Laboratory Manual for

EC718–Embedded Systems

B. Tech.

SEM. VII (EC)

Department of Electronics & Communication
Faculty of Technology
Dharmsinh Desai University
Nadiad

TABLE OF CONTENTS

PART – 1 LABORATORY MANUAL

PART I

LAB MANUAL

LAB 1 INTRODUCTION TO KEIL µVISION 5

Aim: To study the environment of Keil µVision 5 and write a program in "C" to print "Hello World"

Theory:

The µVision5 IDE is a Windows-based software development platform that combines a robust editor, project manager, and makes facility. µVision5 integrates all tools including the C compiler, macro assembler, linker/locator, and HEX file generator. µVision5 helps expedite the development process of your embedded applications by providing the following:

- Full-featured source code editor
- Device database for configuring the development toolset
- Project manager for creating and maintaining your projects
- Integrated make facility for assembling, compiling, and linking your embedded applications,
- Dialogues for all development tool settings,
- True integrated source-level Debugger with high-speed CPU and peripheral simulator.
- Advanced GDI interface for software debugging in the target hardware and for connection to KeilULINK
- Flash programming utility for downloading the application program into Flash ROM,
- Links to development tools manuals, device datasheets & user‟s guides.

The µVision5 IDE offers numerous features and advantages that help you quickly and successfully develop embedded applications. They are easy to use and are guaranteed to help you achieve your design goals.

Getting Started with µVision5

Create new project by clicking **Project→ New project,** select the device STM32F10C38 (or a device from other family, for ex.STM32F4xxxx) as shown in Fig. 1.1

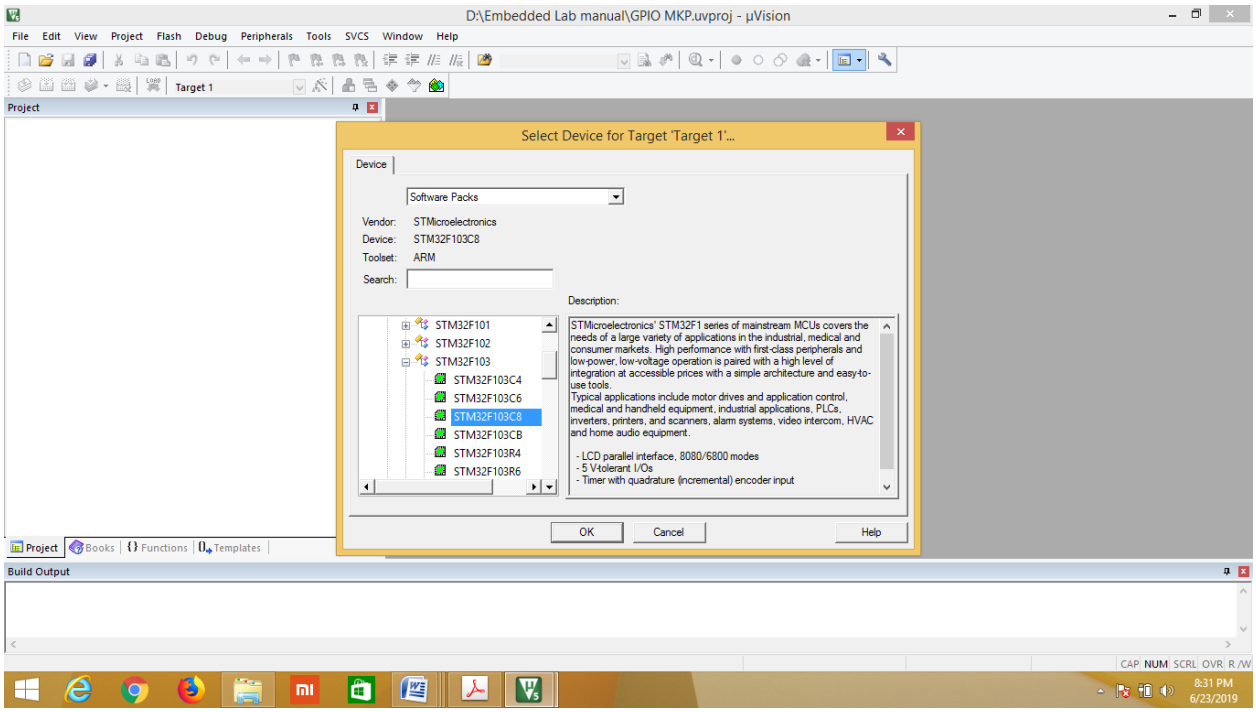


Fig. 1.1 Selection of microcontroller STM32F10C38

Select Startup file from Devices in Manage Run-Time environment window as shown in fig.1.2

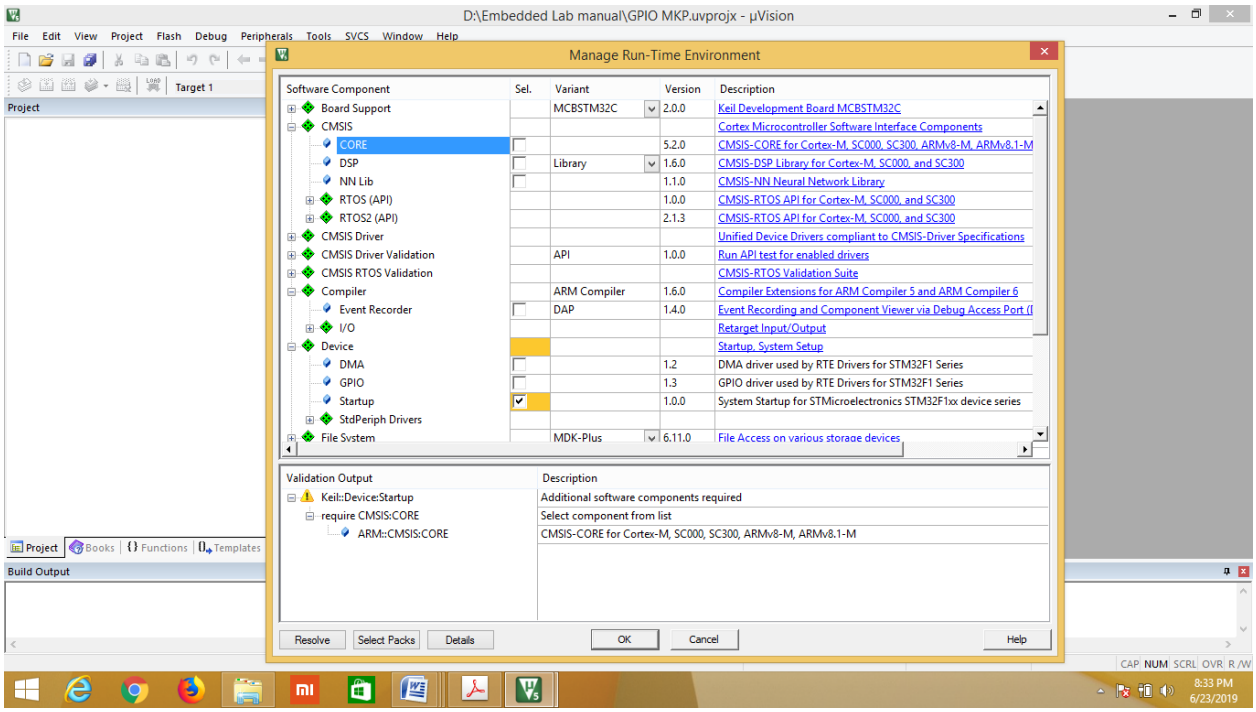


Fig 1.2 Selection of Startup file

As Shown in Fig 1.2, Lower half of Manage Run-Time environment shows that to include startup file ‘additional software components required’. To include additional software component click on “Resolve” button, it will automatically include additional components (in this case it will include “Core” as shown in Fig 1.3

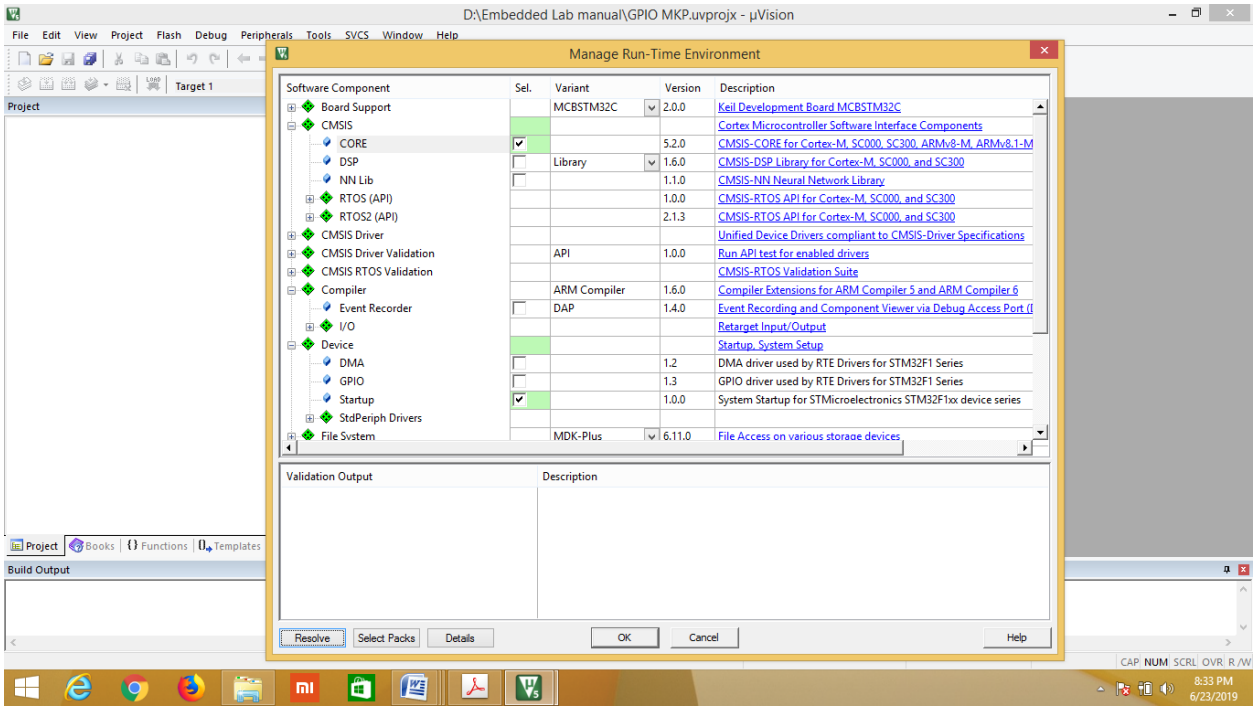


Fig 1.3 Inclusion of additional software component(CORE)

Now to get standard output in the debug window (output of printf function for simulation) we need to select STDOUT option in COMPILER→IO as shown in Fig 1.4

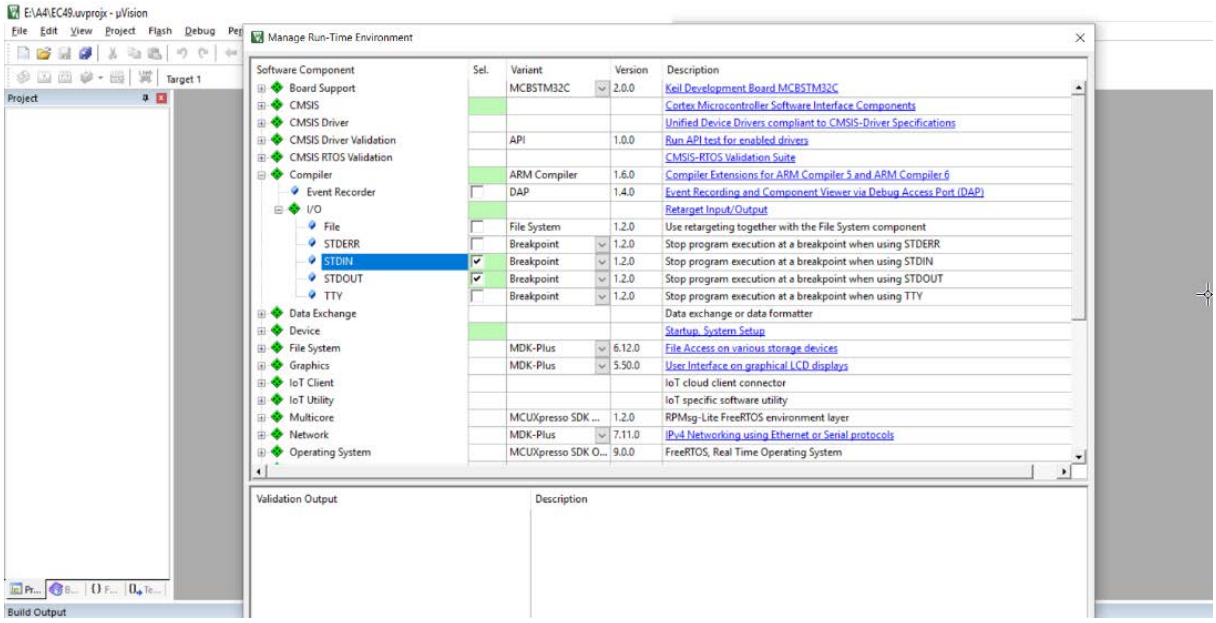


Fig 1.4 Selection of STDOUT for debug window

Add new empty C file in the project and type the sample program in the C file and then build the project as shown in fig 1.5

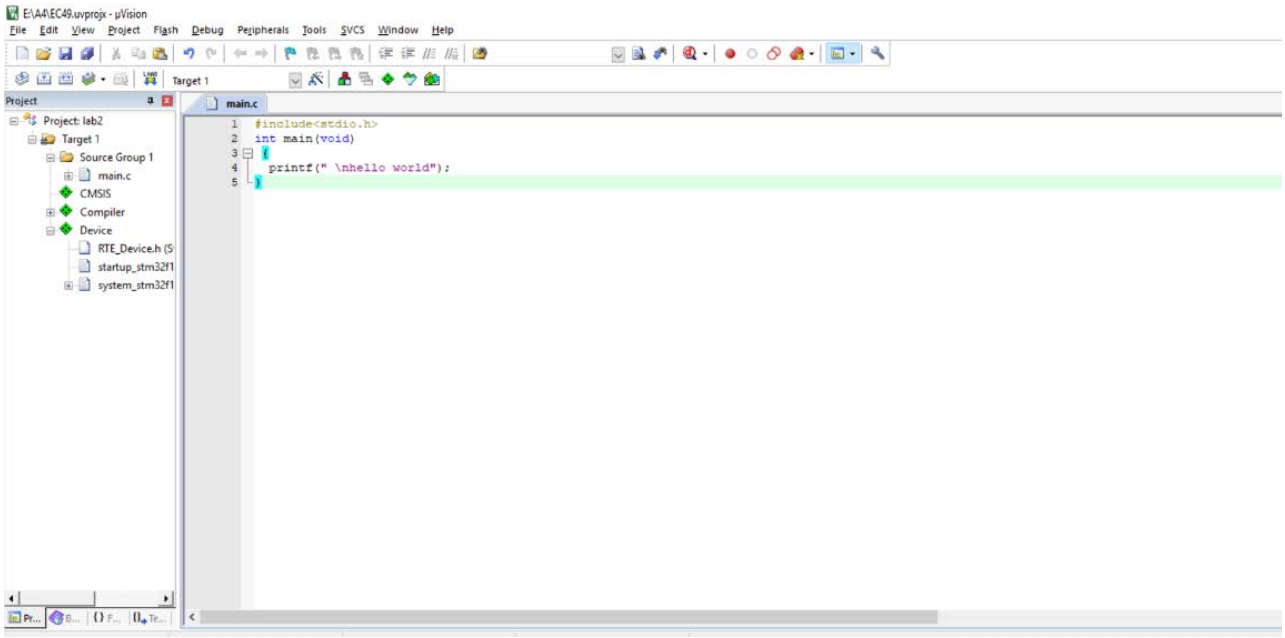


Fig 1.5 Add source code in the project

Now click on “Build” to compile the program, after compilation click on the debug button to run on the simulator. Now open debug viewer from **VIEW→ Serial windows** as shown in Fig 1.6

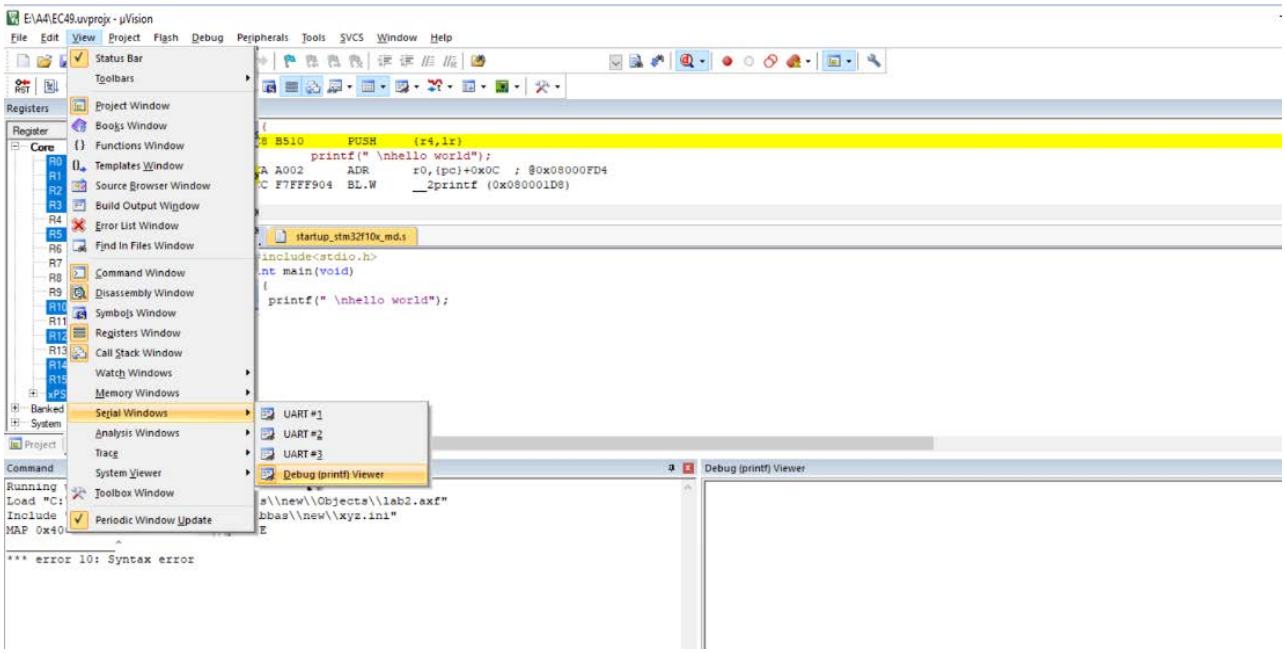


Fig 1.6 Selecting debug viewer for output

Click on free run and you can see the output in the debug viewer as shown in fig 1.7

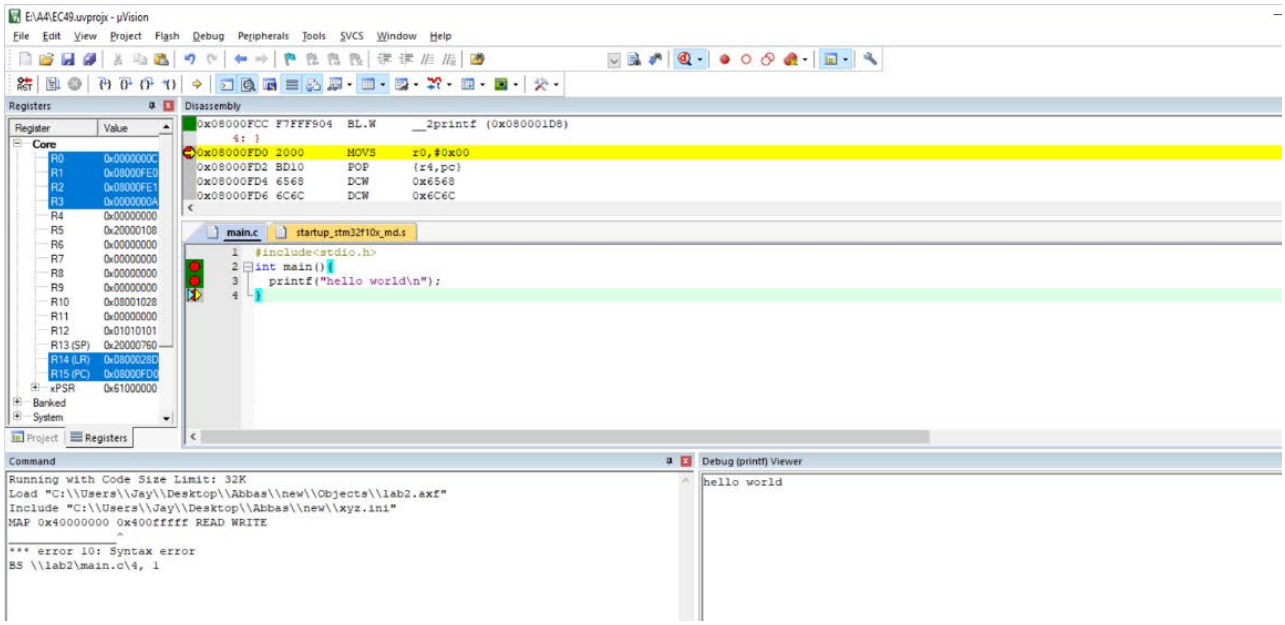


Fig 1.7 Output of printf() in debug viewer

Assignment

1) What is breakpoint and how it is useful for debugging ? How printf() statements are used for debugging?
2) What is the difference between build and rebuild?
3) What is Disassembly window?
4) List different windows commonly used for debugging
5) Create a new project for a device from STM32F4xxxx family and repeat the experiment

Conclusion

LAB 2 GENERAL PURPOSE INPUT OUTPUT PORTS PROGRAMMING

Aim: To program General Purpose Input Output Ports of STM32F10x/4xx in Keil µVision 5

Theory:

Each of the general-purpose I/O ports in STM32F10x family has

- **Two 32-bit configuration registers**
  Port configuration register low : GPIOx*_CRL,
  Port configuration register High: GPIOx*_CRH
- **Two 32-bit data registers**
  Port input data register: GPIOx_IDR,
  Port output data register: GPIOx_ODR
- **A 32-bit set/resetregister**
  Port bit set/reset register: GPIOx_BSRR
- **A 16-bit reset register**
  Port bit reset register: GPIOx_BRR
- **A 32-bit locking register**
  Port configuration lock register: GPIOx_LCKR)
  *x=A to G port

Subject to the specific hardware characteristics of each I/O port listed in the *datasheet*, each port bit of the General Purpose IO (GPIO) Ports, can be individually configured by software in several modes:

● Input floating
● Input pull-up
● Input-pull-down
● Analog Input
● Output open-drain
● Output push-pull
● Alternate function push-pull
● Alternate function open-drain

Each I/O port bit is freely programmable, however the I/O port registers have to be accessed as 32-bit words (half-word or byte accesses are not allowed). The purpose of the GPIOx_BSRR and GPIOx_BRR registers is to allow atomic read/modify accesses to any of the GPIO registers. Details of these registers is shown in fig 2.1 to 2.3

GPIO Registers:

- **Port configuration register high (GPIOx_CRL) (x=A..G)**

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30, 27:26, 23:22, 19:18, 15:14, 11:10, 7:6, 3:2 **CNFy[1:0]:** Port x configuration bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table.*
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
11: Reserved
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

Bits 29:28, 25:24, 21:20, 17:16, 13:12, 9:8, 5:4, 1:0 **MODEy[1:0]:** Port x mode bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table.*
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

Fig 2.1 GPIOx CRL

- **Port configuration register high (GPIOx_CRH)(x=A..G)**

Address offset: 0x04

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Fig 2.2 GPIOx CRH

Configuration same as CRL

- **Port bit set/reset register (GPIOx_BSRR)(x=A..G)**

Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x Reset *bit y (y= 0 .. 15)*
These bits are write-only and can be accessed in Word mode only.
0: No action on the corresponding ODRx bit
1: Reset the corresponding ODRx bit
Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x *Set bit y (y= 0 .. 15)*
These bits are write-only and can be accessed in Word mode only.
0: No action on the corresponding ODRx bit
1: Set the corresponding ODRx bit

Fig 2.3 GPIOx BSRR

Sample program: Program to Blink LED connected to PORTC pin 13

```
#include<stm32f10x.h>

int main()
	{
	int i;

	RCC->APB2ENR |= (1<<4);  enable  I/O port C clock
	; Configure GPIO pins: Push-pull Output Mode with Pull-down resistors
	GPIOC->CRH |= ( (1<<20) | (1<<21)) ;

	GPIOC->CRH &= ~( (1<<22) | (1<<23)) ;

	while(1)
		{
		GPIOC->BSRR =(1<<13); turn ON LED connected to PORT C pin13
		for(i=0;i<2000000;i++) ;    delay

		GPIOC->BSRR =1<<(13+16); turn OFF LED connected to PORT C pin13
		for(i=0;i<2000000;i++) ; delay
		}
	}
```

Simulation Steps: (Using Keil µVision 5 IDE)

Create new project by clicking **Project→ New project,** select the device STM32F10C38 as shown in Fig. 2.4

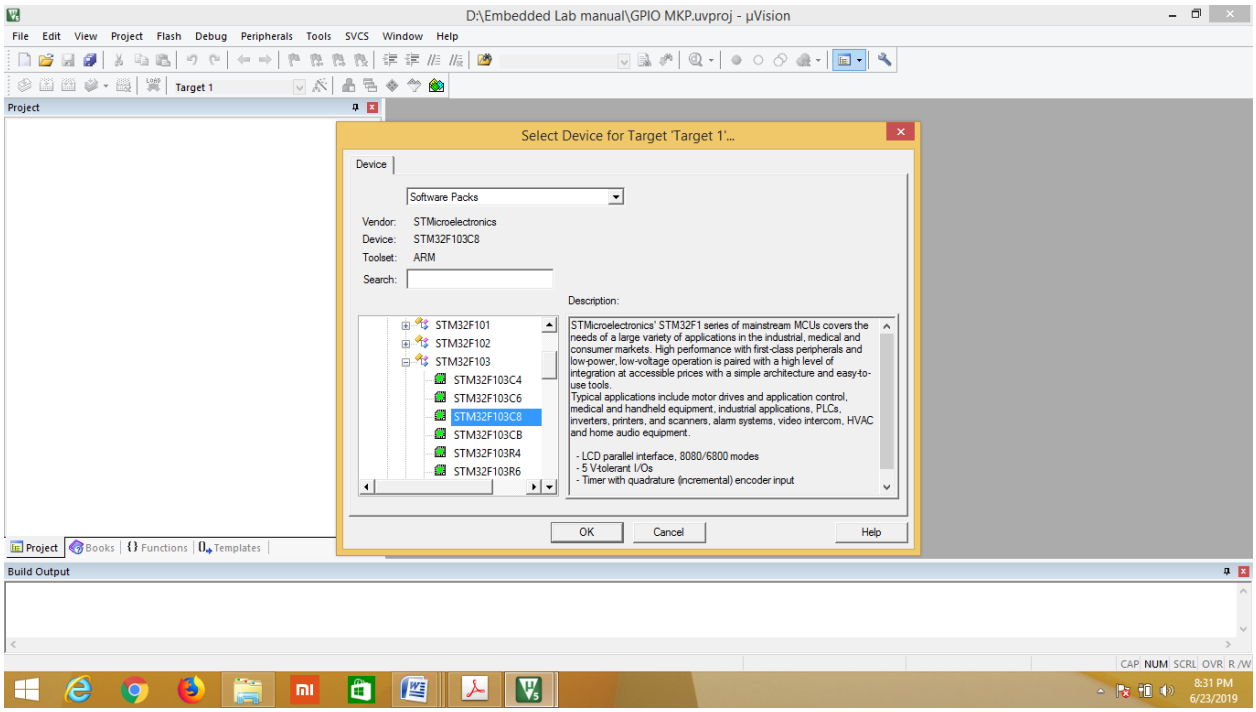


Fig. 2.4 Selection of microcontroller STM32F10C38

Select Startup file from Devices in Manage Run-Time environment window as shown in fig.2.5

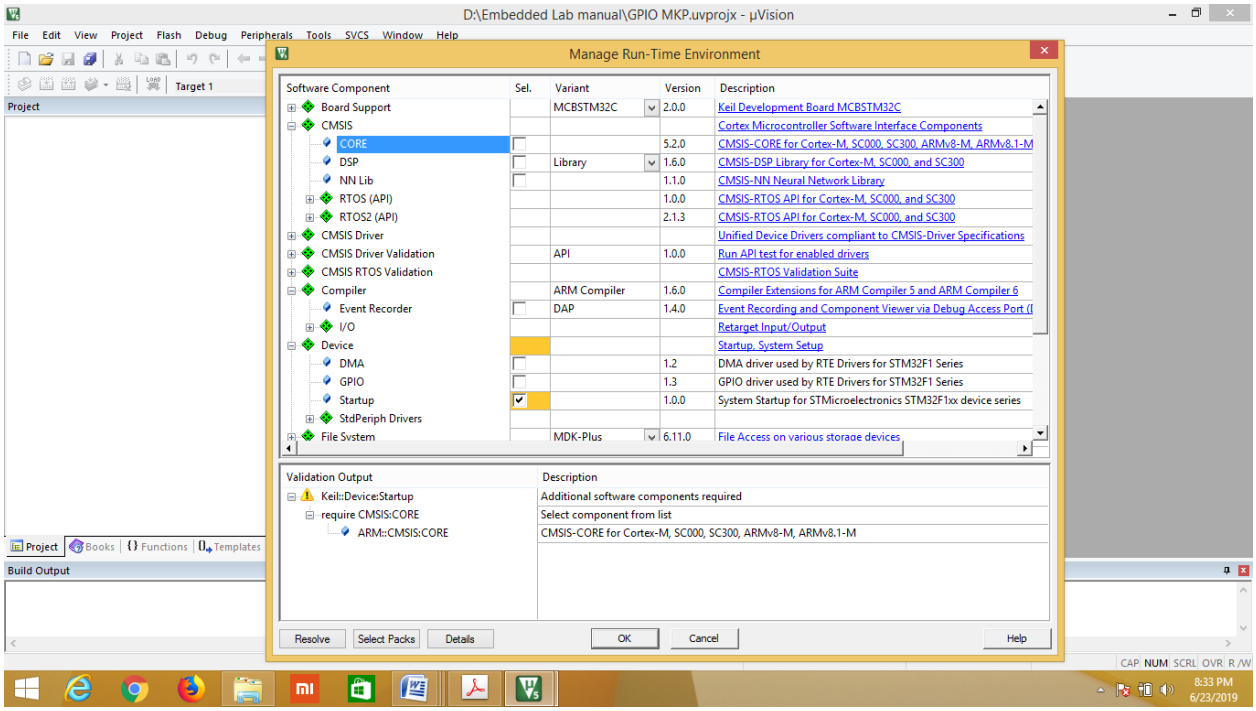


Fig 2.5 Selection of Startup file

As Shown in Fig 2.5, Lower half of Manage Run-Time environment shows that to include startup file 'additional software components required'. To include additional software component click on "Resolve" button, it will automatically include additional components (in this case it will include "Core" as shown in Fig 2.6

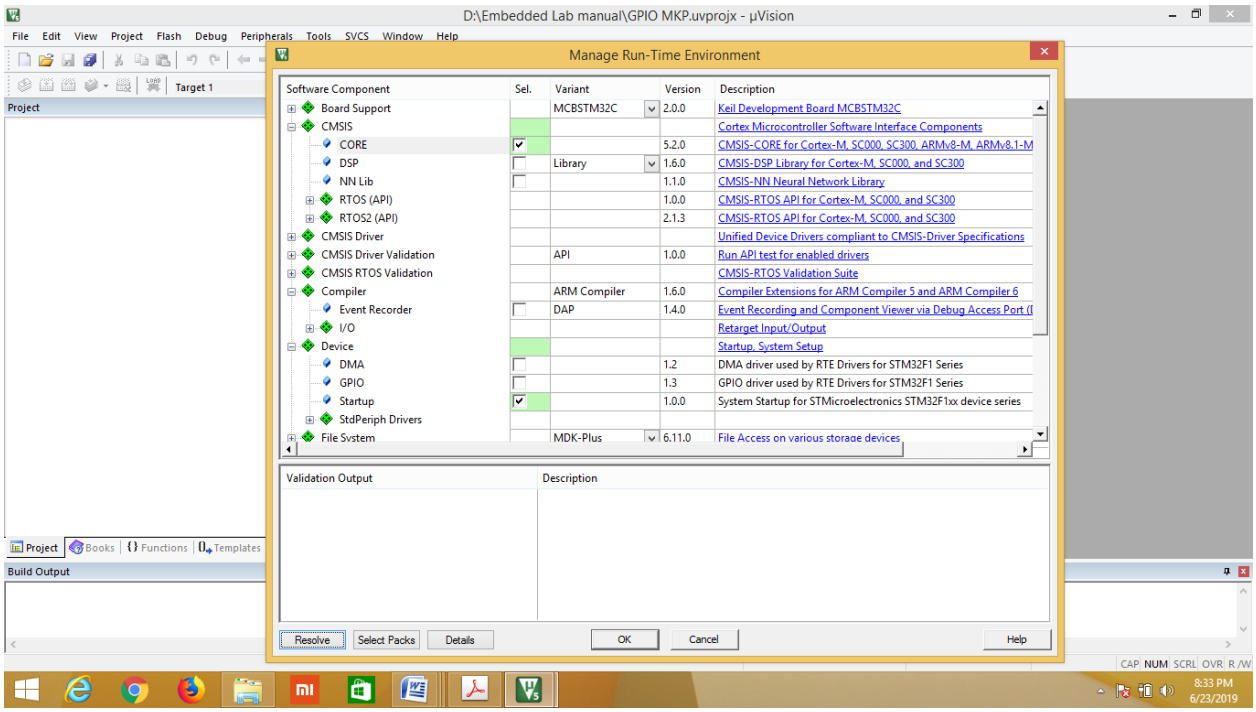


Fig 2.6 Inclusion of additional software component(CORE)

Add new empty C file in the project and type the sample program in the C file and then build the project as shown in fig 2.7

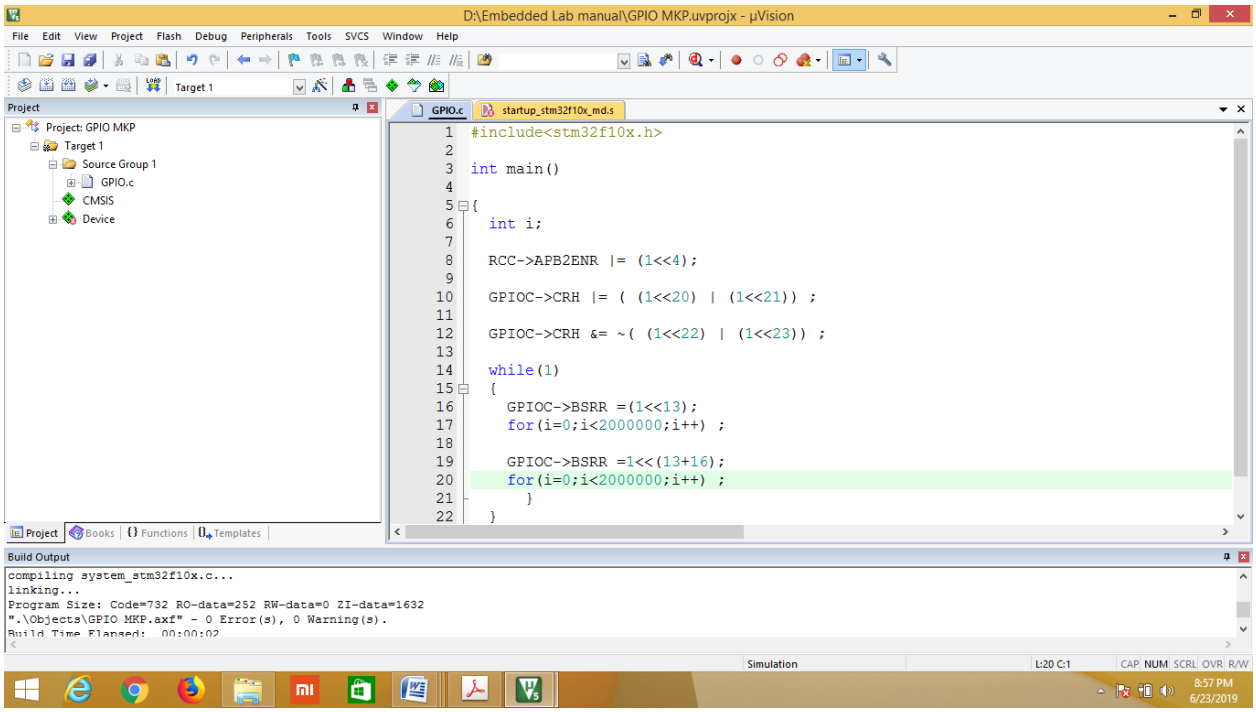


Fig 2.6 Adding C program file and building the project

Now select the Initialization file from Options for Target ‘Target 1’ window, in this case it is ‘XYZ.ini’ as shown in Fig 2.7. The initialization file is required for the simulation only.

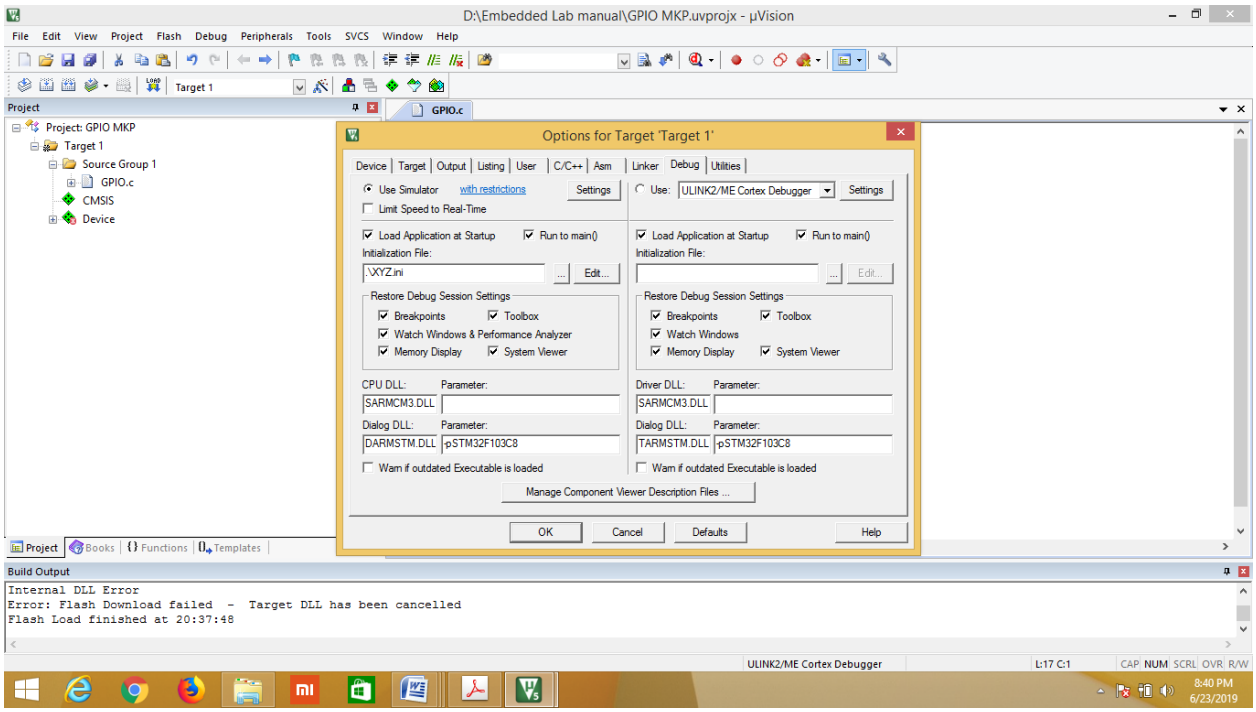


Fig 2.7 Selection of Initialization file

If initialization file does not exist, then create new one type the contents as shown in Fig 2.8

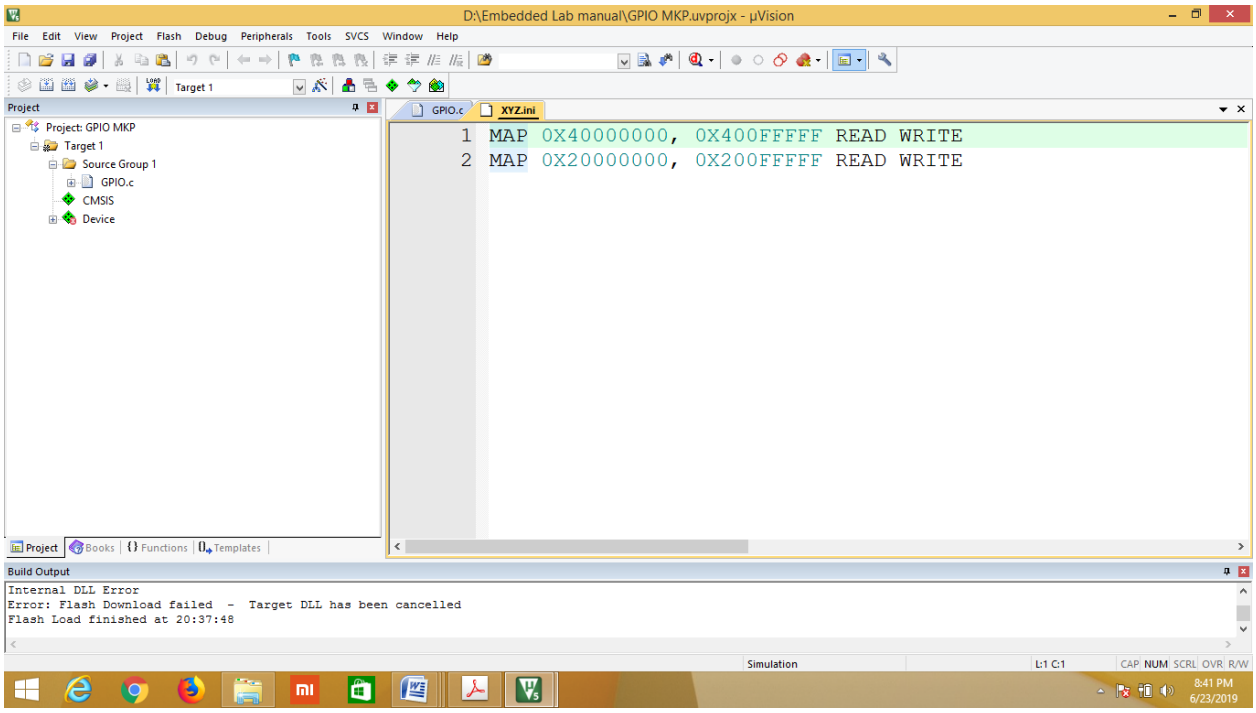


Fig 2.8 Contents of the Initialization file

Start the debug session and debug up to line 17 as shown in Fig2.9, the 'General Purpose I/O C' windows shows that the LED connected to PORT C pin 13 is ON

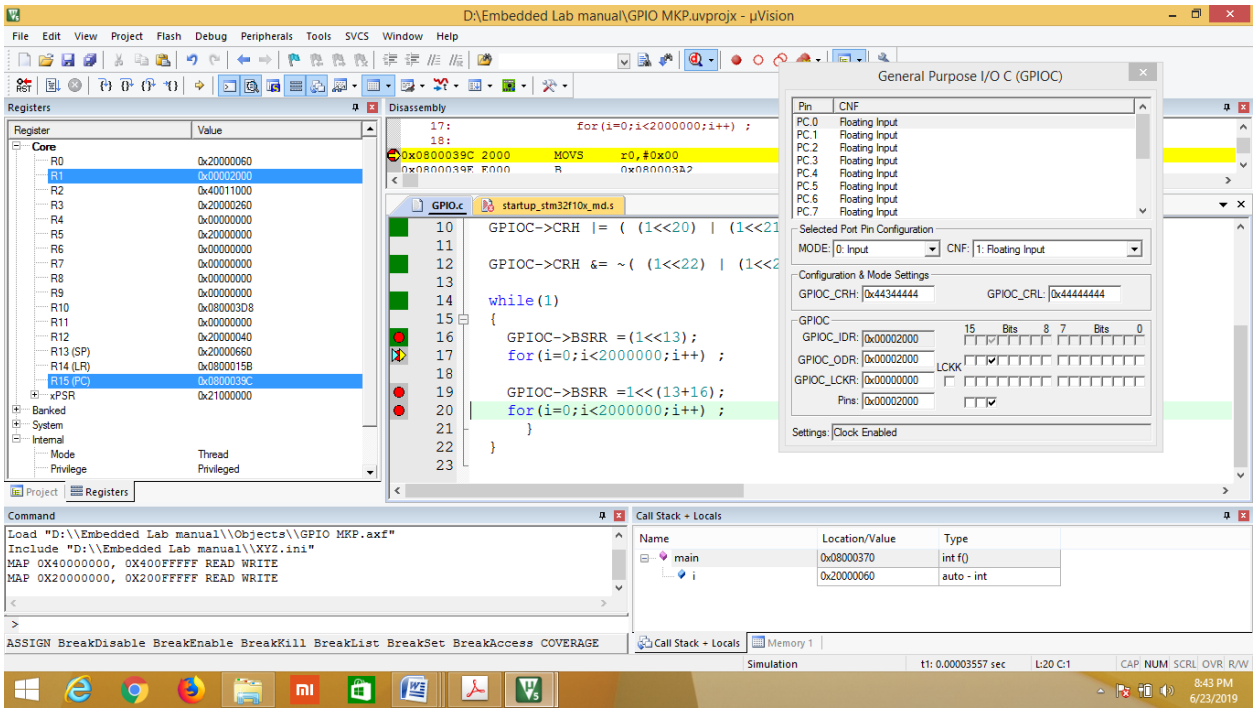


Fig 2.9 Debugging the program-LED ON

Continue the debug session and debug up to line 20 as shown in Fig 2.10, the 'General Purpose I/O C' windows shows that the LED connected to PORT C pin 13 is OFF

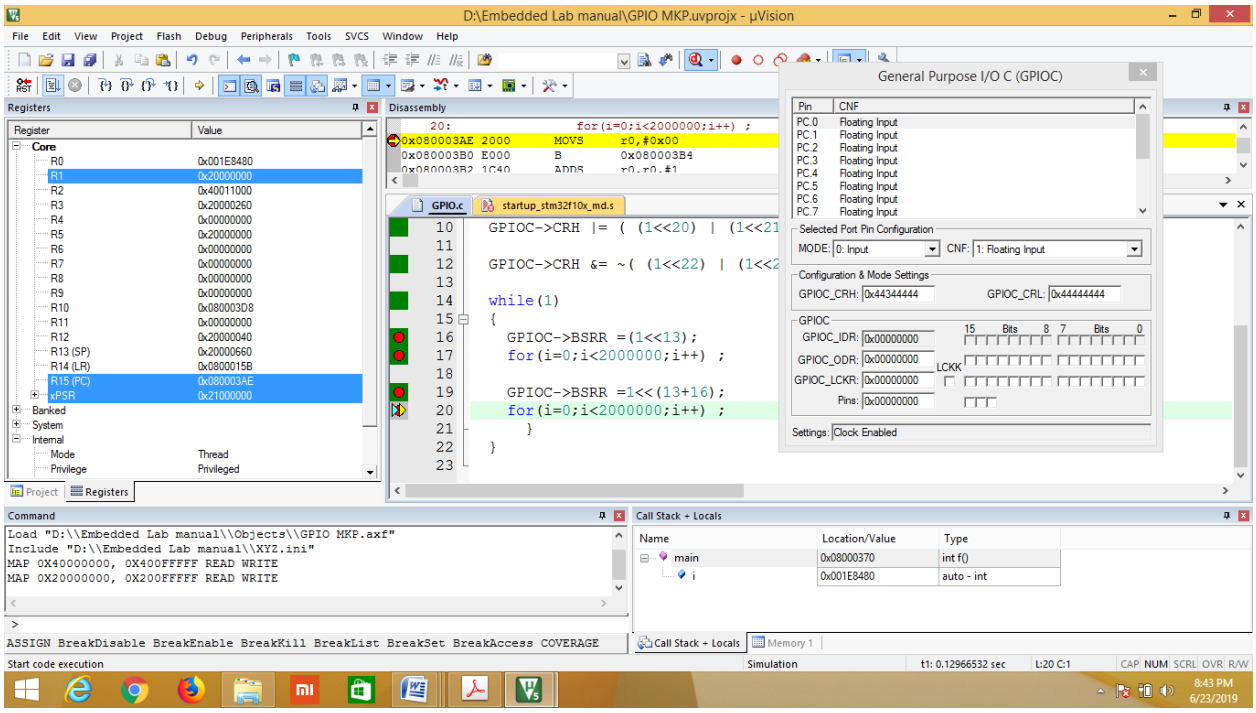


Fig 2.10 Debugging the program-LED OFF

Assignment:

1. Modify above program to scroll the LED (turned ON) from right (Port C Pin 0) to left (Port C Pin 7)

2. Modify above program to Blink all LEDs connected to PORT C
3. List and briefly explain the SFRs involved with the GPIO pins of STM32F10x family
4. Develop similar project for the microcontroller from other vendor for ex. LPC1768 from NXP

Conclusion:

_____
_____
_____
_____
_____

LAB 3 GPIO PROGRAMMING USING CMSIS

Aim: To program General Purpose Input Output Ports using CMSIS

Theory:

CMSIS

CMSIS is the Cortex Microcontroller Software Interface Standard defined by ARM. ARM provides this software package and specification for all ARM vendors. It is not an API specification; it's an organizational and style standard. It is a vendor-independent hardware abstraction layer for the Cortex-M processor series and defines generic tool interfaces. The CMSIS enables consistent device support and simple software interfaces to the processor and the peripherals, simplifying software re-use, reducing the learning curve for microcontroller developers, and reducing the time to market for new devices.

The CMSIS is defined in close cooperation with various silicon and software vendors and provides a common approach to interface to peripherals, real-time operating systems, and middleware components. The CMSIS is intended to enable the combination of software components from multiple middleware vendors.

CMSIS has been created to help the industry in standardization. It enables consistent software layers and device support across a wide range of development tools and microcontrollers. CMSIS is not a huge software layer that introduces overhead and does not define standard peripherals. The silicon industry can therefore support the wide variations of Cortex-M processor-based devices with this common standard.

The benefits of the CMSIS are:

- Overall CMSIS reduces the learning curve, development costs, and time-to-market. Developers can write software quicker through a variety of easy-to-use, standardized software interfaces.
- Consistent software interfaces improve the software portability and re-usability. Generic software libraries and interfaces provide consistent software framework.
- Provides interfaces for debug connectivity, debug peripheral views, software delivery, and device support to reduce time-to-market for new microcontroller deployment.
- Provides a compiler independent layer that allows using different compilers. CMSIS is supported by mainstream compilers.
- Enhances program debugging with peripheral information for debuggers and ITM channels for printf-style output and RTOS kernel awareness.
- CMSIS is delivered in CMSIS-Pack format which enables fast software delivery, simplifies updates, and enables consistent integration into development tools.

GPIO registers of STM32F10x

Each of the general-purpose I/O ports has

- **Two 32-bit configuration registers**
  Port configuration register low : GPIOx*_CRL,
  Port configuration register High: GPIOx*_CRH
- **Two 32-bit data registers**
  Port input data register: GPIOx_IDR,
  Port output data register: GPIOx_ODR
- **A 32-bit set/resetregister**
  Port bit set/reset register: GPIOx_BSRR
- **A 16-bit reset register**
  Port bit reset register: GPIOx_BRR
- **A 32-bit locking register**
  Port configuration lock register: GPIOx_LCKR)
  *x=A to G port

CMSIS Functions used in the project:

- **int32_t LED_Initialize (void); Initialize LEDs, configure pins as outputs**

```
int32_t LED_Initialize (void) {
  uint32_t n;

  /* Configure pins: Push-pull Output Mode (50 MHz) with Pull-down resistors */
  for (n = 0; n < LED_COUNT; n++) {
    GPIO_PortClock   (Pin_LED[n].port, true);
    GPIO_PinWrite    (Pin_LED[n].port, Pin_LED[n].num, 0);
    GPIO_PinConfigure(Pin_LED[n].port, Pin_LED[n].num,
                      GPIO_OUT_PUSH_PULL,
                      GPIO_MODE_OUT2MHZ);
  }

return 0; }
```

- **int32_t LED_Uninitialize (void); De-initialize LEDs, configure pins as inputs**

```
int32_t LED_Uninitialize (void) {
  uint32_t n;

  /* Configure pins: Input mode, without Pull-up/down resistors */
  for (n = 0; n < LED_COUNT; n++) {
```

```
    GPIO_PinConfigure(Pin_LED[n].port, Pin_LED[n].num,
                      GPIO_IN_FLOATING,
                      GPIO_MODE_INPUT);
  }
  return 0;
}
```

- **int32_t LED_On (uint32_t num); turn on specified LED**

```
int32_t LED_On (uint32_t num) {
  int32_t retCode = 0;

  if (num < LED_COUNT) {
    GPIO_PinWrite(Pin_LED[num].port, Pin_LED[num].num, 1);
  }
  else {
    retCode = -1;
  }
  return retCode;
  }
```

- **int32_t LED_Off (uint32_t num); turn of specified LED**

```
int32_t LED_Off (uint32_t num) {
  int32_t retCode = 0;

  if (num < LED_COUNT) {
    GPIO_PinWrite(Pin_LED[num].port, Pin_LED[num].num, 0);
  }
  else {
    retCode = -1;
  }
  return retCode;
}
```

- **int32_t LED_SetOut   (uint32_t val); write value to LEDs**

```
int32_t LED_SetOut (uint32_t val) {
  uint32_t n;

  for (n = 0; n < LED_COUNT; n++) {
    if (val & (1<<n)) {
      LED_On (n);
    } else {
      LED_Off(n);
    }
  }
 return 0;
}
```

Sample program: Program to turn ON/OFF LED connected to PORTE pin 8

```
#include "STM32F10x.h"

extern int32_t  LED_Initialize   (void);
extern int32_t  LED_Uninitialize (void);
extern int32_t  LED_On           (uint32_t num);
extern int32_t  LED_Off          (uint32_t num);
extern int32_t  LED_SetOut       (uint32_t val);

int main(void)
{
 LED_Initialize();
        while(1)
        {
        LED_On(0);
        LED_Off(0);
          }
}
```

Simulation Steps: (Using Keil µVision 5 IDE)

Create new project by clicking **Project→ New project,** select the device STM32F10C38 as shown in Fig. 3.1

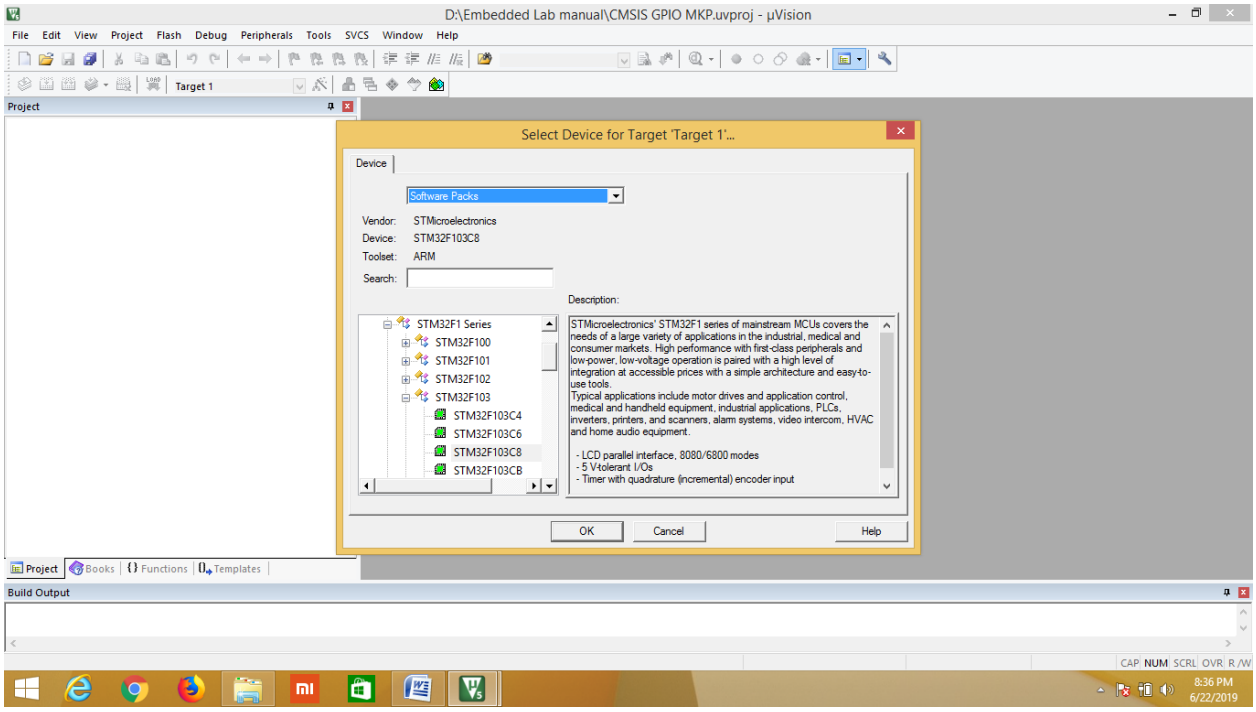


Fig. 3.1 Selection of microcontroller STM32F10C38

Select board support package MCBSTM32C and CMSIS core along with required peripheral APIs as shown in fig.3.2

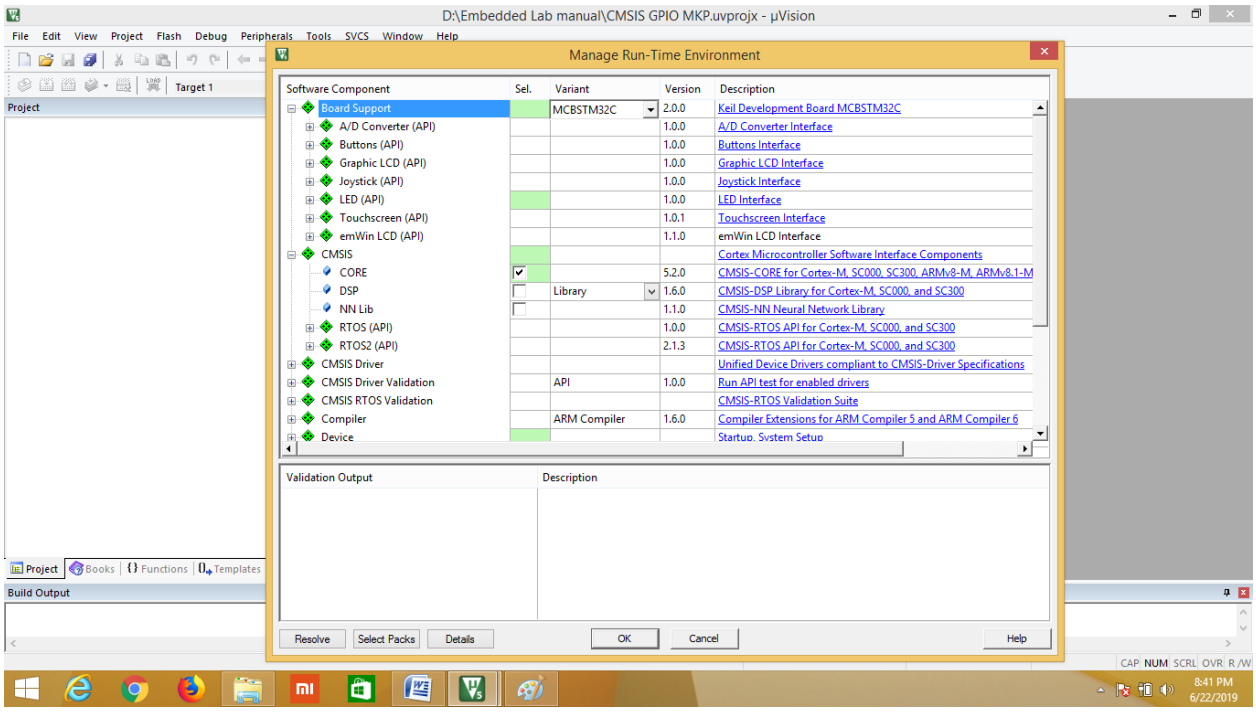


Fig.3.2 Selection of CMSIS core and board support package

Add new empty C file in the project and type the sample program in the C file and then build the project as shown in fig 3.3

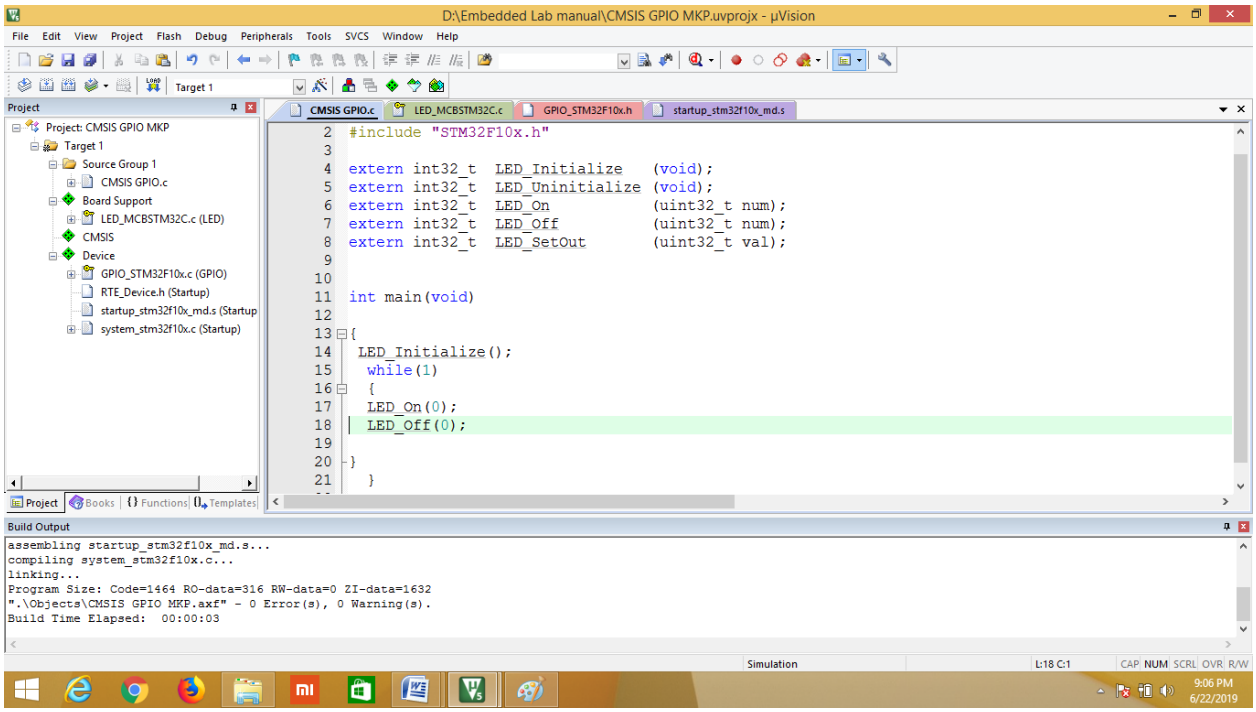


Fig 3.3 Adding C program file and building the project

Start the debug session and debug up to LED_On() function as shown in Fig 3.4

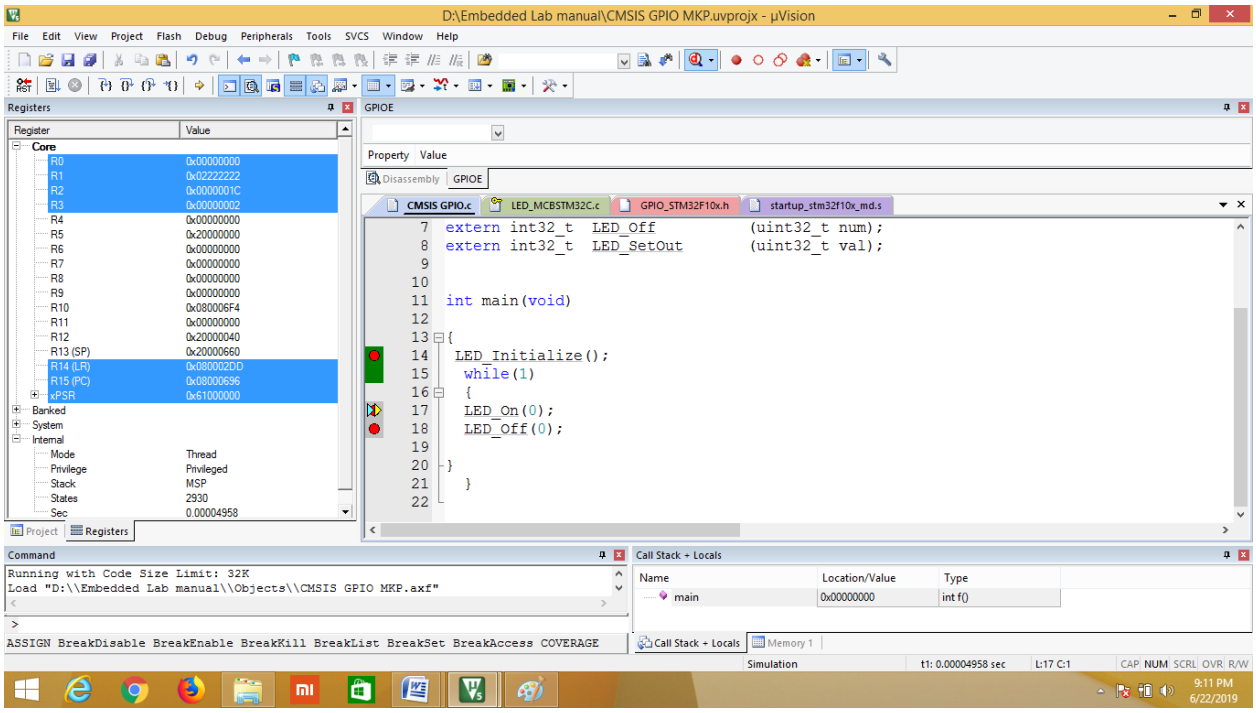


Fig.3.4 Debugging the program

Now single step the LED_On() function, the execution will go to the definition of the function as shown in Fig. 3.5

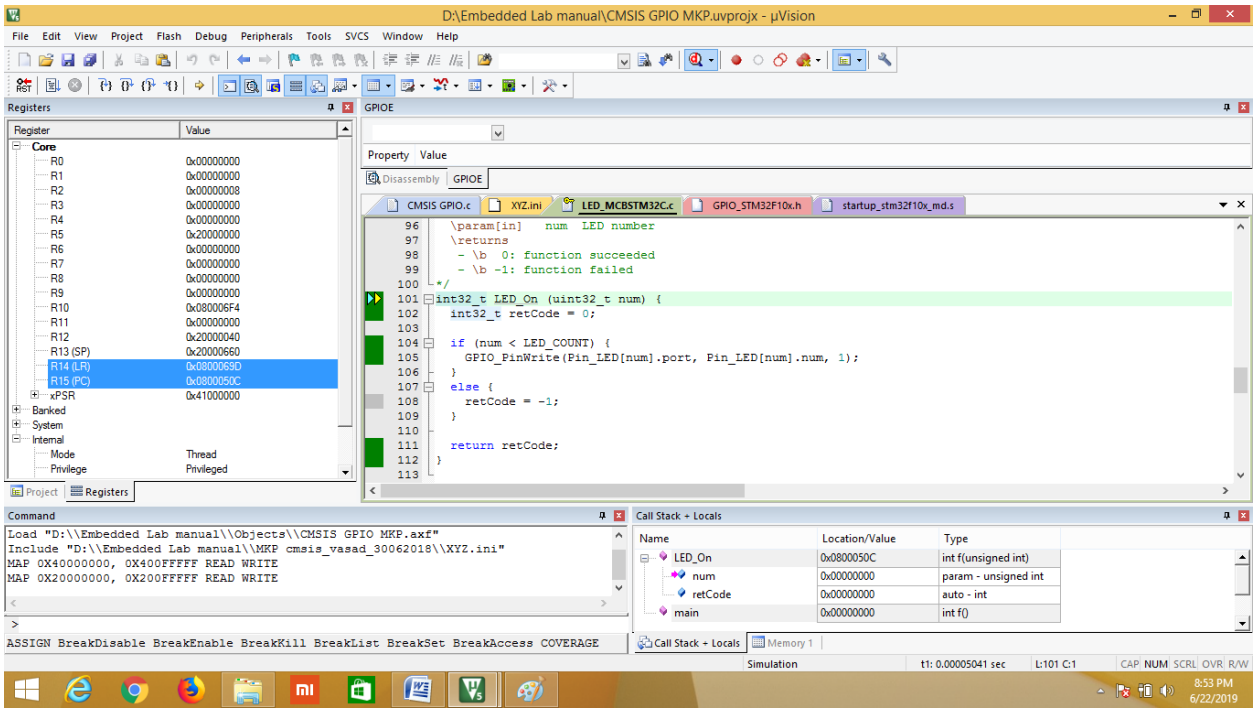


Fig. 3.5 Single stepping the LED_On() function

When the function is completely executed, it will turn ON LED connected to PORTE pin 8 as shown in Fig 3.6

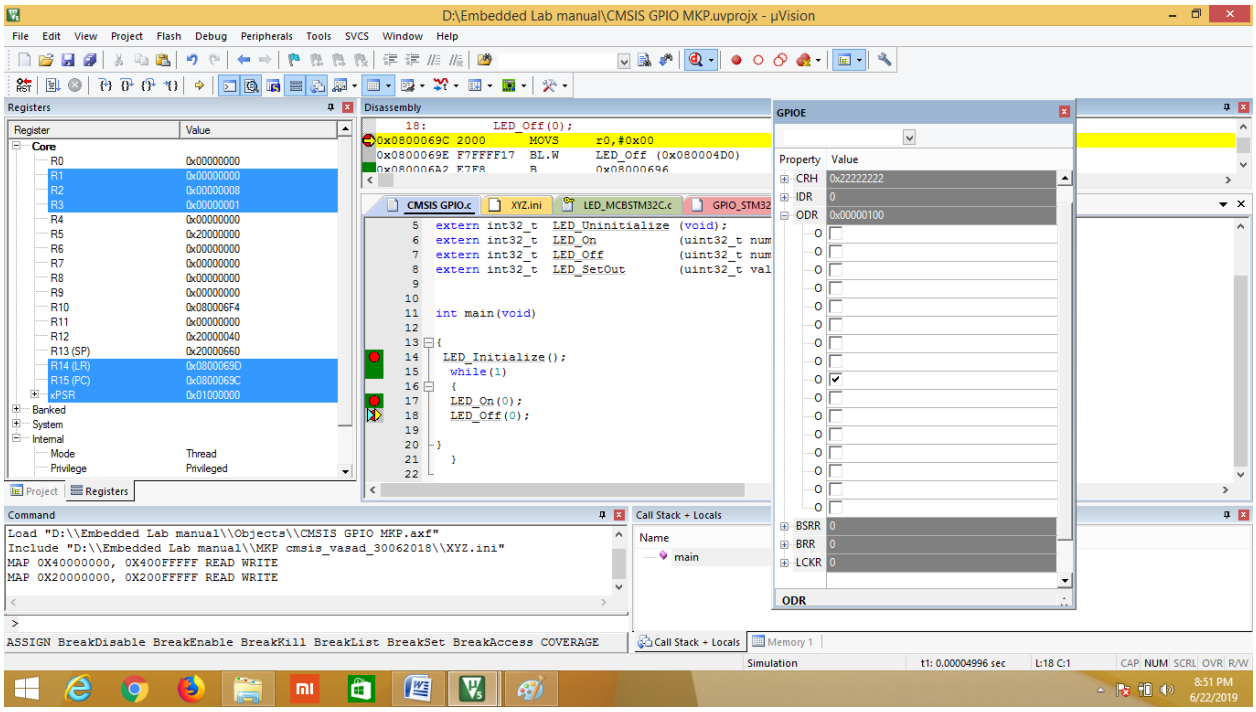


Fig 3.6 Output of LED_On() function

Similarly execute LED_Off() function and observe that the LED connected to PORTE pin 8 is turned OFF as shown in Fig. 3.7

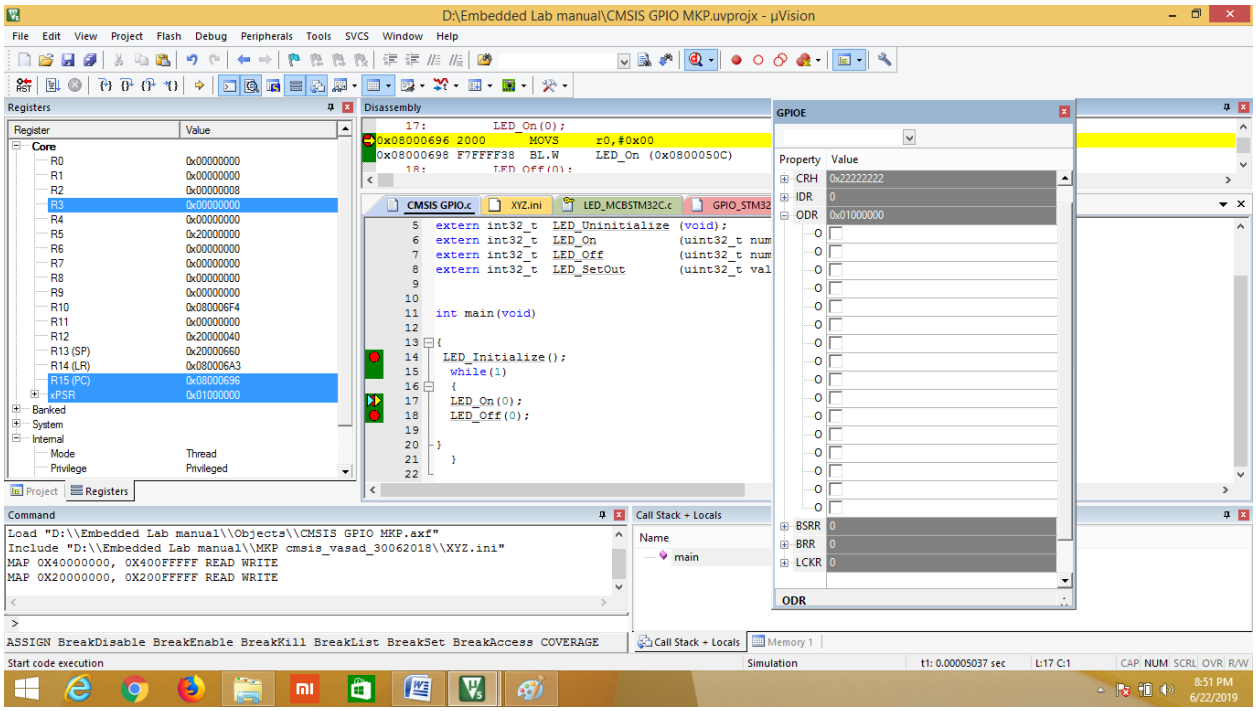


Fig 3.7 Output of LED_Off() function

Assignment:

1. Call LED_On and LED_Off fuction with 2 as parameter, which Port pins are affected? How?
2. Modify the project to turn ON/OFF the LED connected to PORTC pin 13
3. Develop similar project for the microcontroller of other family, for ex. STM32429ZITx and observe which port pins are modified if same sample program is used.
4 Locate all the functions involved in the project
5. Modify above project to send binary number to the port E
6. List and briefly explain the GPIO SFRs involved in the project

Conclusion:

_____
_____
_____
_____
_____

LAB 4 INTER INTEGRATED CIRCUIT BUS PROTOCOL

Aim: To study about I2C (Inter-Integrated Circuit) bus protocol and its programming

Theory:

Features:

• Multimaster capability
• Generation and detection of 7-bit/10-bit addressing and General Call
• Supports different communication speeds:
  – Standard Speed (up to 100 kHz)
  – Fast Speed (up to 400 kHz)
• Status flags:
  – Transmitter/Receiver mode flag
  - End-of-Byte transmission flag
  – I2C busy flag

I2C overview:

Communication flow

In Master mode, the I2C interface initiates a data transfer and generates the clock signal as shown in Fig. 4.1. A serial data transfer always begins with a start condition and ends with a stop condition as shown in following Fig. 4.1. Both start and stop conditions are generated in master mode by software. Data and addresses are transferred as 8-bit bytes, MSB first. The first byte(s) following the start condition contain the address (one in 7-bit mode, two in 10-bit mode). The address is always transmitted in Master mode. A 9th clock pulse follows the 8 clock cycles of a byte transfer, during which the receiver must send an acknowledge bit to the transmitter. Acknowledge may be enabled or disabled by software.

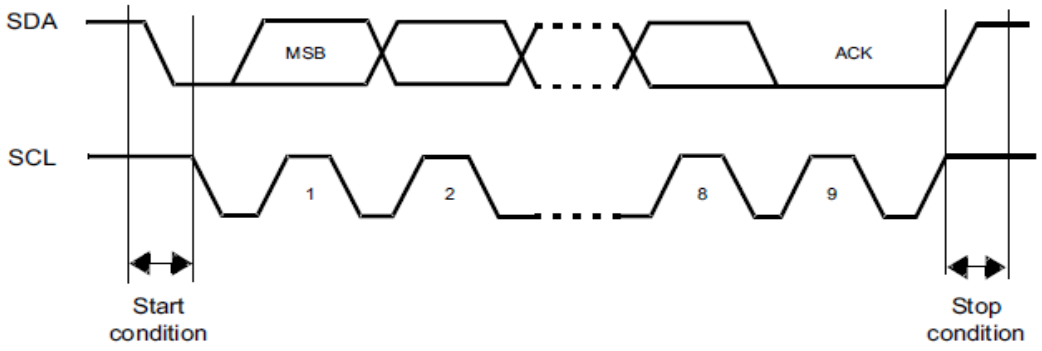
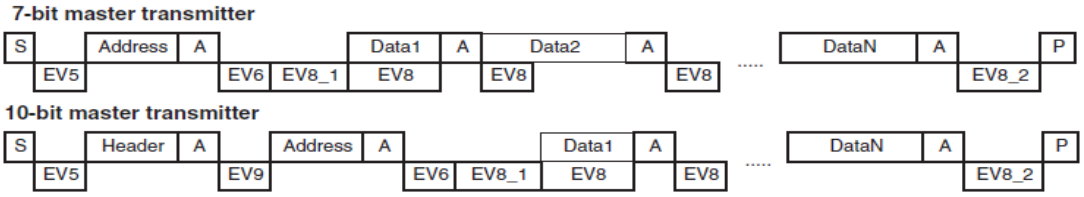


Fig. 4. 1 Communication flow

I2C master transmitter mode

Transfer sequence diagram for master transmitter mode is as shown in Fig. 4. 2



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with Address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV8: TxE=1, shift register not empty,.data register empty, cleared by writing DR register
EV8_2: TxE=1, BTF = 1, Program Stop request. TxE and BTF are cleared by hardware by the Stop condition
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

lotes: 1- The EV5, EV6, EV9, EV8_1 and EV8_2 events stretch SCL low until the end of the corresponding software sequence.
2- The EV8 software sequence must complete before the end of the current byte transfer. In case EV8 software sequence can not be managed before the current byte end of transfer, it is recommended to use BTF instead of TXE with the drawback of slowing the communication.

Fig. 4. 2 Transfer sequence diagram for master transmitter

Various I2C registers:

1. I2C Control Register 2 I2C_CR2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			LAST	DMA EN	ITBUF EN	ITEVT EN	ITERR EN	Reserved		FREQ[5:0]					
			rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

Bit 12	LAST: DMA last transfer.
Bit 11	DMAEN: DMA requests enable
Bit 10	ITBUFEN: Buffer interrupt enable
Bit 9	ITEVTEN: Event interrupt enable
Bit 8	ITERREN: Error interrupt enable
Bits 5:0	FREQ[5:0]: Peripheral clock frequency in MHz.

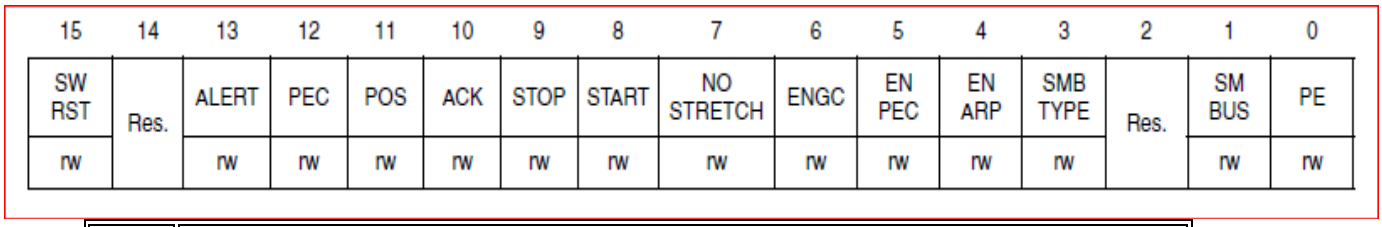
2. I2C Clock Control Register I2C_CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	Reserved		CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15	F/S:	I2C master mode selection. 0: Standard Mode I2C. 1: Fast Mode I2C
Bit 14	DUTY:	Fast mode duty cycle.
Bits	CCR[11:0]:	Clock control register in Fast/Standard mode (Master mode) Controls the

11:0		SCL clock in master mode. Standard mode or SMBus: Thigh = CCR * TPCLK1. Tlow = CCR * TPCLK1 See manual for Fast Mode

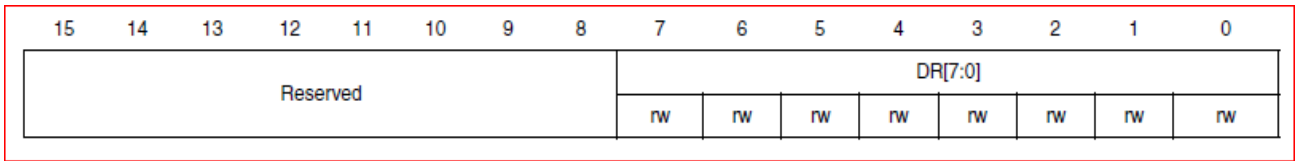
3. I2C Control Register I2C_CR1



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SW RST	Res.	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	EN PEC	EN ARP	SMB TYPE	Res.	SM BUS	PE
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

Bit 15	SWRST: 0: I2C Peripheral not under reset. 1: I2C Peripheral under reset state
Bit 13	ALERT: SMBus alert
Bit 12	PEC: Packet error checking.
Bit 11	POS: Acknowledge/PEC Position (for data reception)
Bit 10	ACK: Acknowledge enable. 0: No acknowledge returned
Bit 9	STOP: Stop generation
Bit 8	START: Start generation
Bit 7	NOSTRETCH: 0: Clock stretching enabled. 1: Clock stretching disabled
Bit 6	ENGC: General call enable.
Bit 5	ENPEC: PEC enable. 0: PEC calculation disabled. 1: PEC calculation enabled
Bit 4	ENARP: ARP enable.
Bit 3	SMBTYPE: SMBus type, 0: SMBus Device. 1: SMBus Host
Bit 1	SMBUS: SMBus mode, 0: I2C mode. 1: SMBus mode
Bit 0	PE: Peripheral enable. 0: Peripheral disable. 1: Peripheral enable.

4. I2C Data Register I2C_DR



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								DR[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:8	Reserved	forced by hardware to 0.	
Bits 7:0	DR[7:0]	8-bit data register Byte received or to be transmitted to the bus.	
		Transmitter mode:	Byte transmission starts automatically when a byte is written in the DR register. A continuous transmit stream can be maintained if the next data is put in DR once the transmission is started (TxE=1)

		Receiver mode:	Received byte is copied into DR (RxNE=1). A continuous transmit stream can be maintained if DR is read before the next data byte is received (RxNE=1).

5. I2C Status Register1 I2C_SR1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMB ALERT	TIME OUT	Res.	PEC ERR	OVR	AF	ARLO	BERR	TxE	RxNE	Res.	STOP F	ADD10	BTF	ADDR	SB
rc_w0	rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	r	r		r	r	r	r	r

Bit 15	SMBALERT:	SMBus alert
Bit 14	TIMEOUT:	Timeout or Tlow error
Bit 12	PECERR:	PEC Error in reception
Bit 11	OVR:	Overrun/Underrun
Bit 10	AF:	Acknowledge failure
Bit 9	ARLO:	Arbitration lost (master mode)
Bit 8	BERR:	Bus error
Bit 7	TxE:	Data register empty (transmitters)
Bit 6	RxNE:	Data register not empty (receivers)
Bit 4	STOPF:	Stop detection (slave mode)
Bit 3	ADD10:	10-bit header sent (Master mode)
Bit 2	BTF:	Byte transfer finished
Bit 1	ADDR:	Address sent (master mode)/matched (slave mode)
Bit 0	SB:	Start bit (Master mode)

6. I2C Status Register 2 I2C_SR2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PEC[7:0]								DUALF	SMB HOST	SMB DEF AULT	GEN CALL	Res.	TRA	BUSY	MSL
r	r	r	r	r	r	r	r	r	r	r	r		r	r	r

Bits 15:8	PEC[7:0]	Packet error checking register -contains the internal PEC when ENPEC=1.
Bit 7	DUALF:	Dual flag (Slave mode)
Bit 6	SMBHOST:	SMBus host header (Slave mode)
Bit 5	SMBDEFAULT:	SMBus device default address (Slave mode)

Bit 4	GENCALL:	General call address (Slave mode)
Bit 2	TRA:	Transmitter/receiver
Bit 1	BUSY:	Bus busy
Bit 0	MSL:	Master/slave. 0: Slave Mode. 1: Master Mode

SAMPEL CODE: (1) I2C Master Transmitter Mode

```
#include<stm32f10x.h>
char res;
void I2C_Init()
{
RCC->APB1ENR |= (1<<21);
I2C1->CR2=0x0008;
I2C1->CCR=0x0028;
I2C1->CR1=0x0001;
}
void I2C_Start()
{
I2C1->CR1|=1<<8;
I2C1->CR1|=1<<10;
while(!(I2C1->SR1&0x0001));
}
void I2C_Addr(unsigned char adr)
{
I2C1->DR=adr|0;
while(!(I2C1->SR1&0x0002));
res=(I2C1->SR2);
}
void I2C_Write()
        {
I2C1->DR=0xAA;
while(!(I2C1->SR1&(1<<7)));
}
void I2C_Stop()
{
I2C1->CR1|=0x0200;
while(I2C1->SR2&0x0002);
}
int main()
{
I2C_Init();
I2C_Start();
I2C_Addr(0x70);
I2C_Write();
I2C_Stop();
}
```

OUTPUT:

1. Step run the code of I2C_Start routine.. Open I2C interface window from Peripherals on toolbar to observe the start condition as shown in Fig. 4.3 after debugging statements of Start routine.

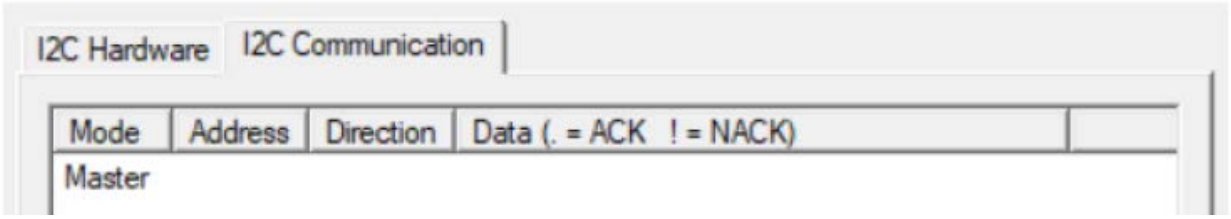


Fig. 4.3 STRAT condition on the I2C interface

2. Step run the code of I2C_Addr routine. The I2C communication tab will show the following result after entering the while loop in the address () function .

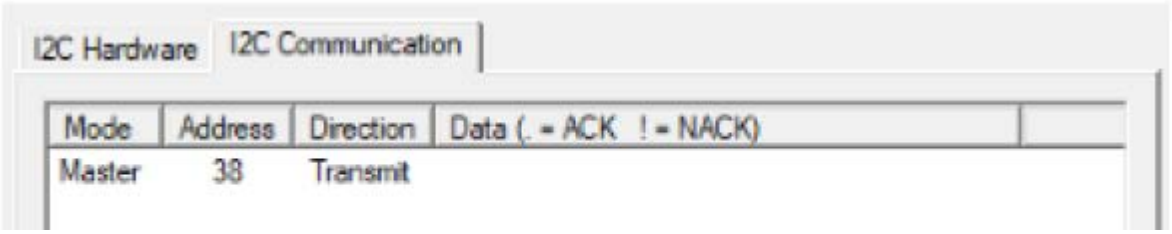


Fig. 4.4 Master is transmitting

3. Acknowledgement is to be provided from Command console using a virtual register.
Enter in command window:
I2C1_IN = 0xFF00 (I2C1_IN is a virtual register)

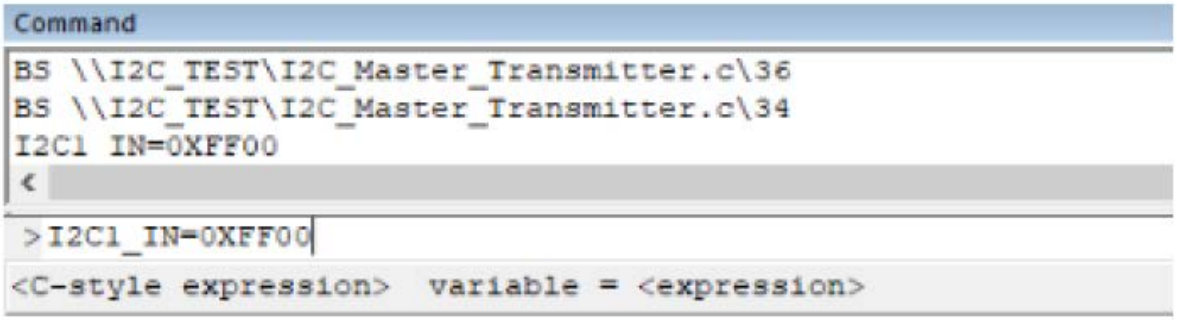


Fig 4.5 I2C Acknowledgement using simulator command

4. After providing the acknowledgement step run the program with single step till the acknowledgement is received (check for '.' after 38 in Address column).

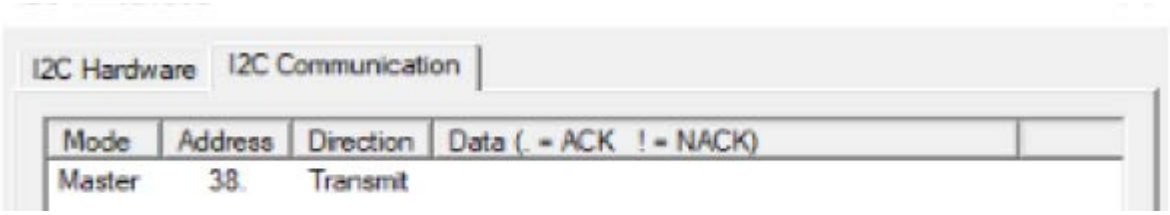


Fig 4.6 Slave has sent the acknowledge to the Master

5. Step run the code of I2C_Write routine. It can be seen from following Fig. 4.7 that data register I2C1_DR is updated with data 0x00AA

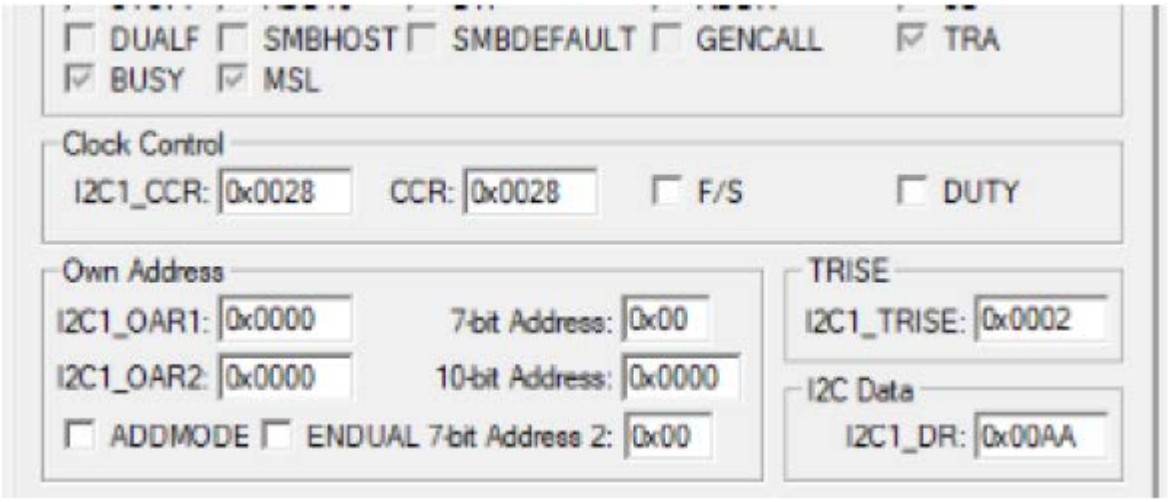


Fig. 4.7 Data Register updated

6. Acknowledgement is to be provided from Command console using a virtual register (As discussed above).

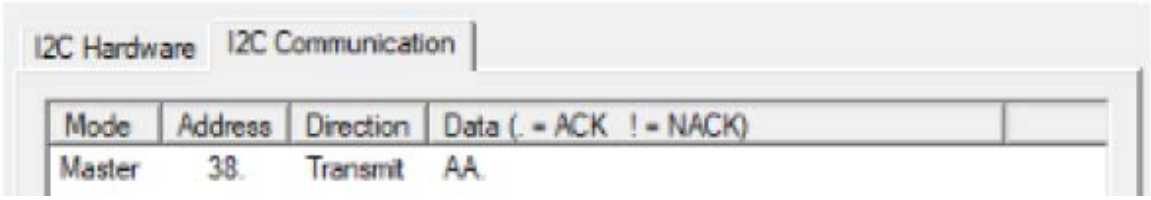


Fig. 4.8 ACK received

7. Stop bit is set (i.e. data is transfer successfully)

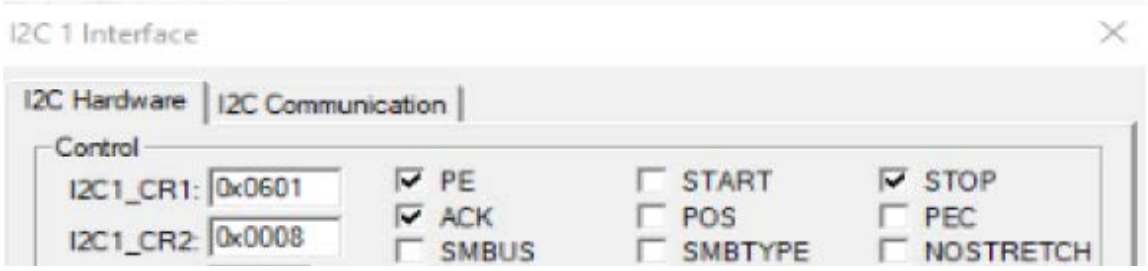


Fig. 4.9 STOP bit is set

Sample code : (2) Slave as transmitter and Master as receiver.

```
#include <stm32f10x.h>
#include <stdio.h>
void Initialize(void)
{
I2C1 -> CR2 = 0x08; // Set freq to 8MHz
I2C1 -> CCR = 0x28; // set clock freq
I2C1 -> CR1 = 0x0401;
// enable pheripheral and acknowledgement bit[0] and bit[10] resp.
}
void Start(void)
{
I2C1 -> CR1 |= 0x0100; // start generation bit[8]
while(!(I2C1 -> SR1 & 0x0001)); // confirming start bit generation (checking bit[0])
}
void Address(char x)
{
I2C1 -> DR = x | 0; // address kept in DATA REGISTER
while(!(I2C1 -> SR1 & 0x0002)); // confirming address sent (check bit[1])
I2C1 -> SR1 = ((I2C1 -> SR1) & 0xFFFD); // clearing ADDR bit for master receiver mode (clear
bit[1])
}

void Data()
{
char d ; //I2C1 -> DR = 0x00;
while(!(I2C1 -> SR1 & 0x40));
// Wait for data transfer to complete (checking bit[1])
}
void Stop(void)
{
I2C1 -> CR1 |= 0x0200; // Set stop bit, bit[9]
while(I2C1 -> SR2 & 0x0002);
// confirming stop bit generation (checking bit[1])
}
int main(void)
{
Initialize();
Start();
Address(0x71);
Data();
Stop();
return 0;
}
```

Output:

1. Step run the code of Start routine.. Open I2C interface window from Peripherals on toolbar to observe the start condition as shown in Fig. 4.3 after debugging statements of Start routine.

2. Step run the code ofAddress routine. Acknowledgement is to be provided from Command console using a virtual register as discussed above.

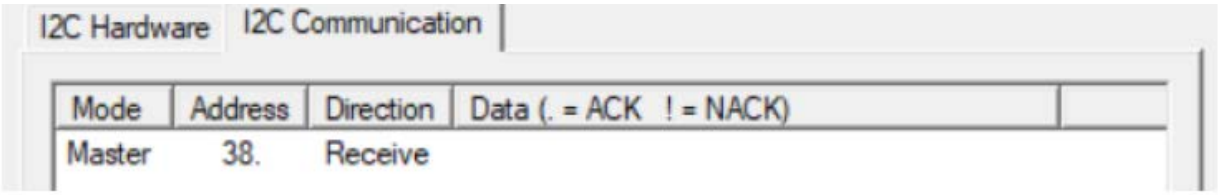


Fig. 4.10 Slave has sent the acknowledge to the Master

3. I2C data input is provided using a virtual register using command:

I2C1_IN = 0x11

4. Step run the program till the data is received successfully.

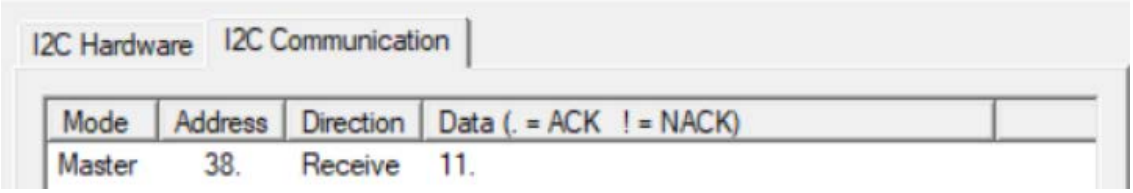


Fig. 4.11 Master has received the data successfully

Assingment:

1. Modify the above sample code by using various I2C registers of LPC 2378 kit.

Conclusion:

_____
_____
_____
_____

LAB 5 SPI PROTOCOL

Aim: To establish connection and transfer data using SPI protocol for STM32F10x in Keil µVision 5

Theory:

The SPI communication stands for serial peripheral interface communication protocol, which was developed by the Motorola in 1972. SPI interface is available on popular communication controllers such as PIC, AVR, and ARM controller, etc. It has synchronous serial communication data link that operates in full duplex, which means the data signals carry on both the directions simultaneously.

SPI protocol consists of four wires such as MISO, MOSI, CLK, SS used for master/slave communication. The master is a microcontroller, and the slaves are other peripherals like sensors, GSM modem and GPS modem, etc. The multiple slaves are interfaced to the master through a SPI serial bus. The SPI protocol does not support the Multi-master communication and it is used for a short distance within a circuit board.

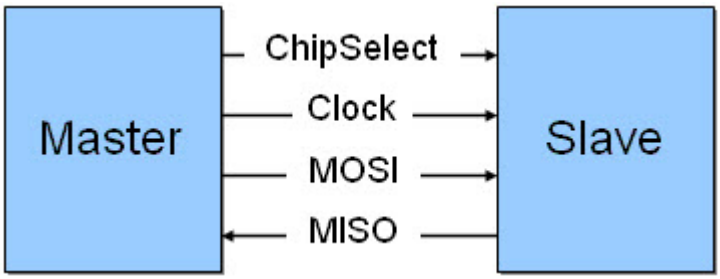


Fig 5.1 Block Diagram of SPI Protocol

SPI Lines:

MISO (Master in Slave out): The MISO line is configured as an input in a master device and as an output in a slave device.
MOSI (Master out Slave in): The MOSI is a line configured as an output in a master device and as an input in a slave device wherein it is used to synchronize the data movement.
SCK (serial clock): This signal is always driven by the master for synchronous data transfer between the master and the slave. It is used to synchronize the data movement both in and out through the MOSI and MISO lines.
SS (Slave Select) and CS (Chip Select): This signal is driven by the master to select individual slaves/Peripheral devices. It is an input line used to select the slave devices.

SPI features

• Full-duplex synchronous transfers on three lines
• Simplex synchronous transfers on two lines with or without a bidirectional data line

• 8- or 16-bit transfer frame format selection
• Master or slave operation
• Multimaster mode capability
• 8 master mode baud rate prescalers (fPCLK/2 max.)
• Slave mode frequency (fPCLK/2 max)
• Faster communication for both master and slave
• NSS management by hardware or software for both master and slave: dynamic change of master/slave operations
• Programmable clock polarity and phase
• Programmable data order with MSB-first or LSB-first shifting
• Dedicated transmission and reception flags with interrupt capability
• SPI bus busy status flag
• Hardware CRC feature for reliable communication:
– CRC value can be transmitted as last byte in Tx mode
– Automatic CRC error checking for last received byte
• Master mode fault, overrun and CRC error flags with interrupt capability
• 1-byte transmission and reception buffer with DMA capability: Tx and Rx requests

Master operation:
In the master configuration, the serial clock is generated on the SCK pin.

Procedure
1. Select the BR[2:0] bits to define the serial clock baud rate (see SPI_CR1 register).
2. Select the CPOL and CPHA bits to define one of the four relationships between the data transfer and the serial clock.
3. Set the DFF bit to define 8- or 16-bit data frame format
4. Configure the LSBFIRST bit in the SPI_CR1 register to define the frame format.
5. If the NSS pin is required in input mode, in hardware mode, connect the NSS pin to a high-level signal during the complete byte transmit sequence. In NSS software mode, set the SSM and SSI bits in the SPI_CR1 register. If the NSS pin is required in output mode, the SSOE bit only should be set.
6. The MSTR and SPE bits must be set (they remain set only if the NSS pin is connected to a high-level signal). In this configuration the MOSI pin is a data output and the MISO pin is a data input.

Slave operation:

In the slave configuration, the serial clock is received on the SCK pin from the master device. The value set in the BR[2:0] bits in the SPI_CR1 register, does not affect the data transfer rate.

Procedure
1. Set the DFF bit to define 8- or 16-bit data frame format
2. Select the CPOL and CPHA bits to define one of the four relationships between the data transfer and the serial clock. For correct data transfer, the CPOL and CPHA bits must be configured in the same way in the slave device and the master device.

3. The frame format (MSB-first or LSB-first depending on the value of the LSBFIRST bit in the SPI_CR1 register) must be the same as the master device.
4. In Hardware mode, the NSS pin must be connected to a low level signal during the complete byte transmit sequence. In NSS software mode, set the SSM bit and clear the SSI bit in the SPI_CR1 register.
5. Clear the MSTR bit and set the SPE bit (both in the SPI_CR1 register) to assign the pins to alternate functions. In this configuration the MOSI pin is a data input and the MISO pin is a data output.

Register Map:
The table provides shows the SPI register map and reset values.

Table 5.1: SPI Register Map and Reset Values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	SPI_CR1	Reserved																BIDIMODE	BIDIOE	CRCEN	CRCNEXT	DFF	RXONLY	SSM	SSI	*LSBFIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	SPI_CR2	Reserved																								TXEIE	RXNEIE	ERRIE	Reserved		*SSOE*	TXDMAEN	RXDMAEN
	Reset value																									0	0	0			0	0	0
0x08	SPI_SR	Reserved																								BSY	OVR	MODF	CRCERR	UDR	CHSIDE	TXE	RXNE
	Reset value																									0	0	0	0	0	0	1	0
0x0C	SPI_DR	Reserved																DR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	SPI_CRCPR	Reserved																CRCPOLY[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0x14	SPI_RXCRCR	Reserved																RxCRC[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	SPI_TXCRCR	Reserved																TxCRC[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	SPI_I2SCFGR	Reserved																				I2SMOD	I2SE	I2SCFG		PCMSYNC	Reserved	I2SSTD		CKPOL	DATLEN		CHLEN
	Reset value																					0	0	0	0	0		0	0	0	0	0	0
0x20	SPI_I2SPR	Reserved																						MCKOE	ODD	I2SDIV							
	Reset value																							0	0	0	0	0	0	0	0	1	0

SPI MASTER TRANSMITTER FOR STM32F10x in Keil µVision 5:

```
#include <stm32f10x.h>
#include <stdio.h>
```

```
void Initialize()
{
                SPI1->CR1 = 0X0044;                /* Device selected as master, SPI enable */
}

  int main (void) {
        int a=0;
        Initialize();

/* Do forever */
        while(1)
{
        a++;
        /* Write data out */
        SPI1->DR = a; //DATA;

/* Wait for transfer to be completed */
        while (!((SPI_I2S_FLAG_TXE)));

}
}
```

OUTPUT:

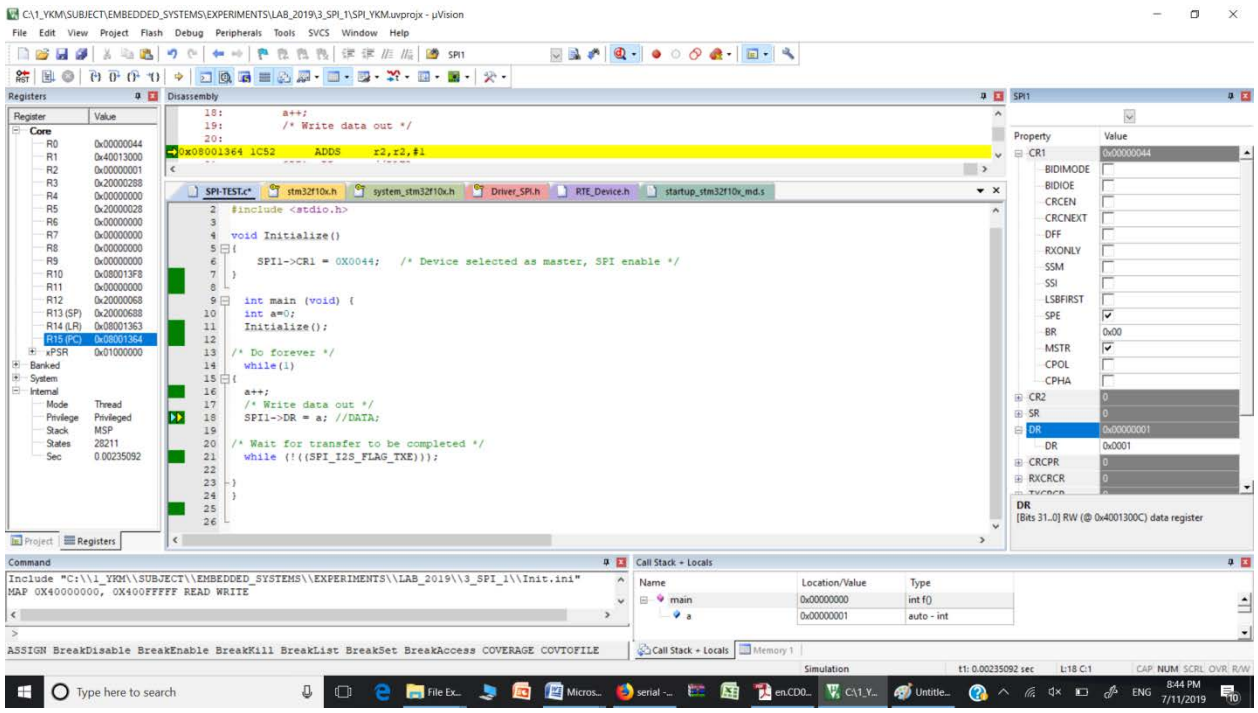


Fig 5.2 Output of SPI for STM32F10x in Keil µVision 5

Assignment:

1. Configure SPI in slave mode.

Conclusion:

LAB 6 INTRODUCTION TO FreeRTOS AND TASK CREATION

Aim: To perform task creation and task switching using scheduler.

Theory:

Free RTOS is designed to be small and simple. The kernel itself consists of only three C files. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembly functions included where needed (mostly in architecture-specific scheduler routines).

Free RTOS provides methods for multiple threads or tasks, mutexes, semaphores and software timers. A tick-less mode is provided for low power applications. Thread priorities are supported. Free RTOS applications can be completely statically allocated. Alternatively RTOS objects can be dynamically allocated with five schemes of memory allocation provided:

- allocate only;
- allocate and free with a very simple, fast, algorithm;
- a more complex but fast allocate and free algorithm with memory coalescence;
- an alternative to the more complex scheme that includes memory coalescence that allows a heap to be broken across multiple memory areas and C library allocate and free with some mutual exclusion protection.

xTaskCreate():-Each task requires RAM that is used to hold the task state (the task control block, or TCB),and used by the task as its stack. If a task is created using xTaskCreate() then the requiredRAM is automatically allocated from the FreeRTOS heap.

BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask);

pvTaskCode	Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function (in effect, just the function name) that implements the task.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used in a call to xTaskGetHandle() to obtain a task handle. The application-defined constant configMAX_TASK_NAME_LEN defines the maximum length of the name in characters – including the NULL terminator. Supplying a string longer than this maximum will result in the string being silently truncated.

usStackDepth	Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack. The value specifies the number of words the stack can hold, not the number of bytes. For example, on an architecture with a 4 byte stack width, if usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes). The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t. The size of the stack used by the idle task is defined by the application-defined constant configMINIMAL_STACK_SIZE. The value assigned to this constant in the demo application provided for the chosen microcontroller architecture is the minimum recommended for any task on that architecture. If your task uses a lot of stack space, then you must assign a larger value.
pvParameters	Task functions accept a parameter of type 'pointer to void' (void*). The value assigned to pvParameters will be the value passed into the task. This parameter has the type 'pointer to void' to allow the task parameter to effectively, and indirectly by means of casting, receive a parameter of any type. For example, integer types can be passed into a task function by casting the integer to a void pointer at the point the task is created, then by casting the void pointer parameter back to an integer in the task function definition itself.
uxPriority	Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority. configMAX_PRIORITIES is a user defined constant. If configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0 then there is no upper limit to the number of priorities that can be available (other than the limit of the data types used and the RAM available in your microcontroller), but it is advised to use the lowest number of priorities required, to avoid wasting RAM. Passing a uxPriority value above (configMAX_PRIORITIES – 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value.
pxCreatedTask	pxCreatedTask can be used to pass out a handle to the task

being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

vTaskStartScheduler():-Typically, before the scheduler has been started, main() (or a function called by main()) will beexecuting. After the scheduler has been started, only tasks and interrupts will ever execute.Starting the scheduler causes the highest priority task that was created while the schedulerwas in the Initialization state to enter the Running state.

vTaskDelay(): Places the task that calls vTaskDelay() into the Blocked state for a fixed number of tick interrupts. Specifying a delay period of zero ticks will not result in the calling task being placed into the Blocked state, but will result in the calling task yielding to any Ready state tasks that share its priority. Calling vTaskDelay(0) is equivalent to calling taskYIELD().

void vTaskDelay(TickType_t xTicksToDelay);

xTicksToDelay	The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state and remain in the Blocked state until the tick count reached 10,100.

1) **Two task creation:**

```
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"

/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";

/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);
```

```
int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock
*/
        SystemCoreClockUpdate();


        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                      /* This task will run at priority 1. */
        NULL );                 /* We are not using the task handle. */

      /* Create the other task in exactly the same way. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2,
1, NULL );
        /* Start the scheduler so our tasks start executing. */
        vTaskStartScheduler();
        /* If all is well we will never reach here as the scheduler will now be
        running.  If we do reach here then it is likely that there was insufficient
        heap available for the idle task to be created. */
        for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
```

```
                /* Delay for a period. */
            vTaskDelay( 100 );
            }
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
            /* Task is implemented in an infinite loop. */
            for( ;; )
            {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
                vTaskDelay( 100 );
            }
}
```

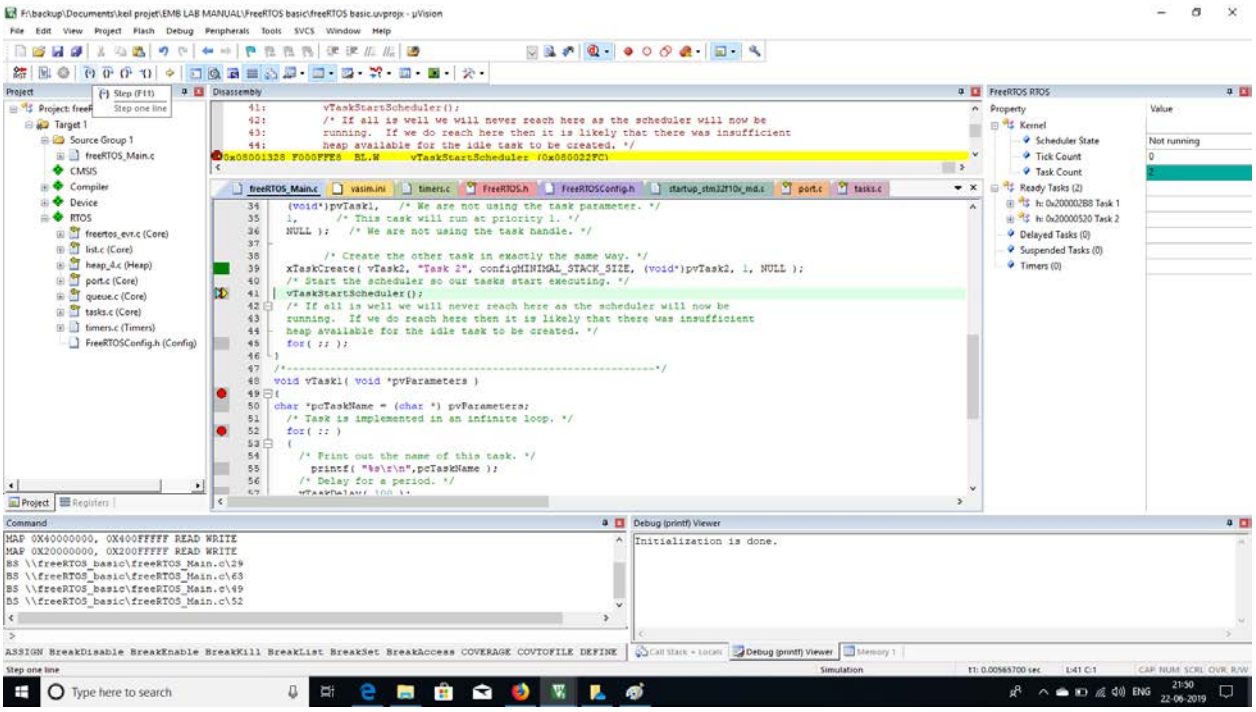


Fig 6.1 : Two task created

In this program two task (Task 1 and Task 2) are created using xTaskCreate (), both task are in ready state before Scheduler started.

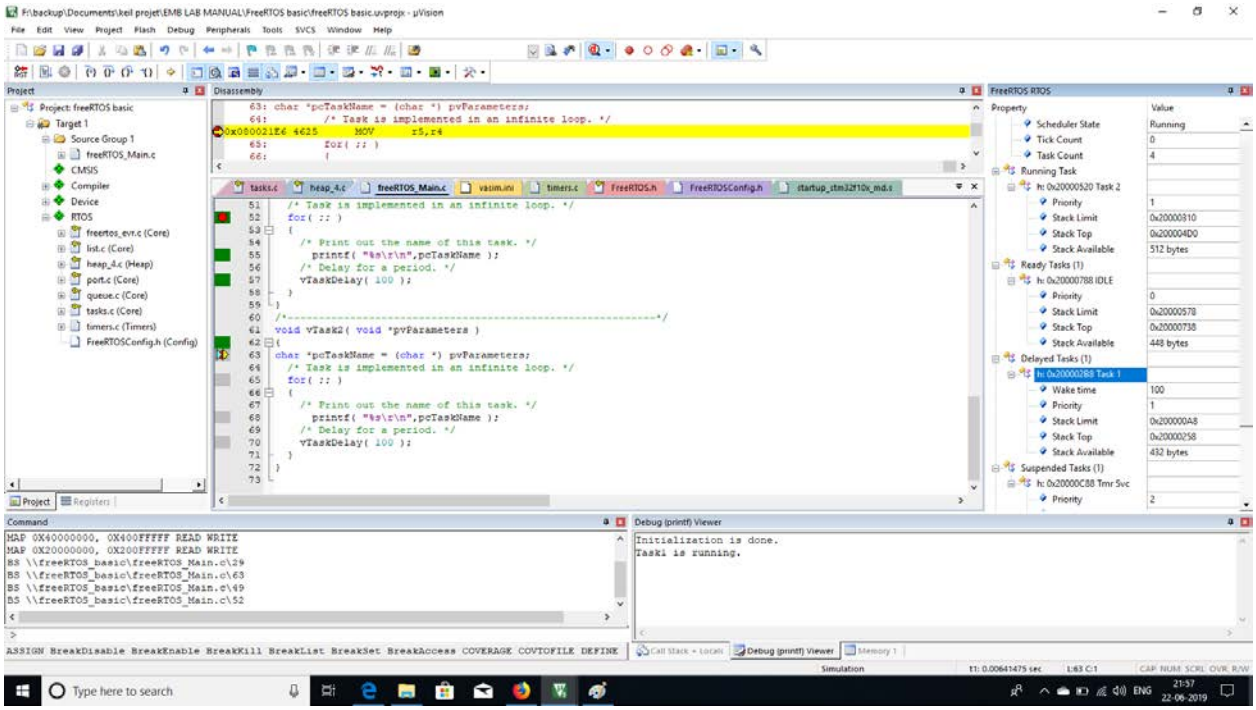


Fig 6.2: Four task created

after starting the scheduler there are four task(Task 1, Task 2, IDLE task and Tmr Svc task) as shown in above fig. FreeRTOS RTOS window.

Task 1 is already executed as shown in Debug(printf) window, due to vTaskDelay() function now Task2 is running , Task 1 is delayed (due to 100 tick delay provided) and IDLE task is always ready.

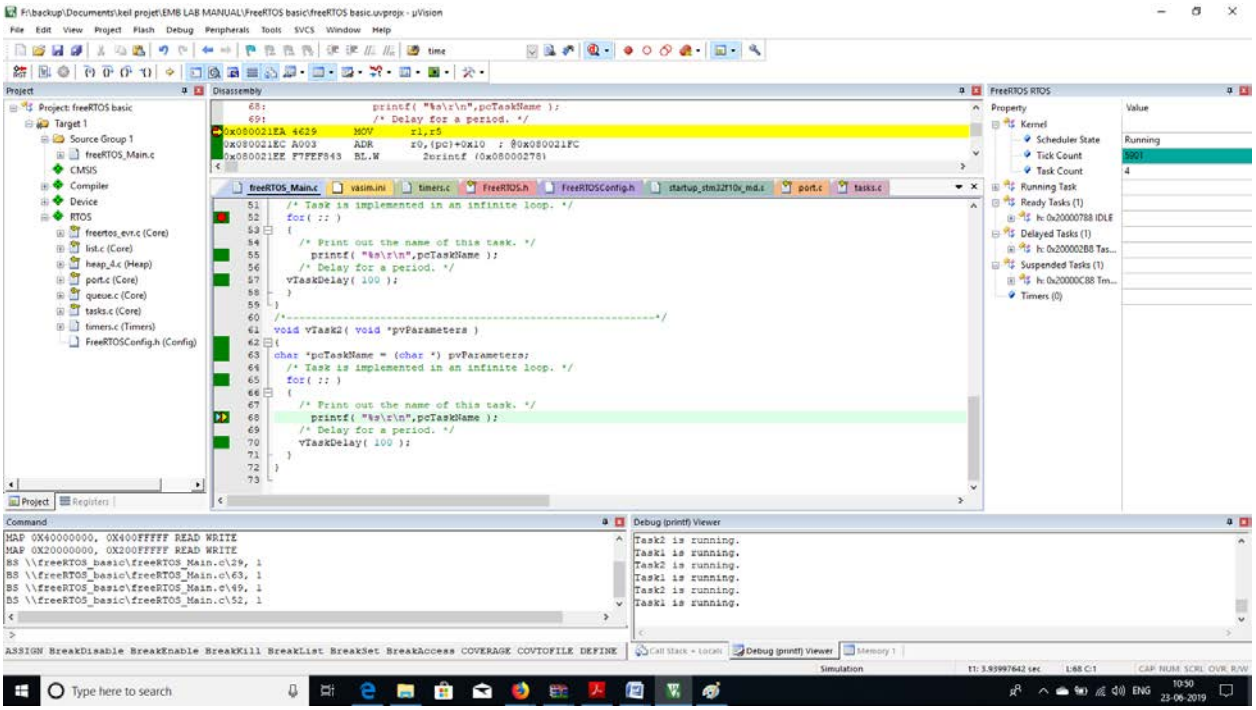


Fig 6.3: Task 1 and Task 2 are running

Assignment:

1) Set the task1 delay to 50 and task 2 delay to 100 and observe output.
2) Now change the priority of task and observe output.

2. Task creation from the task

```
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"

/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";

/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);

int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock */
        SystemCoreClockUpdate();


        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                          /* This task will run at priority 1. */
```

```
        NULL );                 /* We are not using the task handle. */

        /* Start the scheduler so our tasks start executing. */
        vTaskStartScheduler();
        /* If all is well we will never reach here as the scheduler will now be
        running.  If we do reach here then it is likely that there was insufficient
        heap available for the idle task to be created. */
        for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
         /* Create the other task from task 1. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );

        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
        vTaskDelay( 100 );



        }
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
                vTaskDelay( 100 );
```

```
        }
}
```

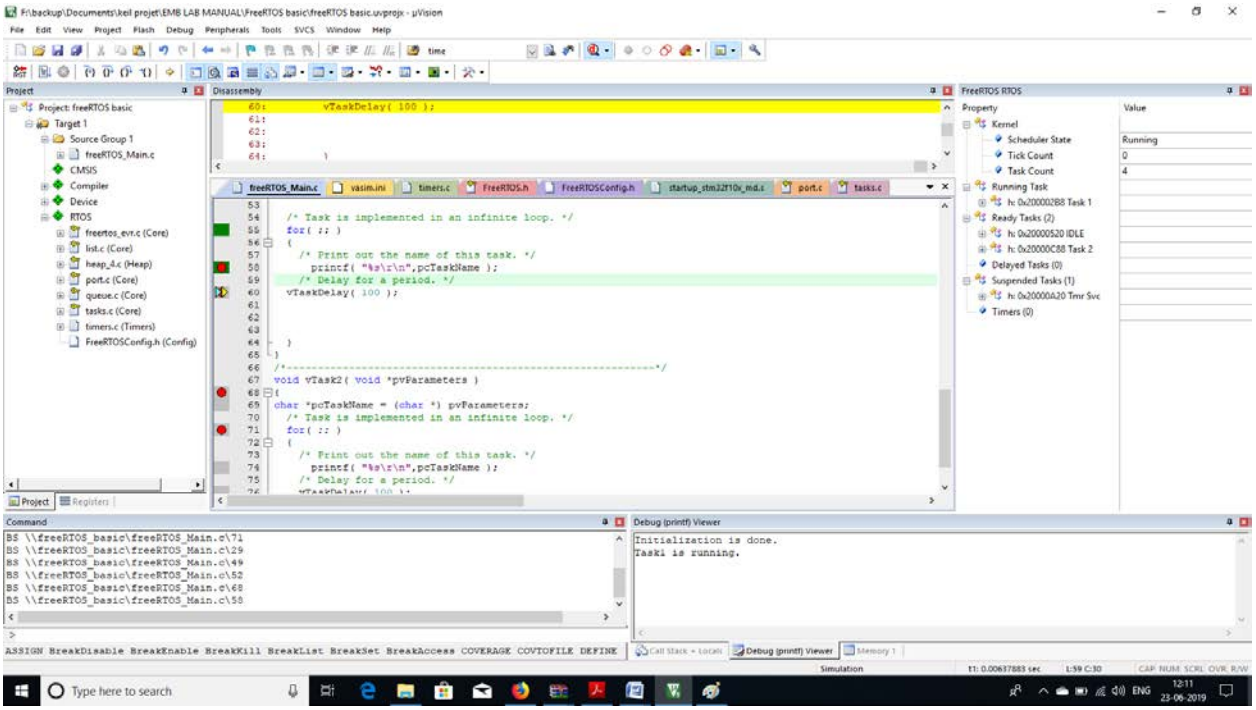


Fig 6.4: task 2 created from task 1

As shown in output after Task 2 is created from the Task 1, here Task 2 is ready to run while Task 1 is running.

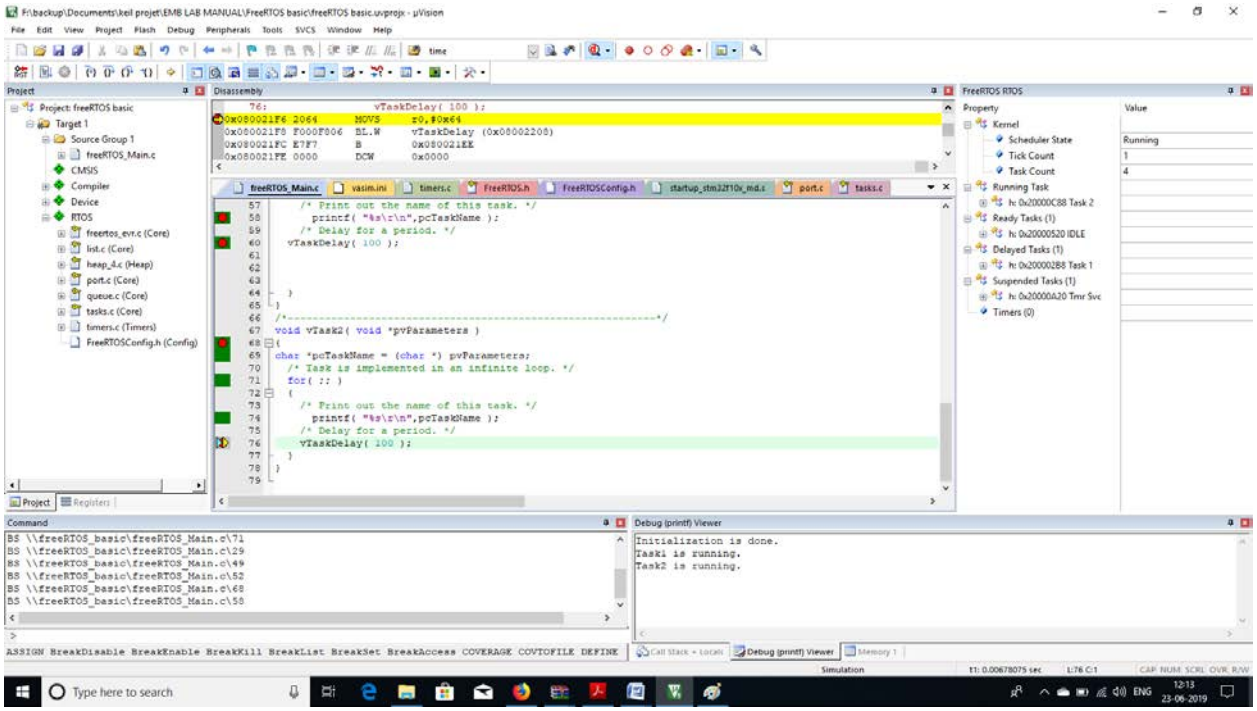


Fig 6.5: Task 2 is running and Task 1 is delayed

As shown in fig 6.5 now task 2 is running and task 1 is delayed due to the vTaskDelay() function.

3. creat task 2 from task 1 and then delete the task2

```
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"

/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";

/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);

int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock */
        SystemCoreClockUpdate();


        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                      /* This task will run at priority 1. */
        NULL );                 /* We are not using the task handle. */

      /* Create the other task in exactly the same way. */
```

```
//        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );
        /* Start the scheduler so our tasks start executing. */
        vTaskStartScheduler();
        /* If all is well we will never reach here as the scheduler will now be
        running.  If we do reach here then it is likely that there was insufficient
        heap available for the idle task to be created. */
        for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
         /* Create the other task from task 1. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );

        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
        vTaskDelay( 100 );
        }
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
                vTaskDelay( 100 );
                        printf( "TASK 2 IS ABOUT GET DELETED \n");
        vTaskDelete(NULL);
        }
```

}

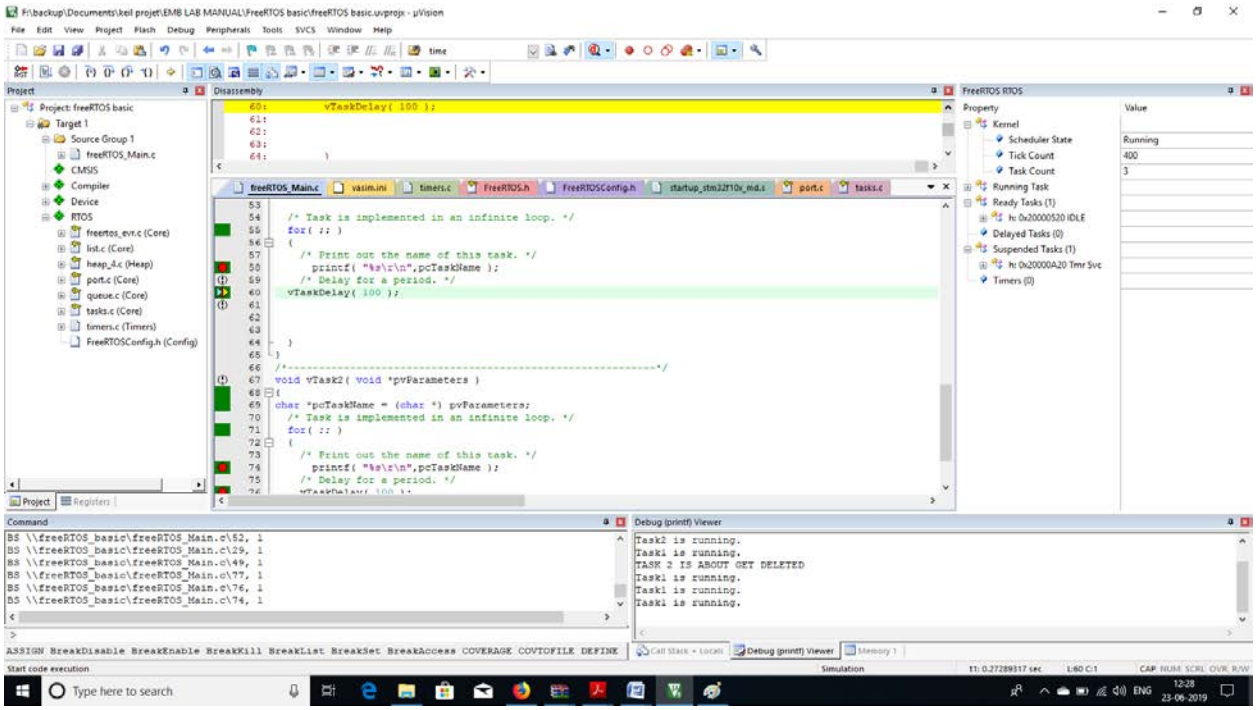


Fig 6.6: Task 2 created and deleted it self

As shown in output task 2 created from task 1 and after running one time task 2 deleted itself.

Assignment : Delete the Task 2 from Task 1 (hint: read vTaskDelete() function)

Solution:

```
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"

/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );
TaskHandle_t xTask2Handle = NULL;
/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";

/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);
```

```
int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock */
        SystemCoreClockUpdate();


        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                      /* This task will run at priority 1. */
        NULL );                 /* We are not using the task handle. */

      /* Create the other task in exactly the same way. */
//        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );
        /* Start the scheduler so our tasks start executing. */
        vTaskStartScheduler();
        /* If all is well we will never reach here as the scheduler will now be
        running.  If we do reach here then it is likely that there was insufficient
        heap available for the idle task to be created. */
        for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
         /* Create the other task from task 1. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
xTask2Handle );

        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
```

```
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
        vTaskDelay( 100 );
                        printf( "TASK 2 IS ABOUT GET DELETED \n");
        vTaskDelete(xTask2Handle);




        }
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
        /* Task is implemented in an infinite loop. */
        for( ;; )
        {
                /* Print out the name of this task. */
                printf( "%s\r\n",pcTaskName );
                /* Delay for a period. */
                vTaskDelay( 100 );

        }
}
```

Assignment

1. Set the task 1 delay to 50 and task 2 delay to 100.
2. Change the priority of tasks.

Conclusion

LAB 7 TASK SCHEDULING USING FreeRTOS

Aim: To study Task scheduling and pre-emption

Theory:

The task that is actually running (using processing time) is in the Running state. On a single core processor there can only be one task in the Running state at any given time.

Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state. Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state.

Tasks can wait in the Blocked state for an event and are automatically moved back to the Ready state when the event occurs. Temporal events occur at a particular time, for example, when a block time expires, and are normally used to implement periodic or timeout behavior. Synchronization events occur when a task or interrupt service routine sends information using a task notification, queue, event group, or one of the many types of semaphore. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

Configuring the Scheduling Algorithm

The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.

All the examples so far have used the same scheduling algorithm, but the algorithm can be changed using the configUSE_PREEMPTION and configUSE_TIME_SLICING configuration constants. Both constants are defined in FreeRTOSConfig.h.

Prioritized Pre-emptive Scheduling with Time Slicing

The configuration shown in Table 14 sets the FreeRTOS scheduler to use a scheduling algorithm called 'Fixed Priority Pre-emptive Scheduling with Time Slicing', which is the scheduling algorithm used by most small RTOS applications, and the algorithm used by all the examples presented in this book so far.

Table 14. The FreeRTOSConfig.h settings that configure the kernel to use Prioritized Pre-emptive Scheduling with Time Slicing

Constant	Value
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

Creat two task without any delay having same priority.

/* Include files */

```
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"

/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";

/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);

int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock */
        SystemCoreClockUpdate();



        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                      /* This task will run at priority 1. */
        NULL );                 /* We are not using the task handle. */

      /* Create the other task in exactly the same way. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );
        /* Start the scheduler so our tasks start executing. */
```

```
	vTaskStartScheduler();
	/* If all is well we will never reach here as the scheduler will now be
	running.  If we do reach here then it is likely that there was insufficient
	heap available for the idle task to be created. */
	for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
	/* Task is implemented in an infinite loop. */
	for( ;; )
	{
		/* Print out the name of this task. */
		printf( "%s\r\n",pcTaskName );
		/* Delay for a period. */
	//vTaskDelay( 100 );
	}
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
	/* Task is implemented in an infinite loop. */
	for( ;; )
	{
		/* Print out the name of this task. */
		printf( "%s\r\n",pcTaskName );
		/* Delay for a period. */
	//	vTaskDelay( 100 );
	}
}
```

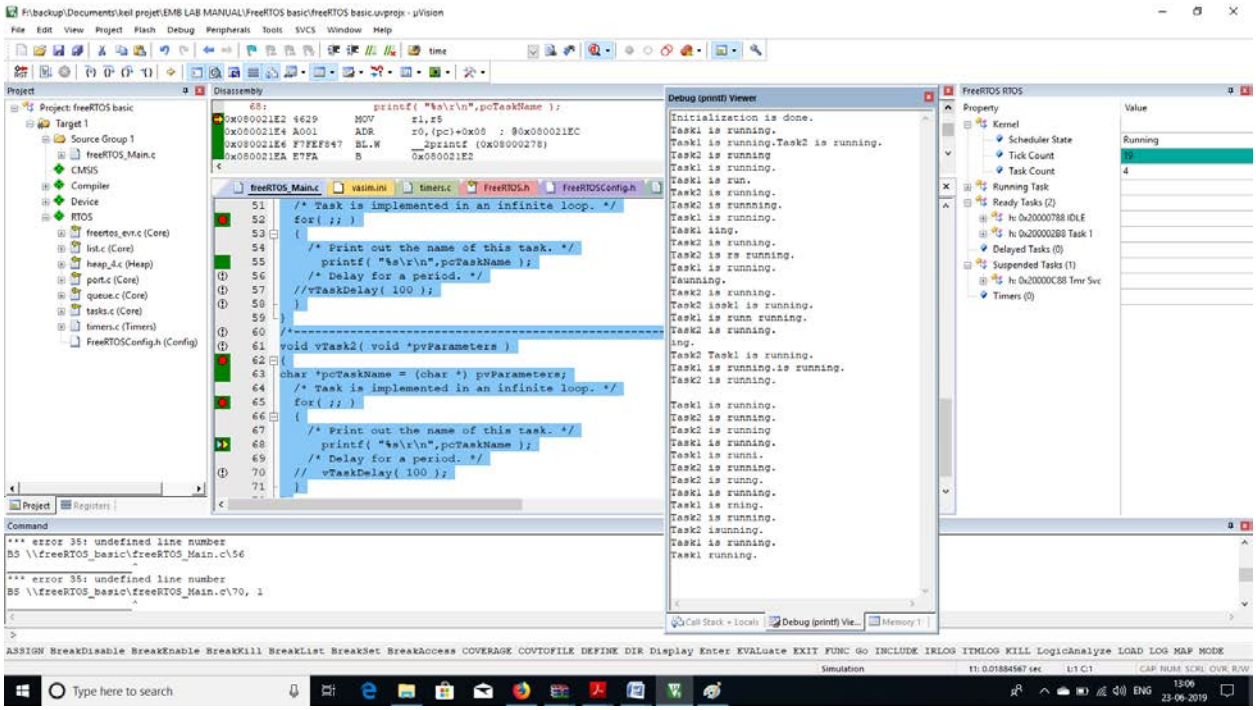


Fig7.1: two task without any delay output

As shown in output debug (printf) viewer, few of the print statement are not printed completed properly, as pre-emptive is 1 and time slicing is 1 at every time tick scheduler change the task in execution.

Set pre emption to 0 now.

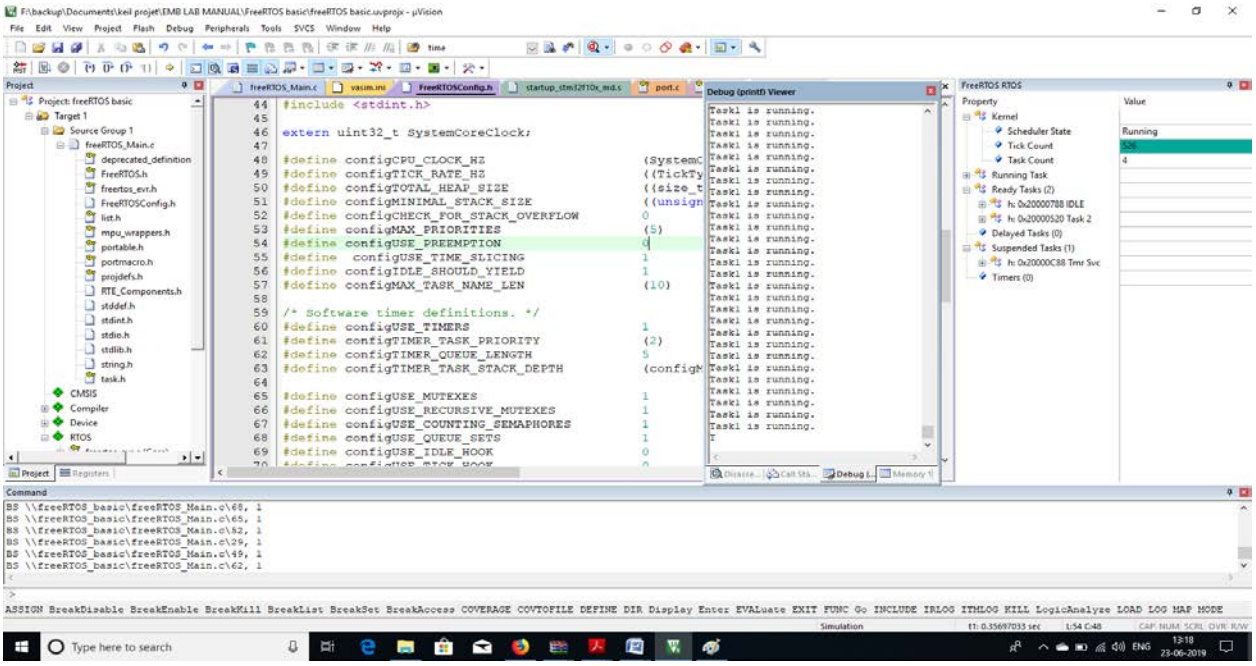


Fig7.2: output with preemption 0

As pre-emption is set to 0 only task 1 is running and task 2 is in ready state but not allowed to run.

Now provide delay of 10 in task 1 and no delay in task 2 and keep pre emption 1

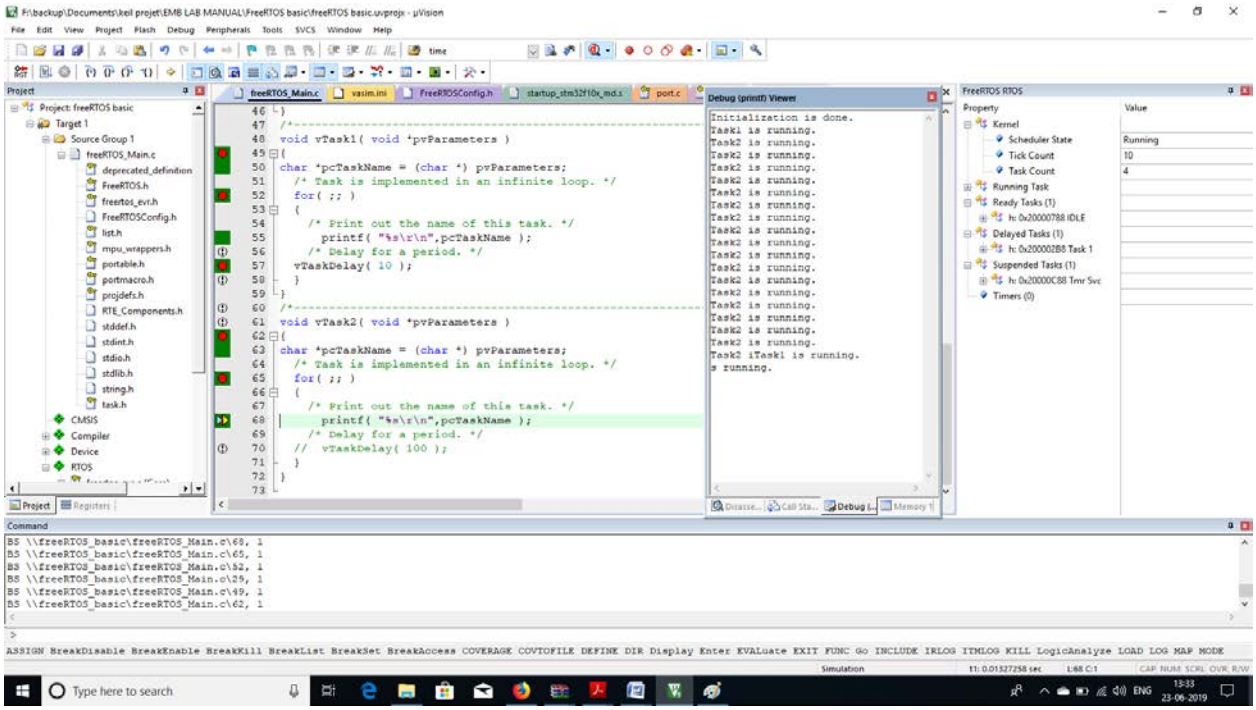


Fig :7.3 task 1 with delay of 10 and task 2 with no delay

As shown in output after running task 1 due to delay of 10 ticks, task 2 runs few times and short terminated, once wait time of 10 tick is over by task 1.

In above case now set pre emption to 0.

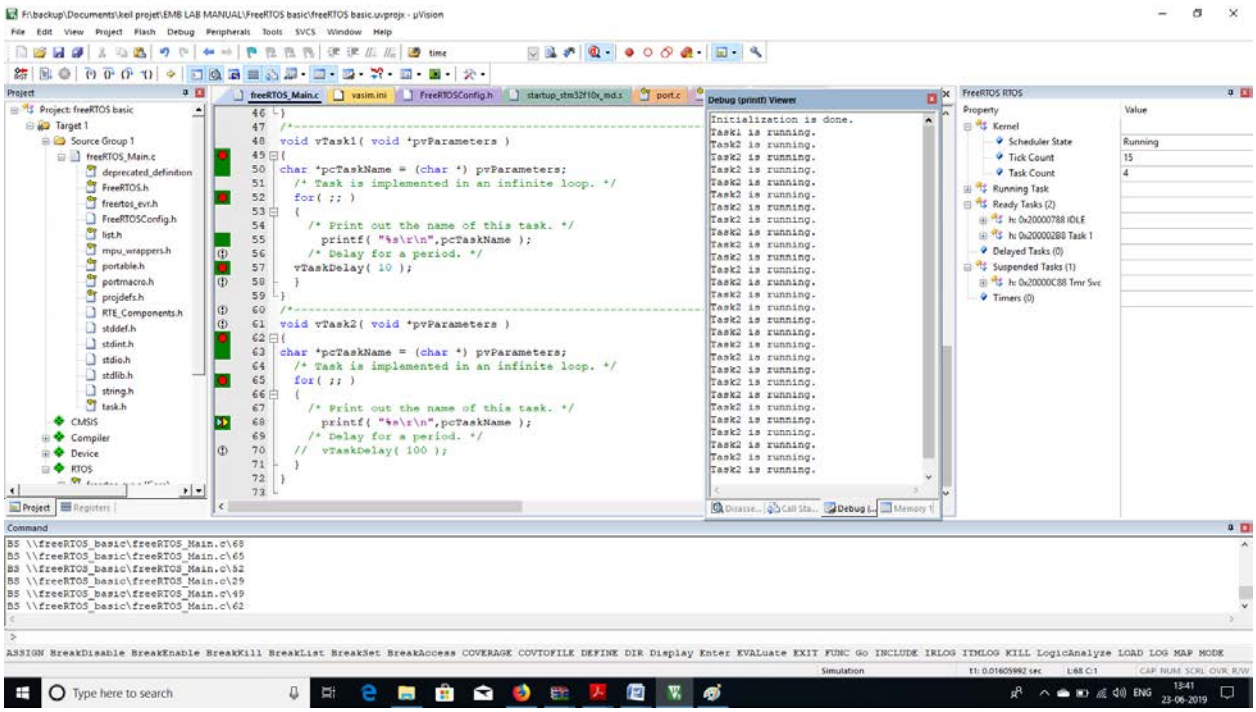


Fig:7.4 pre emption set to 0

As pre emption is disabled first task1 will execute, then due to the delay after that task 2 will run and even after task 1 delay is over and task 1 is ready to run but as due to preemption is disabled it will be not allowed to run.

Assignment:

Create three task (task 1, task 2 and task 3) and set task 1 and task 2 on same priority and task 3 at lower priority. Set delay of 10 in task 1, 20 in delay of task 2 and no delay in task 3. Observe the output with pre emption enabled and disabled.

Conclusion

LAB 8 SEMAPHORE USING FreeRTOS

Aim: To study and implement the concept of semaphore for resource sharing in FreeRTOS.

Theory:

FreeRTOS is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements.

Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly might make a system seem annoyingly unresponsive without actually making it unusable. Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag has the potential to do more harm than good if it responded to crash sensor inputs too slowly.

SemaphoreA semaphore is a variable or abstract data type used to control access to a common resource by multiple processors in a concurrent system.

xSemaphoreCreateBinary() Creates a binary semaphore, and returns a handle by which the semaphore can bereferenced.The semaphore created by this function takes two values only.

xSemaphoreGive() 'Gives' (or releases) a semaphore that has previously been created using a call tovSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex() and has also been successfully 'taken'.

xSemaphoreTake() 'Takes' (or obtains) a semaphore that has previously been created using a call tovSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex().

SAMPLE PROGRAM:

```
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
/* The task functions prototype*/
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

SemaphoreHandle_t xSemaphore_pri;

/* Task parameter to be sent to the task function */
const char *pvTask1  = "Task1 is running.";
const char *pvTask2  = "Task2 is running.";
```

```
/* Extern functions */
extern void SystemInit(void);
extern void SystemCoreClockUpdate(void);

int main( void )
{
        /* Board initializations */
      SystemInit();


/* This function initializes the MCU clock, PLL will be used to generate Main MCU clock */
        SystemCoreClockUpdate();


        printf("Initialization is done.\r\n");
        /* Create one of the two tasks. */
        xTaskCreate(vTask1, /* Pointer to the function that implements the task. */
        "Task 1", /* Text name for the task.  This is to facilitate debugging only. */
      configMINIMAL_STACK_SIZE, /* Stack depth in words. */
        (void*)pvTask1,                 /* We are not using the task parameter. */
        1,                      /* This task will run at priority 1. */
        NULL );                 /* We are not using the task handle. */

      /* Create the other task in exactly the same way. */
        xTaskCreate( vTask2, "Task 2", configMINIMAL_STACK_SIZE, (void*)pvTask2, 1,
NULL );
        /* Start the scheduler so our tasks start executing. */

        xSemaphore_pri = xSemaphoreCreateBinary();
  xSemaphoreGive(xSemaphore_pri);

        vTaskStartScheduler();
        /* If all is well we will never reach here as the scheduler will now be
        running.  If we do reach here then it is likely that there was insufficient
        heap available for the idle task to be created. */
        for( ;; );
}
/*-----------------------------------------------------------*/
void vTask1( void *pvParameters )
{
```

```
char *pcTaskName = (char *) pvParameters;
	/* Task is implemented in an infinite loop. */
	for( ;; )
	{

		if(xSemaphoreTake(xSemaphore_pri,portMAX_DELAY)==pdTRUE)
			{


		/* Print out the name of this task. */
		printf( "%s\r\n",pcTaskName );
				vTaskDelay( 10 );
				xSemaphoreGive(xSemaphore_pri);
		/* Delay for a period. */

			}
				vTaskDelay( 10);

	}
}
/*-----------------------------------------------------------*/
void vTask2( void *pvParameters )
{
char *pcTaskName = (char *) pvParameters;
	/* Task is implemented in an infinite loop. */
	for( ;; )
	{
		if(xSemaphoreTake(xSemaphore_pri,portMAX_DELAY)==pdTRUE)
			{

	/* Print out the name of this task. */
		printf( "%s\r\n",pcTaskName );
		xSemaphoreGive(xSemaphore_pri);
				vTaskDelay( 10);
			}

				vTaskDelay( 20 );

	}
}
```

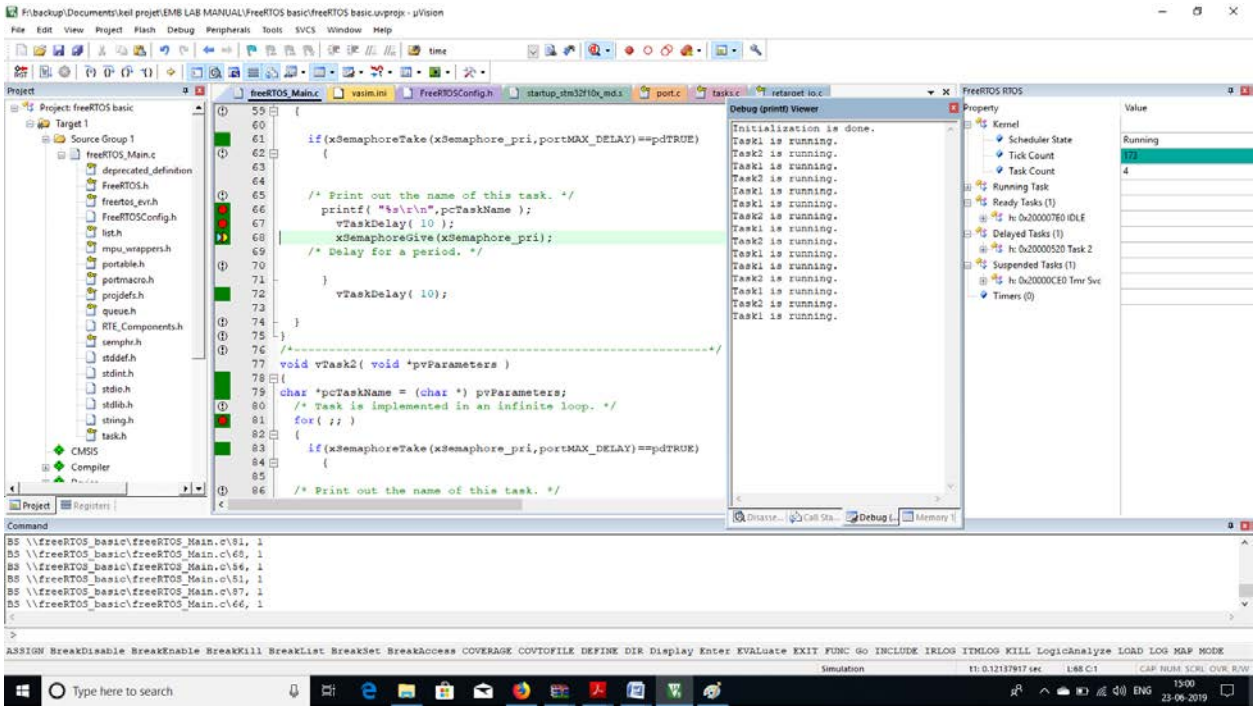


Fig 8.1: output of program

Here shared resource is debug(printf) viewer, as seen in previous experiment due to pre emption enabled few of the printf statement were terminated. Now due to semaphore until one function release the semaphore other function cannot use the same semaphore.

Assignment:

1) Create three tasks with different priority and with time delay use semaphore reserve shared resource and observe output.
2) In assignment 1 now remove delay from the lowest priority task and observe output.
3) In assignment 1 from the lowest priority task do not release the semaphore and observe output.

Conclusion

LAB 9 Exception and Fault

Aim: - 1. Write a program that shows software interrupts with their handlers.
2. Write a sample code to demonstrate various types of faults

Theory:

(1) Exception

Once configured, this more advanced operating mode provides a partition between the exception/interrupt code running in handler mode and the background application code running in thread mode. Each operating mode can have its own code region, RAM region, and stack. This allows the interrupt handler code full access to the chip without the risk that it may be corrupted by the application code. However, at some point, the application code will need to access features of the Cortex-M processor that are only available in the handler mode with its full privileged access. To allow this to happen, the Thumb-2 instruction set has an instruction called Supervisor Call (SVC). When this instruction is executed, it raises a supervisor exception that moves the processor from executing the application code in thread\unprivileged mode to an exception routine in handler/privileged mode as shown in Fig. 9.1. The SVC has its own location within the vector table and behaves like any other exception.

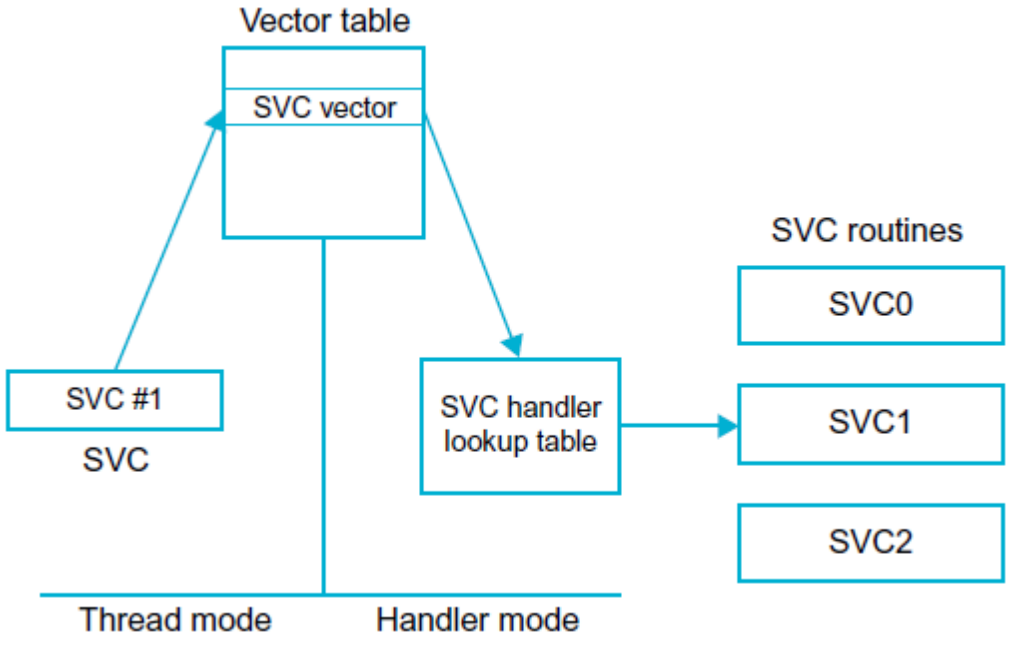


Fig. 9. 1 SVC Call

The supervisor instruction may also be encoded with an 8-bit value called an ordinal. When the SVC is executed, this ordinal value can be read and used as an index to call one of 256 different supervisor functions. The toolchain provides a dedicated SVC support function that is used to

extract the ordinal value and call the appropriate function. This number is then used as an index into a lookup table to load the address of the function that is being called.

The project consists of the standard project startup file and the initializing system file. The application source code is in the file Test.c. There is an additional source file SVC.c that provides support for handling SVC exceptions. The SVC.c file contains the SVC exception handler; this is a standard support file that is provided with the ARM compiler. The application code in Test.c is calling four simple functions that in turn call routines to perform basic arithmetic operations. Each of the arithmetic functions is designed to be called with an SVC instruction so that all of these functions run in handler mode rather than thread mode. In order to convert the arithmetic functions from standard functions to software interrupt functions, we need to change the way the function prototype is declared. The way this is done will vary between compilers, but in the ARM compiler there is a function qualifier __svc. This is used as shown below to convert the function to an SVC and allows you to pass up to four parameters and get a return value. The __svc qualifier defines this function as an SVC and defines the ordinal number of the function.

SAMPLE APPLICATION CODE:

File Name: Test.c

```
#include <stm32f10x.h>                  /* STM32F103 definitions        */

  int __svc(0) add (int i1, int i2);
  int __SVC_0      (int i1, int i2) {
  return (i1 + i2);
}
 int __svc(1) mul4(int i);
 int __SVC_1      (int i) {
 return (i << 2);
}

int __svc(2) div (int i1, int i2);
int __SVC_2      (int i1, int i2) {
  return (i1 / i2);
}

int __svc(3) mod (int i1, int i2);
int __SVC_3      (int i1, int i2) {
  return (i1 % i2);
}

int res;
```

```
/*----------------------------------------------------------------------------
  Test Function
 *----------------------------------------------------------------------------*/
void test_t (void) {
  res  = div (res, 10);                    /* Call SWI Functions       */
  res  = mod (res,  3);
}

/*----------------------------------------------------------------------------
  Test Function
 *----------------------------------------------------------------------------*/
void test_a (void) {
  res  = add (74,  27);                     /* Call SWI Functions       */
  res += mul4(res);
}

/*----------------------------------------------------------------------------
  MAIN function
 *----------------------------------------------------------------------------*/
int main (void) {

  test_a();
  test_t();

  while (1) ;
}
```

SVC HANDLER CODE:

File Name: SVC.c

```
__asm void SVC_Handler (void) {
      PRESERVE8

            TST     LR,#4                ; Called from Handler Mode?
            MRSNE   R12,PSP                ; Yes, use PSP
            MOVEQ   R12,SP                 ; No, use MSP
            LDR     R12,[R12,#24]          ; Read Saved PC from Stack
            LDRH    R12,[R12,#-2]           ; PC-2 to get op-ccode of SVC
            BICS    R12,R12,#0xFF00         ; Extract SVC Number
                  PUSH    {R4,LR}             ; Save Registers

            LDR     LR,=SVC_Count
            LDR     LR,[LR]
            CMP     R12,LR
```

```
        BHS     SVC_Dead            ; Overflow

       LDR    LR,=SVC_Table
        LDR     R12,[LR,R12,LSL#2]    ; Load SVC Function Address
        BLX     R12                 ; Call SVC Function

        POP     {R4,LR}
        TST     LR,#4
        MRSNE   R12,PSP
        MOVEQ   R12,SP
        STM     R12,{R0-R3}         ; Function return values
        BX      LR                  ; RETI

SVC_Dead
        B       SVC_Dead            ; None Existing SVC

SVC_Cnt       EQU    (SVC_End-SVC_Table)/4
SVC_Count      DCD     SVC_Cnt

; Import user SVC functions here.
        IMPORT  __SVC_0
        IMPORT  __SVC_1
        IMPORT  __SVC_2
        IMPORT  __SVC_3

SVC_Table
; Insert user SVC functions here
        DCD     __SVC_0             ; SVC 0 Function Entry
        DCD     __SVC_1             ; SVC 1 Function Entry
        DCD     __SVC_2             ; SVC 2 Function Entry
        DCD     __SVC_3             ; SVC 2 Function Entry

SVC_End

        ALIGN
}
```

OUTPUT: Following Fig. 9.3 to Fig. 9.7 shows step by step program execution by using the above code.

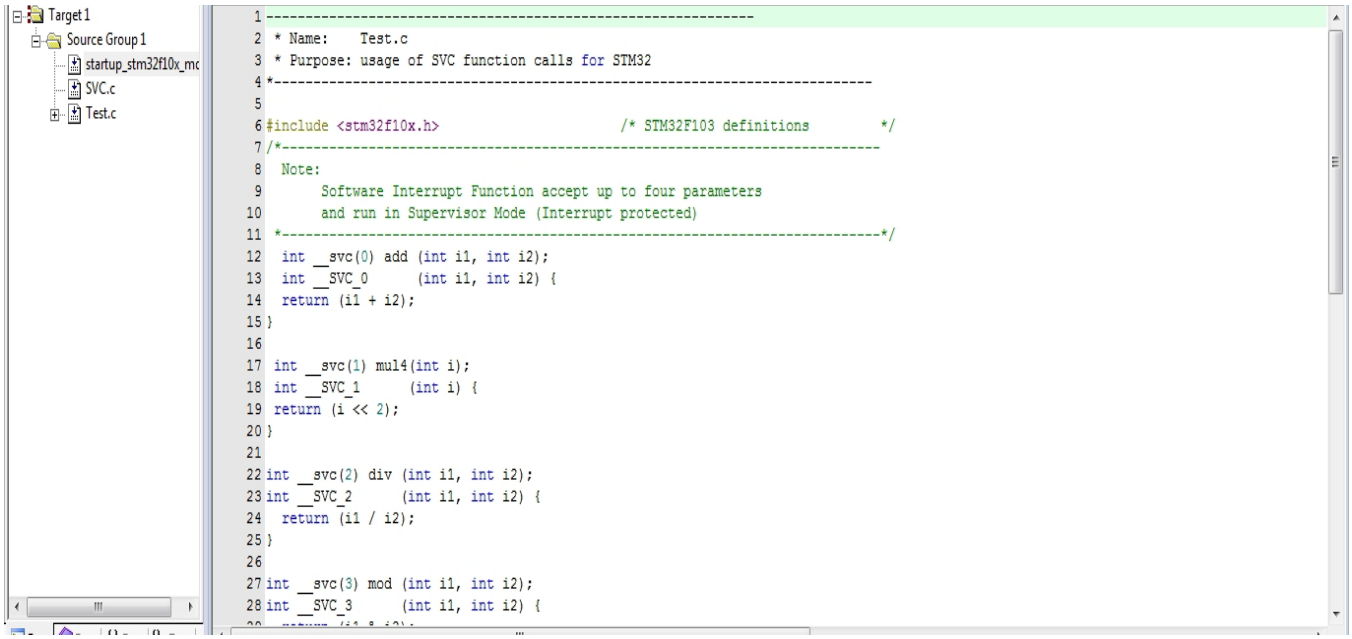


Fig. 9. 3 Project Window

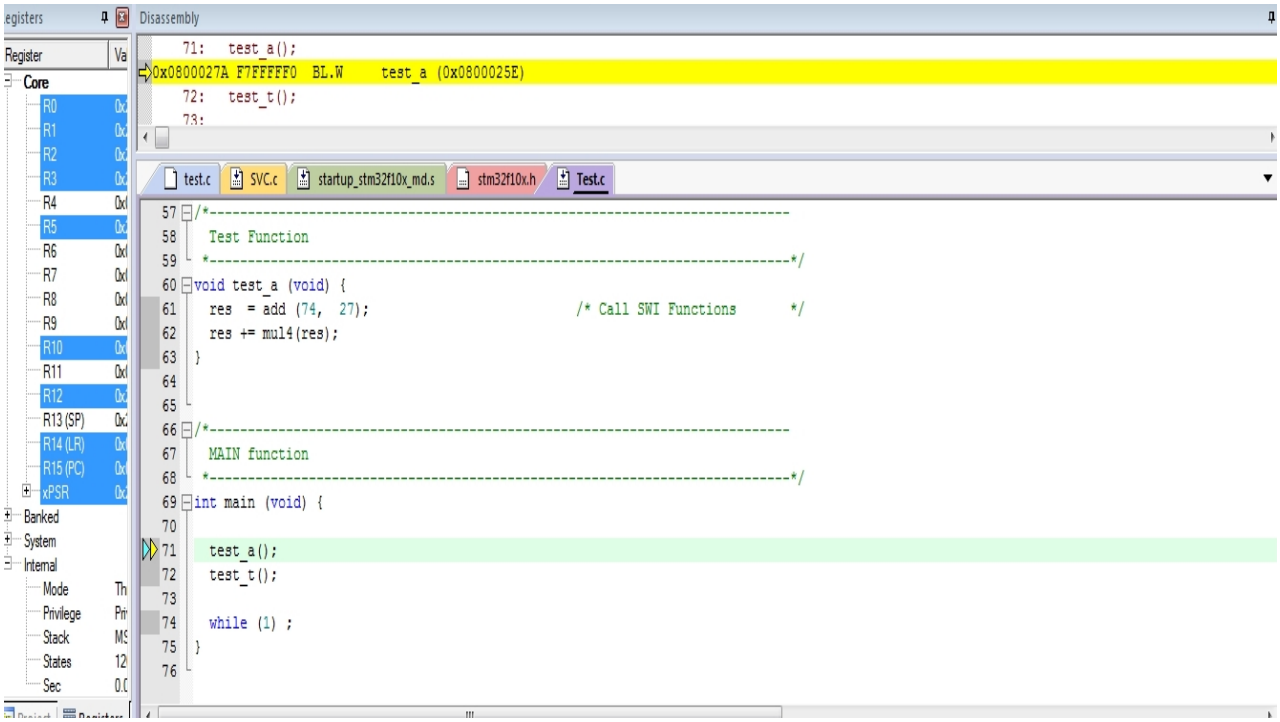


Fig. 9.4 Beginning of program execution

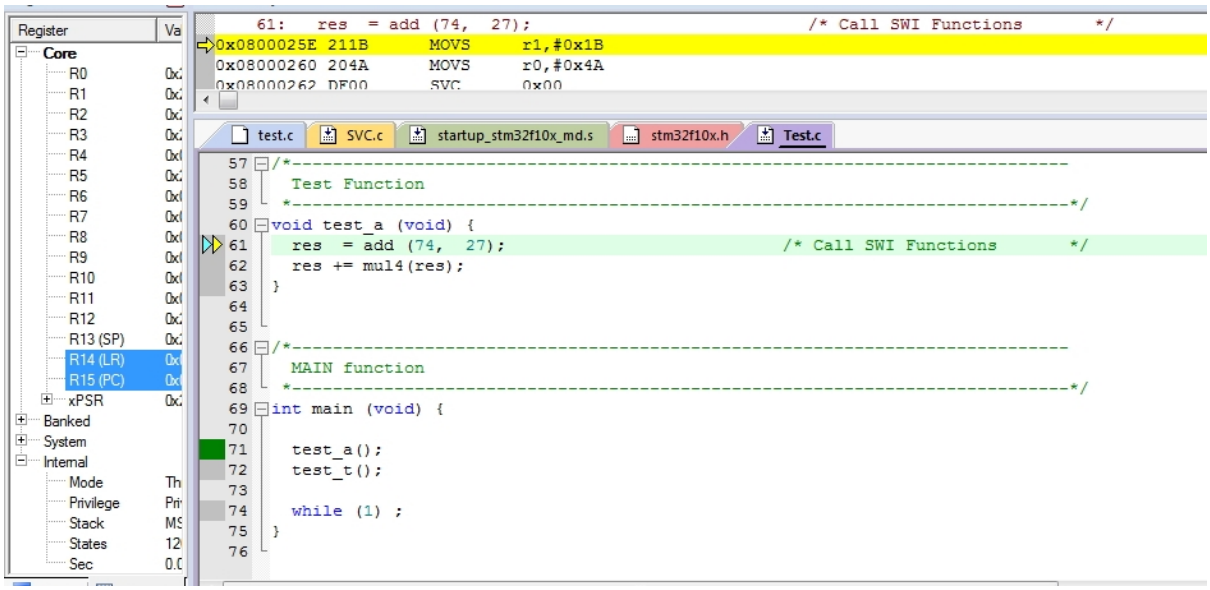


Fig. 9.5 Program enter to execute add function

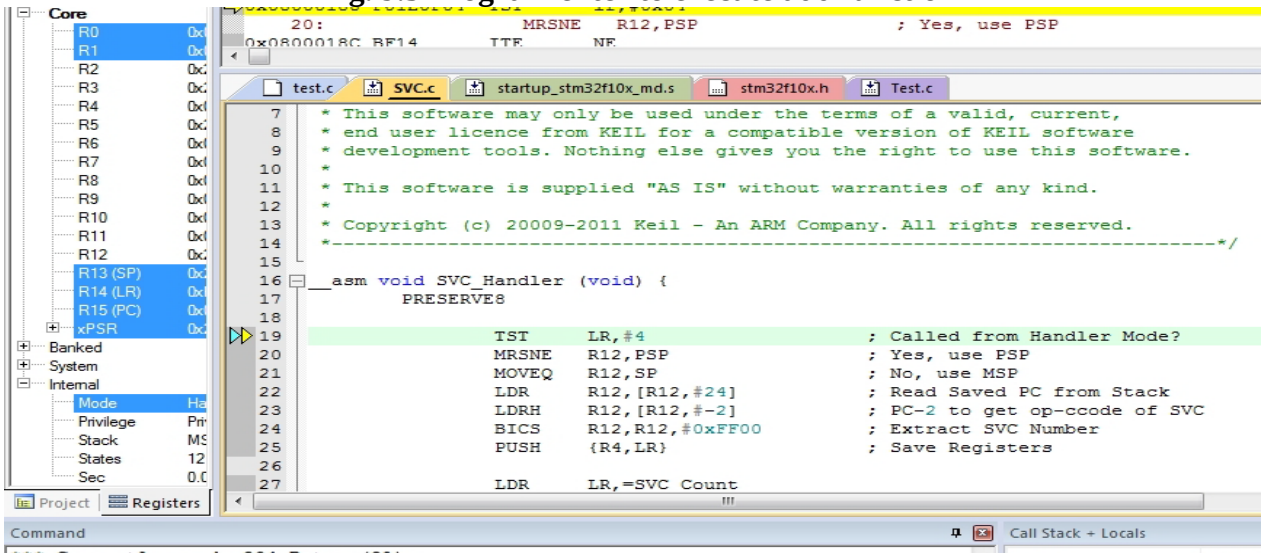


Fig. 9.6 Program entered into SVC Handler

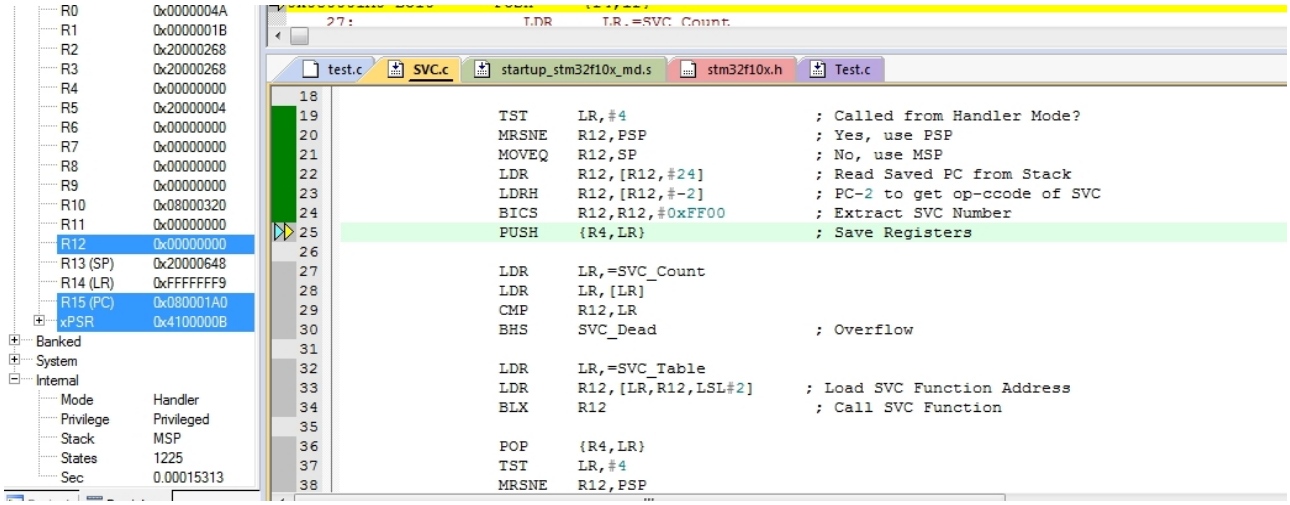


Fig. 9.7 SVC number is extracted and kept in R12

(2) Faults

The following smalpe code demonstrates memory mangment fault.

```
MOV R0,#0x40000000; Non-exectable regin
BX R0;
```

The foloowing Fig. 9.8 and Fig. 9.9 shows steps to generate the memory mangment fault. Enter

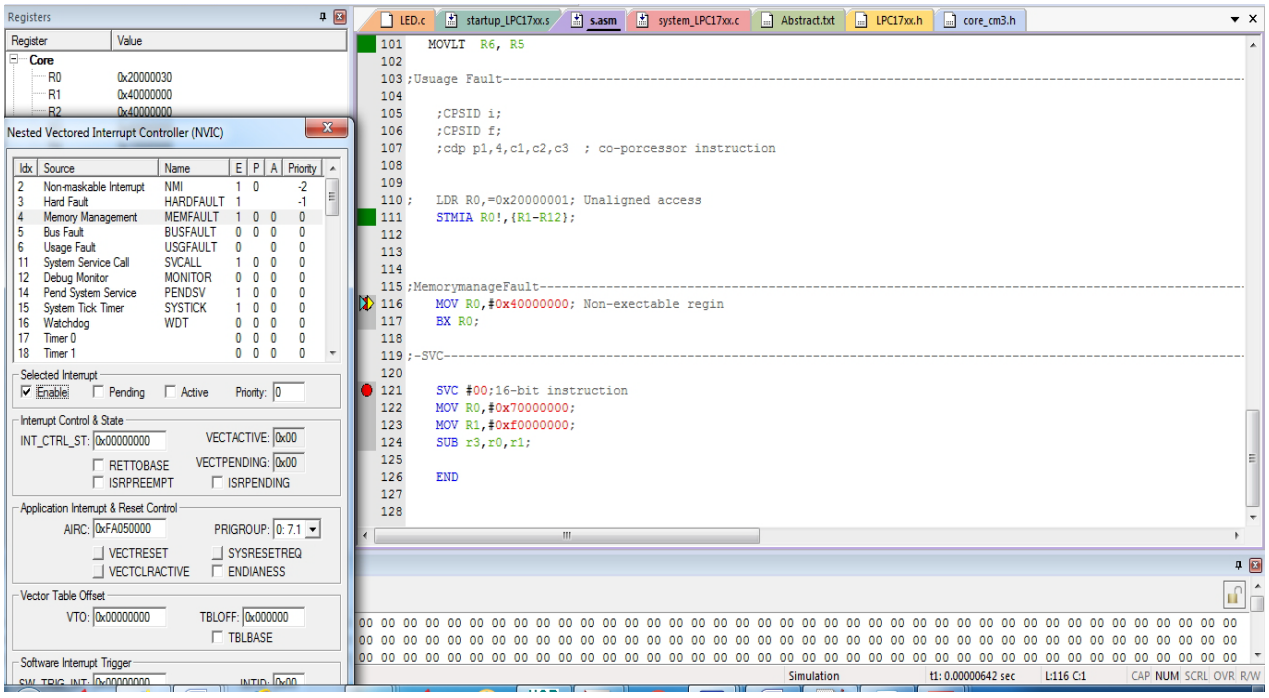


Fig.9.8 Memory Management Fault is enabled in NVIC using GUI support

Enter in to Debug and enable Memory Management Exception Fault from Peripherals->Core Peripherals->Nested Vectored Interrupt Controller.

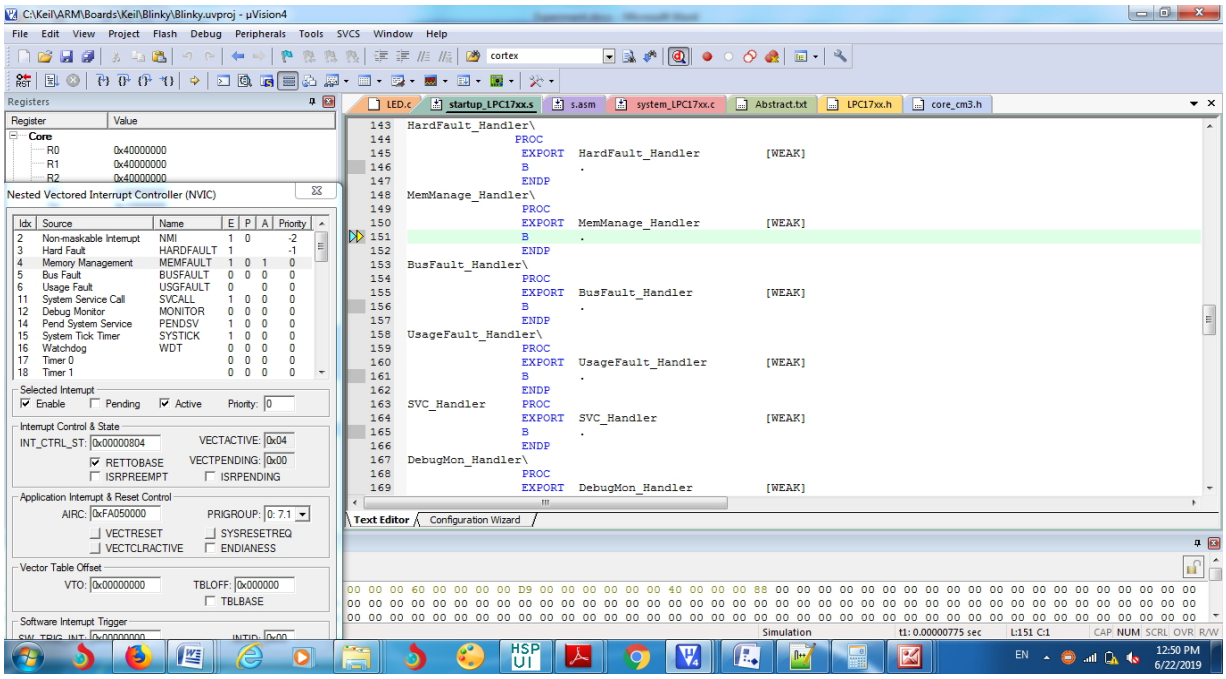


Fig. 9.9 Program enters into corresponding Memory Mangment Fault Handler

Assignment:

1. Generate the software interrupt int 4 to find maximum number among 10 data bytes.
2. Write a sample code to generate other types of faults like usage fault, hard fault etc.

Conclusion:

_____

_____

_____

_____

_____

_____

LAB 10 DEVICE DRIVERS

Aim: To implement the character device driver.

Theory:

A device driver is a program that controls a particular type of device that is attached to our computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on. When we buy an operating system, many device drivers are built into the product. However, if we later buy a new type of device that the operating system didn't anticipate, we'll have to install the new device driver. A device driver essentially converts the more general input/output instructions of the operating system to messages that the device type can understand.

There are several kinds of device drivers, each handling a different kind of I/O devices. Blockdevice drivers manage devices with physically addressable storage media, such as disks, memory. All other devices are considered character devices.

Block Device Drivers

Devices that support a file system are known as block devices. Drivers written for thesedevices are known as block device drivers. Block device drivers take a file systemrequest, in the form of a buffer structure, and issue the I/O operations to the disk totransfer the specified block. The main interface to the file system is the strategy routine.

Character Device

Drivers Character device drivers normally perform I/O in a byte stream. Examples of devicesusing character drivers include tape drives and serial ports. Character device driverscan also provide additional interfaces not present in block drivers, such as I/O control(IOCTL) commands, memory mapping, and device polling.

SAMPLE PROGRAM:

joshichar.c

```
#include<linux/init.h>                  //initialization
#include<linux/module.h>
#include<linux/kernel.h>                    //kernel support
#include<linux/device.h>
#include<linux/uaccess.h>                   //for memory copy
#include<linux/fs.h>                        // linux file system

#define DEVICE_NAME "joshichar"             // /dev/joshichar
#define CLASS_NAME "joshi"                  //device class

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dr. N Joshi");
MODULE_DESCRIPTION("A simple device driver");
MODULE_VERSION("0.1");
```

```
static int majorNumber;                                 // stores device-number
static char message[256] = {0};                         //input Buffer
static short sz;                                        //size
static struct class* joshicharClass = NULL;             //class
static struct device* joshicharDevice = NULL;           //device

static int dev_open(struct inode*, struct file*);
static int dev_release(struct inode*, struct file*);
static ssize_t dev_read(struct file*, char*, size_t, loff_t*);
static ssize_t dev_write(struct file*, const char*, size_t, loff_t*);

static struct file_operations fops =
{
        .open = dev_open,
        .read = dev_read,
        .write = dev_write,
        .release = dev_release
};                                                      //mapping functions

static int __init joshichar_init(void)                  //initialize device
{
        printk(KERN_INFO "JOSHIChar: Initializing the JOSHIChar LKM (DDU).\n");
   majorNumber = register_chrdev(0,DEVICE_NAME,&fops);
   if(majorNumber < 0)
   {
        printk(KERN_ALERT "JOSHIChar: failure to register major number.\n");
        return majorNumber;
   }
   joshicharClass = class_create(THIS_MODULE, CLASS_NAME);
   if(IS_ERR(joshicharClass))
   {
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failure to register class\n");
        return PTR_ERR(joshicharClass);
   }
        joshicharDevice = device_create(joshicharClass, NULL,
        MKDEV(majorNumber,0),NULL,DEVICE_NAME);
   if(IS_ERR(joshicharDevice))
   {
        class_destroy(joshicharClass);
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failure to create the device\n");
        return PTR_ERR(joshicharDevice);
   }
   return 0;
}
```

```
static void __exit joshichar_exit(void)                                 //for removing device
{
        device_destroy(joshicharClass, MKDEV(majorNumber,0));
        class_unregister(joshicharClass); //unregister class
        class_destroy(joshicharClass); //remove device class
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_INFO "JOSHIChar: Device has been removed\n");
}
static int dev_open(struct inode* inodep, struct file* filep)                   //opening device
{
        printk(KERN_INFO "JOSHIChar: Device has been opened\n");
        return 0;
}
static ssize_t dev_read(struct file* filep, char* buffer, size_t len, loff_t* offset)
                                                                                //accessing  device
{
        int error_count = 0;
        printk(KERN_INFO "Device Reading.....\n");
        error_count = copy_to_user(buffer,message,sz);
        if(error_count == 0)
        {
                return(sz=0);
        }
        else
        {
                printk(KERN_INFO "JOSHIChar: Failure to dispatch message to user.\n");
                return(-EFAULT);
        }
}
static ssize_t dev_write(struct file* filep, const char* buffer, size_t len, loff_t* offset)
                                                                        //writing to device
{
      printk(KERN_INFO "Writing to the device.....\n");
        sprintf(message,"%s(%zu letters)",buffer,len);
        sz = strlen(message);
        return len;
}
static int dev_release(struct inode* inodep, struct file* filep)                //releasing device
{
        printk(KERN_INFO "JOSHIChar: Device closed.\n");
        return 0;
}
module_init(joshichar_init);
module_exit(joshichar_exit);
```

test_joshichar.c

/* A Linux user space progrm that communicates with the joshichar.c LKM character device driver. It passes a string to the LKM and reads the response from the LKM. For this program to work, the device must be called /dev/joshichar.*/

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>
#define DEVICE_PATH "/dev/joshichar"
#define BUFFER_LENGTH 256
static char receive[BUFFER_LENGTH];
int main()
{
        int ret, fd;
        char stringToSend[BUFFER_LENGTH];
        printf("Opening char-device...\n");
        fd = open(DEVICE_PATH,O_RDWR);                  //open device with read-write
        if(fd<0)
        {
                perror("Failed to open the char-device..");
                return errno;
        }
        printf("Successfully opened char-device: %s\n",DEVICE_PATH);
        printf("Enter a string to send it to char-device: ");
        gets(stringToSend);
        printf("Writing message [%s]to the char-device %s\n",stringToSend,
                DEVICE_PATH);
        ret = write(fd,stringToSend,strlen(stringToSend));
        if(ret < 0)
        {
                perror("Failed to write the message to the char-device.");
                return errno;
        }
        printf("Press ENTER to read back from the char-device...\n");
        getchar();
        printf("Reading from the char-device...\n");
        ret = read(fd, receive,BUFFER_LENGTH);          //read the response from LKM
        if(ret < 0)
        {
                perror("Failed to read the message from the char-device.");
                return errno;
        }
        printf("The received message from char-device: [%s]\n", receive);
```

```
        printf("Good Bye.\n");
        return 0;
}
```

Makefile

```
obj-m+=joshichar.o

all:
        make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
        $(CC) test_joshichar.c -o test
clean:
        make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
        rm test
```

STEPS TO CREATE A CHARACTER DEVICE DRIVER:

i. To include kernel header files first we have to find out present kernel version using uname –r in terminal window.the ouput is shown in fig 10.1

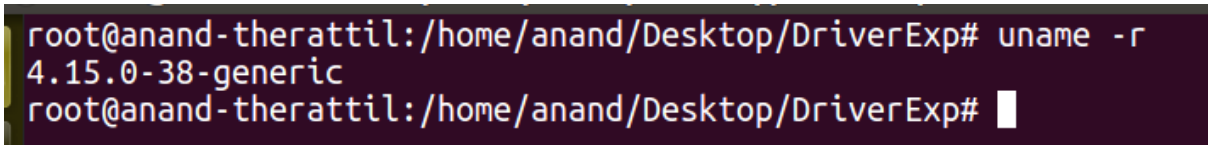


Fig 10.1 output of uname –r

This command is used for getting the kernel version of the current system.

ii. Search the appropriate kernel header files using sudo apt-cache search linux-header-$(uname –r).output is shown in fig 10.2.

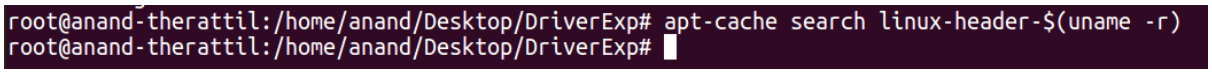


Fig 10.2 output of searching required linux header

iii. Upgrade the kernel header files using sudo apt-get install linux-header-$(uname –r).
The output of this command is shown in fig 10.3

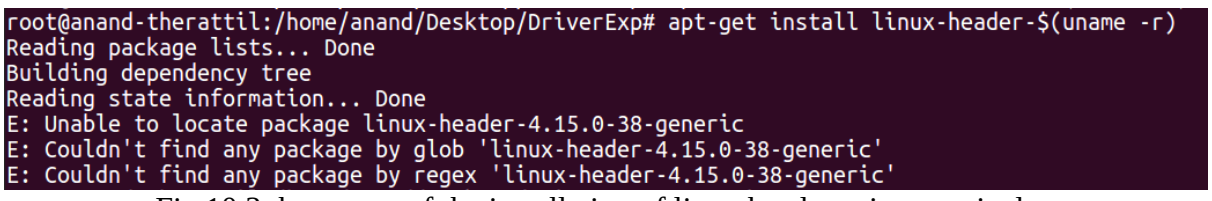


Fig 10.3 the output of the installation of linux header using terminal

If by using this command an error occurred then the synaptics command should be used for installation process of the linux header.simply write synaptic on terminal which will open window shown in fig 10.4.

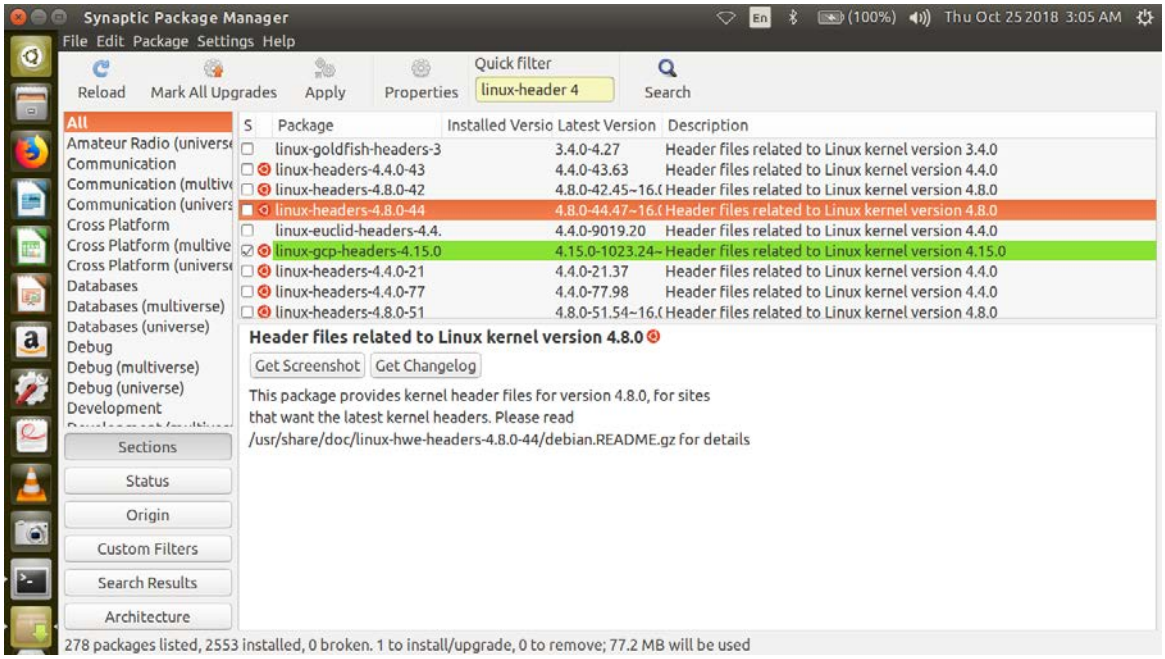


Fig 10.4 synaptic window

Find the linux-header of the appropriate kernel version and install the header by right clicking it and apply.

iv. Change directory from present directory to the code consisting directory.

v. By using make command, various files are created including kernel object file, code object file executable file and modules.the output of the command is shown in fig x.5

```
root@anand-therattil:/home/anand/Desktop/DriverExp# make
make -C /lib/modules/4.15.0-38-generic/build/ M=/home/anand/Desktop/DriverExp modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-38-generic'
  CC [M]  /home/anand/Desktop/DriverExp/joshichar.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/anand/Desktop/DriverExp/joshichar.mod.o
  LD [M]  /home/anand/Desktop/DriverExp/joshichar.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-38-generic'
cc test_joshichar.c -o test
test_joshichar.c: In function ‘main’:
test_joshichar.c:35:2: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
  gets(stringToSend);
  ^
/tmp/cc2dt1Fd.o: In function `main':
test_joshichar.c:(.text+0xa7): warning: the `gets' function is dangerous and should not be used.
root@anand-therattil:/home/anand/Desktop/DriverExp#
```

Figs 10.5 output of make command.

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. For executing the main command a Makefile must be present. In this sample we are using the Makefile is

used for the compiling the files joshichar.c and test_joshichar.c using the file Makefile,it creates the obeject file,kernel object file,executable file.

vi. Using insmod command insert the character device to the kernel.which is shown in fig 10.6

vii. To verify device has been inserted use lsmod command.the ouput of the command is shown in fig 10.6

```
root@anand-therattil:/home/anand/Desktop/DriverExp# insmod joshichar.ko
root@anand-therattil:/home/anand/Desktop/DriverExp# lsmod
Module                  Size  Used by
joshichar              16384  0
nls_iso8859_1          16384  0
uas                    24576  0
usb_storage            69632  1 uas
```

Fig 10.6 output of insmod and lsmod.

insmod command is used for inserting the joshichar module to the kernel. This could be verified using the lsmod command which gives all the modules available with the kernel, its size and the number of user accessing it

.

viii. To run the executable file use ./test.the output is shown if fig 10.7.

```
root@anand-therattil:/home/anand/Desktop/DriverExp# ./test
Opening char-device...

 the value of FD is 3
Successfully opened char-device: /dev/joshichar
Enter a string to send it to char-device:hello
Writing message [ hello ]to the char-device /dev/joshichar
Killed
root@anand-therattil:/home/anand/Desktop/DriverExp#
```

Fig 10.7 output of ./test

After running the test file we execute the both the joshichar.c & test_char.c which will open the character device write to the character device and read from the character device and at the end will remove the device from the kernel.

ix. Use rmmod command to remove the character device from the kernel.the output of the removing the device is shown in fig 10.8.

```
root@anand-therattil:/home/anand/Desktop/DriverExp# rmmod joshichar
root@anand-therattil:/home/anand/Desktop/DriverExp# lsmod
Module                  Size  Used by
nls_iso8859_1          16384  0
uas                    24576  0
usb_storage            69632  1 uas
rfcomm                 77824  0
```

Fig 10.8 removing the device from the kernel

The command rmmod is used for removing the device module from the kernel.As shown in fig 10.8 on excecuting the command rmmod joshichar we remove the the device from the kernel, this could be verified using th lsmod command which is also shown in the same fig 10.8.

x. The initializing of the device,writing to device removing of device can be seen using the dmesg command.the output of dmesg is shown in fig 10.9.and 10.10.

```
[ 5269.158295] Initializing the charater device driver LKM.
[ 5277.591553] Charater Device has been opened
[ 5281.304075] Writing to the device........
[ 5281.304095] BUG: unable to handle kernel paging request at 00007ffe6d685890
[ 5281.304109] IP: string+0x25/0x70
```

Fig 10.9. dmesg output of initializing and writing of device.

```
[ 5281.304546] CR2: 00007ffe6d685890
[ 5281.304551] ---[ end trace bb217b22d2c1dc1c ]---
[ 5281.304725] Device removed.
[ 5287.036964] charater Device has been removed
```

Fig 10.10 dmesg output for removing device

CONCLUSION:

PART II

APPENDIX

(DATASHEET OF STM32F103X8 FAMILY)

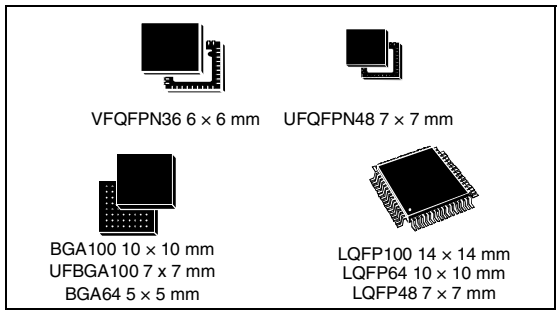
STM32F103x8
STM32F103xB

Medium-density performance line ARM®-based 32-bit MCU with 64 or 128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 com. interfaces

Datasheet - production data

Features

- ARM® 32-bit Cortex®-M3 CPU Core
  - 72 MHz maximum frequency, 1.25 DMIPS/MHz (Dhrystone 2.1) performance at 0 wait state memory access
  - Single-cycle multiplication and hardware division
- Memories
  - 64 or 128 Kbytes of Flash memory
  - 20 Kbytes of SRAM
- Clock, reset and supply management
  - 2.0 to 3.6 V application supply and I/Os
  - POR, PDR, and programmable voltage detector (PVD)
  - 4-to-16 MHz crystal oscillator
  - Internal 8 MHz factory-trimmed RC
  - Internal 40 kHz RC
  - PLL for CPU clock
  - 32 kHz oscillator for RTC with calibration
- Low-power
  - Sleep, Stop and Standby modes
  - V_{BAT} supply for RTC and backup registers
- 2 x 12-bit, 1 µs A/D converters (up to 16 channels)
  - Conversion range: 0 to 3.6 V
  - Dual-sample and hold capability
  - Temperature sensor
- DMA
  - 7-channel DMA controller
  - Peripherals supported: timers, ADC, SPIs, I^2Cs and USARTs
- Up to 80 fast I/O ports
  - 26/37/51/80 I/Os, all mappable on 16 external interrupt vectors and almost all 5 V-tolerant



- Debug mode
  - Serial wire debug (SWD) & JTAG interfaces
- 7 timers
  - Three 16-bit timers, each with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
  - 16-bit, motor control PWM timer with dead-time generation and emergency stop
  - 2 watchdog timers (Independent and Window)
  - SysTick timer 24-bit downcounter
- Up to 9 communication interfaces
  - Up to 2 x I^2C interfaces (SMBus/PMBus)
  - Up to 3 USARTs (ISO 7816 interface, LIN, IrDA capability, modem control)
  - Up to 2 SPIs (18 Mbit/s)
  - CAN interface (2.0B Active)
  - USB 2.0 full-speed interface
- CRC calculation unit, 96-bit unique ID
- Packages are ECOPACK®

Table 1. Device summary

Reference	Part number
STM32F103x8	STM32F103C8, STM32F103R8 STM32F103V8, STM32F103T8
STM32F103xB	STM32F103RB STM32F103VB, STM32F103CB, STM32F103TB

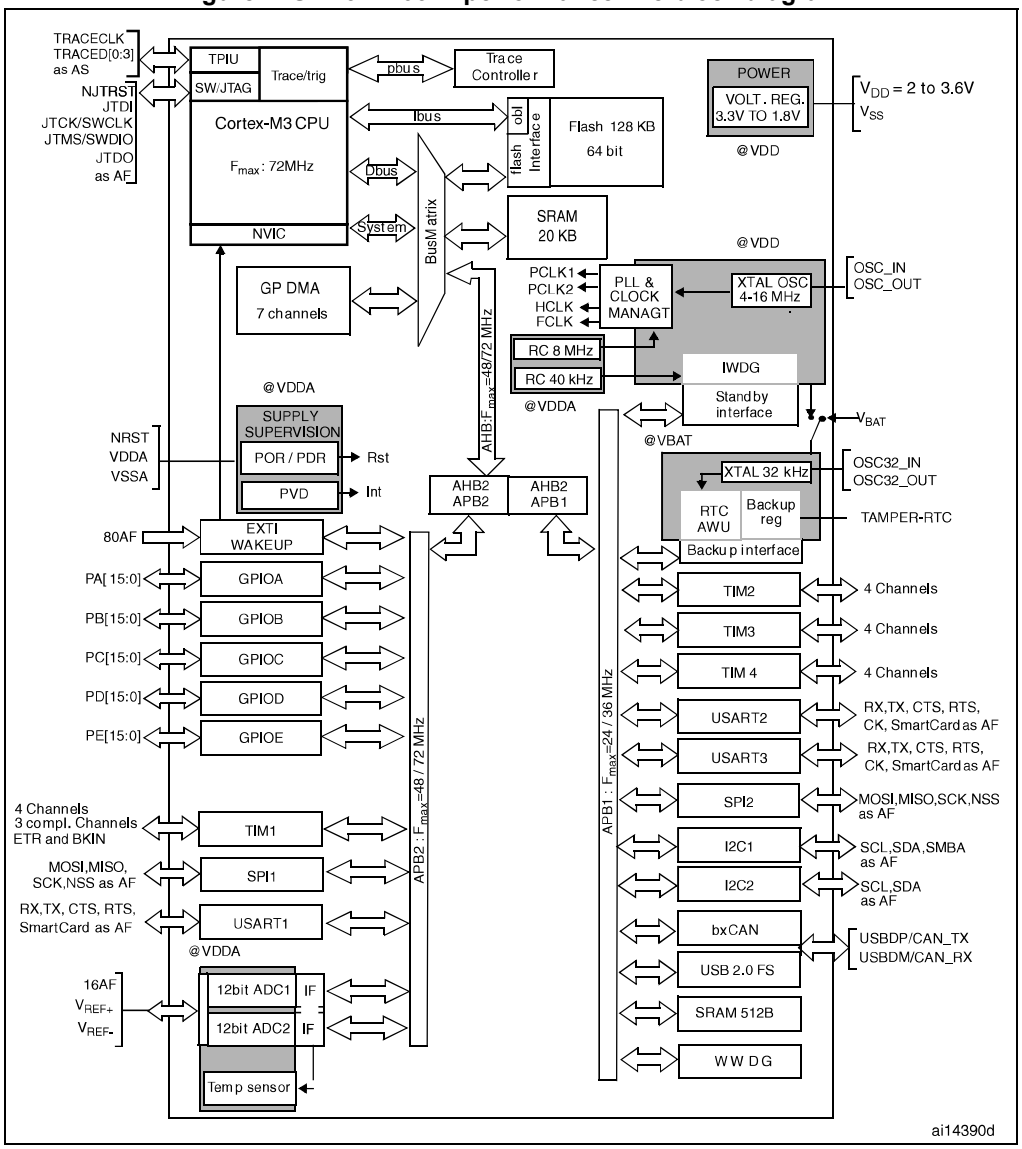
2.1 Device overview

Table 2. STM32F103xx medium-density device features and peripheral counts

Peripheral		STM32F103Tx		STM32F103Cx		STM32F103Rx		STM32F103Vx	
Flash - Kbytes		64	128	64	128	64	128	64	128
SRAM - Kbytes		20		20		20		20	
Timers	General-purpose	3		3		3		3	
	Advanced-control	1		1		1		1	
Communication	SPI	1		2		2		2	
	I^2C	1		2		2		2	
	USART	2		3		3		3	
	USB	1		1		1		1	
	CAN	1		1		1		1	
GPIOs		26		37		51		80	
12-bit synchronized ADC Number of channels		2 10 channels		2 10 channels		2 16 channels(1)		2 16 channels	
CPU frequency		72 MHz							
Operating voltage		2.0 to 3.6 V							
Operating temperatures		Ambient temperatures: -40 to +85 °C / -40 to +105 °C (see *Table 9*) Junction temperature: -40 to + 125 °C (see *Table 9*)							
Packages		VFQFPN36		LQFP48, UFQFPN48		LQFP64, TFBGA64		LQFP100, LFBGA100, UFBGA100	

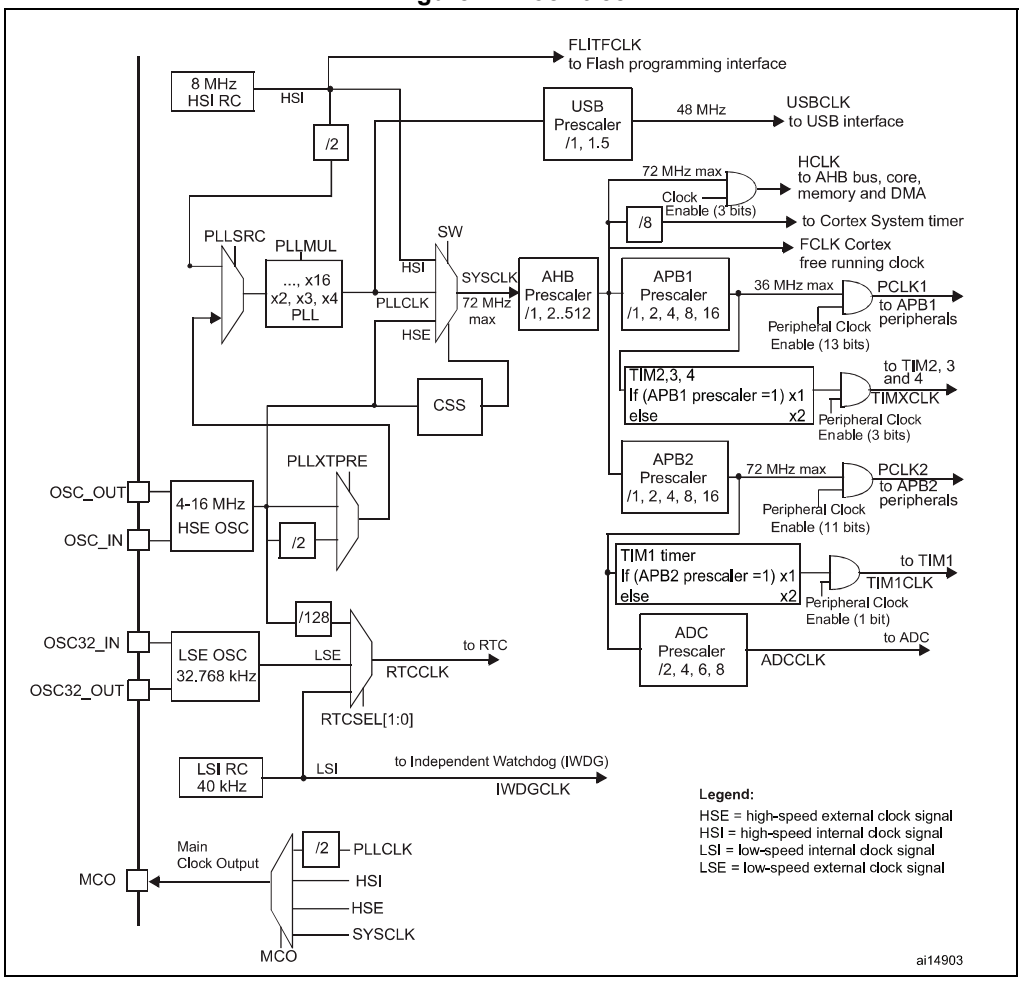
1. On the TFBGA64 package only 15 channels are available (one analog input pin has been replaced by 'Vref+').

Figure 1. STM32F103xx performance line block diagram



1. T_A = –40 °C to +105 °C (junction temperature up to 125 °C).
2. AF = alternate function on I/O port pin.

Figure 2. Clock tree



1. When the HSI is used as a PLL clock input, the maximum system clock frequency that can be achieved is 64 MHz.
2. For the USB function to be available, both HSE and PLL must be enabled, with USBCLK running at 48 MHz.
3. To have an ADC conversion time of 1 µs, APB2 must be at 14 MHz, 28 MHz or 56 MHz.

2.2 Full compatibility throughout the family

The STM32F103xx is a complete family whose members are fully pin-to-pin, software and feature compatible. In the reference manual, the STM32F103x4 and STM32F103x6 are identified as low-density devices, the STM32F103x8 and STM32F103xB are referred to as medium-density devices, and the STM32F103xC, STM32F103xD and STM32F103xE are referred to as high-density devices.

Low- and high-density devices are an extension of the STM32F103x8/B devices, they are specified in the STM32F103x4/6 and STM32F103xC/D/E datasheets, respectively. Low-density devices feature lower Flash memory and RAM capacities, less timers and peripherals. High-density devices have higher Flash memory and RAM capacities, and additional peripherals like SDIO, FSMC, I^2S and DAC, while remaining fully compatible with the other members of the STM32F103xx family.

The STM32F103x4, STM32F103x6, STM32F103xC, STM32F103xD and STM32F103xE are a drop-in replacement for STM32F103x8/B medium-density devices, allowing the user to try different memory densities and providing a greater degree of freedom during the development cycle.

Moreover, the STM32F103xx performance line family is fully compatible with all existing STM32F101xx access line and STM32F102xx USB access line devices.

Table 3. STM32F103xx family

<table>
<tr><th rowspan="3">Pinout</th><th colspan="2">Low-density devices</th><th colspan="2">Medium-density devices</th><th colspan="3">High-density devices</th></tr>
<tr><th>16 KB Flash</th><th>32 KB Flash</th><th>64 KB Flash</th><th>128 KB Flash</th><th>256 KB Flash</th><th>384 KB Flash</th><th>512 KB Flash</th></tr>
<tr><th>6 KB RAM</th><th>10 KB RAM</th><th>20 KB RAM</th><th>20 KB RAM</th><th>48 KB RAM</th><th>64 KB RAM</th><th>64 KB RAM</th></tr>
<tr><td>144</td><td>-</td><td>-</td><td>-</td><td>-</td><td colspan="3" rowspan="2">5 × USARTs
4 × 16-bit timers, 2 × basic timers
3 × SPIs, 2 × I^2Ss, 2 × I2Cs
USB, CAN, 2 × PWM timers
3 × ADCs, 2 × DACs, 1 × SDIO
FSMC (100 and 144 pins)</td></tr>
<tr><td>100</td><td>-</td><td>-</td><td colspan="3" rowspan="3">3 × USARTs
3 × 16-bit timers
2 × SPIs, 2 × I^2Cs, USB, CAN, 1 × PWM timer
2 × ADCs</td></tr>
<tr><td>64</td><td colspan="2" rowspan="3">2 × USARTs
2 × 16-bit timers
1 × SPI, 1 × I^2C, USB, CAN, 1 × PWM timer
2 × ADCs</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>48</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>36</td><td>-</td><td>-</td><td>-</td></tr>
</table>

2.3 Overview

2.3.1 ARM® Cortex®-M3 core with embedded Flash and SRAM

The ARM® Cortex®-M3 processor is the latest generation of ARM processors for embedded systems. It has been developed to provide a low-cost platform that meets the needs of MCU implementation, with a reduced pin count and low-power consumption, while delivering outstanding computational performance and an advanced system response to interrupts.

The ARM® Cortex®-M3 32-bit RISC processor features exceptional code-efficiency, delivering the high-performance expected from an ARM core in the memory size usually associated with 8- and 16-bit devices.

The STM32F103xx performance line family having an embedded ARM core, is therefore compatible with all ARM tools and software.

Figure 1 shows the general block diagram of the device family.

2.3.2 Embedded Flash memory

64 or 128 Kbytes of embedded Flash is available for storing programs and data.

2.3.3 CRC (cyclic redundancy check) calculation unit

The CRC (cyclic redundancy check) calculation unit is used to get a CRC code from a 32-bit data word and a fixed generator polynomial.

Among other applications, CRC-based techniques are used to verify data transmission or storage integrity. In the scope of the EN/IEC 60335-1 standard, they offer a means of verifying the Flash memory integrity. The CRC calculation unit helps compute a signature of the software during runtime, to be compared with a reference signature generated at link-time and stored at a given memory location.

2.3.4 Embedded SRAM

Twenty Kbytes of embedded SRAM accessed (read/write) at CPU clock speed with 0 wait states.

2.3.5 Nested vectored interrupt controller (NVIC)

The STM32F103xx performance line embeds a nested vectored interrupt controller able to handle up to 43 maskable interrupt channels (not including the 16 interrupt lines of Cortex®-M3) and 16 priority levels.

- Closely coupled NVIC gives low-latency interrupt processing
- Interrupt entry vector table address passed directly to the core
- Closely coupled NVIC core interface
- Allows early processing of interrupts
- Processing of *late arriving* higher priority interrupts
- Support for tail-chaining
- Processor state automatically saved
- Interrupt entry restored on interrupt exit with no instruction overhead

This hardware block provides flexible interrupt management features with minimal interrupt latency.

2.3.6 External interrupt/event controller (EXTI)

The external interrupt/event controller consists of 19 edge detector lines used to generate interrupt/event requests. Each line can be independently configured to select the trigger event (rising edge, falling edge, both) and can be masked independently. A pending register maintains the status of the interrupt requests. The EXTI can detect an external line with a pulse width shorter than the Internal APB2 clock period. Up to 80 GPIOs can be connected to the 16 external interrupt lines.

2.3.7 Clocks and startup

System clock selection is performed on startup, however the internal RC 8 MHz oscillator is selected as default CPU clock on reset. An external 4-16 MHz clock can be selected, in which case it is monitored for failure. If failure is detected, the system automatically switches back to the internal RC oscillator. A software interrupt is generated if enabled. Similarly, full interrupt management of the PLL clock entry is available when necessary (for example on failure of an indirectly used external crystal, resonator or oscillator).

Several prescalers allow the configuration of the AHB frequency, the high-speed APB (APB2) and the low-speed APB (APB1) domains. The maximum frequency of the AHB and the high-speed APB domains is 72 MHz. The maximum allowed frequency of the low-speed APB domain is 36 MHz. See *Figure 2* for details on the clock tree.

2.3.8 Boot modes

At startup, boot pins are used to select one of three boot options:

- Boot from User Flash
- Boot from System Memory
- Boot from embedded SRAM

The boot loader is located in System Memory. It is used to reprogram the Flash memory by using USART1. For further details please refer to AN2606.

2.3.9 Power supply schemes

- V_{DD} = 2.0 to 3.6 V: external power supply for I/Os and the internal regulator. Provided externally through V_{DD} pins.
- V_{SSA}, V_{DDA} = 2.0 to 3.6 V: external analog power supplies for ADC, reset blocks, RCs and PLL (minimum voltage to be applied to V_{DDA} is 2.4 V when the ADC is used). V_{DDA} and V_{SSA} must be connected to V_{DD} and V_{SS}, respectively.
- V_{BAT} = 1.8 to 3.6 V: power supply for RTC, external clock 32 kHz oscillator and backup registers (through power switch) when V_{DD} is not present.

For more details on how to connect power pins, refer to *Figure 14: Power supply scheme*.

2.3.10 Power supply supervisor

The device has an integrated power-on reset (POR)/power-down reset (PDR) circuitry. It is always active, and ensures proper operation starting from/down to 2 V. The device remains

in reset mode when V_{DD} is below a specified threshold, $V_{POR/PDR}$, without the need for an external reset circuit.

The device features an embedded programmable voltage detector (PVD) that monitors the V_{DD}/V_{DDA} power supply and compares it to the V_{PVD} threshold. An interrupt can be generated when V_{DD}/V_{DDA} drops below the V_{PVD} threshold and/or when V_{DD}/V_{DDA} is higher than the V_{PVD} threshold. The interrupt service routine can then generate a warning message and/or put the MCU into a safe state. The PVD is enabled by software.

Refer to *Table 11: Embedded reset and power control block characteristics* for the values of $V_{POR/PDR}$ and V_{PVD}.

2.3.11 Voltage regulator

The regulator has three operation modes: main (MR), low-power (LPR) and power down.

- MR is used in the nominal regulation mode (Run)
- LPR is used in the Stop mode
- Power down is used in Standby mode: the regulator output is in high impedance: the kernel circuitry is powered down, inducing zero consumption (but the contents of the registers and SRAM are lost)

This regulator is always enabled after reset. It is disabled in Standby mode, providing high impedance output.

2.3.12 Low-power modes

The STM32F103xx performance line supports three low-power modes to achieve the best compromise between low-power consumption, short startup time and available wakeup sources:

- **Sleep** mode

  In Sleep mode, only the CPU is stopped. All peripherals continue to operate and can wake up the CPU when an interrupt/event occurs.

- **Stop** mode

  The Stop mode achieves the lowest power consumption while retaining the content of SRAM and registers. All clocks in the 1.8 V domain are stopped, the PLL, the HSI RC and the HSE crystal oscillators are disabled. The voltage regulator can also be put either in normal or in low-power mode.
  The device can be woken up from Stop mode by any of the EXTI line. The EXTI line source can be one of the 16 external lines, the PVD output, the RTC alarm or the USB wakeup.

- **Standby** mode

  The Standby mode is used to achieve the lowest power consumption. The internal voltage regulator is switched off so that the entire 1.8 V domain is powered off. The PLL, the HSI RC and the HSE crystal oscillators are also switched off. After entering Standby mode, SRAM and register contents are lost except for registers in the Backup domain and Standby circuitry.

  The device exits Standby mode when an external reset (NRST pin), an IWDG reset, a rising edge on the WKUP pin, or an RTC alarm occurs.

Note: The RTC, the IWDG, and the corresponding clock sources are not stopped by entering Stop or Standby mode.

2.3.13 DMA

The flexible 7-channel general-purpose DMA is able to manage memory-to-memory, peripheral-to-memory and memory-to-peripheral transfers. The DMA controller supports circular buffer management avoiding the generation of interrupts when the controller reaches the end of the buffer.

Each channel is connected to dedicated hardware DMA requests, with support for software trigger on each channel. Configuration is made by software and transfer sizes between source and destination are independent.

The DMA can be used with the main peripherals: SPI, I^2C, USART, general-purpose and advanced-control timers TIMx and ADC.

2.3.14 RTC (real-time clock) and backup registers

The RTC and the backup registers are supplied through a switch that takes power either on V_{DD} supply when present or through the V_{BAT} pin. The backup registers are ten 16-bit registers used to store 20 bytes of user application data when V_{DD} power is not present.

The real-time clock provides a set of continuously running counters which can be used with suitable software to provide a clock calendar function, and provides an alarm interrupt and a periodic interrupt. It is clocked by a 32.768 kHz external crystal, resonator or oscillator, the internal low-power RC oscillator or the high-speed external clock divided by 128. The internal low-power RC has a typical frequency of 40 kHz. The RTC can be calibrated using an external 512 Hz output to compensate for any natural crystal deviation. The RTC features a 32-bit programmable counter for long-term measurement using the Compare register to generate an alarm. A 20-bit prescaler is used for the time base clock and is by default configured to generate a time base of 1 second from a clock at 32.768 kHz.

2.3.15 Timers and watchdogs

The medium-density STM32F103xx performance line devices include an advanced-control timer, three general-purpose timers, two watchdog timers and a SysTick timer.

Table 4 compares the features of the advanced-control and general-purpose timers.

Table 4. Timer feature comparison

Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
TIM1	16-bit	Up, down, up/down	Any integer between 1 and 65536	Yes	4	Yes
TIM2, TIM3, TIM4	16-bit	Up, down, up/down	Any integer between 1 and 65536	Yes	4	No

Advanced-control timer (TIM1)

The advanced-control timer (TIM1) can be seen as a three-phase PWM multiplexed on 6 channels. It has complementary PWM outputs with programmable inserted dead-times. It can also be seen as a complete general-purpose timer. The 4 independent channels can be used for

- Input capture
- Output compare
- PWM generation (edge- or center-aligned modes)
- One-pulse mode output

If configured as a general-purpose 16-bit timer, it has the same features as the TIMx timer. If configured as the 16-bit PWM generator, it has full modulation capability (0-100%).

In debug mode, the advanced-control timer counter can be frozen and the PWM outputs disabled to turn off any power switch driven by these outputs.

Many features are shared with those of the general-purpose TIM timers which have the same architecture. The advanced-control timer can therefore work together with the TIM timers via the Timer Link feature for synchronization or event chaining.

General-purpose timers (TIMx)

There are up to three synchronizable general-purpose timers embedded in the STM32F103xx performance line devices. These timers are based on a 16-bit auto-reload up/down counter, a 16-bit prescaler and feature 4 independent channels each for input capture/output compare, PWM or one-pulse mode output. This gives up to 12 input captures/output compares/PWMs on the largest packages.
The general-purpose timers can work together with the advanced-control timer via the Timer Link feature for synchronization or event chaining. Their counter can be frozen in debug mode. Any of the general-purpose timers can be used to generate PWM outputs. They all have independent DMA request generation.

These timers are capable of handling quadrature (incremental) encoder signals and the digital outputs from 1 to 3 hall-effect sensors.

Independent watchdog

The independent watchdog is based on a 12-bit downcounter and 8-bit prescaler. It is clocked from an independent 40 kHz internal RC and as it operates independently of the main clock, it can operate in Stop and Standby modes. It can be used either as a watchdog to reset the device when a problem occurs, or as a free-running timer for application timeout management. It is hardware- or software-configurable through the option bytes. The counter can be frozen in debug mode.

Window watchdog

The window watchdog is based on a 7-bit downcounter that can be set as free-running. It can be used as a watchdog to reset the device when a problem occurs. It is clocked from the main clock. It has an early warning interrupt capability and the counter can be frozen in debug mode.

SysTick timer

This timer is dedicated for OS, but could also be used as a standard downcounter. It features:

- A 24-bit downcounter
- Autoreload capability
- Maskable system interrupt generation when the counter reaches 0
- Programmable clock source

2.3.16 I²C bus

Up to two I²C bus interfaces can operate in multimaster and slave modes. They can support standard and fast modes.

They support dual slave addressing (7-bit only) and both 7/10-bit addressing in master mode. A hardware CRC generation/verification is embedded.

They can be served by DMA and they support SM Bus 2.0/PM Bus.

2.3.17 Universal synchronous/asynchronous receiver transmitter (USART)

One of the USART interfaces is able to communicate at speeds of up to 4.5 Mbit/s. The other available interfaces communicate at up to 2.25 Mbit/s. They provide hardware management of the CTS and RTS signals, IrDA SIR ENDEC support, are ISO 7816 compliant and have LIN Master/Slave capability.

All USART interfaces can be served by the DMA controller.

2.3.18 Serial peripheral interface (SPI)

Up to two SPIs are able to communicate up to 18 Mbits/s in slave and master modes in full-duplex and simplex communication modes. The 3-bit prescaler gives 8 master mode frequencies and the frame is configurable to 8 bits or 16 bits. The hardware CRC generation/verification supports basic SD Card/MMC modes.

Both SPIs can be served by the DMA controller.

2.3.19 Controller area network (CAN)

The CAN is compliant with specifications 2.0A and B (active) with a bit rate up to 1 Mbit/s. It can receive and transmit standard frames with 11-bit identifiers as well as extended frames with 29-bit identifiers. It has three transmit mailboxes, two receive FIFOs with 3 stages and 14 scalable filter banks.

2.3.20 Universal serial bus (USB)

The STM32F103xx performance line embeds a USB device peripheral compatible with the USB full-speed 12 Mbs. The USB interface implements a full-speed (12 Mbit/s) function interface. It has software-configurable endpoint setting and suspend/resume support. The dedicated 48 MHz clock is generated from the internal main PLL (the clock source must use a HSE crystal oscillator).

2.3.21 GPIOs (general-purpose inputs/outputs)

Each of the GPIO pins can be configured by software as output (push-pull or open-drain), as input (with or without pull-up or pull-down) or as peripheral alternate function. Most of the GPIO pins are shared with digital or analog alternate functions. All GPIOs are high current-capable.

The I/Os alternate function configuration can be locked if needed following a specific sequence in order to avoid spurious writing to the I/Os registers.

I/Os on APB2 with up to 18 MHz toggling speed.

2.3.22 ADC (analog-to-digital converter)

Two 12-bit analog-to-digital converters are embedded into STM32F103xx performance line devices and each ADC shares up to 16 external channels, performing conversions in single-shot or scan modes. In scan mode, automatic conversion is performed on a selected group of analog inputs.

Additional logic functions embedded in the ADC interface allow:

- Simultaneous sample and hold
- Interleaved sample and hold
- Single shunt

The ADC can be served by the DMA controller.

An analog watchdog feature allows very precise monitoring of the converted voltage of one, some or all selected channels. An interrupt is generated when the converted voltage is outside the programmed thresholds.

The events generated by the general-purpose timers (TIMx) and the advanced-control timer (TIM1) can be internally connected to the ADC start trigger, injection trigger, and DMA trigger respectively, to allow the application to synchronize A/D conversion and timers.

2.3.23 Temperature sensor

The temperature sensor has to generate a voltage that varies linearly with temperature. The conversion range is between 2 V < V_{DDA} < 3.6 V. The temperature sensor is internally connected to the ADC12_IN16 input channel which is used to convert the sensor output voltage into a digital value.

2.3.24 Serial wire JTAG debug port (SWJ-DP)

The ARM SWJ-DP Interface is embedded. and is a combined JTAG and serial wire debug port that enables either a serial wire debug or a JTAG probe to be connected to the target. The JTAG TMS and TCK pins are shared with SWDIO and SWCLK, respectively, and a specific sequence on the TMS pin is used to switch between JTAG-DP and SW-DP.

PART III

SESSIONAL QUESTION PAPER

DHARMSINH DESAI UNIVERSITY, NADIAD
FACULTY OF TECHNOLOGY
THIRD SESSIONALEXAMINATION
SUBJECT: (EC718) EMBEDDED SYSTEMS

Examination	**:** B. TECH. SEMESTER VII [EC]	**Seat No**	**: _____**
Date	**: 11/9/2019**	**Day**	**:** Friday
Time	**: 1:45 pm to 3:00 pm**	**Max. Marks**	**:** 36

INSTRUCTIONS:

1. Figures to the right indicate maximum marks for that question.
2. Assume suitable data, if required & mention them clearly.
3. Draw neat sketches wherever necessary.

Q.1 Do as directed. **[6]**

(A) Choose the most appropriate alternate (s)

i) Non preemptive scheduling algorithms are (1)
a) easy to implement b) suitable for general purpose systems with multiple users
c) suitable for interactive users d) none

ii) Identify block device(s) from the following (1)
a) Printer b) network interface c) mice d) disk

iii) Providing device independent block size is function of (1)
a) Interrupt handler b) device driver c) device independent OS software
d) user level software

iv) Which of the followings dominates hard disk access? (1)
a) Rotational delay b) data transfer c) seek time d) DMA

(B) Match the following with respect to Bankers algorithm for avoiding deadlock (2)

a)Banker	p)Resource
b)Customer	q)Assigning a resource
c)Money	r)Releasing a resource
d) Giving loan	s)Scheduler
	t)Process

Q.2 Attempt the following **[12]**

(A) i) Can a generic OS be used in a real time system if it is equipped with real time scheduling algorithm? Explain your answer (3)

ii)A scheduler uses multiple queues for priority classes and offers double quanta whenever a process moves down to the lower class. Compare performance of this scheme with round robin algorithm for a process that requires total 200 quanta to complete. (3)

(B) i) Show how global numbering of all resources in a system can be used to avoid deadlock. (2)

ii) List strategies for dealing with deadlocks (2)

iii)What are RAM disks and their uses? (2)

OR

Q.2 Attempt the following **[12]**

(A) While seek to cylinder 5 is in progress, new requests come in for cylinders 5, 26, 16, 37, 9, and 18. Find number of cylinders the arm have to travel for following algorithms a) FCFS b) SSF c) Elevator. Discuss the situation in which goals of minimum response time and fairness are in conflict for SSF algorithm (6)

(B) List the layers of I/O system? Discuss the main function of each layer. (6)

Q.3 Do as directed. **[6]**

(A) Choose the most appropriate alternate (s): (1)

A typical operating system running on the Cortex-M processor will use _____exception to allow user thread to make calls to the OS API.
(a) PendSV (b) SysTick (c) SVC (d) all of above (e) none of above

(B) Determine the role of following code. (1)

```
x=1;
__set_CONTROL (x);
```

(C) Differentiate the CMSIS functions *__enable_irq* and *__enable_fault_irq*.based on their functionality. (2)

(D) "The supervisor calls provide interfaces between the operating system and the system process". State true/false with justifications. (2)

Q.4 Attempt any two **[12]**

(A) Determine the role of each of following codes with reference to fault handler of SVC exception. (6)

(1)
```
LDR   R12, [LR,R12,LSL#2]
BLX   R12
```
(2)
```
BICS  R12,R12,#0xFF00
```
(3)
```
BX    LR
```
(4)
```
TST   LR,#4
MRSNE R12,PSP
MOVEQ R12,SP
```

(B) **(i)** How to trigger a SysTick exception at a rate of 1KHz using CMSIS function? Assume that the system clock frequency is of *30MHz*. (2)

(ii) Convert the following arithmetic function from standard functions to software interrupt functions. (2)

```
res  = add (40, 23);
```

(iii) How CMSIS functions are advantageous to Cortex-M processor series? (2)

(C) (6)

```
void __svc(0) systemCode (void);
void __SVC_0 (void) {
unsigned int i;
unsigned int pend;
for(i=0; i<100;i++); //Time critical part
pend = NVIC_GetPendingIRQ(SysTick_IRQn);
if(pend==1)
        {          SCB->ICSR |=1<<28;          }
else
         {         for(i=0; i<100;i++);          } //Time non- critical part
                    }
int main (void) {
NVIC_SetPriority(PendSV_IRQn,2);
SysTick_Config(100);
NVIC_SetPriority(SysTick_IRQn,1);
systemCode();   }
```

Answer the followings with reference to above code.

(i) During execution of *systemCode* function, which exception is in active state?

(ii) After execution of *systemCode* function, fault handler of which exception will be triggered?

(ii) What happens if priority levels of PendSV and SysTick exceptions are interchanged?

(iii) What is the use of code SCB->ICSR |=1<<28?

(iv) What is the use of *pend* variable in above code.