USING I2C COMMUNICATION PROTOCOL WITH THE STM32F051C6 MICROCONTROLLER

****

I2C PINS ON THE STM32F051C6

****

	SCL PIN	SDA PIN
I2C1	PB6	PB7
	PB8	PB9
I2C2	PB10	PB11
	PF6	PF7

****

****

LIBRARIES

****

```
#include “stm32f0xx.h”
```

****

****

SYMBOLIC VARIABLES

****

```
#define DEVICE_ID 0b1101000
#define TIMING_REG_VALUE 0x…
#define WRITE ~I2C_CR2_RD_WRN
#define READ I2C_CR2_RD_WRN
```

****

```
*****************************************************************************************************
FUNCTION DECLARATIONS
*****************************************************************************************************
/* Initialise the SDA and SCL pins on the STM32F051C6 */
void initGPIO_for_I2C(void);

/*
  Setup I2C for selected pins
  @ timing_reg_value = the value to which the I2C timing register of the selected pins must be
                       set to
  @ deviceID = the unique I2C address of the SLAVE device
*/
void init_I2C(int device_address);

/*
  Select mode(write or read), indicate number of bytes(registers) to write to or read from and
  establish START condition
*/
void START_I2C( uint32_t mode, uint8_t data_length_in_bytes);

/*
  Send data to the SLAVE device
  @ bytes = number of bytes to write to/read from register on the SLAVE
  @ register_address = address of 1st register(byte) on the sequence of registers that we
                       want to write to or read from  OR just the address of the
                       register we want to write to and/or read from in the case of one
*/
void I2C_write( uint8_t bytes, uint8_t register_address);

/*
  Receive data from the SLAVE device
  @ bytes = expected number of bytes to receive from SLAVE
*/
uint8_t I2C_read( uint8_t bytes );

/*
  Stop I2C communication between MASTER and SLAVE
*/
void STOP_I2C(void);
*****************************************************************************************************
```

```
*****************************************************************************************
FUNCTION DEFINITIONS
*****************************************************************************************
void initGPIO_for_I2C(void)
{
/* connect clock signal to port(s) of the SDA and SCL pins */
RCC -> AHBENR |= RCC_AHBENR_GPIOxEN;

/* Set the SDA and SCL pins to ALT FUNCTION MODE */
GPIOx -> MODER &= ~GPIO_MODER_MODERx_0;
GPIOx -> MODER |= GPIO_MODER_MODERx_1;

/* Set the Output Type Register of the SDA and SCL pins to open drain */
GPIOx -> OTYPER |= GPIO_OTYPER_OT_x | GPIO_OTYPER_OT_x;

/* Set the Output Speed Register for the SDA and SCL pins to low speed */
GPIOx -> OSPEEDR &= ~(GPIO_OSPEEDR_OSPEEDRx_0 | GPIO_OSPEEDR_OSPEEDRx_01);

/* No internal PU or PD resistors for the SDA and SCL pins needed */
GPIOx -> PUPDR &= ~(GPIO_PUPDR_PUPDRx | GPIO_PUPDR_PUPDRx);

/* Select ALT FUNCTIONS for SDA and SCL pins through the GPIOx Alternate Function Register */
GPIOx -> AFR[x] |= …;
GPIOx -> AFR[x] &= ~(…);
}

void init_I2C(int device_address)
{
/* Enable I2C for selected pins in the APB register*/
RCC -> APB1ENR |= RCC_APB1ENR_I2CxEN;

/* Configure timing for the I2C. Use the AN4235 xls file to compute the timing register value*/
I2Cx -> TIMINGR = (uint32_t) TIMING_REG_VALUE;

/* Enable the peripheral for I2C */
I2Cx -> CR1 |= I2C_CR1_PE;
```

Note: When PE=0, the I2C SCL and SDA lines are released. Internal state machines and status bits are put back to their reset value. When cleared, PE must be kept low for at least 3 APB clock cycles.

```
/* Set I2C address of slave device(s) */
I2Cx -> CR2 |= (device_address << 1);
}
```

```
void START_I2C( uint32_t mode, uint8_t data_length_in_bytes)
{
/*
  Configure direction of data transfer.
  Set the RD_WRN bit in the CR2 to configure direction of data transfer.
*/
I2Cx -> CR2 |= mode; // write mode = I2C_CR2_RD_WRN and read mode = ~I2C_CR2_RD_WRN

/* Indicate number of bytes of the register address sequence */
I2Cx -> CR2 |= (data_length_in_bytes << 16);

/*
   Open comm channels by setting the START bit equal to 1 in the CR2 to generate a start
   or restart and wait until start generation is complete.
*/
I2Cx -> CR2 |= I2C_CR2_START;
while( I2Cx -> CR2 & I2C_CR2_START );
}

void I2C_write( uint8_t bytes, uint8_t register_address)
{
START_I2C(WRITE, bytes);
/*
   Send the byte(s) to SLAVE through the Transmit Data Register and wait until the
   Transmit Data Empty(TXE) bit is set to 1 indicating data transfer is complete
*/
I2Cx -> TXDR = register_address;
while( !(I2Cx -> ISR & IC2_ISR_TXE) );
}

uint8_t I2C_read( uint8_t bytes )
{
START_I2C( READ , uint8_t bytes);
/*
  wait for the Receive_data_register Not Empty(RXNE) bit in the ISR to be
  set to 0 before copying the received byte into the Receive Data Register
  because the received previous byte(s) is(are) not yet read
*/
while( !(I2Cx -> ISR & I2C_ISR_RXNE) );

/* Read and return the received data in the Receive Data Register
   The received data is from the registers whose addresses were wrote to by the TXDR*/
return I2Cx -> RXDR;
}
```

```
void STOP_I2C(void)
{
/* Initiate a STOP condition and wait until it is confirmed */
I2Cx -> CR2 |= I2C_CR2_STOP;
while( I2Cx -> CR2 & I2C_CR2_STOP );
}
*****************************************************************************************
```