

Learning Report – Embedded C



LTTTS
GLOBAL
ENGINEERING
ACADEMY



L&T Technology Services



Document History

Ver. Rel. No.	Release Date	Prepared. By	Reviewed By	Approved By	Remarks/Revision Details
	18-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	
	19-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	
	20-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	
	21-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	
	22-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	
	23-02-2021	Saloni Adanna		Dr Vivek Kaundal & Bhargav N	

TABLE OF TABLES

ACTIVITY 1

CREATE A MAKE FILE

CREATE A STARTUP FILE

CREATE A LINKER SCRIPT

DEBUGGING TECHNIQUES

DRIVER DEVELOPMENT

ACTIVITY 2

GPIO

SPI

UART

PWM

ADC

Activity 3

Mini Project

Activity 1

1.1 Introduction

A sample c application program is taken and its cross compilation is done. Cross compilation is a process where compiling is staged and each stage produces a file which can be viewed. The cross-compilation process involves in converting a main code into .asm, .s and .elf files. These are the files produced at each stage of cross compilation. The below image shows the steps involved in cross compilation.

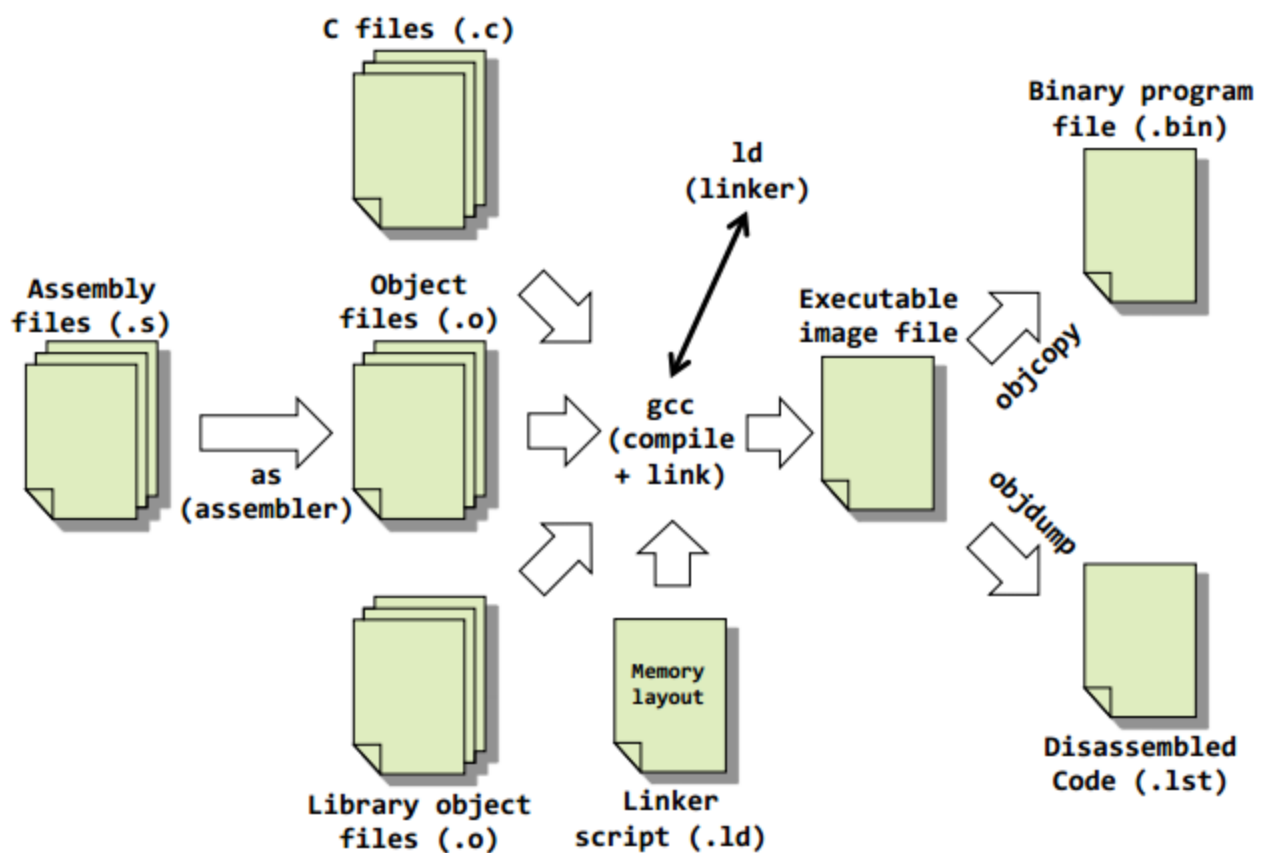
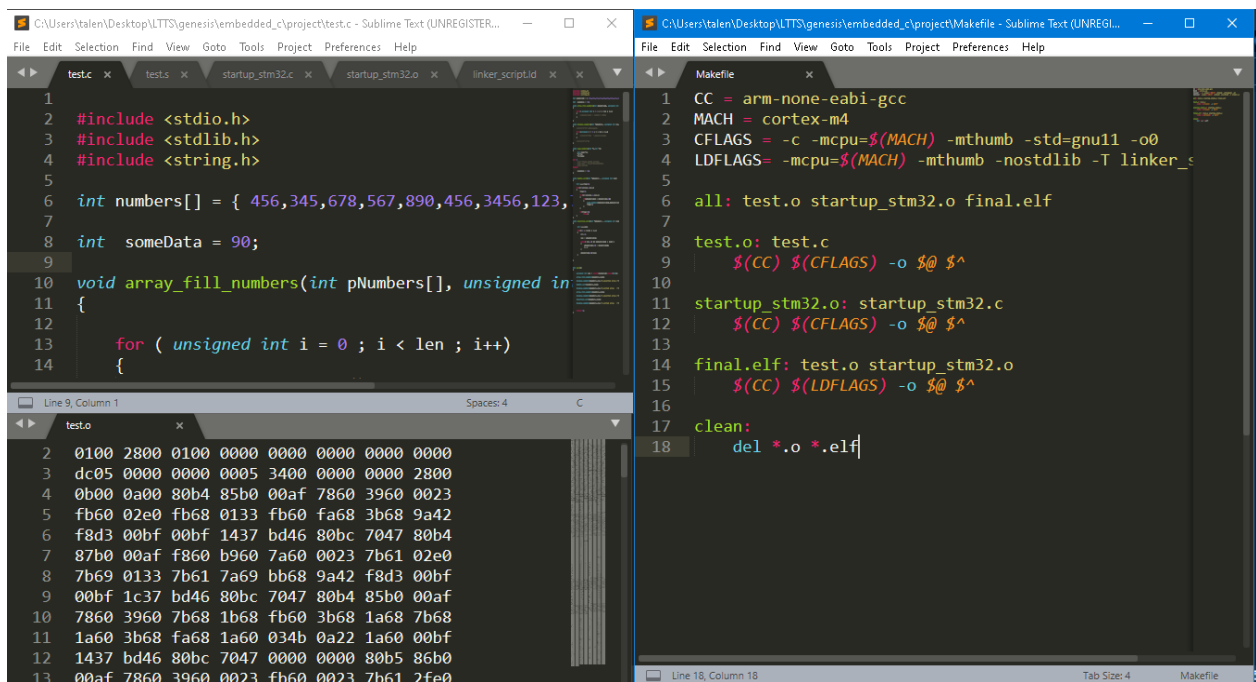


Figure 1:

1. Creating Makefile

Makefiles contains basically the shell comments. Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command every time you need to compile. Makefiles are the solution to simplify this task. Makefiles are special format files that help build and manage the projects automatically. Its acts as an automation tool.



The screenshot shows two Sublime Text windows. The left window displays a C program named 'test.c' with the following code:

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int numbers[] = { 456,345,678,567,890,456,3456,123,
7
8 int someData = 90;
9
10 void array_fill_numbers(int pNumbers[], unsigned in
11 {
12
13     for ( unsigned int i = 0 ; i < len ; i++)
14     {

```

The right window displays a Makefile with the following code:

```

1 CC = arm-none-eabi-gcc
2 MACH = cortex-m4
3 CFLAGS = -c -mcpu=$(MACH) -mthumb -std=gnu11 -o0
4 LDFLAGS= -mcpu=$(MACH) -mthumb -nostdlib -T linker_s
5
6 all: test.o startup_stm32.o final.elf
7
8 test.o: test.c
9     $(CC) $(CFLAGS) -o $@ $^
10
11 startup_stm32.o: startup_stm32.c
12     $(CC) $(CFLAGS) -o $@ $^
13
14 final.elf: test.o startup_stm32.o
15     $(CC) $(LDFLAGS) -o $@ $^
16
17 clean:
18     del *.o *.elf

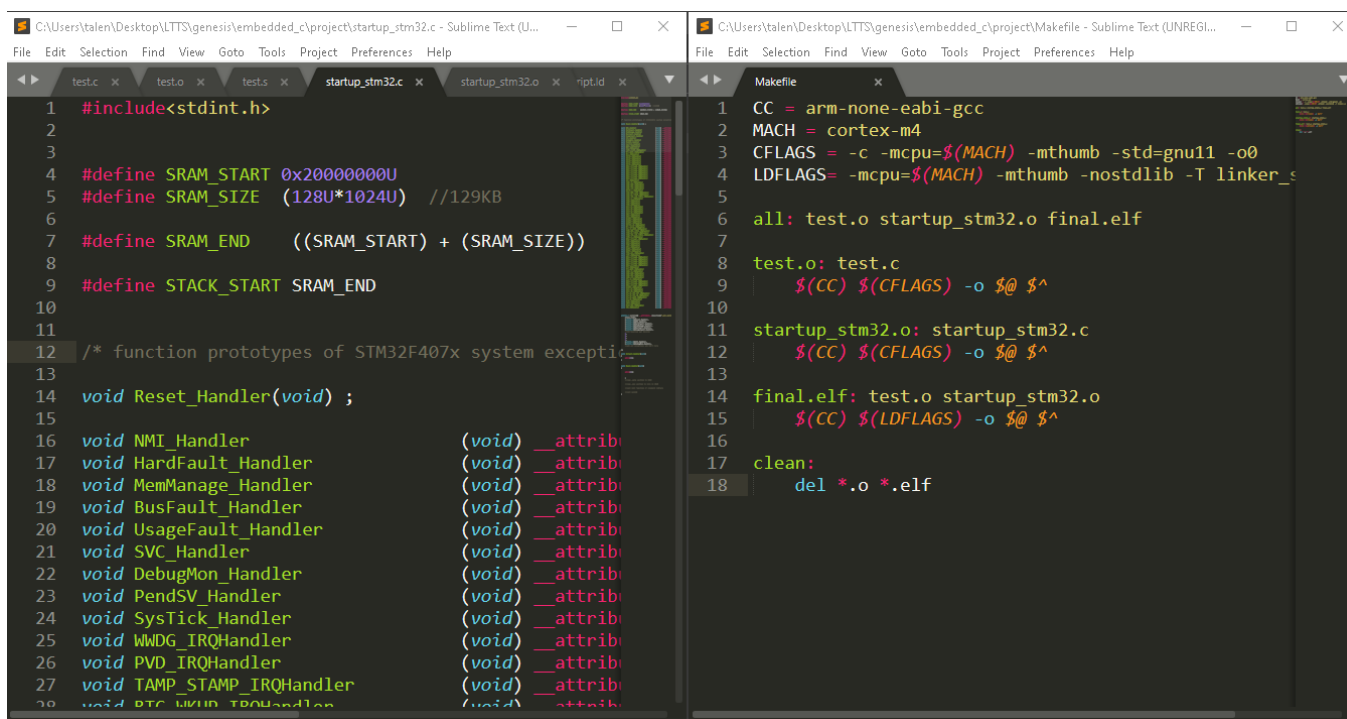
```

Figure 2:

The above figure 2 shows the sample c program along with make file which has generated an object file of the given program which is an intermediate compilation stage.

2. Startup File

Startup files are usually in your home directory. Their names begin with a period, which keeps the ls command from displaying them under normal circumstances. None of the files are required; all the affected programs are smart enough to use defaults when the file does not exist. Figure 3 shows the start-up file for the given application. It tells the computer to where to start the program from and also the address location. Figure 4 shows the object file created from the start-up file.

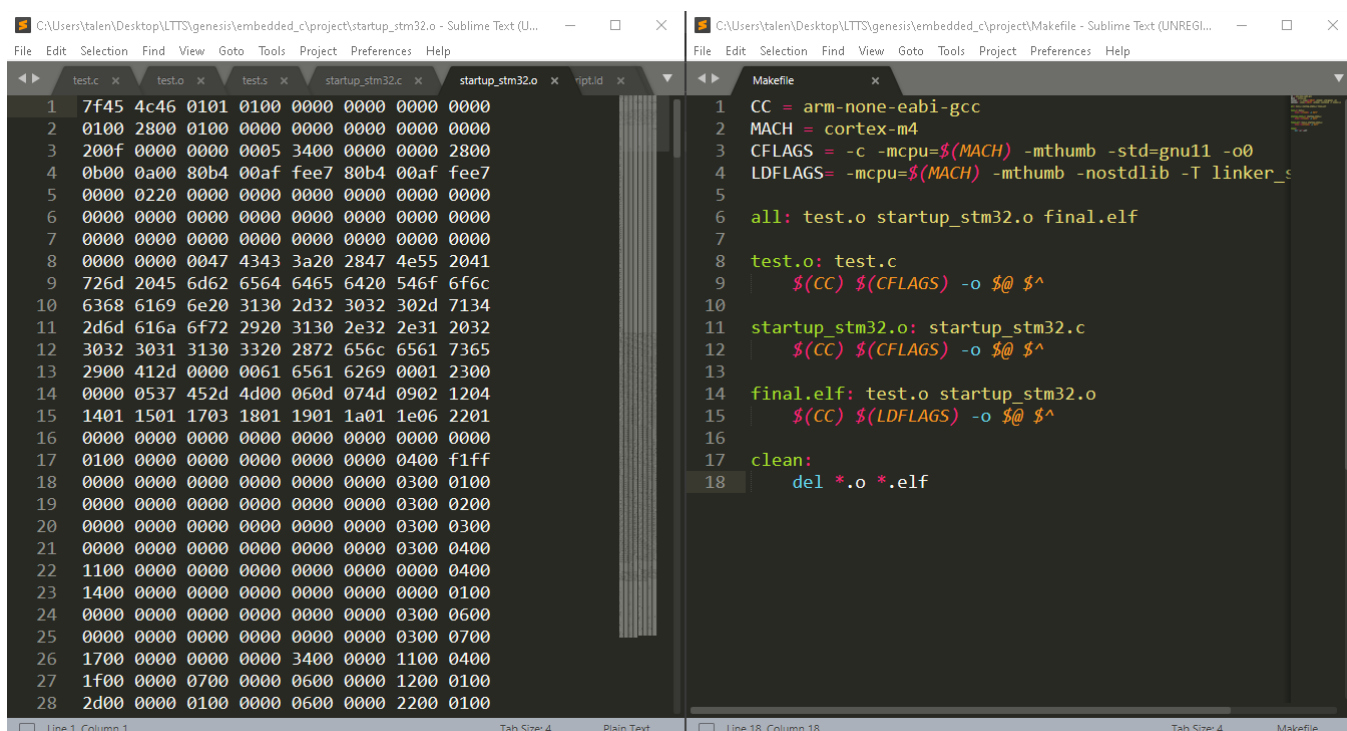


The screenshot shows two Sublime Text windows. The left window, titled 'startup_stm32.c', contains C code for a STM32F407x system. It includes a header for `<stdint.h>`, defines memory addresses for SRAM and stack, and lists function prototypes for various handlers. The right window, titled 'Makefile', contains a Makefile for the project. It sets the compiler to `arm-none-eabi-gcc`, the architecture to `cortex-m4`, and defines flags for C and linker. The targets include `test.o`, `startup_stm32.o`, and `final.elf`.

```
1 #include<stdint.h>
2
3
4 #define SRAM_START 0x20000000U
5 #define SRAM_SIZE (128U*1024U) //129KB
6
7 #define SRAM_END ((SRAM_START) + (SRAM_SIZE))
8
9 #define STACK_START SRAM_END
10
11
12 /* function prototypes of STM32F407x system exceptions
13
14 void Reset_Handler(void) ;
15
16 void NMI_Handler (void) __attribute__((weak));
17 void HardFault_Handler (void) __attribute__((weak));
18 void MemManage_Handler (void) __attribute__((weak));
19 void BusFault_Handler (void) __attribute__((weak));
20 void UsageFault_Handler (void) __attribute__((weak));
21 void SVC_Handler (void) __attribute__((weak));
22 void DebugMon_Handler (void) __attribute__((weak));
23 void PendSV_Handler (void) __attribute__((weak));
24 void SysTick_Handler (void) __attribute__((weak));
25 void WWDG_IRQHandler (void) __attribute__((weak));
26 void PVD_IRQHandler (void) __attribute__((weak));
27 void TAMP_STAMP_IRQHandler (void) __attribute__((weak));
28 void RTC_WKUP_IRQHandler (void) __attribute__((weak));
```

```
1 CC = arm-none-eabi-gcc
2 MACH = cortex-m4
3 CFLAGS = -c -mcpu=$(MACH) -mthumb -std=gnu11 -o0
4 LDFLAGS= -mcpu=$(MACH) -mthumb -nostdlib -T linker_
5
6 all: test.o startup_stm32.o final.elf
7
8 test.o: test.c
9     $(CC) $(CFLAGS) -o $@ $^
10
11 startup_stm32.o: startup_stm32.c
12     $(CC) $(CFLAGS) -o $@ $^
13
14 final.elf: test.o startup_stm32.o
15     $(CC) $(LDFLAGS) -o $@ $^
16
17 clean:
18     del *.o *.elf
```

Figure 3:



The screenshot shows two Sublime Text windows. The left window, titled 'startup_stm32.o', displays a memory dump of the object file. The dump shows a series of hexadecimal values arranged in columns, representing the raw data of the object file. The right window, titled 'Makefile', contains the same Makefile as in Figure 3.

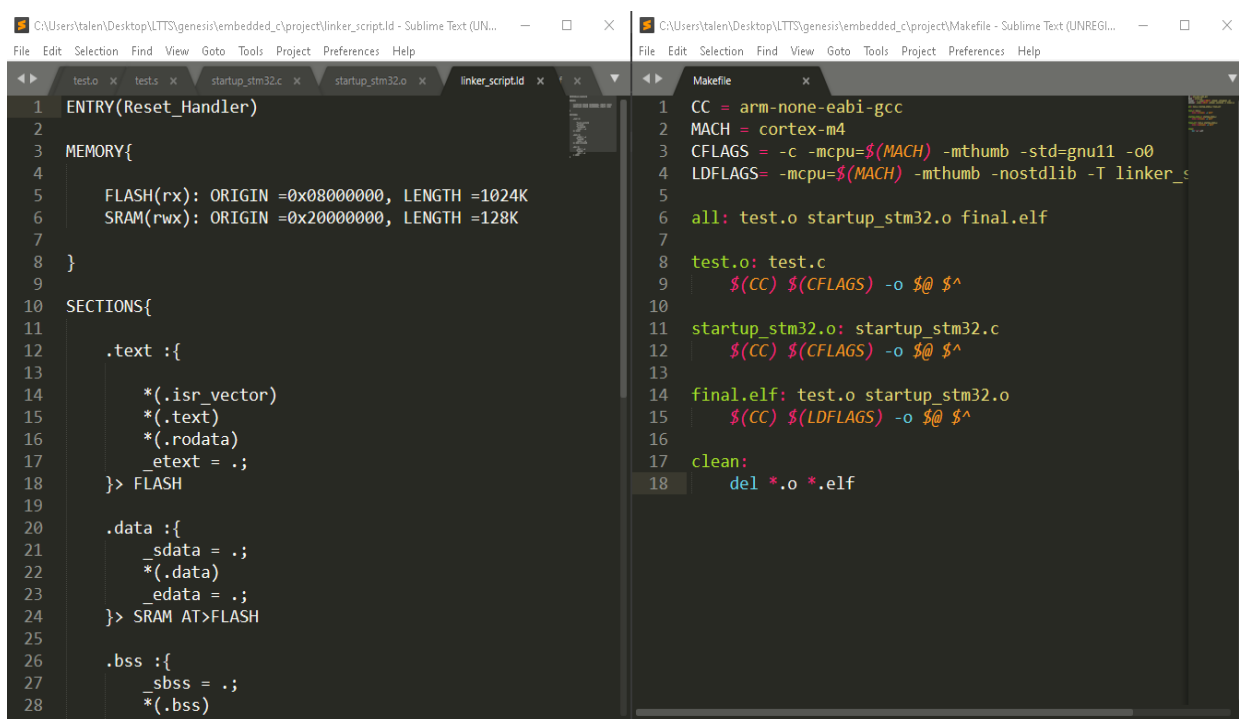
```
1 7f45 4c46 0101 0100 0000 0000 0000 0000
2 0100 2800 0100 0000 0000 0000 0000 0000
3 200f 0000 0000 0005 3400 0000 0000 2800
4 0b00 0a00 80b4 00af fee7 80b4 00af fee7
5 0000 0220 0000 0000 0000 0000 0000 0000
6 0000 0000 0000 0000 0000 0000 0000 0000
7 0000 0000 0000 0000 0000 0000 0000 0000
8 0000 0000 0047 4343 3a20 2847 4e55 2041
9 726d 2045 6d62 6564 6465 6420 546f 6f6c
10 6368 6169 6e20 3130 2d32 3032 302d 7134
11 2d6d 616a 6f72 2920 3130 2e32 2e31 2032
12 3032 3031 3130 3320 2872 656c 6561 7365
13 2900 412d 0000 0061 6561 6269 0001 2300
14 0000 0537 452d 4d00 060d 074d 0902 1204
15 1401 1501 1703 1801 1901 1a01 1e06 2201
16 0000 0000 0000 0000 0000 0000 0000 0000
17 0100 0000 0000 0000 0000 0000 0400 f1ff
18 0000 0000 0000 0000 0000 0000 0300 0100
19 0000 0000 0000 0000 0000 0000 0300 0200
20 0000 0000 0000 0000 0000 0000 0300 0300
21 0000 0000 0000 0000 0000 0000 0300 0400
22 1100 0000 0000 0000 0000 0000 0000 0400
23 1400 0000 0000 0000 0000 0000 0000 0100
24 0000 0000 0000 0000 0000 0000 0300 0600
25 0000 0000 0000 0000 0000 0000 0300 0700
26 1700 0000 0000 0000 3400 0000 1100 0400
27 1f00 0000 0700 0000 0600 0000 1200 0100
28 2d00 0000 0100 0000 0600 0000 2200 0100
```

```
1 CC = arm-none-eabi-gcc
2 MACH = cortex-m4
3 CFLAGS = -c -mcpu=$(MACH) -mthumb -std=gnu11 -o0
4 LDFLAGS= -mcpu=$(MACH) -mthumb -nostdlib -T linker_
5
6 all: test.o startup_stm32.o final.elf
7
8 test.o: test.c
9     $(CC) $(CFLAGS) -o $@ $^
10
11 startup_stm32.o: startup_stm32.c
12     $(CC) $(CFLAGS) -o $@ $^
13
14 final.elf: test.o startup_stm32.o
15     $(CC) $(LDFLAGS) -o $@ $^
16
17 clean:
18     del *.o *.elf
```

Figure 4:

3. Linker Scripts

Every link is controlled by a linker script. This script is written in the linker command language. The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. Most linker scripts do nothing more than this. However, when necessary, the linker script can also direct the linker to perform many other operations, using the commands described below. The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled into the linker executable. You can use the ‘--verbose’ command-line option to display the default linker script. Certain command-line options, such as ‘-r’ or ‘-N’, will affect the default linker script. You may supply your own linker script by using the ‘-T’ command line option. When you do this, your linker script will replace the default linker script.

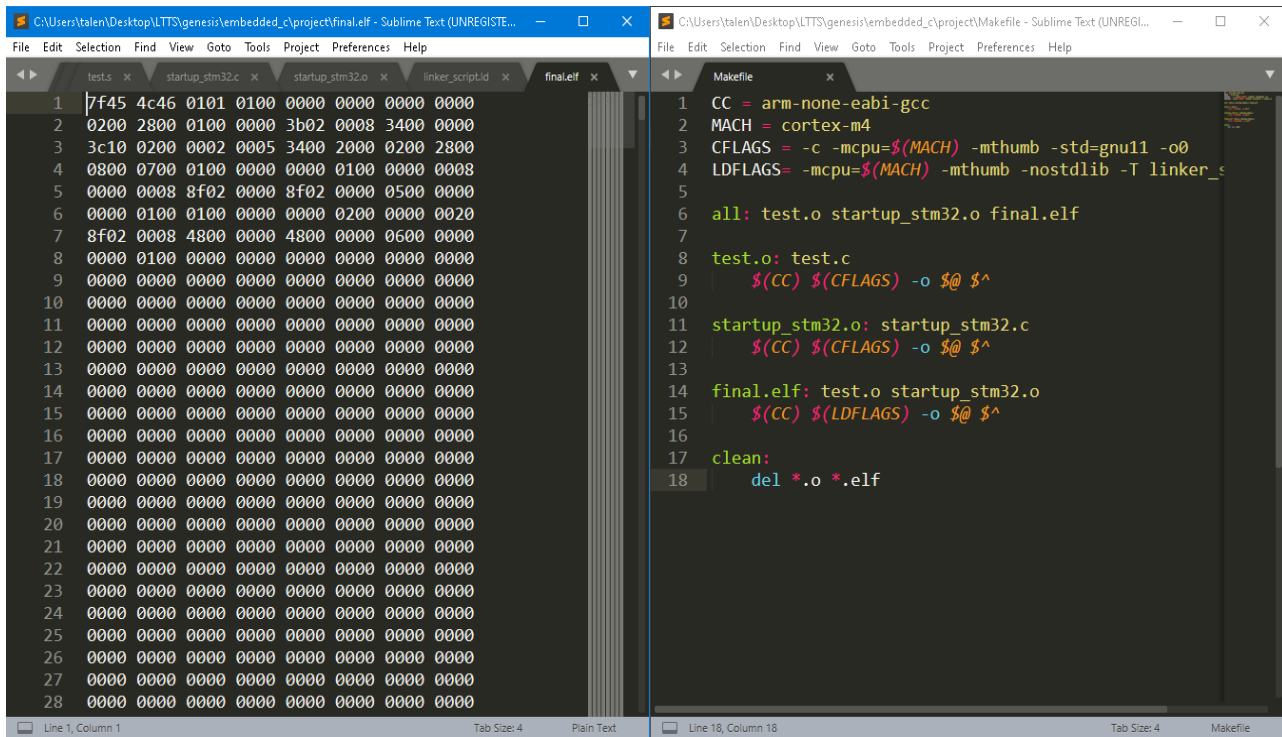


The image shows two side-by-side Sublime Text editor windows. The left window, titled 'linker_script.ld', contains a linker script for an ARM Cortex-M4. It defines memory regions for FLASH (0x08000000, 1024K) and SRAM (0x20000000, 128K). It also defines sections for .text, .data, and .bss, mapping them to the appropriate memory regions. The right window, titled 'Makefile', contains a Makefile for the project. It defines the compiler (arm-none-eabi-gcc), the target architecture (cortex-m4), and the linker flags (-mthumb, -std=gnu11, -o0, -mthumb, -nostdlib, -T linker_script.ld). It also defines the target (all) and the rules for building the test.o, startup_stm32.o, and final.elf files.

```
1 ENTRY(Reset_Handler)
2
3 MEMORY{
4     FLASH(rx): ORIGIN =0x08000000, LENGTH =1024K
5     SRAM(rwx): ORIGIN =0x20000000, LENGTH =128K
6 }
7
8 SECTIONS{
9
10     .text :{
11         *(.isr_vector)
12         *(.text)
13         *(.rodata)
14         _etext = .;
15     }> FLASH
16
17     .data :{
18         _sdata = .;
19         *(.data)
20         _edata = .;
21     }> SRAM AT>FLASH
22
23     .bss :{
24         _sbss = .;
25         *(.bss)
26     }
```

```
1 CC = arm-none-eabi-gcc
2 MACH = cortex-m4
3 CFLAGS = -c -mcpu=$(MACH) -mthumb -std=gnu11 -o0
4 LDFLAGS = -mcpu=$(MACH) -mthumb -nostdlib -T linker_script.ld
5
6 all: test.o startup_stm32.o final.elf
7
8 test.o: test.c
9     $(CC) $(CFLAGS) -o $@ $^
10
11 startup_stm32.o: startup_stm32.c
12     $(CC) $(CFLAGS) -o $@ $^
13
14 final.elf: test.o startup_stm32.o
15     $(CC) $(LDFLAGS) -o $@ $^
16
17 clean:
18     del *.o *.elf
```

Figure 5:



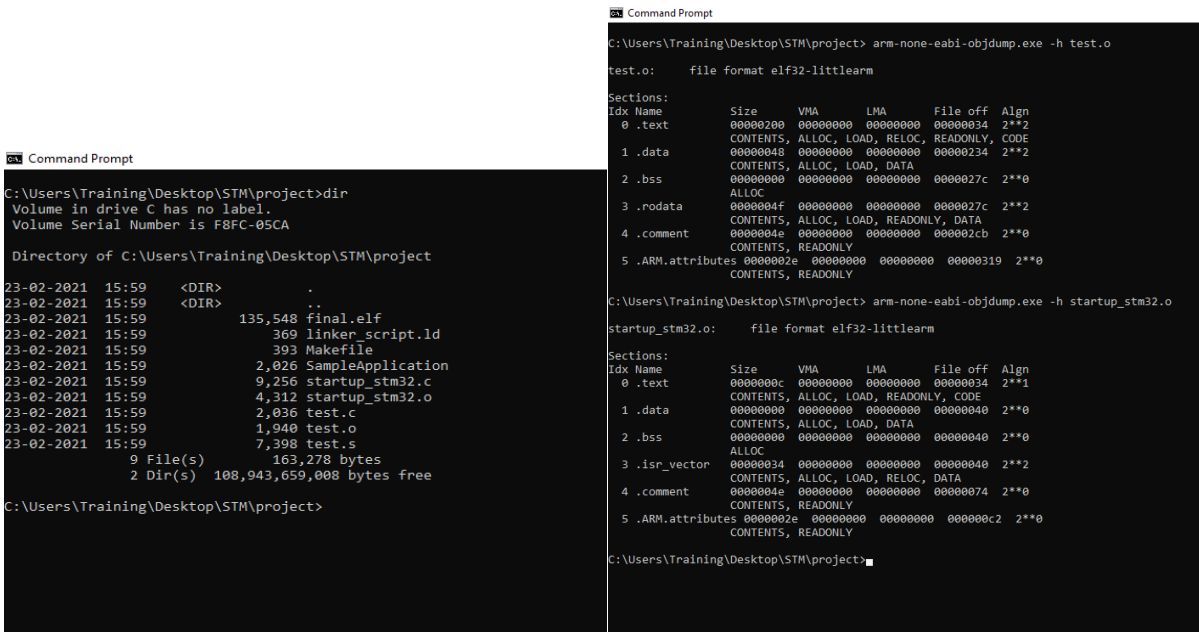
```

C:\Users\talen\Desktop\LTTS\genesis\embedded_c\project\final.elf - Sublime Text (UNREGISTE...
File Edit Selection Find View Goto Tools Project Preferences Help
1 | 7f45 4c46 0101 0100 0000 0000 0000 0000
2 | 0200 2800 0100 0000 3b02 0008 3400 0000
3 | 3c10 0200 0002 0005 3400 2000 0200 2800
4 | 0800 0700 0100 0000 0000 0100 0000 0008
5 | 0000 0008 8f02 0000 8f02 0000 0500 0000
6 | 0000 0100 0100 0000 0000 0200 0000 0020
7 | 8f02 0008 4800 0000 4800 0000 0600 0000
8 | 0000 0100 0000 0000 0000 0000 0000 0000
9 | 0000 0000 0000 0000 0000 0000 0000 0000
10 | 0000 0000 0000 0000 0000 0000 0000 0000
11 | 0000 0000 0000 0000 0000 0000 0000 0000
12 | 0000 0000 0000 0000 0000 0000 0000 0000
13 | 0000 0000 0000 0000 0000 0000 0000 0000
14 | 0000 0000 0000 0000 0000 0000 0000 0000
15 | 0000 0000 0000 0000 0000 0000 0000 0000
16 | 0000 0000 0000 0000 0000 0000 0000 0000
17 | 0000 0000 0000 0000 0000 0000 0000 0000
18 | 0000 0000 0000 0000 0000 0000 0000 0000
19 | 0000 0000 0000 0000 0000 0000 0000 0000
20 | 0000 0000 0000 0000 0000 0000 0000 0000
21 | 0000 0000 0000 0000 0000 0000 0000 0000
22 | 0000 0000 0000 0000 0000 0000 0000 0000
23 | 0000 0000 0000 0000 0000 0000 0000 0000
24 | 0000 0000 0000 0000 0000 0000 0000 0000
25 | 0000 0000 0000 0000 0000 0000 0000 0000
26 | 0000 0000 0000 0000 0000 0000 0000 0000
27 | 0000 0000 0000 0000 0000 0000 0000 0000
28 | 0000 0000 0000 0000 0000 0000 0000 0000
Line 1, Column 1 Tab Size: 4 Plain Text

C:\Users\talen\Desktop\LTTS\genesis\embedded_c\project\Makefile - Sublime Text (UNREGI...
File Edit Selection Find View Goto Tools Project Preferences Help
1 | CC = arm-none-eabi-gcc
2 | MACH = cortex-m4
3 | CFLAGS = -c -mcpu=$(MACH) -mthumb -std=gnu11 -o0
4 | LDFLAGS = -mcpu=$(MACH) -mthumb -nostdlib -T linker_s
5 |
6 | all: test.o startup_stm32.o final.elf
7 |
8 | test.o: test.c
9 |     $(CC) $(CFLAGS) -o $@ $^
10 |
11 | startup_stm32.o: startup_stm32.c
12 |     $(CC) $(CFLAGS) -o $@ $^
13 |
14 | final.elf: test.o startup_stm32.o
15 |     $(CC) $(LDFLAGS) -o $@ $^
16 |
17 | clean:
18 |     del *.o *.elf
Line 18, Column 18 Tab Size: 4 Makefile

```

Figure 6



```

C:\Users\Training\Desktop\STM\project> dir
Volume in drive C has no label.
Volume Serial Number is F8FC-05CA

Directory of C:\Users\Training\Desktop\STM\project

23-02-2021  15:59          <DIR>          .
23-02-2021  15:59          <DIR>          ..
23-02-2021  15:59             135,548 final.elf
23-02-2021  15:59             360 linker_script.ld
23-02-2021  15:59             393 Makefile
23-02-2021  15:59             2,026 SampleApplication
23-02-2021  15:59             9,256 startup_stm32.c
23-02-2021  15:59             4,312 startup_stm32.o
23-02-2021  15:59             2,036 test.c
23-02-2021  15:59             1,940 test.o
23-02-2021  15:59             7,398 test.s
                9 File(s)          163,278 bytes
                2 Dir(s)  108,943,659,008 bytes free

C:\Users\Training\Desktop\STM\project>

C:\Users\Training\Desktop\STM\project> arm-none-eabi-objdump.exe -h test.o
test.o:          file format elf32-littlearm

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
0 .text          00000200  00000000  00000000  00000034  2**2
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data          00000048  00000000  00000000  00000234  2**2
CONTENTS, ALLOC, LOAD, DATA
2 .bss           00000000  00000000  00000000  0000027c  2**0
ALLOC
3 .rodata        0000004f  00000000  00000000  0000027c  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .comment       0000004e  00000000  00000000  000002cb  2**0
CONTENTS, READONLY
5 .ARM.attributes 0000002e  00000000  00000000  00000319  2**0
CONTENTS, READONLY

C:\Users\Training\Desktop\STM\project> arm-none-eabi-objdump.exe -h startup_stm32.o
startup_stm32.o:          file format elf32-littlearm

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
0 .text          0000000c  00000000  00000000  00000034  2**1
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
1 .data          00000000  00000000  00000000  00000040  2**0
CONTENTS, ALLOC, LOAD, DATA
2 .bss           00000000  00000000  00000000  00000040  2**0
ALLOC
3 .isr_vector    00000034  00000000  00000000  00000040  2**2
CONTENTS, ALLOC, LOAD, RELOC, DATA
4 .comment       0000004e  00000000  00000000  00000074  2**0
CONTENTS, READONLY
5 .ARM.attributes 0000002e  00000000  00000000  000000c2  2**0
CONTENTS, READONLY

C:\Users\Training\Desktop\STM\project>

```

Figure 7

Activity 2

1. GPIO

GPIO'S also known as general purpose input output are the ports that are available on any micro-controller. These vary from one micro controller to other. GPIO's basically helps in controlling the input and output. But sometimes these ports are used for special functioning too. Depending on the size of micro controller they vary from 8 bit to 32,64 bits. In the below project, stm32f407, a 32-bit micro controller is used to toggle a LED at the output of one of the GPIO ports. This has been done using the stm IDE where the headers and main code is generated automatically.

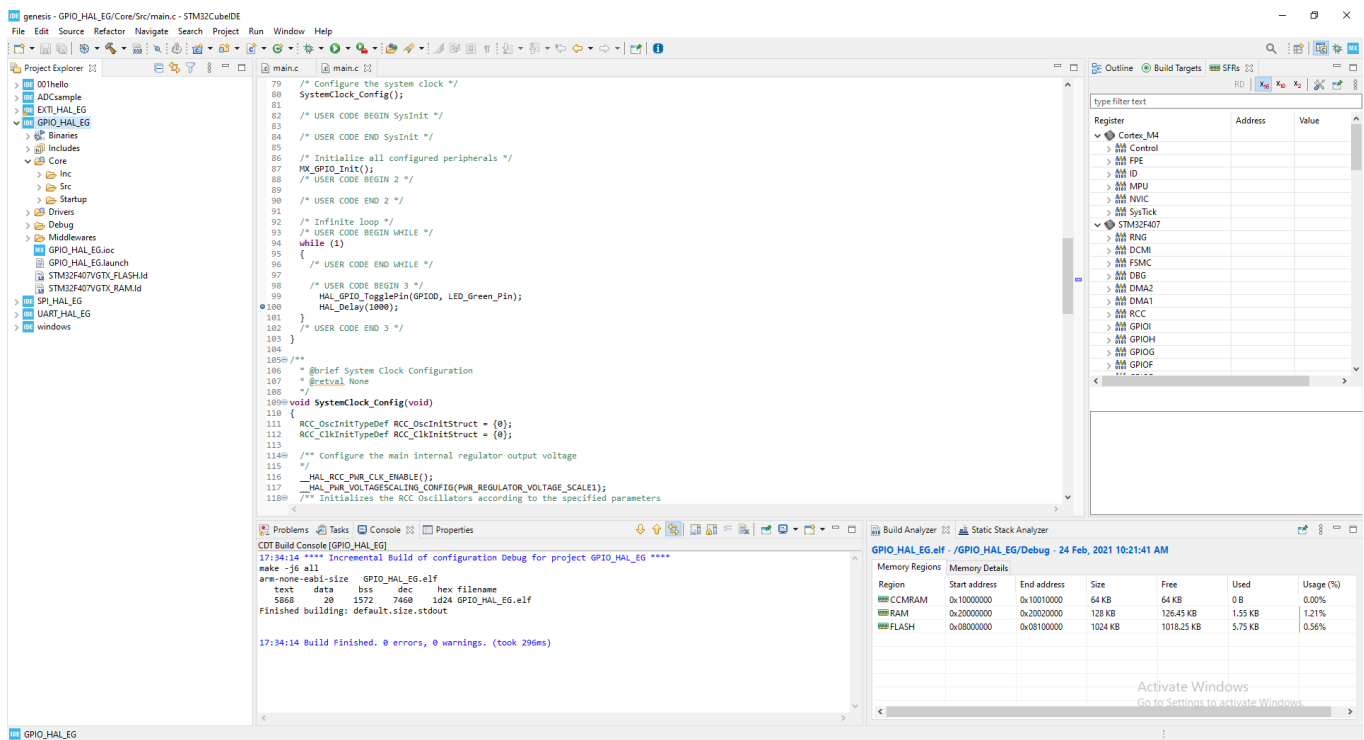


Figure 8

2. EXT INTERRUPTS

Interrupts are a piece of code which halts the main function and tells to execute this piece of code. Once the interrupt is executed, the main function continues from where it has started. Depending on the micro-controller interrupts are present.

There are two types of internals in a micro controller. Ones that control from inside are the internal interrupts and the ones that are controlled externally like buttons, switches are called external interrupts. In this stm32 board we are controlling the LED through an external interrupt.

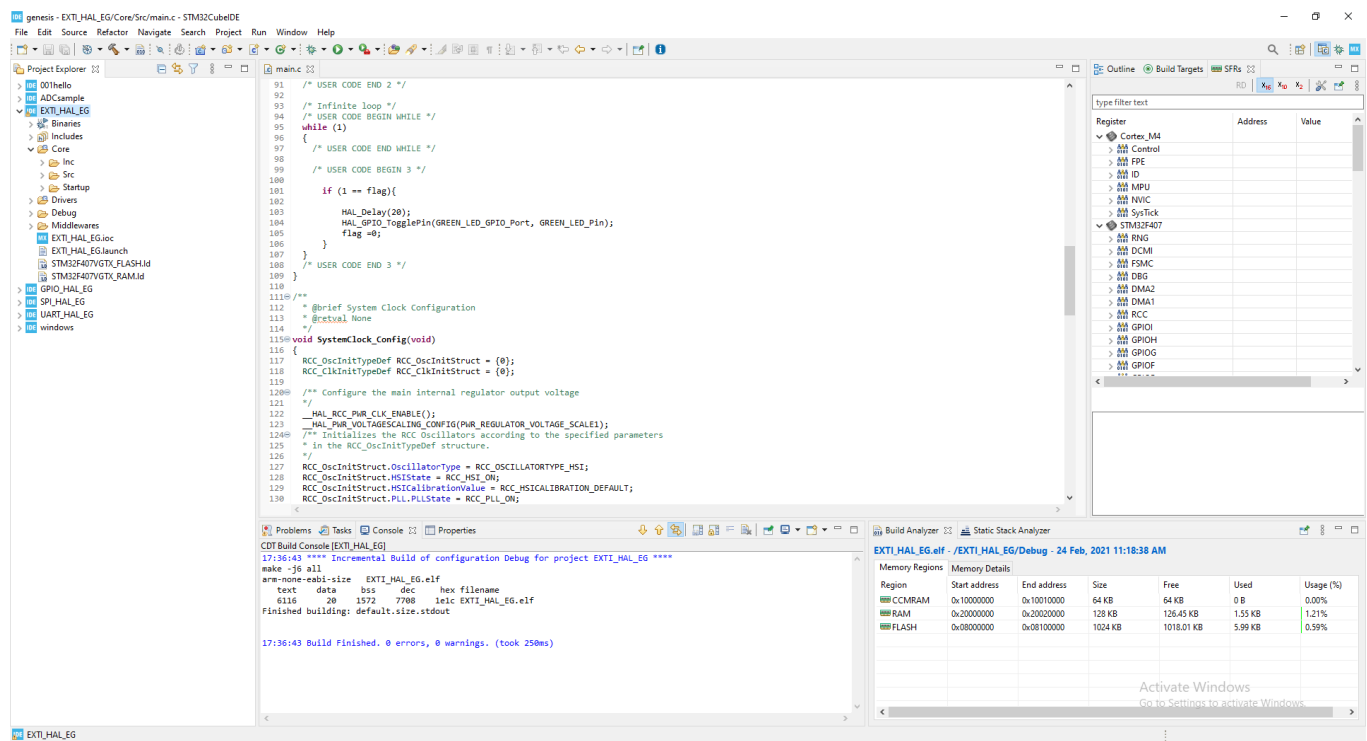
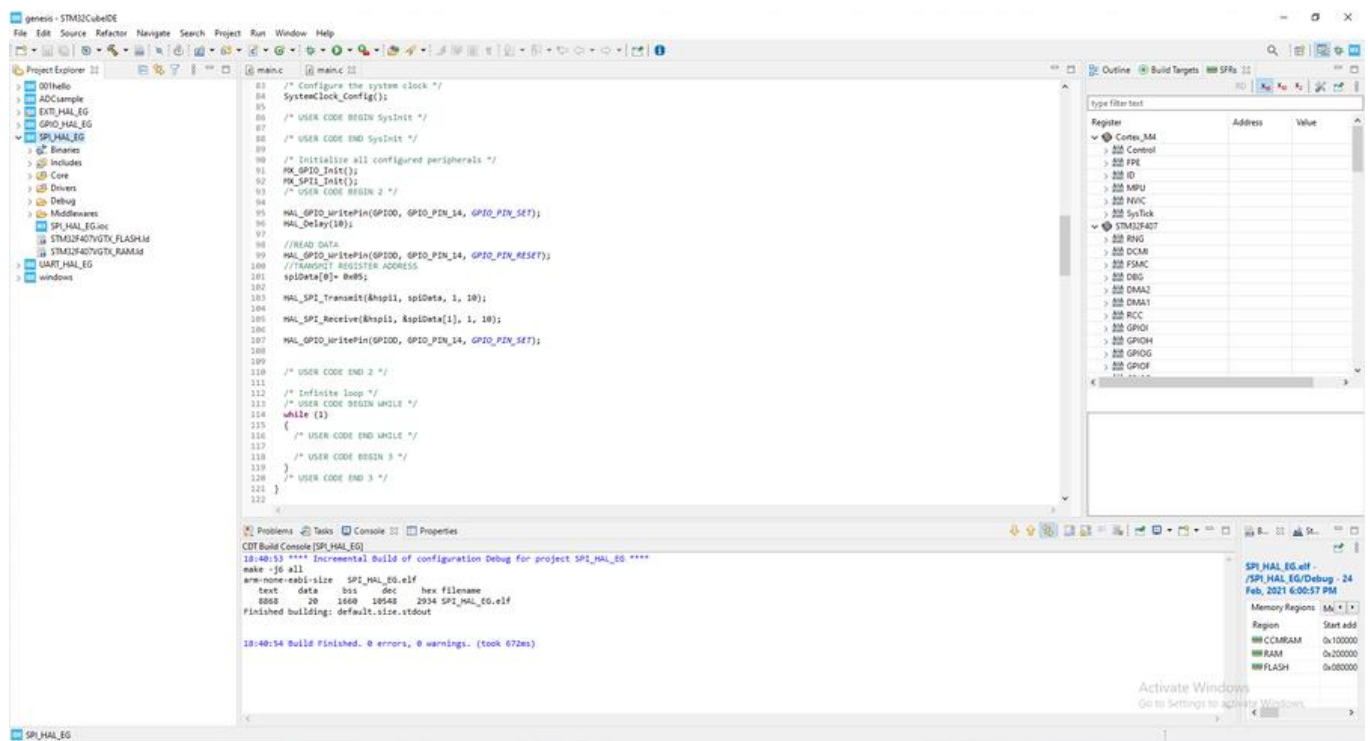
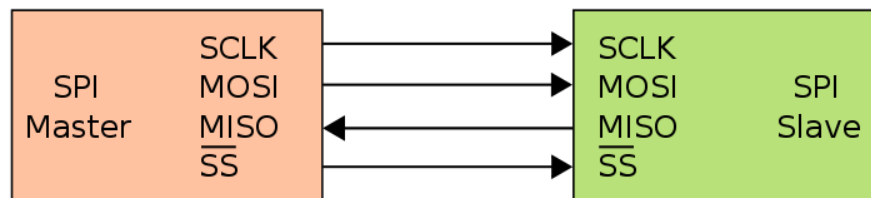


Figure 9

3. SPI

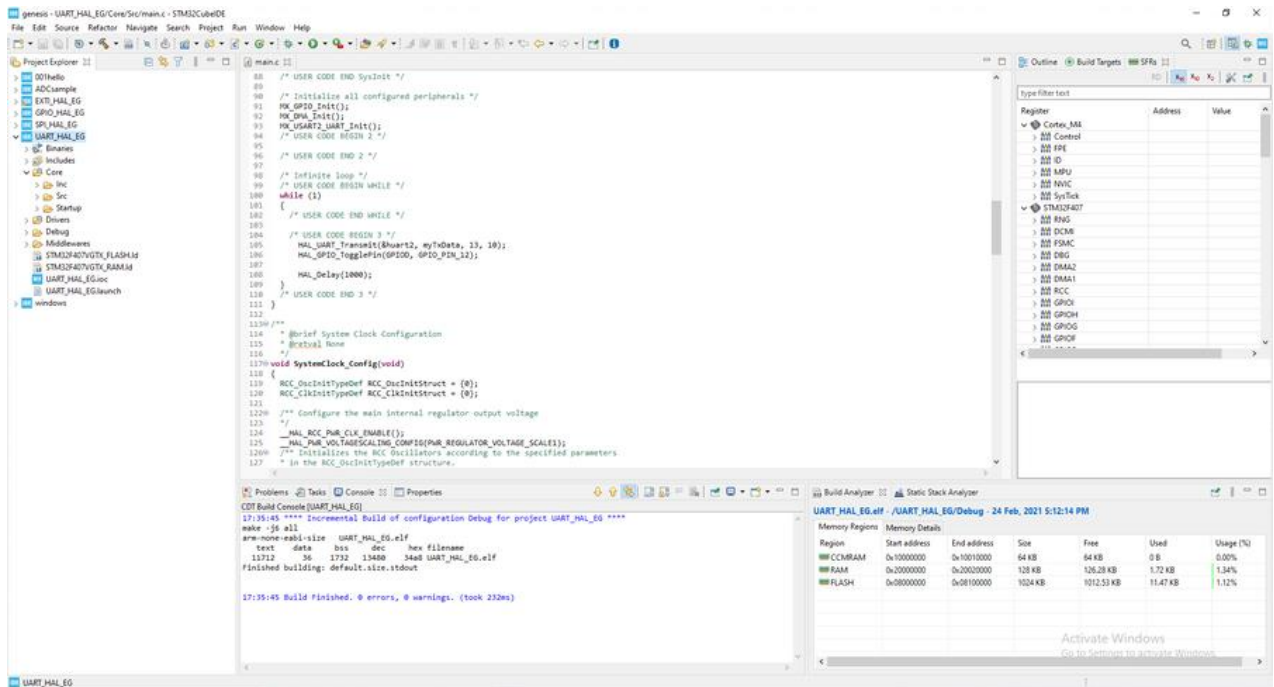
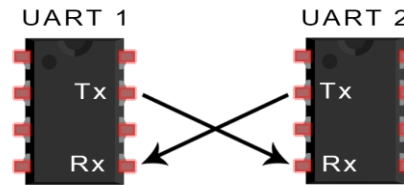
SPI is a serial communication protocol. The full form is Synchronous Data transfer Protocol. These are used for high -speed communication. SPI operation based upon shift registers. Master or Slave has an 8bit registers inside it. SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave-devices are supported through selection with individual slave select (SS), sometimes called chip select (CS), lines.



Here we are transmitting and receiving data using SPI

4. UART

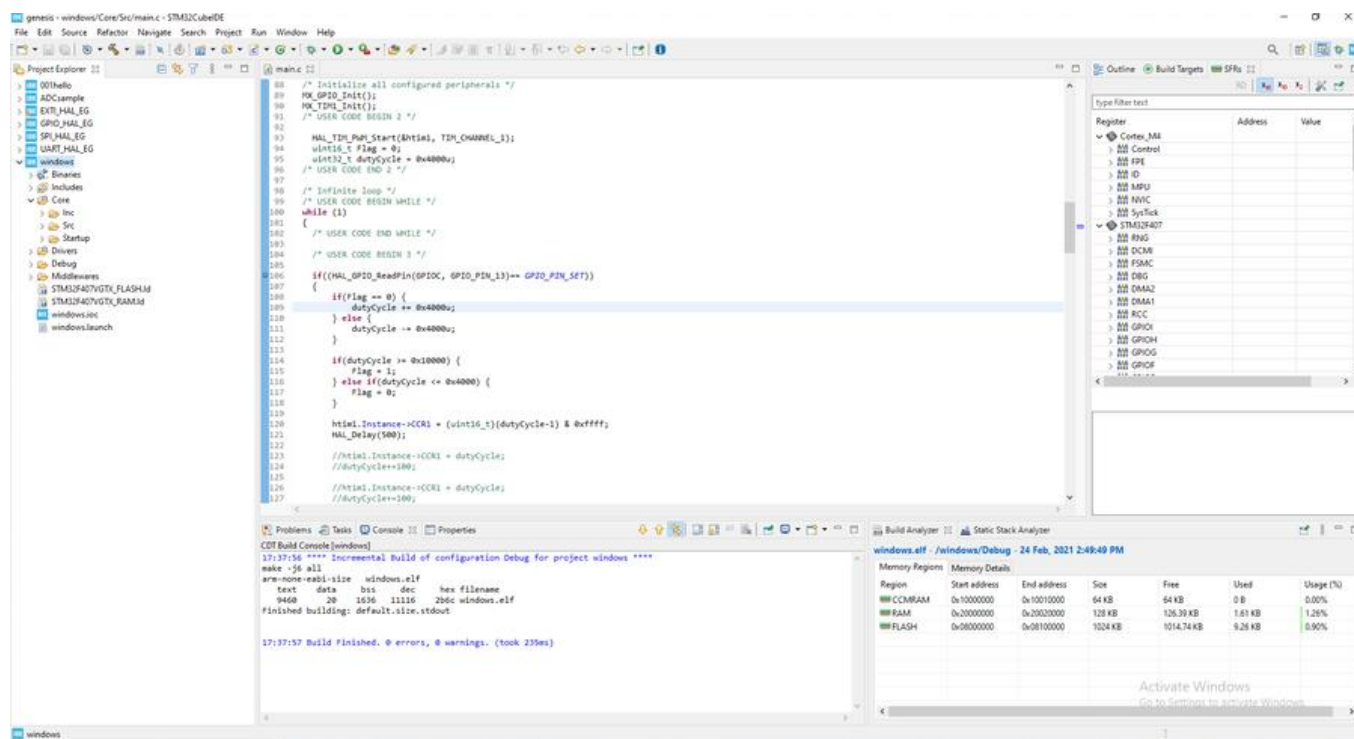
In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART



SPI interface of the STM32 devices using the STM32CubeMX HAL API.

5. PWM

I used the STM32Cube initialization code generator to generate an initialized Timer function. To generate a fixed duty cycle PWM signal I added HAL_TIM_Base_Start ; //Starts the TIM Base generation and HAL_TIM_PWM_Start //Starts the PWM signal generation to the Timer initialization function as shown below.



```

108  /* Initialize all configured peripherals */
109  MX_GPIO_Init();
110  MX_TIM1_Init();
111  /* USER CODE BEGIN 2 */
112
113  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
114  uint16_t flag = 0;
115  uint32_t dutyCycle = 0x4000u;
116  /* USER CODE END 2 */
117
118  /* Infinite loop */
119  /* USER CODE BEGIN WHILE */
120  while (1)
121  {
122      /* USER CODE END WHILE */
123
124      /* USER CODE BEGIN 3 */
125      if((HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET))
126      {
127          if(flag == 0) {
128              dutyCycle += 0x4000u;
129          } else {
130              dutyCycle -= 0x4000u;
131          }
132
133          if(dutyCycle >= 0x10000) {
134              flag = 1;
135          } else if(dutyCycle <= 0x4000) {
136              flag = 0;
137          }
138
139          htim1.Instance->CCR1 = (uint16_t)(dutyCycle-1) & 0xffff;
140          HAL_Delay(500);
141
142          //htim1.Instance->CCR1 = dutyCycle;
143          //dutyCycle+=100;
144
145          //htim1.Instance->CCR1 = dutyCycle;
146          //dutyCycle-=100;
147      }
148  }
149  }

```

Build Console [windows]

```

17:37:56 **** Incremental Build of configuration Debug for project windows ****
make -j6 all
arm-none-eabi-size: windows.elf
text      data      bss      dec      hex filename
9460      20      1636      11116      2bdc windows.elf
Finished building: default.size.txtout
17:37:57 Build Finished. 0 errors, 0 warnings. (took 239us)

```

Build Analyzer Static Stack Analyzer

windows.elf - /windows/Debug - 24 Feb, 2021 2:49:49 PM

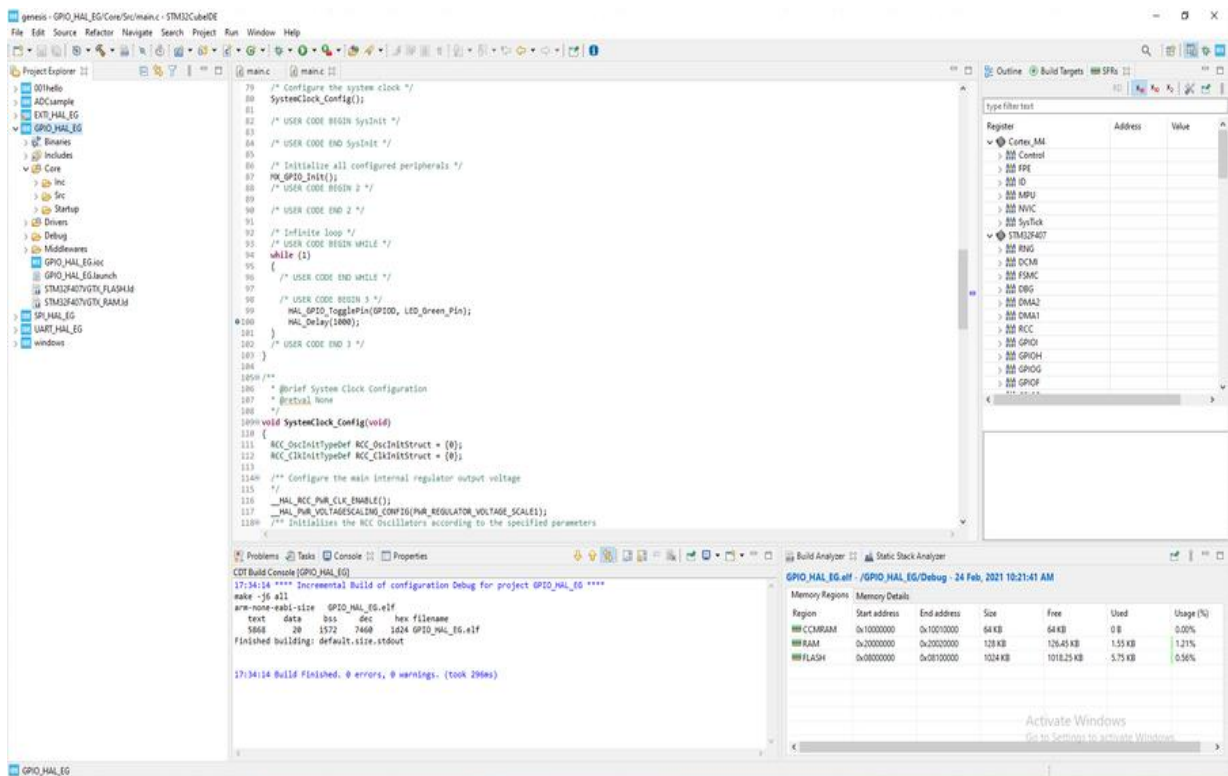
Memory Region	Start address	End address	Size	Free	Used	Usage (%)
CCM/DRAM	0x10000000	0x10010000	64 KB	64 KB	0 B	0.00%
RAM	0x20000000	0x20020000	128 KB	126.39 KB	1.61 KB	1.25%
FLASH	0x08000000	0x08100000	1024 KB	1014.74 KB	9.26 KB	0.90%

Reading GPIO and setting duty signal, we set flag and adjust PWM

6. ADC

One of the most common peripherals on many modern microcontrollers is the analog-to-digital converter (ADC). These embedded devices read an analog voltage (usually somewhere between 0 V and the given reference voltage) and report it as a binary value. The exact implementation of the ADC can change among STM32 chips, as some use the successive-approximation register (SAR) technique while others rely on sigma-delta modulation for more resolution (but lower speeds).

ADC on the STM32L476 with STM32CubeIDE and HAL. Direct memory access (DMA) controller to demonstrate how you might handle moving (relatively) large amounts of data in your microcontroller.



The screenshot displays the STM32CubeIDE environment for a project named 'GPIO_HAL_EG'. The main.c file is open, showing the initialization and main loop of the program. The console window shows the build process, and the memory regions window displays the memory layout.

main.c Code Snippet:

```

79 /* Configure the system clock */
80 SystemClock_Config();
81
82 /* USER CODE BEGIN SysInit */
83
84 /* USER CODE END SysInit */
85
86 /* Initialize all configured peripherals */
87 MX_GPIO_Init();
88 /* USER CODE BEGIN 2 */
89
90 /* USER CODE END 2 */
91
92 /* Infinite loop */
93 /* USER CODE BEGIN WHILE */
94 while (1)
95 {
96     /* USER CODE END WHILE */
97
98     /* USER CODE BEGIN 3 */
99     HAL_GPIO_TogglePin(GPIOID, LED_GREEN_Pin);
100     HAL_Delay(1000);
101 }
102 /* USER CODE END 3 */
103 }
104
105 /**
106  * @brief System Clock Configuration
107  * @retval None
108  */
109 void SystemClock_Config(void)
110 {
111     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
112     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
113
114     /** Configure the main internal regulator output voltage
115     */
116     __HAL_RCC_PWR_CLK_ENABLE();
117     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
118     /** Initializes the RCC Oscillators according to the specified parameters
119     */

```

Console Output:

```

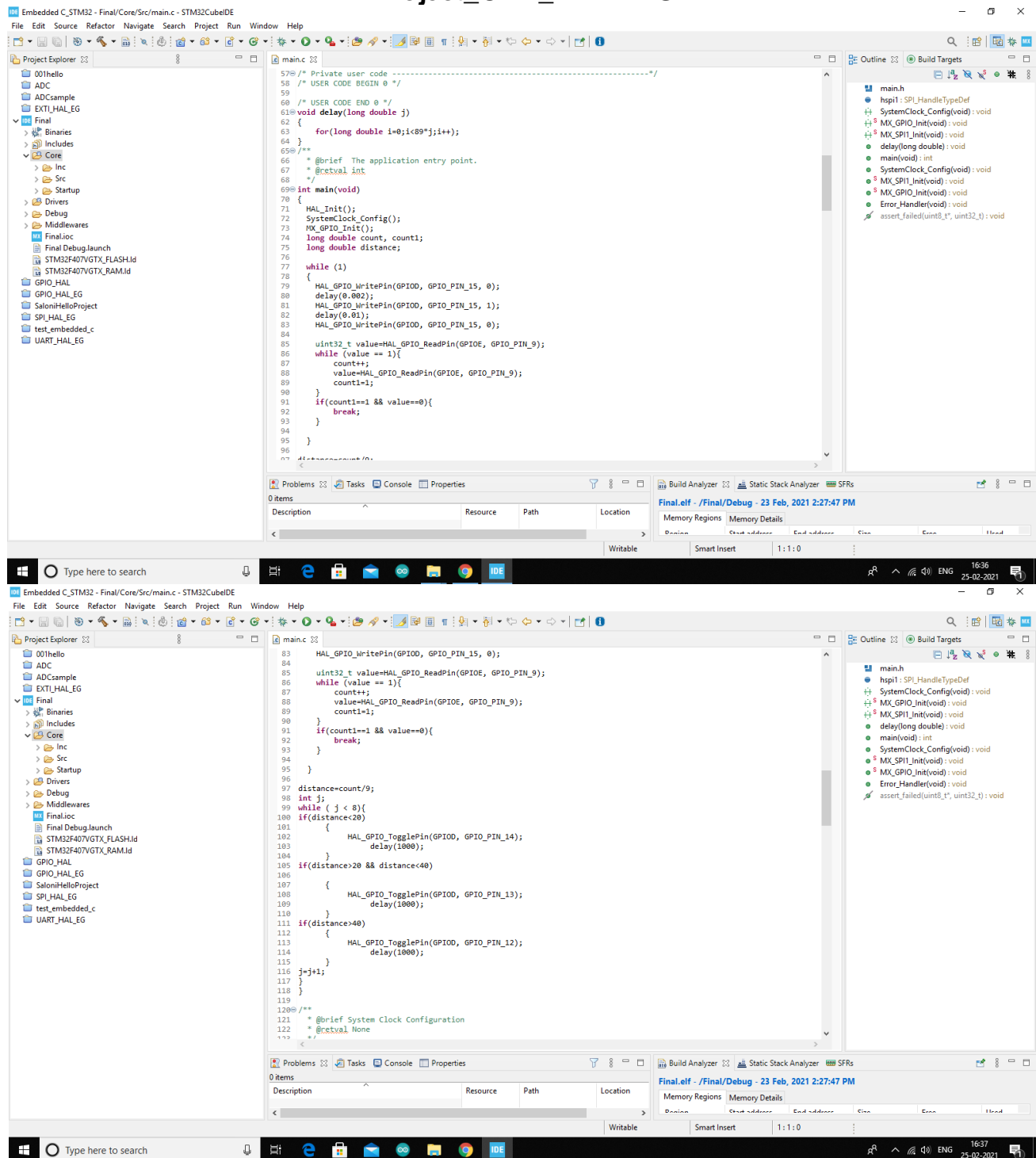
17:34:14 *** Incremental Build of configuration Debug for project GPIO_HAL_EG ***
make -j6 all
arm-none-eabi-size GPIO_HAL_EG.elf
text      data      bss      hex  filename
5068      20      5572      7460      1624  GPIO_HAL_EG.elf
Finished building: default.sizestdout
17:34:14 Build Finished. 0 errors, 0 warnings. (took 296ms)

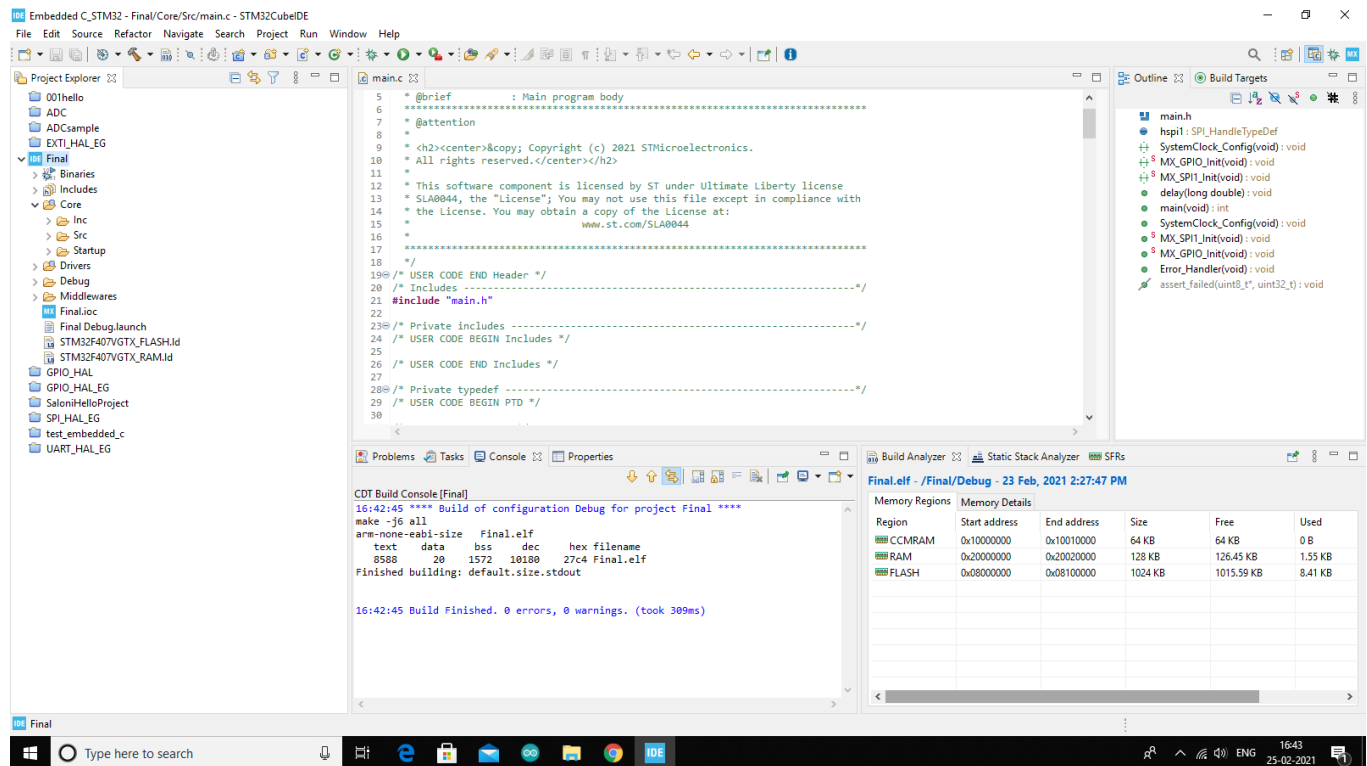
```

Memory Regions Table:

Region	Start address	End address	Size	Free	Used	Usage (%)
MMIO/CMRAM	0x10000000	0x10010000	64 KB	64 KB	0 B	0.00%
MMRAM	0x20000000	0x20020000	128 KB	126 KB	1.55 KB	1.21%
MMFLASH	0x08000000	0x08100000	1024 KB	1018.25 KB	5.75 KB	0.56%

Project_CAR_PARKING





Here we are using ultrasonic sensor for measuring distance between object behind and Car. So that user can easily park car.

We are using three LEDs- Green , Orange , Red

Green – Nothing is there, Go fast and park

Orange – some distance is there between car and object , Be careful and Park

Red- Stop immediately, Object is so close

