**(waiting);

**extern** **void** (\*AM\_State)();

**#endif**