dsPIC33C I^{2}C Software Library

1.0 OVERVIEW

This library provides a software implementation of an I^{2}C interface for dsPIC33C using standard I/Os. This library supports both Master and Slave modes. The performance and resources needed are listed in Table 1. The software I^{2}C Master code is blocking, which means that the Master I^{2}C procedures consume all CPU time when executed. The software I^{2}C Master task should be implemented as a low priority process interrupted by other high priority tasks in the application.

For the software I^{2}C Slave, one edge processing takes about 67 instruction cycles. In this case, when the nested interrupts are disabled, other interrupts may be delayed by 67 cycles.

> **Note:** **This software library should be used with the physical I2Cx pins (SCLx and SDAx) without enabling the respective I2Cx module. Configure the Port pins multiplexed with the I2Cx module to use I/O pins for Software I^{2}C implementation.**

TABLE 1: PERFORMANCE AND RESOURCES

Mode	Max Speed	Program (bytes)	RAM (bytes)	Instruction Cycles per Byte	Instruction Cycles per SCL Clock Edge
Master	1 MHz	290	0	—	—
Slave	650 kHz	390	14	1950	67

2.0 I^{2}C MASTER INTERFACE

2.1 Files

The files in the table below must be added to the application project to implement the Master I^{2}C interface:

File	Description
i2c_master.h	This header contains I/O definitions/selection and timing/clock speed settings used for the Master I^{2}C interface. This file also includes prototypes of Master I^{2}C functions.
i2c_master.c	This source file contains Master I^{2}C function implementations.

The library package includes demo projects.

The demo project located in the **master_demo** folder shows how to use the Master I^{2}C library functions to access EEPROM 24FC256. The files related to this demo are:

File	Description
i2c_master_eeprom_24fc256.h	This header contains prototypes of functions to access 24FC256 EEPROM.
i2c_master_eeprom_24fc256.c	This C source file contains functions implementations to access 24FC256 EEPROM.
master_demo_main.c	The main demonstration source file. It contains code to initialize the I^{2}C interface and write and read values to and from the 24FC256 EEPROM.

2.2 Library Settings

The library settings are separate for the Master and Slave. The following parameters must be configured for **I2C MASTER INTERFACE** in the **i2c_master.h** header:

File	Description
I2C_CLOCK_DELAY	This parameter is a quarter of the I^{2}C clock period in instruction cycles and defines timing for the I^{2}C interface. **Note:** I^{2}C may not work if this parameter is wrong (clock is too fast).
SCL_TRIS	This parameter sets the TRIS bit of I/O used for SCL signal. Verify that, in the application, the SCL pin is configured as a digital input in the ANSEL register.
SCL_ODC	This parameter sets the ODC bit of I/O used for SCL signal.
SCL_LAT	This parameter sets the LAT bit of I/O used for SCL signal.
SCL_PORT	This parameter sets the PORT bit of I/O used for SCL signal.
SDA_TRIS	This parameter sets the TRIS bit of I/O used for SDA signal. Verify that, in the application, the SDA pin in the application is configured as a digital input in the ANSEL register.
SDA_ODC	This parameter sets the ODC bit of I/O used for SDA signal.
SDA_LAT	This parameter sets the LAT bit of I/O used for SDA signal.
SDA_PORT	This parameter sets the PORT bit of I/O used for SDA signal.

The library does not set SCL and SDA pins input type. Setting the input type must be done in the application. SCL and SDA must be configured as digital inputs using ANSELx registers.

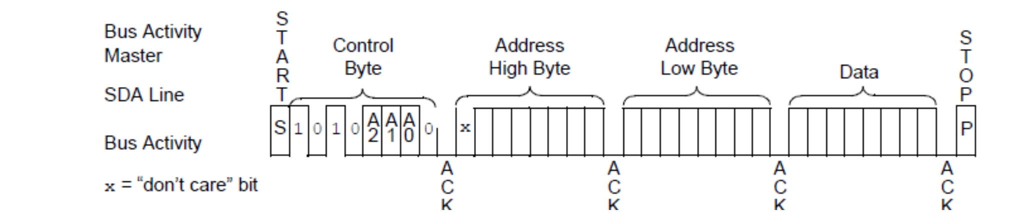
2.3 Functions and Macros

TABLE 2: MASTER FUNCTIONS AND MACROS

Function	Description	Parameters	Returned Data
`void I2CM_Init()`	This function initializes SDA and SCL I/Os.	None.	None.
`short I2CM_Start()`	This function generates an I^{2}C start signal.	None.	The function returns non-zero value if the bus collision is detected.
`short I2CM_Stop()`	This function generates an I^{2}C stop signal.	None.	The function returns non-zero value if the bus collision is detected.
`short I2CM_Write(unsigned char data)`	This function transmits 8-bit data to Slave.	unsigned char data – data to be transmitted.	This function returns acknowledgment from Slave (0 means ACK and 1 means `NACK`).
`unsigned char I2CM_Read(short ack)`	This function reads 8-bit data from Slave.	long ack – acknowledgment to be sent to Slave.	This function returns 8-bit data read.

2.4 Getting Started with Master

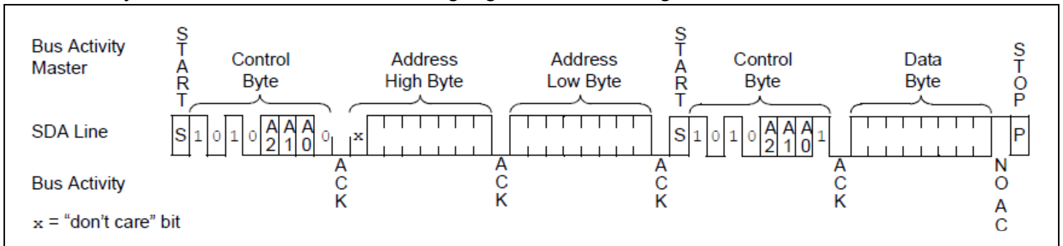
When the dsPIC33C device is communicating with 24FC256 EEPROM, the following signals should be generated to write a byte:



The signals can be generated to write a byte by using the following function calls:

```
I2CM_Start();

I2CM_Write(0xA0);

I2CM_Write(address>>8);

I2CM_Write(address&0x00FF);

I2CM_Write(data);

I2CM_Stop();
```

To read a byte from EEPROM the following signals should be generated:



The write can be done by the following functions calls:

```
I2CM_Start();

I2CM_Write(0xA0);

I2CM_Write(address>>8);

I2CM_Write(address&0x00FF);

I2CM_Start();

I2CM_Write(0xA1); // READ

data = I2CM_Read(1); // NO ACKNOWLEDGE

I2CM_Stop();
```


3.0 I^{2}C SLAVE INTERFACE

The files in the table below must be added to the application project to implement the Slave I^{2}C interface:

File	Description
i2c_slave.h	This header contains I/O definitions/selection used for the Slave I^{2}C interface. Also, this file includes prototypes of Slave I^{2}C functions.
i2c_slave.c	This C source file contains Slave I^{2}C functions implementations.

The library package includes demo projects.

The demo project located in the **slave_demo** folder shows how to use the Slave I^{2}C library functions to emulate EEPROM 24FC256. The files related to this demo are:

File	Description
i2c_slave_eeprom_24fc256.h	This C source file contains I^{2}C callback functions implementations to emulate 24FC256 EEPROM.
slave_demo_main.c	The main demonstration source file. The code in this file initializes I^{2}C Slave interface and runs I^{2}C task.

3.1 Library Settings

The following parameters must be configured for **I2C SLAVE INTERFACE** in **i2c_slave.h** header:

Parameter	Description
SCL_TRIS	This parameter sets the TRIS bit of I/O used for SCL signal. Verify that, in the application, the SCL pin is configured as a digital input in the ANSEL register.
SCL_ODC	This parameter sets the ODC bit of I/O used for SCL signal.
SCL_LAT	This parameter sets the LAT bit of I/O used for SCL signal.
SCL_PORT	This parameter sets the PORT bit of I/O used for SCL signal.
SDA_TRIS	This parameter sets the TRIS bit of I/O used for SDA signal. Verify that, in the application, the SDA pin is configured as a digital input in the ANSEL register.
SDA_ODC	This parameter sets the ODC bit of I/O used for SDA signal.
SDA_LAT	This parameter sets the LAT bit of I/O used for SDA signal.
SDA_PORT	This parameter sets the PORT bit of I/O used for SDA signal.
I2C_DISABLE_-CLOCK_STRETCHING	Add/uncomment the definition of this parameter to disable clock stretching.

The library does not set SCL and SDA pins input type. It must be done in the application. SCL and SDA must be configured as digital inputs using ANSELx registers.

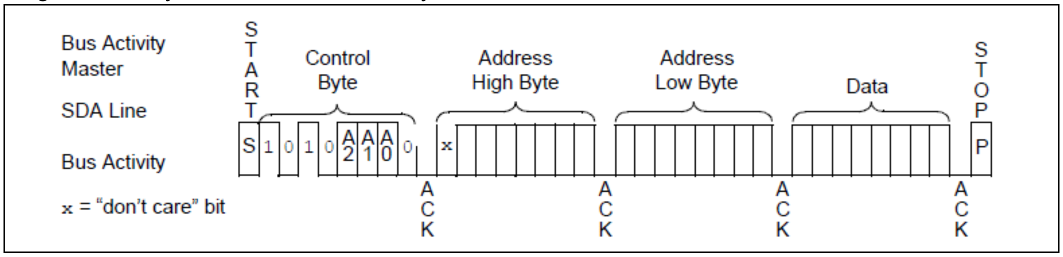
3.2 Functions and Macros

TABLE 3: SLAVE FUNCTIONS AND MACROS

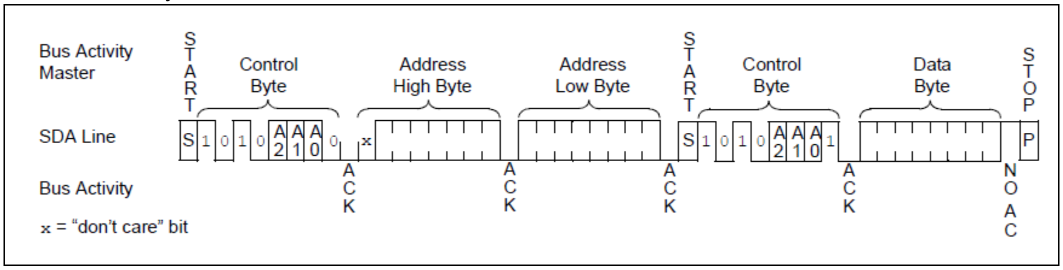
Function	Description	Parameters	Returned Data
`void I2CS_Init()`	This function initializes SDA and SCL I/Os.	None.	None.
`void I2CS_Task()`	This function is an engine to process signals on SDA and SCL I/Os. If some I²C event will be detected, this function will pass control to the corresponding callback function implemented in the user's code. The `I2CS_Task()` function must be executed periodically. It can be done by: • Change Notification interrupts on both SCL and SDA I/Os (the interrupts must detect positive and negative edges/transitions) • Calling it in main idle loop • Using a Timer interrupt	None.	None.
`void I2CS_Start()`	This is a callback function controlled by the `I2CS_Task()` function. If it is implemented in the application, it will be called each time when I²C start signal is detected.	None.	None.
`void I2CS_Stop()`	This is a callback function controlled by the `I2CS_Task()` function. If it is implemented in the application, it will be called each time when I²C stop signal is detected.	None.	None.
`short I2CS_Read(unsigned char data)`	This is a callback function controlled by the `I2CS_Task()` function. If it is implemented in the application, it will be called each time when 8-bit data are received from the I²C Master.	unsigned char data – data received from I²C Master.	Return/Pass the Acknowledgment (bit #0) and Write Mode (bit #1) flags to the library. If the data received must be acknowledged, then clear bit #0. For `NACK`, return one in bit #0. If for the next transaction, the I²C Slave must transmit data to the Master, then return/pass one in bit #1. If for the next transaction, the I²C Slave must still receive the data from Master and then clear bit #1.
`unsigned char I2CS_Write(short prev_ack)`	This is a callback function controlled by the `I2CS_Task()` function. If it is implemented in the application, it will be called each time when I²C Master requests 8-bit data from Slave.	long prev_ack – acknowledgment for the previous transaction. In most cases if the Master answered with `NACK (=1)` before, the new data are not required, and the Master will generate a stop event soon.	Return/Pass the 8-bit data to be transmitted to I2C Master.

3.3 Getting Started with Slave

Assuming that the dsPIC33C device must emulate the 24FC256 EEPROM, the following signals will be generated by the Master to write a byte:



and to read a byte:



To emulate this protocol, the I²C Slave must:

1. Detect a start condition using the `I2CS_Start()` callback function.
2. Receive the first byte after start and decode DEVICE address and read RW bit (bit #0) in the first byte.
3. If the DEVICE address matches the required address and next byte will be read (RW bit = 0), then read 2 address bytes and store data byte received using `I2CS_Read(…)` callback function.
4. If RW bit = 1, the Slave must transmit data to Master using `I2CS_Write(…)` callback function.

The EEPROM emulation protocol can be done if the callback functions are implemented as shown in Example 1.

EXAMPLE 1: SLAVE COMMUNICATION EXAMPLE

```
// 24FC256 COMMUNICATION PROTOCOL STATES
typedef enum {
    STATE_DEV_ADDRESS,                          // device address will be received
    STATE_ADDRESS_HIGH_BYTE,                    // high byte of memory address will be recieved
    STATE_ADDRESS_LOW_BYTE,                     // low byte of memory address will be recieved
    STATE_DATA_READ,                            // data byte will be read from master
    STATE_DATA_WRITE                            // data byte will be sent to master
} I2C_STATE;

// current state
I2C_STATE state = STATE_DEV_ADDRESS;

// 24FC256 device 7-bit address
#define EEPROM_DEV_ADDRESS     0x50             // from 24FC256 datasheet
#define EEPROM_SIZE            256              // 256 bytes

unsigned char eeprom_data[EEPROM_SIZE];         // memory storage
short          eeprom_address = 0;              // current memory address

// This callback function is called every time when I2C start is detected
void I2CS_Start(){
    state = STATE_DEV_ADDRESS; // after start the device address byte will be transmitted
}

// This callback function is called every time when data from I2C master are received
short I2CS_Read(unsigned char data){

    switch(state){

        case STATE_DEV_ADDRESS:
            if((data >> 1) == EEPROM_DEV_ADDRESS){ // bits from #7 to #1 are device address
                if(data&1){ // if bit #0 is set (=1) it indicates that the next data go from
                           // slave to master
                    state = STATE_DATA_WRITE;
                  return 2; // ACK to master (bit #0 = 0), master reads data on next transaction
                           //(bit #1 = 1)
              }else{ // if bit #0 is cleared (=0) it indicates that the next data go from master
to slave
                    state = STATE_ADDRESS_HIGH_BYTE;
                    return 0; // ACK to master
                }
            }
            return 1;     // NACK to master if device address doesn't match EEPROM_DEV_ADDRESS
        case STATE_ADDRESS_HIGH_BYTE:
            state = STATE_ADDRESS_LOW_BYTE;
            eeprom_address = (data<<8);
            return 0;                           // ACK to master
        case STATE_ADDRESS_LOW_BYTE:
            state = STATE_DATA_READ;
            eeprom_address |= data;
            return 0;                           // ACK to master
        case STATE_DATA_READ:
            state = STATE_DEV_ADDRESS;
            if(eeprom_address >= EEPROM_SIZE){
                return 1;                       // NACK to master, the memory address is wrong
            }
            eeprom_data[eeprom_address] = data; // store the data received
            return 0;                           // ACK to master
        default:
           state = STATE_DEV_ADDRESS;
            return 1;                           // NACK to master / unknown state
    }

    return 1;                                   // NACK to master
}

// This callback function is called every time when data must be sent to I2C master
unsigned char I2CS_Write(short prev_ack){
```

EXAMPLE 1: SLAVE COMMUNICATION EXAMPLE (CONTINUED)

```
        if(eeprom_address >= EEPROM_SIZE)
    {
        return 0;
    }
    return eeprom_data[eeprom_address];              // send memory data to master
}

void main()
{
    I2CS_Init();

    while(1){
        I2CS_Task();
    }
}
```

