Freescale Semiconductor
Application Note

Document Number: AN4803
Rev. 0, 10/2013

I²C Non-Blocking Communication

by: Matus Plachy

1 Introduction

This application note describes one of the possible ways to perform the non-blocking communication on the I²C bus. It also provides the example c-code for the I²C master.

The blocking communication means that the MCU stalls until the byte is transferred from the data register. In the code, this is usually implemented as checking of the status bit in the "while" loop. However, this approach might not be applicable for some time-critical applications. Assuming an I2C speed of 100 kHz, transmission of 1 byte will stall the CPU for 9 µs. The total time while the CPU is waiting for data transmitting or receiving is much higher because the I²C message consists of at least three bytes.

The algorithm described in this application note, enables sending whole I²C packet without waiting of the CPU in loops. Fault detection and evaluation leverages the value of this example code by introducing a more robust solution. The algorithm assumes only one master on the bus is presented, so it does not check and evaluate the loss of arbitration condition.

Contents

2 I²C bus

The Inter-Integrated Circuit (I²C, I2C, or IIC) is the serial bidirectional two-wire communication interface. Serial data (SDA) and serial clock (SCL) carry information between the devices connected to the bus. Each device is recognized by a unique address. A master is the device that initiates data transfer on the bus and generates the SCL clock that synchronizes the data transfer. At the time of transmission, any device addressed is considered as a slave.

2.1 I²C data transfer formats

The I²C byte format, timing diagrams of the SDA and SCL lines, and definition of Start, Stop, and other signals can be found in the reference manual of any microcontroller that possesses the I²C module (in the list of referenced documentation [1] or [2]). The message formats for writing and reading data to and from the slave are described in the subsequent subsections.

2.1.1 Write data message format

The message format for data transmission to a slave consists of five parts:

1. Start signal
2. Slave address transmission followed by “write” bit
3. Transmission of the register address where the data will be written
4. Data transmission (multiple)
5. Stop signal

After each byte received by the slave, the slave generates an acknowledge bit to signal the master successful completion of the byte transmission. Some I²C slave devices might allow writing multiple data. The I²C slave device features then auto-increment or decrement of previously accessed memory locations.

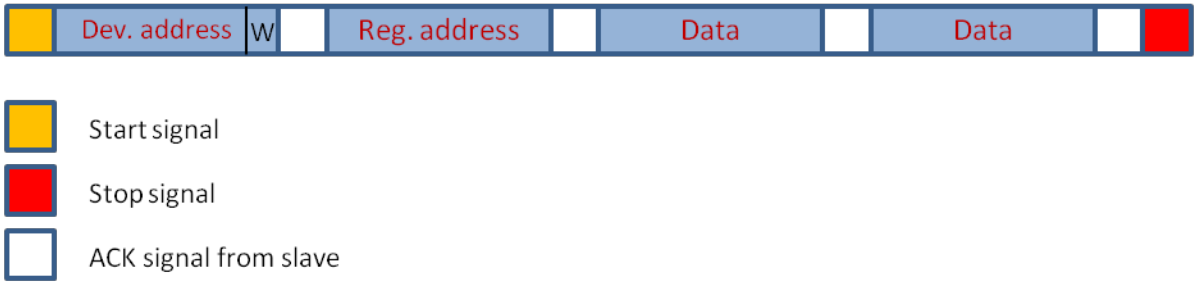

Figure 1. Write data transmission message

2.1.2 Read data message format

The message format for data transmission from the slave (the master is a receiver) in the case of combined format as defined in [3] consists of nine parts:

1. Start signal

2. Slave address transmission followed by “read” bit
3. Register address transmission from which the data will be read
4. Repeated start signal
5. Slave address transmission followed by “write” bit
6. Reading first dummy data byte from the data register (valid slave data arrives after first master ACK)
7. Data transmission from the slave to master (multiple)
8. Not acknowledge signal generation
9. Stop signal

After each byte received by the master, the master generates and acknowledge bit to signal the slave about successful completion of the byte transmission. Similarly, some I^{2}C slave devices allow reading multiple data.

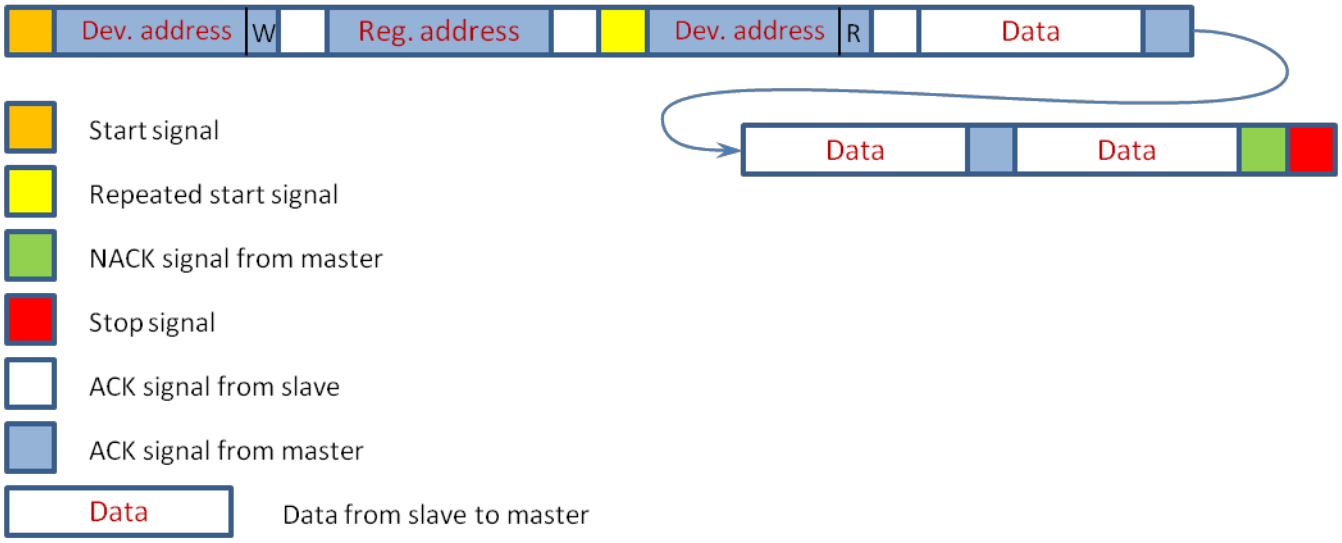

Figure 2. Read Data Transmission Message

The I^{2}C bus specification enables also another message formats. The I^{2}C slave device can support the combine message format also for data writing to a slave. Other I^{2}C messages format are described in the [3] and are not supported by the example code which is provided in this document.

3 Transmission of I^{2}C messages with non-blocking algorithm

This section describes the algorithms for the multi-byte read and write I2C messages. Because the aim is the non-blocking communication, the CPU will not wait until each byte is transferred, but the algorithm allows the application to continue the execution. After the byte is transferred and ACK signal is received, the I^{2}C module triggers the interrupt. In this interrupt the state machine of the communication driver then initiates the transfer of next byte of the I^{2}C message.

3.1 Non-blocking I²C communication driver

3.1.1 Data type structure, flags and faults

As mentioned in I²C bus section, the I²C write and read message consists from several parts. In order to identify in what phase the communication currently is, the control structure contains flags identifying current transmission stage:

```
typedef enum
{
        I2C_TRM_STAGE_NONE = 0,
        I2C_TRM_STAGE_WRITE_DATA,
        I2C_TRM_STAGE_WRITE_DEV_ADDRESS_W,
        I2C_TRM_STAGE_WRITE_DEV_ADDRESS_R,
        I2C_TRM_STAGE_WRITE_REG_ADDRESS,
        I2C_TRM_STAGE_READ_DUMMY_DATA,
        I2C_TRM_STAGE_READ_DATA,
        I2C_TRM_STAGE_NACK,
} tI2C_trm_stage;
```

The next enumeration data type is used to identify read or write message format:

```
typedef enum
{
        I2C_MODE_READ = 0,
        I2C_MODE_WRITE,
} tI2C_mode;
```

The transmission in progress is signaled by flag in the following data type:

```
typedef enum
{
        I2C_FLAG_NONE = 0,
        I2C_FLAG_TRANSMISSION_PROGRESS,
} tI2C_flag;
```

The software also checks and evaluates fault states:

```
typedef enum
{
        I2C_NO_FAULT = 0,
        I2C_BUS_BUSY,
        I2C_TIMEOUT,
        I2C_PERMANENT_BUS_FAULT,
} tI2C_fault;
```

The `I2C_BUS_BUSY` fault flag is set when attempting to send new message over the I²C bus, but the bus is in the busy state. It means that the transfer of previous message has not been completed. The I²C module sets the BUSY flag when the Start signal is detected and clears the flag when the stop signal is detected.

The `I2C_TIMEOUT` fault flag is set, when the `I2C_BUS_BUSY` fault persists more than 150 µs (number determined as double of longest I²C message). The message transfer is not completed and the bus is still in busy state because the data transmission on the I2C bus is affected by the noise. Timeout feature

prevents processor to stick for ever when this condition occurs on the I²C bus. After the timeout fault occurs, the software disables the I²C module and sends nine clock pulses followed by NACK and the stop signal. The slave should release the SDA line and the I²C module is reinitialized. Then the software tries to send the message once again. If the transfer of the message fails again, then `I2C_PERMANENT_BUS_FAULT` fault flag is set and the communication would be reenabled only by power-on reset of the slave, if another hardware fault is not preventing proper I²C communication. This procedure is not part of the example code. When the `I2C_TIMEOUT` fault flag is set, it means, the previous message also might not be transferred completely. This must be considered during initialization sequence of the slave, some registers might not be initialized correctly. So it is advisable to also send the previous message (not part of the example code).

The I²C control structure contains following variable except above listed enumeration types:

- Device address followed by write bit.
- Device address followed by read bit.
- Address of the slave register where the first byte of data will be written or read from.
- Size of the data in bytes.
- Data index which points to the current position in the `iic_data[]`read or write data array.

```
typedef struct
{
        tI2C_trm_stage          eI2C_trm_stage;
        tI2C_flag               eI2C_flag;
        tI2C_mode               eI2C_mode;
        tI2C_fault              eI2C_fault;
        unsigned char           device_address_w;
        unsigned char           device_address_r;
        unsigned char           register_address;
        unsigned char           data_size;
        unsigned char           data_index;
} tI2C_com_ctr;
```

3.1.2 Algorithm description

Before the I²C transmission starts, the control structure has to be initialized (device address for read and write, address of the target register, size of the data to be sent or received). The message transmission is initiated by calling `I2C_write` or `I2C_read` functions. These functions send the first byte of data (the slave address) to the I²C bus. The name of the variables and the flags are abbreviated in the flowcharts.

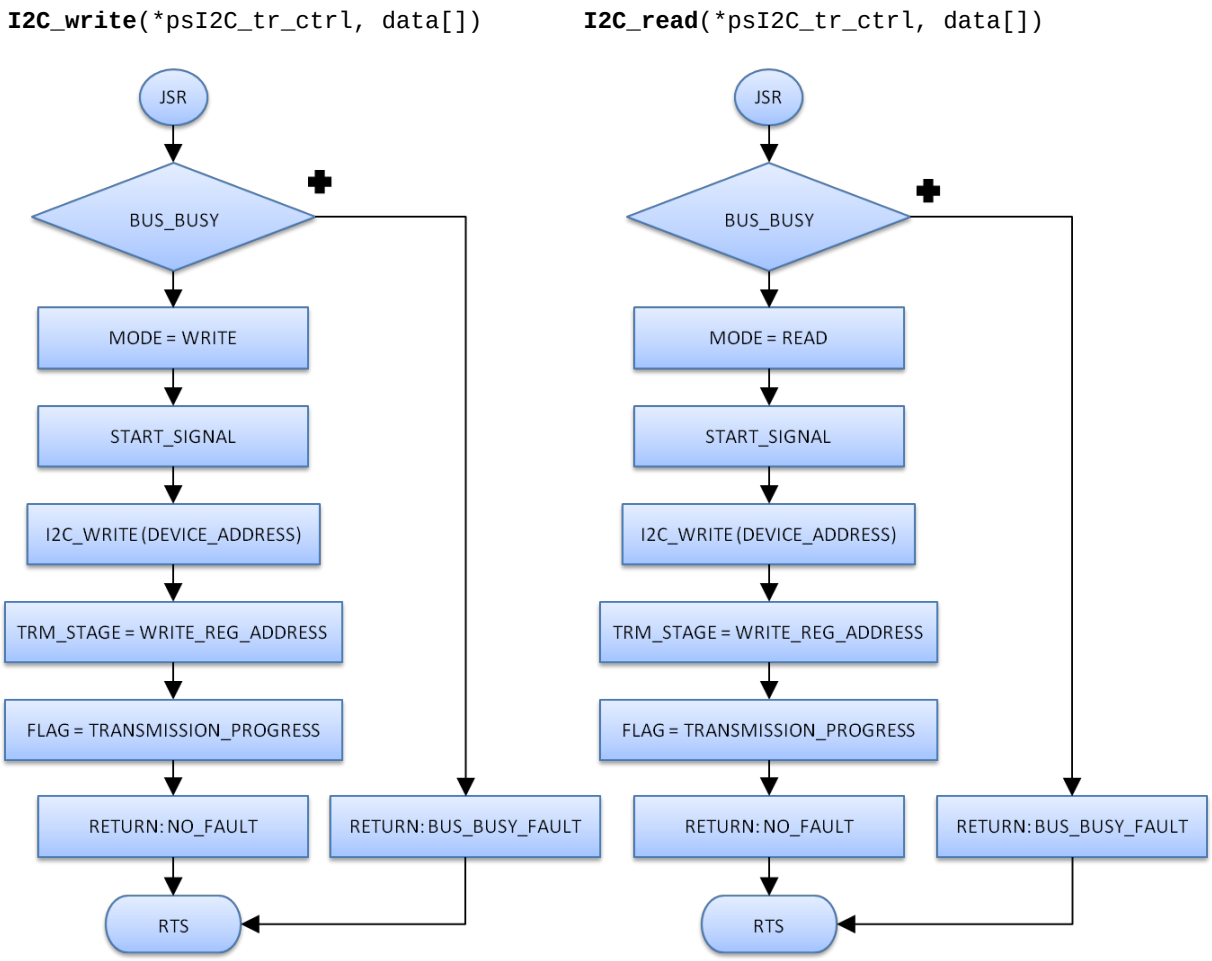

Figure 3. Flowcharts of I2C_write and I2C_read functions

The control flag `I2C_FLAG_TRANSMISSION_PROGRESS` is set at the end of `I2C_write` and `I2C_read` functions. The main application must not initiate the next message transfer until this flag is changed back to `I2C_FLAG_NONE`.

When the first byte of the message is successfully sent, the main application can continue with execution of other tasks. While attempting to send the first byte, and the I2C bus is in busy state, the return value of the `I2C_write` or `I2C_read` functions is fault flag. This not necessarily means that the I^2C bus is in fault condition, the transfer of the previous message might be still in the progress. The timeout counter is incrementing and the attempt of the data transfer is repeated. The initial value of the timeout time-down counter should be selected with respect to the size of the largest message that is transferred, but consider the value appropriately so the MCU will not wait uselessly.

If the transfer of the byte is completed and the slave confirmed the data transfer by sending the acknowledge signal to the master, the I^2C module on the MCU generates an interrupt request. In some cases in the I^2C bus, the slave do not confirm the receiving of the data by sending the acknowledge signal. The following points describe the possible conditions when no acknowledge (NACK) is generated [3]:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

Note

The last point does not represent a fault condition and is used at the end of the read data message. The generation of the NACK signal by the slave on the I^2C bus (points 1-4) is not detected by the example code. However, this state on the I^2C bus causes triggering the timeout fault, so the application can indirectly handle also this condition on the I^2C bus.

In the interrupt service routine `I2C_isr_Callback` then algorithm continues sending the bytes of the I^2C message. The algorithm is different for transmitting (write) and receiving (read) data. For the receiving data it is more complicated that results from the structure of read I^2C message.

Flowchart of the I^2C ISR Callback when the master is in transmitting mode (write data message):

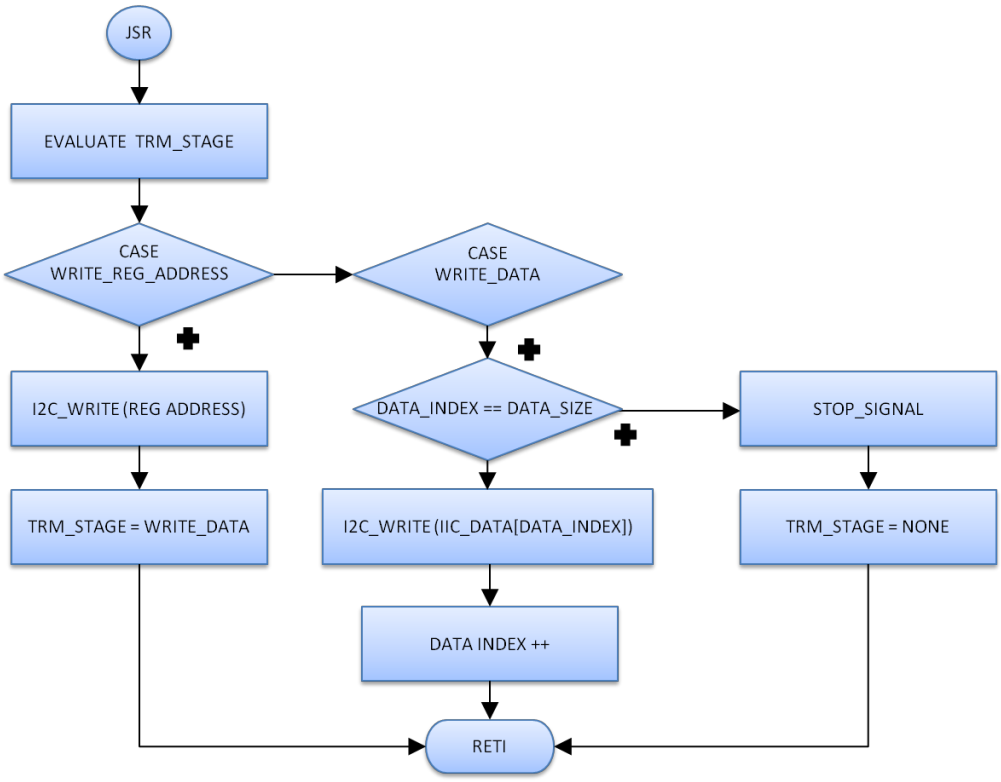

Figure 4. Flowchart of the ISR - Write Data Message

Flowchart of the I²C ISR Callback when the master is in receiving mode (read data message):

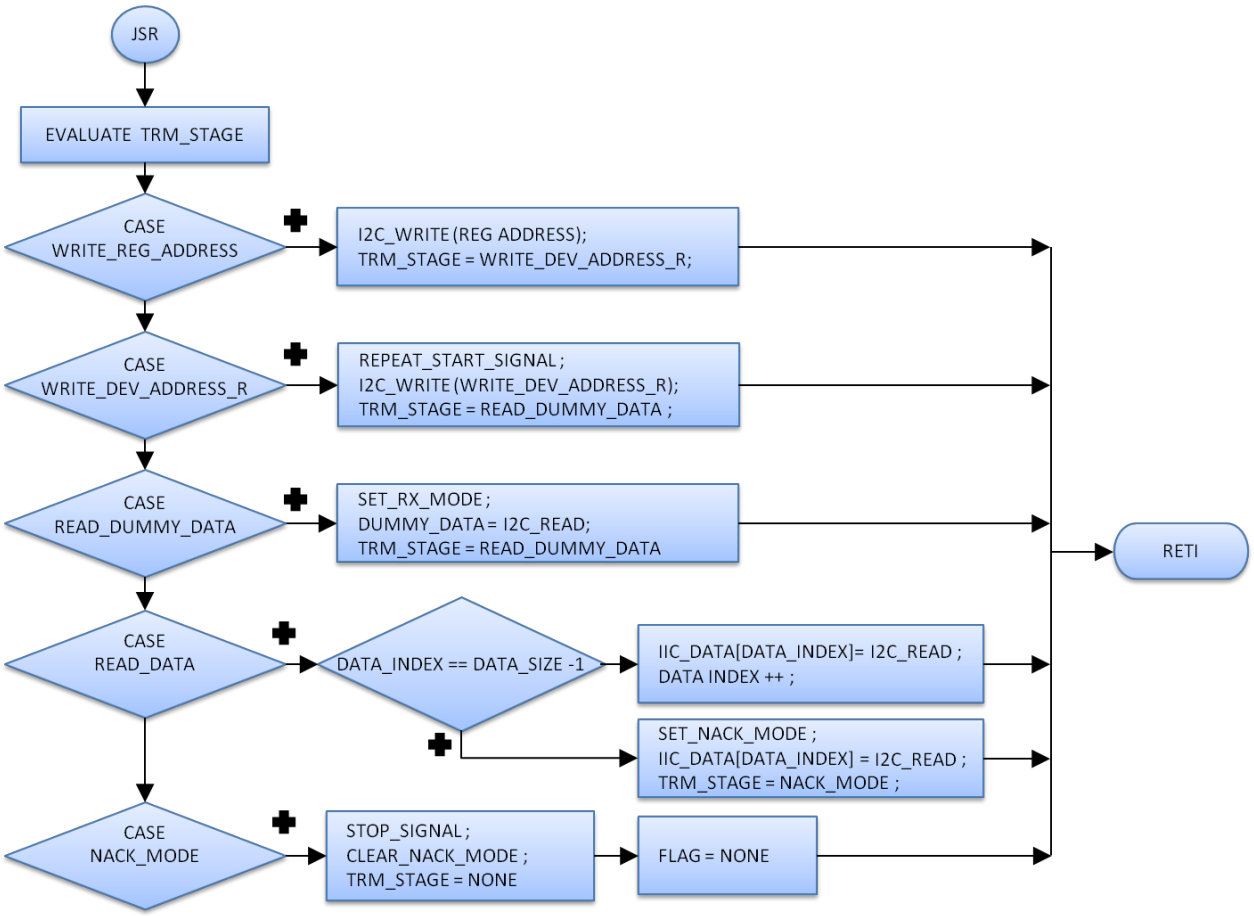

Figure 5. Flowchart of the ISR - read data message

The ISR starts with evaluation of current stage of the message transfer (TRM_STAGE) using "switch" statement. Then follows the execution of that branch, which corresponds to the actual transmission stage. At the end of each branch the TRM_STAGE variable is updated with the next stage, so the next time the ISR is executed, the algorithm passes the next branch of the "switch" statement. At the end of the last branch, the transmission control flag is changed to `I2C_FLAG_NONE`. This flag can be used for signaling to the main application that the transfer of whole I²C message was finished and the I²C bus is ready to start the transfer of the next message. In the example code the end of message transfer is evaluated by checking the BUSY status bit in the I²C module.

3.2 Example of using of the software driver

Except the algorithm that performs the non-blocking I²C communication, this document contains also example showing the way how to use it in the application. The example shows initialization and communication with some fictitious temperature sensor, so the values of addresses and registers are randomly selected. The reader should also consider the comments provided within the code example.

The example code below represents the initialization of the I²C control structure, sensor initialization, and timer interrupt routine that serves for periodical sense of the temperature from the sensor that is connected to the master MCU via I²C interface.

At first device address for read and write is defined and the control structure that is used for evaluation of the readiness of actual temperature value is defined:

```
#define SENSOR_I2C_ADDRESS 0x5A // as defined in sensor datasheet
#define IICWRITE(iicaddress) ((iicaddress<<1) & 0xFE)
#define IICREAD(iicaddress)  ((iicaddress<<1) | 0x01)
#define SENSOR_I2C_ADDRESS_W  IICWRITE( SENSOR_I2C_ADDRESS )    //0xB4
#define SENSOR_I2C_ADDRESS_R  IICREAD( SENSOR_I2C_ADDRESS )     //0xB5

typedef enum
{
        SENSOR_NO_FAULT                     = 0,
        SENSOR_INIT_FAILED,
        SENSOR_READ_FAILED,

} tSensor_fault;

typedef enum
{
        SENSOR_NO_TRANSFER                  = 0,
        SENSOR_TRANSFER_PROGRESS,
        SENSOR_TRANSFER_FINISHED,
} tSensor_data_transfer;


typedef struct
{
        tSensor_fault              eSensor_fault;
        tSensor_data_transfer      eSensor_data_transfer;
        unsigned char              measured_data;
} tSensor_com_ctr;

// variable of “tSensor_com_ctr”  type declaration:
        tSensor_com_ctr            sTMPR_com_ctr;

// array of data that are sent (received) during I2C message transfer. The size depends on
//the maximum size of the message that is sent over I2C bus (e.g. during initialization of
//the sensor
unsigned char                      iic_data[10];
```

The following function serves for initialization of the I²C control structure:

```
void TemperatureSensingInit (void)
{
   // init I2C_com_ctr structure members that are the same for all application
   sI2C_com_ctr.device_address_w = SENSOR_I2C_ADDRESS_W;
   sI2C_com_ctr.device_address_r = SENSOR_I2C_ADDRESS_R;
   // init flags before first use
   sI2C_com_ctr.eI2C_flag = I2C_FLAG_NONE;
   sI2C_com_ctr.eI2C_fault = I2C_NO_FAULT;

        if (Sensor_Init() == SENSOR_INIT_FAILED)
        {
```

```
// if the I2C bus error caused fail of the sensor init process, the error flag //is set. The
main application then should evaluate this condition
sTMPR_com_ctr.eSensor_fault = SENSOR_INIT_FAILED;
                return;
        }
        ENABLE_TIMER_INTERRUPT;
}
```

The following function provides an example of the sensor initialization at the application start-up:

```
tSensor_fault Sensor_Init (void)
{
        static unsigned int time_out_cnt = 1000;
        // clear the data array first
        for (i=0; i<10; i++)
                iic_data[i] = 0;
        //clear all registers
        sI2C_com_ctr.data_size = 10; // assuming the sensor has 10 registers
        sI2C_com_ctr.eI2C_trm_stage = I2C_TRM_STAGE_WRITE_DEV_ADDRESS_W;
        sI2C_com_ctr.register_address = 0x00;
        if (I2C_write_data(&sI2C_com_ctr, iic_data) != I2C_NO_FAULT)
                return SENSOR_INIT_FAILED;
        // Move to next command after previous message is sent
        // Here the blocking communication is used. The application has to wait anyway until
// the sensor is initialized.
        while ((( sI2C_com_ctr.eI2C_flag == I2C_FLAG_NONE) && (time_out_cnt != 0)))
        {
          time_out_cnt--
        }
        if (time_out_cnt == 0)  // if the counter counts to zero
                return SENSOR_INIT_FAILED;
        // prepare data for next message transfer
        sI2C_com_ctr.data_size = 2;
        sI2C_com_ctr.eI2C_trm_stage = I2C_TRM_STAGE_WRITE_DEV_ADDRESS_W;
        sI2C_com_ctr.register_address = 0x00;
        iic_data[0] = 0x63; // some setting
        iic_data[1] = 0x10; // some setting
        if (I2C_write_data(&sI2C_com_ctr, iic_data) != I2C_NO_FAULT)
                return SENSOR_INIT_FAILED;
        // move to next command after previous "packet" sent
        time_out_cnt = 1000;
        while (!(( sI2C_com_ctr.eI2C_flag == I2C_FLAG_NONE) && (time_out_cnt != 0)))
        {
          time_out_cnt--
        }
        if (time_out_cnt == 0) // if the counter counts to zero
                return SENSOR_INIT_FAILED;
 //… configuration continues by writing data to other registers. The initialization process
 // is divided to several messages, some registers might remained untouched, some might
 // require other registers to be written first.
return SENSOR_NO_FAULT;
}
```

Timer interrupt service routine, where the initiation of transfer of the sensed values is performed:

```
void Timer1_Isr(void)
```

```
{
 // initialization of I2C control structure
sI2C_com_ctr.register_address = 0x04; // address of the register with actual measured
      // temperature
 sI2C_com_ctr.data_size = 1;
 sI2C_com_ctr.eI2C_flag = I2C_FLAG_NONE;
 sI2C_com_ctr.eI2C_trm_stage = I2C_TRM_STAGE_WRITE_DEV_ADDRESS_W;
 if (I2C_read_data(&sI2C_com_ctr, iic_data) != I2C_NO_FAULT)
          sTMPR_com_ctr.eSensor_fault SENSOR_READ_FAILED;
 else
{
 sTMPR_com_ctr.eSensor_data_transfer = SENSOR_TRANSFER_PROGRESS;
// IMPORTANT: The SENSOR_TRANSFER_FINISHED flag has to be set in the I2C_isr_Callback
// function, when all bytes of the I2C message are transferred. The main application should
// evaluate eSensor_data_transfer flags in order to check the availability of actual
// temperature data. Then the flag should be changed to SENSOR_NO_TRANSFER.
// The value of actual temperature will be stored in first element of iic_data array
sTMPR_com_ctr.eSensor_fault = SENSOR_NO_FAULT;
}
   CLEAR_TIMER_INTERRUPT_FLAG;
}
```

3.3 Functions description

`void I2C_Init()`

Function performs initialization of the I^{2}C module on the MCU.

`void I2C_DeInit()`

Deinitialization of the I^{2}C module on the MCU, function is called from the I2C_Restore().

`void I2C_Restore()`

Reinitialization of the I^{2}C module in case time-out condition occurs on the bus.

`tI2C_fault` **`I2C_write`**`(tI2C_com_ctr *psI2C_tr_ctrl,` **`unsigned char`** `data[])`

The function initiates the transmission of the first byte of the I^{2}C write message.

`tI2C_fault` **`I2C_read`**`(tI2C_com_ctr *psI2C_tr_ctrl,` **`unsigned char`** `data[])`

The function initiates the transmission of the first byte of the I^{2}C read message, it checks the state of the bus.

`tI2C_ fault` **`I2C_write_data`**`(tI2C_com_ctr *psI2C_tr_ctrl,` **`unsigned char`** `data[])`

The function is used for initiation of the I^{2}C write message transmission from the main application. The function evaluates the *`I2C_BUS_BUSY`* fault flag (the return value of **`I2C_write`** function). In case the flag is set, then the function repeatedly calls the **`I2C_write`** function until

the time-out condition occurs. Then the I²C bus is reinitialized and the function attempts to write the data on the I²C bus again. If the time-out condition occurs again, the *I2C_PERMANENT_BUS_FAULT* fault flag is set. The main application should then evaluate this flag.

`tI2C_fault` **`I2C_read_data`**`(tI2C_com_ctr *psI2C_tr_ctrl,` **`unsigned char`** `data[])`

The function is used for initiation of the I²C write message transmission from the main application, it performs the same procedure as **`I2C_write_data`** function.

`tI2C_fault` **`I2C_isr_Callback`** `(tI2C_com_ctr *psI2C_tr_ctrl,` **`unsigned char`** `data[])`

The function executes the non-blocking algorithm of the I²C read and write messages transmission. The function algorithm description and the flowchart can be found in Algorithm description section. After the transmission of the I²C message is completed, the flag *SENSOR_TRANSFER_FINISHED* has to be set to `sTMPR_com_ctr.eSensor_data_transfer`.

`void I2C_Isr(void)`

This function is the interrupt service routine of I²C module. The interrupt is generated when all bits from the data registers are sent and the acknowledge signal is received. In this ISR the **`I2C_isr_Callback`** function is called and the interrupt flag is cleared.

4 Conclusion

This application note describes one of the possible methods of designing master side I²C non-blocking communication software driver. Algorithm described in this application note, enables to send data to a slave as well as request and receive the data from the slave. The algorithm can be used for communication between the MCU and a sensor. The write-data part of the software might be used for sensor configuration at the application startup, while the read-data part of the software driver can be used for requesting and receiving the measured data.

5 Code listing

i2c.h file

```
/*****************************************************************************
*
* Freescale Semiconductor Inc.
* (c) Copyright 2013 Freescale Semiconductor
* ALL RIGHTS RESERVED.
*
*****************************************************************************/
/*
 * @file      i2c.h
 * @author    B02785, plcm001
 * @version   1.0.2.0
 * @date      July-22-2013
 * @brief     I2C driver header file.
 *****************************************************************************/
#ifndef __I2C_H
#define __I2C_H

/*****************************************************************************
 * definition of transmission control structure                              *
 *****************************************************************************/
typedef enum
{
        I2C_TRM_STAGE_NONE                          = 0,
        I2C_TRM_STAGE_WRITE_DATA,
        I2C_TRM_STAGE_WRITE_DEV_ADDRESS_W,
        I2C_TRM_STAGE_WRITE_DEV_ADDRESS_R,
        I2C_TRM_STAGE_WRITE_REG_ADDRESS,
        I2C_TRM_STAGE_READ_DUMMY_DATA,
        I2C_TRM_STAGE_READ_DATA,
        I2C_TRM_STAGE_NAK,
} tI2C_trm_stage;   // transmission stages

typedef enum
{
        I2C_MODE_READ                       = 0,
        I2C_MODE_WRITE,
}tI2C_mode;


typedef enum
{
        I2C_FLAG_NONE                       = 0,
        I2C_FLAG_TRANSMISSION_PROGRESS,
} tI2C_flag;

typedef enum
{
        I2C_NO_FAULT                        = 0,
        I2C_BUS_BUSY,
        I2C_TIMEOUT,
        I2C_PERMANENT_BUS_FAULT,
}tI2C_fault;
```

```
typedef struct
{
        tI2C_trm_stage            eI2C_trm_stage;
        tI2C_flag                         eI2C_flag;
        tI2C_mode                         eI2C_mode;
        tI2C_fault                                eI2C_fault;
        unsigned char                     device_address_w;
        unsigned char                     device_address_r;
        unsigned char                     register_address;
        unsigned char                     data_size;
        unsigned char                     data_index;
} tI2C_com_ctr;



/*****************************************************************************
 * command definitions                                                       *
 *****************************************************************************/
/* I2C macro definitions                                                     */


#if defined (TOWER)

#define I2C_MASTER_SDA_PIN_1              GPIOF_DR |= GPIOF_DR_D_3
#define I2C_MASTER_SDA_PIN_0              GPIOF_DR &= ~GPIOF_DR_D_3
#define I2C_MASTER_SCL_PIN_1              GPIOF_DR |= GPIOF_DR_D_2
#define I2C_MASTER_SCL_PIN_0              GPIOF_DR &= ~GPIOF_DR_D_2

#define I2C_MASTER_SDA_PIN_AS_IN   GPIOF_DDR        &= ~GPIOF_DDR_DD_3
#define I2C_MASTER_SDA_PIN_AS_OUT  GPIOF_DDR        |= GPIOF_DDR_DD_3

#define I2C_MASTER_SCL_PIN_AS_IN   GPIOF_DDR        &= ~GPIOF_DDR_DD_2
#define I2C_MASTER_SCL_PIN_AS_OUT  GPIOF_DDR        |= GPIOF_DDR_DD_2

#define I2C_MASTER_SDA_PIN_AS_GPIO          GPIOF_PER &= ~GPIOF_PER_PE_2
#define I2C_MASTER_SCL_PIN_AS_GPIO          GPIOF_PER &= ~GPIOF_PER_PE_3

#define I2C_MASTER_SDA_PIN_AS_I2C           GPIOF_PER |= GPIOF_PER_PE_2
#define I2C_MASTER_SCL_PIN_AS_I2C           GPIOF_PER |= GPIOF_PER_PE_3
#endif

/*****************************************************************************//*!
 * @brief   Macro generate Start I2C signal
 * @param   module  - I2C0|I2C1
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_START_SIGNAL         (I2C_C1 |= I2C_C1_MST)

/*****************************************************************************//*!
 * @brief   Macro generate Stop I2C signal
 * @param   module  - I2C0|I2C1
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_STOP_SIGNAL         (I2C_C1 &= ~I2C_C1_MST)

/*****************************************************************************//*!
```

```
 * @brief   Macro generate Repeat Start I2C signal
 * @param   module  - I2C0|I2C1
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_REPEAT_START_SIGNAL  (I2C_C1 |= I2C_C1_RSTA)

/*****************************************************************************//*!
 * @brief   Macro Write data for transfer
 * @param   module  - I2C0|I2C1
 * @param   data    - Data for send
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_WRITE_BYTE(data)     (I2C_D = data)


/*****************************************************************************//*!
 * @brief   Macro Return data from last transfer
 * @param   module  - I2C0|I2C1
 * @return  data    - Data from last transfer
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_READ_BYTE            (unsigned char)I2C_D


/*****************************************************************************//*!
 * @brief   Macro Return Irq. flag
 * @param   module  - I2C0|I2C1
 * @return  TRUE    - Interrupt. pending    (action finished)
 * @return  FALSE   - No Interrupt. pending (action in proggress)
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_GET_IRQ_FLAG         (I2C_S & I2C_S_IICIF)


/*****************************************************************************//*!
 * @brief   Macro clear Irq. flag
 * @param   module  - I2C0|I2C1
 * @return  NONE
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_CLEAR_IRQ_FLAG       (I2C_S |= I2C_S_IICIF)


/*****************************************************************************//*!
 * @brief   Macro change I2C mode RX
 * @param   module  - I2C0|I2C1
 * @return  NONE
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_SET_RX_MODE          (I2C_C1 &= ~I2C_C1_TX)


/*****************************************************************************//*!
 * @brief   Macro change I2C mode TX
 * @param   module  - I2C0|I2C1
 * @return  NONE
 * @note    Implemented as inlined macro.
```

```
 *****************************************************************************/
#define I2C_SET_TX_MODE          (I2C_C1 |= I2C_C1_TX)


/*****************************************************************************//*!
 * @brief   Macro change I2C mode NACK
 * @param   module  - I2C0|I2C1
 * @return  NONE
 * @note    Implemented as inlined macro.
 *****************************************************************************/
#define I2C_SET_NACK_MODE        (I2C_C1 |= I2C_C1_TXAK)

/*****************************************************************************//*!
 * @brief   Macro change I2C mode back to ACK
 * @param   module  - I2C0|I2C1
 * @return  NONE
 * @note    Implemented as inlined macro.
 *****************************************************************************/

#define I2C_CLEAR_NACK_MODE        (I2C_C1 &= ~I2C_C1_TXAK)

/*****************************************************************************
 * public function prototypes                                                *
 *****************************************************************************/
extern void I2C_Isr(void);
extern tI2C_fault I2C_write_data(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char *data);
extern tI2C_fault I2C_read_data(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char *data);

/*****************************************************************************
 * local function prototypes                                                 *
 *****************************************************************************/

void I2C_Init();
void I2C_DeInit();
tI2C_fault I2C_Restore();
void I2C_delay(void);
tI2C_fault I2C_isr_Callback (tI2C_com_ctr *psI2C_tr_ctrl,unsigned char *data);


#endif /* __I2C_H */

/*****************************************************************************
 * End of module                                                             *
 *****************************************************************************/
```

i2c.c file

```
/*****************************************************************************
 * (c) Copyright 2013, Freescale Semiconductor Inc.
 * ALL RIGHTS RESERVED.
 *****************************************************************************//*!
 * @file      i2c.c
 * @author    B02785, plcm001
 * @version   1.0.4.2
 * @date      July-24-2012
 * @brief     IIC driver implementation.
```

```
 * @par       Driver example
 * @include   i2c.h
 ******************************************************************************/

#include "i2c.h"
#include "derivative.h" // header file that implements peripheral memory map
                                  // here is used MC56F82748.h

/******************************************************************************
 * global variables                                                           *
 ******************************************************************************/
unsigned char             iic_data[0x80];
tI2C_com_ctr              sI2C_com_ctr; // I2C communication control structure

/******************************************************************************
 * local variables                                                            *
 ******************************************************************************/

static volatile unsigned int timeout_cnt;

/******************************************************************************
 * macro definitions                                                          *
 ******************************************************************************/


/******************************************************************************
 * Public functions definitions                                               *
 ******************************************************************************/

/******************************************************************************
*  I2C peripheral module initialization function definition                   *
******************************************************************************/

void I2C_Init(void)
{

  I2C_F     = 0x27; // clock divider 56, I2C frequency: 80 kHz
  I2C_C1    = I2C_C1_IICIE | I2C_C1_IICEN;
// enable interrupt in INTC module
  INTC_IPR6 |= INTC_IPR6_IIC0_0 | INTC_IPR6_IIC0_1;
}

/******************************************************************************
*  I2C peripheral module de-initialization function definition                *
******************************************************************************/
void I2C_DeInit(void)
{
        I2C_C1 = 0;

}
/******************************************************************************
*  I2C dealy function definition                                                *
******************************************************************************/

void I2C_delay(void)  // delay of 200 us @50MHz CPU clock (creates period of 5 kHz)
{
        unsigned int cnt;
```

```
        for ( cnt = 0;cnt < 10000; cnt++)
                { asm(nop); };
}

/*****************************************************************************
*  I2C restore function definition                                           *
*****************************************************************************/

tI2C_fault I2C_Restore(void)
{
        unsigned char tmp = 0;

        I2C_STOP_SIGNAL;
        I2C_DeInit();
        I2C_MASTER_SDA_PIN_AS_GPIO;
        I2C_MASTER_SDA_PIN_GPIO_HIGH_DRIVE;
        I2C_MASTER_SCL_PIN_AS_GPIO;
        I2C_MASTER_SDA_PIN_AS_OUT;
        I2C_MASTER_SDA_PIN_0;
        I2C_MASTER_SCL_PIN_AS_OUT;

        for(tmp = 0; tmp <9; tmp ++)  // nine clock for data
        {
               I2C_MASTER_SCL_PIN_0;
               I2C_delay();
               I2C_MASTER_SCL_PIN_1;
               I2C_delay();
        }

        I2C_MASTER_SCL_PIN_0;
        I2C_MASTER_SDA_PIN_AS_OUT; //SDA pin set to output
        I2C_MASTER_SDA_PIN_1;     //negative acknowledge
        I2C_delay();
        I2C_MASTER_SCL_PIN_1;
        I2C_delay();
        I2C_MASTER_SCL_PIN_0;
        I2C_delay();
        I2C_MASTER_SDA_PIN_0;     //stop
        I2C_delay();
        I2C_MASTER_SCL_PIN_1;
        I2C_delay();

        I2C_MASTER_SDA_PIN_AS_IN;
        tmp = 0;
        if (!((GPIOC_DR>>14) & 1)) // if still SDA is zero, try once again with
// more SCL clocks
        {
               while (!((GPIOC_DR>>14) & 1) && (tmp < 30))
               {
                      I2C_MASTER_SCL_PIN_0;
                      I2C_delay();
                      I2C_MASTER_SCL_PIN_1;
                      I2C_delay();
                      tmp++;
               };
               if (tmp == 30)
                      return I2C_PERMANENT_BUS_FAULT; // giving up, permanent
```

```
 // error, reset required
                I2C_MASTER_SCL_PIN_0;
                I2C_MASTER_SDA_PIN_AS_OUT; //SDA pin set to output
                I2C_MASTER_SDA_PIN_1;     //negative acknowledge
                I2C_delay();
                I2C_MASTER_SCL_PIN_1;
                I2C_delay();
                I2C_MASTER_SCL_PIN_0;
                I2C_delay();
                I2C_MASTER_SDA_PIN_0;     //stop
                I2C_delay();
                I2C_MASTER_SCL_PIN_1;
                I2C_delay();
        }

        I2C_MASTER_SDA_PIN_AS_I2C;
        I2C_MASTER_SCL_PIN_AS_I2C;

        I2C_Init();
        return I2C_NO_FAULT;

}

/*****************************************************************************
*  I2C data write function definition                                        *
*****************************************************************************/
tI2C_fault I2C_write(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char data[])
{
// check if the bus is not busy while there is attempt to write device address
//on the bus
        if ((I2C_S >> 5) & 1)
         return I2C_BUS_BUSY;

        // set the i2C communication mode, this flag will be evaluated in i2c isr
        psI2C_tr_ctrl -> eI2C_mode = I2C_MODE_WRITE;
        psI2C_tr_ctrl -> data_index = 0;            // initialize data index
        psI2C_tr_ctrl -> eI2C_flag = I2C_FLAG_TRANSMISSION_PROGRESS;
// move to next byte
        psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_WRITE_REG_ADDRESS;
        I2C_SET_TX_MODE;
        I2C_START_SIGNAL;
 //initiate write message with device address
        I2C_WRITE_BYTE(psI2C_tr_ctrl -> device_address_w);

        return I2C_NO_FAULT;
}
/*****************************************************************************
*  I2C data read function definition                                         *
*****************************************************************************/
tI2C_fault I2C_read(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char data[])
{
// check if the bus is not busy while there is attempt to write device address
// on the bus
        if ((I2C_S >> 5) & 1)
                return I2C_BUS_BUSY;
// set the i2C communication mode, this flag will be evaluated in i2c isr
        psI2C_tr_ctrl -> eI2C_mode = I2C_MODE_READ;
```

```
        psI2C_tr_ctrl -> data_index = 0;    // initialise data index
        I2C_SET_TX_MODE;
        I2C_START_SIGNAL;
//initiate read message with device address
        I2C_WRITE_BYTE(psI2C_tr_ctrl -> device_address_w);
        psI2C_tr_ctrl -> eI2C_flag = I2C_FLAG_TRANSMISSION_PROGRESS;
// move to next byte
        psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_WRITE_REG_ADDRESS;

        return I2C_NO_FAULT;
}
/*****************************************************************************
 * write data function definition                                            *
 *****************************************************************************/
tI2C_fault I2C_write_data(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char data[])
{
        unsigned int cnt;
        timeout_cnt = 100000;
        while ((( I2C_write(psI2C_tr_ctrl, data) == I2C_BUS_BUSY) && \
(timeout_cnt != 0)))
{timeout_cnt--;}
        if (timeout_cnt == 0)
        {
                psI2C_tr_ctrl-> eI2C_fault = I2C_TIMEOUT;
// the bus signals still busy and the timeout occurs, restore the I2C
                I2C_Restore();
                timeout_cnt = 100000;
                //second attempt to write the data to I2C bus
                while ((( I2C_write(psI2C_tr_ctrl, data) == I2C_BUS_BUSY) && \
                (timeout_cnt != 0)))
{timeout_cnt--;};
                        if (timeout_cnt == 0)
                        {
// try to send the data anyway
                                I2C_write(psI2C_tr_ctrl, data);
// wait till the packet is sent
                                for (cnt =0; cnt <100; cnt++)
                                        I2C_delay();
// the bus signals still busy and the timeout occurs,
// restore the I2C
                                I2C_Restore();
                                timeout_cnt = 100000;
                                //third attempt to write the data to I2C bus
                                while ((( I2C_write(psI2C_tr_ctrl, data) ==
I2C_BUS_BUSY)\
 && (timeout_cnt != 0))) {timeout_cnt--;};
                                        if (timeout_cnt == 0)
                                                return I2C_PERMANENT_BUS_FAULT;
                        }
        }
        return I2C_NO_FAULT;
}

/*****************************************************************************
 * read data function definition                                             *
 *****************************************************************************/
tI2C_fault I2C_read_data(tI2C_com_ctr *psI2C_tr_ctrl, unsigned char data[])
```

```
{
        timeout_cnt = 100000;
        while ((( I2C_read(psI2C_tr_ctrl, data) == I2C_BUS_BUSY) && \
(timeout_cnt != 0))) {timeout_cnt--;}
        if (timeout_cnt == 0)
        {
// the bus is still busy and the timeout occurs, restore the I2C
          I2C_Restore();
          timeout_cnt = 100000;
          //second attempt to write the data to I2C bus
          while ((( I2C_read(psI2C_tr_ctrl, data) == I2C_BUS_BUSY) && \
  (timeout_cnt != 0))) {timeout_cnt--;};
            if (timeout_cnt == 0)
                 return I2C_PERMANENT_BUS_FAULT;
        }
        return I2C_NO_FAULT;
}

/*****************************************************************************
 * interrupt callback function definition                                    *
 *****************************************************************************/
tI2C_fault I2C_isr_Callback (tI2C_com_ctr *psI2C_tr_ctrl, unsigned char* data)
{
  bool dummy;
  register unsigned char dummy_data;
  switch (psI2C_tr_ctrl -> eI2C_mode)
  {
  case  I2C_MODE_WRITE:
    {
      switch (psI2C_tr_ctrl -> eI2C_trm_stage)
      {
        case I2C_TRM_STAGE_WRITE_REG_ADDRESS:
        {
         psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_WRITE_DATA;
         I2C_WRITE_BYTE( psI2C_tr_ctrl -> register_address);
          break;
        };
        case I2C_TRM_STAGE_WRITE_DATA:
        {
    // if this is acknowledge after last byte in the message
          if ((psI2C_tr_ctrl -> data_size ) == psI2C_tr_ctrl -> data_index)
          {
// acknowledge after last data byte received,
// so generate stop signal now
            I2C_STOP_SIGNAL;
            psI2C_tr_ctrl -> eI2C_flag = I2C_FLAG_NONE;  // write data sequence end
// end of message transfer flag
psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_NONE;
            return I2C_NO_FAULT;
          }
          else
          {
         I2C_WRITE_BYTE(data[psI2C_tr_ctrl -> data_index]);
// increase index to the next address of the data array
            psI2C_tr_ctrl -> data_index++;
          }
          break;
```

```
        };
    };
    break;
   }
  case I2C_MODE_READ:
    {
    switch (psI2C_tr_ctrl -> eI2C_trm_stage)
          {
            case I2C_TRM_STAGE_WRITE_REG_ADDRESS:
            {
                 I2C_WRITE_BYTE( psI2C_tr_ctrl -> register_address);
                 psI2C_tr_ctrl -> eI2C_trm_stage = \
                          I2C_TRM_STAGE_WRITE_DEV_ADDRESS_R; // move to next byte
              break;
            };
            case I2C_TRM_STAGE_WRITE_DEV_ADDRESS_R:
            {
               I2C_REPEAT_START_SIGNAL;
               I2C_WRITE_BYTE( psI2C_tr_ctrl -> device_address_r);
               psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_READ_DUMMY_DATA;
              break;
            };
            case I2C_TRM_STAGE_READ_DUMMY_DATA:  // post read dummy data action
            {
                 I2C_SET_RX_MODE;
// reading data register initiates
// receiving of the next byte of data
                 dummy_data = I2C_READ_BYTE;
                 psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_READ_DATA;
              break;
            }
            case I2C_TRM_STAGE_READ_DATA:
            {
              // if this is acknowledge after last byte in the stream
              if ((psI2C_tr_ctrl -> data_size -1) == psI2C_tr_ctrl -> data_index)
              {
                I2C_SET_NACK_MODE; // last read ends with NAK
                iic_data[psI2C_tr_ctrl -> data_index] = I2C_READ_BYTE;
                psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_NAK;
              }
              else
              {
                   iic_data[psI2C_tr_ctrl -> data_index] = I2C_READ_BYTE;
                          if(psI2C_tr_ctrl -> data_index < (psI2C_tr_ctrl -> \
                                   data_size-1))
// increase pointer to the next address of the data array
                              psI2C_tr_ctrl -> data_index++;
              }
              break;
            };
            case I2C_TRM_STAGE_NAK:
            {
              I2C_STOP_SIGNAL;
              I2C_CLEAR_NACK_MODE;
              psI2C_tr_ctrl -> eI2C_flag = I2C_FLAG_NONE;  // read data sequence end
              psI2C_tr_ctrl -> eI2C_trm_stage = I2C_TRM_STAGE_NONE;
              asm{ nop };
```

```
              asm{ nop };
              asm{ nop };
              asm{ nop };
              return I2C_NO_FAULT;
            }
          break;
        }
      }

  }
  return I2C_NO_FAULT;
}
/*****************************************************************************
 * interrupt function definition                                             *
 *****************************************************************************/
void MC56F827xx_ISR_IIC0(void)
#pragma interrupt saveall
{

         if (I2C_S | I2C_S_TCF)
                  if (I2C_isr_Callback(&sI2C_com_ctr, iic_data) == I2C_NO_FAULT)
                  {
                           if (sI2C_com_ctr.eI2C_flag == I2C_FLAG_NONE)
                           {
                             // the transfer of the whole message ended. If the read
  //message was completed, now the data are in the iic_data[]
  // Place a code that process received data.
                           }
                  }
    /* Clear Irq. request */
            I2C_CLEAR_IRQ_FLAG;
}



/*****************************************************************************
 * End of module                                                             *
 *****************************************************************************/
```

6 References

Following are the reference sources related to this application note:

1. *MC56F827XXRM - MC56F82XXX Reference Manual* available on freescale.com.
2. *K40 Sub-Family Reference Manual* available on freescale.com.
3. *UM10204 - I2C-bus specification and user manual, Rev.5,* NXP Semiconductors 2012.

7 Acronyms

The following table summarizes acronyms used in this document:

Table 1. Acronyms

Term	Meaning
ACK	Acknowledge signal – the signal is generated by the I2C device which receives the data (byte)
ISR	Interrupt Service Routine - a part of the code, that is executed when the interrupt is generated
NACK	Not Acknowledge signal – condition on the bus, a receiver signals problems with interpretation of received data or master signals the end of data transfer from slave.

8 Revision history

Revision Number	Date	Substantial Changes
0	10/2013	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4803
Revision 0
October 2013