Freescale Semiconductor
Application Note

Document Number: AN4481
Rev 0.1, 07/2012

Sensor I²C Setup and FAQ

by: Miguel Salhuana

1 Introduction

This document outlines how an MCU interfaces with a Freescale sensor using the I²C communication protocol. It is important to clearly understand the correct method for setting up the I²C for communication between a Freescale sensor and the MCU. All programming of the Freescale sensor registers is done through the I²C protocol as defined by the Philips I²C-Bus Specification version 2.1, slave mode. This document assumes that the user is familiar with I²C protocol and discusses I²C setup for communication with the Freescale sensor. "FAQ" on page 10 reviews the most frequently asked questions of the I²C interface.

This document covers the following topics:

- Configuring the I²C Address
- Single-Byte Write
- Multiple-Byte Write
- Single-Byte Read
- Multiple-Byte Read
- Bus Reset
- FAQ
- Sample Driver Code for Freescale Microprocessors

Contents

1.1 Features

The I^2C features, for Freescale sensors, include:

- Compliance with the Philips I^2C-Bus Specification version 2.1
- Slave-only operation
- Single- and Multiple-Byte Write/read with auto-increment addressing capability.
- 7-bit addressing mode with 8-bit data retrieval

1.2 Keywords

I^2C, IIC, Interrupt, Sensor, Streaming Data, Polling, Slave, Protocol

2 I^2C Configuration

Please note that this document makes reference to the I^2C as well as IIC.

- IIC - refers to the host/MCU controller
- I^2C - refers to the slave/sensor controller

An example is the IIC host controller data register which has the name IICD. Therefore, when referring to the slave/sensor we will use I^2C and when referring to the host/MCU controller we will use IIC.

2.1 Explanation of Dummy Read

Please note that during the explanation of the algorithm we will make reference to a “Dummy Read”. This read is not a true read of the I^2C bus, but is a read of the IICD register in the IIC block. It will not be seen on the bus or in any I^2C specification because it is internal to the design of the I^2C hardware block.

The I^2C hardware block includes a state machine that will perform an IIC byte transfer on the bus each time the IICD register is written to or read. The first byte (I^2C device address) is sent out when the I^2C address value is written to the IICD register. Once this has been written out then it needs the Dummy Read to initiate the transfer of the next byte. Note that it is a Dummy Read because there is no real data in the IICD register yet because the hardware has not actually done the transfer.

Throughout the text of this document you will see references to the “Dummy Read” this is an artifact of the implementation of the I^2C controller block in the host MCU.

2.2 I^2C Device Address

Each slave is assigned an I^2C device address. The address consists of 8 bits, the 7 most-significant bits represent the actual device address and the least significant bit is the read/write toggle. Please refer to the device data sheet for the specific I^2C address.

For example the MPL3115A2 sensor uses the standard 7-bit I^2C slave address of 0x60 or 1100000. The 8-bit I^2C write address is 0xC0 and the 8-bit read address is 0xC1.

2.3 Single-Byte Write

In order to perform a single-byte write, use the following steps:

1. Setup START condition (a HIGH to LOW transition of SDA while SCL is HIGH)
2. Send I^{2}C device address (last bit is 0 to write)
3. Send I^{2}C register address
4. Write data
5. Send STOP (a LOW to HIGH transition of SDA while SCL is HIGH)

Please note that the START and STOP conditions are always generated by the master.

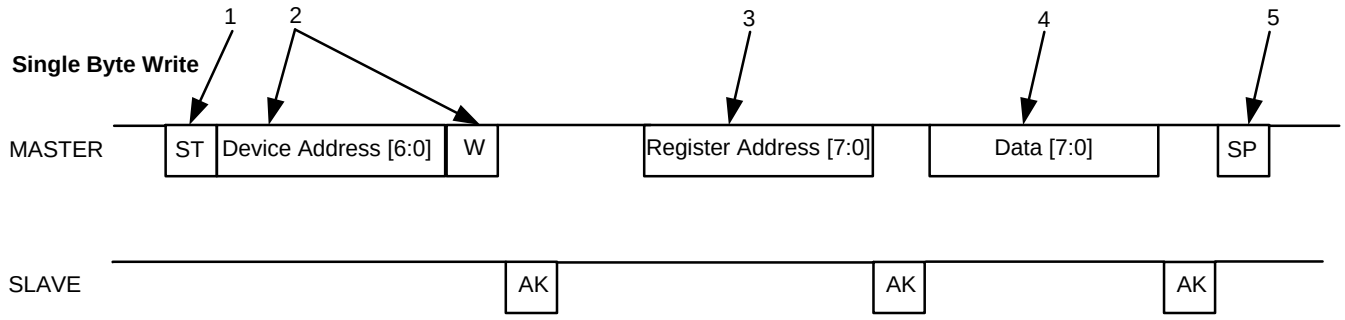

Figure 1. Single-Byte Write

Example 1. Single-Register Write code example

```
/*******************************************************\
* IIC Write Register
*  address - I2C Address
*   reg - register address
*   val - value to be written
\*******************************************************/
void IIC_RegWrite(byte address, byte reg, byte val)
{
  IICC_TX = 1;                                  // Transmit Mode
  IIC_Start();                                  // Send Start
  IIC_CycleWrite(address);                      // Send IIC "Write" Address
  IIC_CycleWrite(reg);                          // Send Register
  IIC_CycleWrite(val);                          // Send Value
  IIC_Stop();                                   // Send Stop
}
```

2.4 Multiple-Byte Write

When doing a multiple-byte write, we use the auto-incrementing address feature of the ASIC embedded with the sensor. The steps to doing a multiple-byte read are:

1. Setup START condition (a HIGH to LOW transition of SDA while SCL is HIGH)
2. Send I^2C device address (last bit is 0 to write)
3. Send register address
4. Write data (using a loop)
5. Send STOP (a LOW to HIGH transition of SDA while SCL is HIGH)

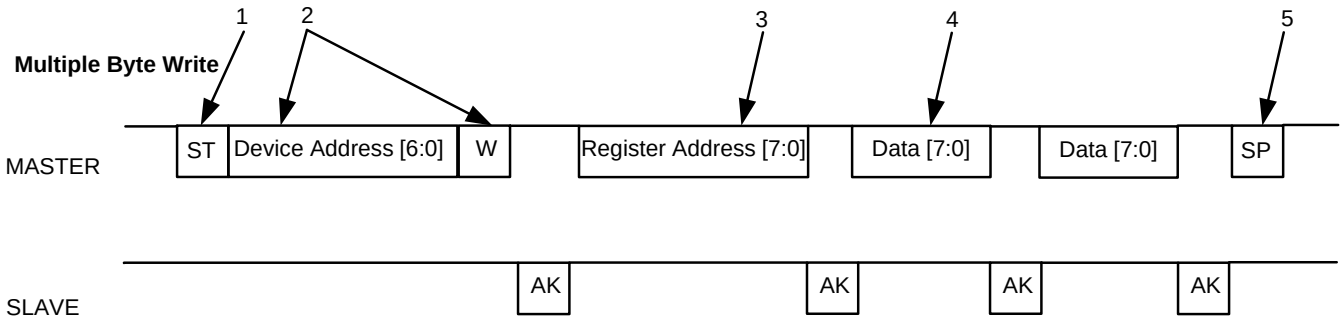

Figure 2. Multiple-Byte Write

Example 2. Multiple-Write Register code example

```
/*********************************************************\
* IIC Write Multiple Registers
*  address - I2C Address
*   reg1 - register address
*   N    - number of bytes to write
*   array - values to be written
*
\*********************************************************/
void IIC_RegWriteN(byte address, byte reg1,byte N,byte *array)
{
  IICC_TX = 1;                                  // Transmit Mode
  IIC_Start();                                  // Send Start
  IIC_CycleWrite(address);                      // Send IIC "Write" Address
  IIC_CycleWrite(reg1);                         // Send Register
  while (N>0)                                   // Send N Values
  {
    IIC_CycleWrite(*array);
    array++;
    N--;
  }
  IIC_Stop();                                   // Send Stop
}
```

2.5 Single-Byte Read

To do a single-byte read we use the following steps:

1. Setup START condition (HIGH to LOW transition of SDA while SCL is HIGH)
2. Send I^{2}C device address (last bit is 0 to write)
3. Send I^{2}C register address
4. Repeated START (HIGH to LOW transition of SDA while SCL is HIGH)
5. Send I^{2}C device address (last bit set to read)
6. Dummy Read (internal only)
7. Read data
8. Send STOP (a LOW to HIGH transition of SDA while SCL is HIGH)

Please note that the START and STOP conditions are always generated by the master.

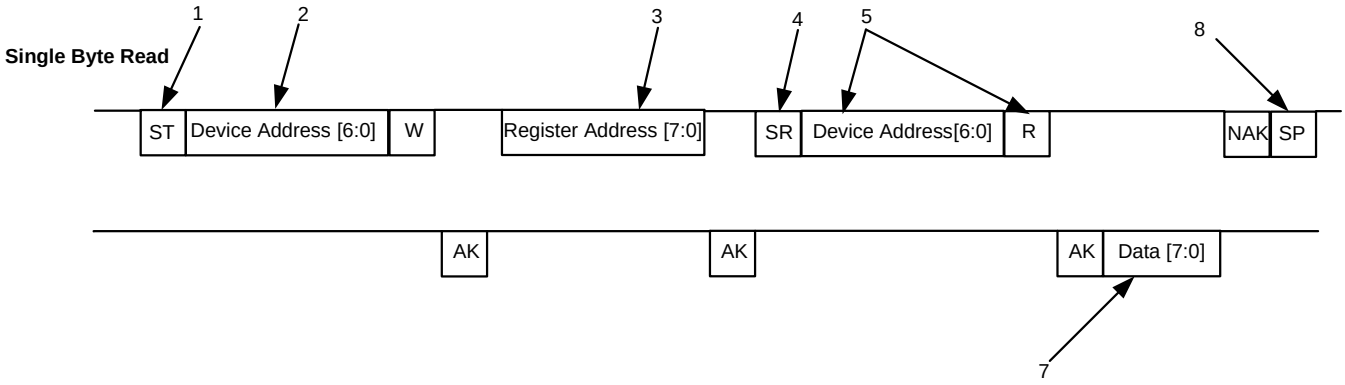

Figure 3. Single-Byte Read

Example 3. Single-Read Register code example

```
/*****************************************************\
* IIC Read Register
*  address - I2C Address
*   reg - register address
\*****************************************************/
byte IIC_RegRead(byte address, byte reg)
{
  byte b;
  IICC_TX = 1;                        // Transmit Mode
  IIC_Start();                        // Send Start
  IIC_CycleWrite(address);            // Send IIC "Write" Address
  IIC_CycleWrite(reg);                // Send Register
  IIC_RepeatStart();                  // Send Repeat Start
  IIC_CycleWrite(address+1);          // Send IIC "Read" Address
  b = IIC_CycleRead(1);               // Dummy read starts read transaction
  b = IIC_CycleRead(1);               // Read Register Value
  IIC_Stop();                         // Send Stop
  return b;
}
```

2.6 Multiple-Byte Read

The multiple-byte read uses the sensor's built in auto-address increment feature. This allows the host controller to cycle through a specific range of registers by only sending the first register's address and then reading the data for the number of registers specified in the function call. This address table is specific for each sensor so please check the data sheet for the specified ranges.

To illustrate the use of the auto-address increment feature, we can take a look at the MPL3115A2 data sheet register address map:

Table 1. Register Address Map

Register Address	Name	Reset	Reset when STBY to Active	Type	Auto-Increment Address		Comment	
0x00	Sensor Status Register (**STATUS**)(1)(2)	0x00	Yes	R	0x01		Alias for DR_STATUS or F_STATUS	
0x01	Pressure Data Out MSB (**OUT P_MSB**)(1)(2)	0x00	Yes	R	0x02	0x01	Bits 12-19 of 20-bit real-time Pressure sample.	Root pointer to Pressure and Temperature FIFO data.
0x02	Pressure Data Out CSB (**OUT_P_CSB**)(1)(2)	0x00	Yes	R	0x03		Bits 4-11 of 20-bit real-time Pressure sample	
0x03	Pressure Data Out LSB (**OUT_P _LSB**)(1)(2)	0x00	Yes	R	0x04		Bits 0-3 of 20-bit real-time Pressure sample	
0x04	Temperature Data Out MSB (**OUT_T_MSB**)(1)(2)	0x00	Yes	R	0x05		Bits 4-11 of 12-bit real-time Temperature sample	
0x05	Temperature Data Out LSB (**OUT_T _LSB**)(1)(2)	0x00	Yes	R	0x00		Bits 1-3 of 12-bit real-time Temperature sample	
0x06/0x00	Sensor Status Register (**DR_STATUS**)(1)(2)	0x00	Yes	R	0x07		Data Ready status information	

1. Register contents are preserved when transitioning from "ACTIVE" to "STANDBY" mode.
2. Register contents are reset when transitioning from "STANDBY" to "ACTIVE" mode.

As you can see in this table you can continually read the status and output register of the MPL3115A2 by initiating a read at register 0x00 and it will auto-increment through the output registers and loop back to the status register to begin the loop again.

The steps to doing a multiple-byte read are:

1. Setup START condition (HIGH to LOW transition of SDA while SCL is HIGH)
2. Send I^2C device address (last bit is 0 to write)
3. Send register address
4. Repeated START (HIGH to LOW transition of SDA while SCL is HIGH)
5. Send I^2C device address (last bit set to read)
6. Dummy Read (internal only)
7. Read data (using a loop)
8. Send STOP (a LOW to HIGH transition of SDA while SCL is HIGH)

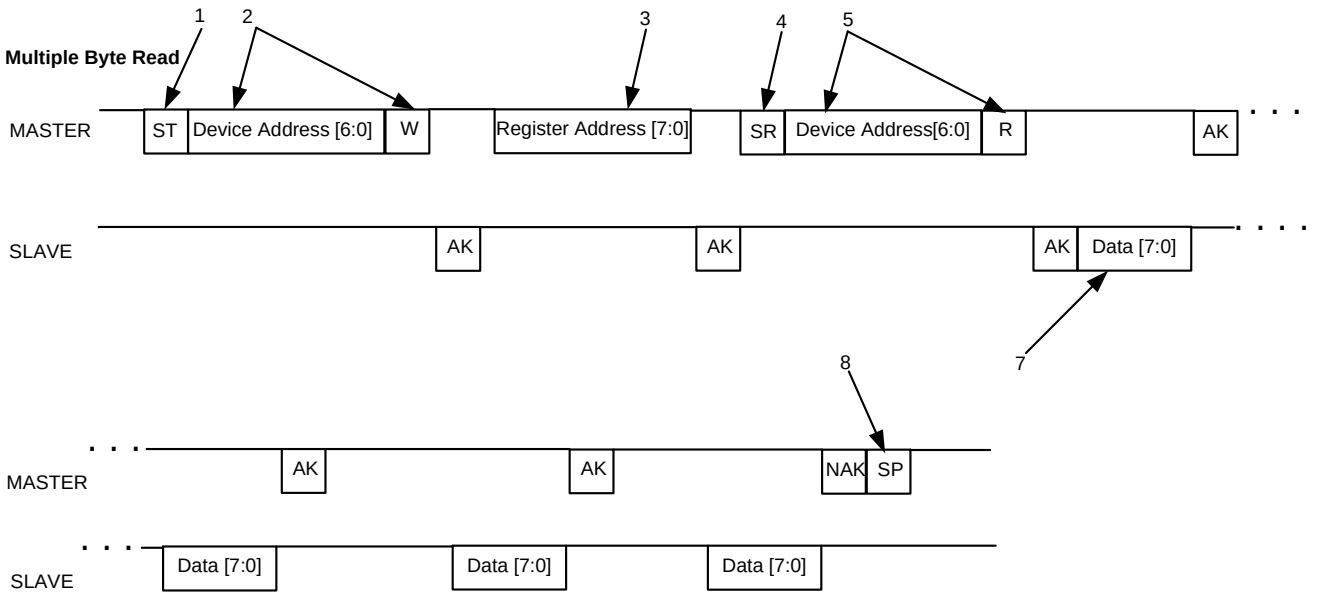

Figure 4. Multiple-Byte Read

Example 4. Multiple-Read Register code example

```
/*******************************************************\
* IIC Read Multiple Registers
*  address - I2C Address
*   reg1 - register address
*   N    - number of bytes to read
*   array - values to be read
\*******************************************************/
void IIC_RegReadN(byte address, byte reg1, byte N, byte *array)
{
  byte b;
  IICC_TX = 1;                                  // Transmit Mode
  IIC_Start();                                  // Send Start
  IIC_CycleWrite(address);                      // Send IIC "Write" Address
  IIC_CycleWrite(reg1);                         // Send Register
  IIC_RepeatStart();                            // Send Repeat Start
  IIC_CycleWrite(address+1);                    // Send IIC "Read" Address
  b = IIC_CycleRead(0);                         // Dummy read
  while (N>1)                                   // Read N-1 Register Values
  {
    b = IIC_CycleRead(0);
    *array = b;
    array++;
    N--;
  }
  b = IIC_CycleRead(1);
  *array = b;                                   // Read Last value
  IIC_Stop();                                   // Send Stop
}
```

2.7 Bus Reset

If the I^2C master stops in the middle of an I^2C transaction, to a slave, then it may leave the slave device in an incomplete state which can prevent future transactions from succeeding. Many slave devices may not have a reset pin due to pin limitation therefore, in order to recover from this condition the I^2C bus on the slave can be reset by bit banging the bus or by power cycling the system and/or the slave devices.

The sequence for a Bus Reset is as follows:

- Disable the host MCU IIC controller
- Create a START condition
- Clock SCL for at least nine clocks
- Check for SDA high
- Create a STOP condition
- Enable the host MCU IIC controller

Example 5. Bus Reset code example

```
/*****************************************************\
* Initiate IIC Bus Reset
*
*   The transitions here are controlled through the data direction pin
*   of the MCU. When writing a 1 to the data direction (DD) pin of the
*   MCU control register this makes the pin into an output and is driven
*   low since the data register is set to 0.
*   When a 0 is written to the DD bit, the pin is set as an input (floating)
*   which is pulled high by the external pullup resistors on the I2C lines.
*   Please see Application Note AN4481 FAQ for more information.
\*****************************************************/
void IIC_Bus_Reset(void)
{
    int loop;
    // Disable the I2C block on the Host Controller
    IICC1 &= ~(init_IICC1);
    PAUSE;

    /* Create START condition (SDA goes low while SCL is high) */
    I2C_SDA_DD = 1; // SDA = 0
    PAUSE;
    I2C_SCL_DD = 0; // SCL = 1
    PAUSE;

    /* Release SDA back high */
    I2C_SDA_DD = 0; // SDA = 1
    PAUSE;

    /* Clock SCL for at least 9 clocks until SDA goes high */
    /* This loop is significantly greater than 9 clocks to */
    /* make sure that this condition is met.               */

    loop = 0;
```

```
while (loop < 100)
{
     loop++;
     /* Apply one SCL clock pulse */
     I2C_SCL_DD = 1; // SCL = 1
     PAUSE;
     I2C_SCL_DD = 0; // SCL = 0
     PAUSE;
     /* If SDA is high and a complete byte was sent then exit the loop */
     if ((I2C_SDA_PIN) && ((loop % 9) == 0))
     break;
}

/* Create STOP condition (SDA goes high while SCL is high) */
I2C_SDA_DD = 1; // SDA = 0
PAUSE;
I2C_SCL_DD = 0; // SCL = 1
PAUSE;
I2C_SDA_DD = 0; // SDA = 1
PAUSE;
//Set operation back to default for all pins on PTBDD and Enable I2C
PTBDD = init_PTBDD;
IICC1 = init_IICC1;
```

3 FAQ

What are the pullup resistor values?

The normal recommended resistor value is 4.7 kΩ, however, for high-capacitive loads and fast busses, this can be reduced to as low as 1 kΩ.

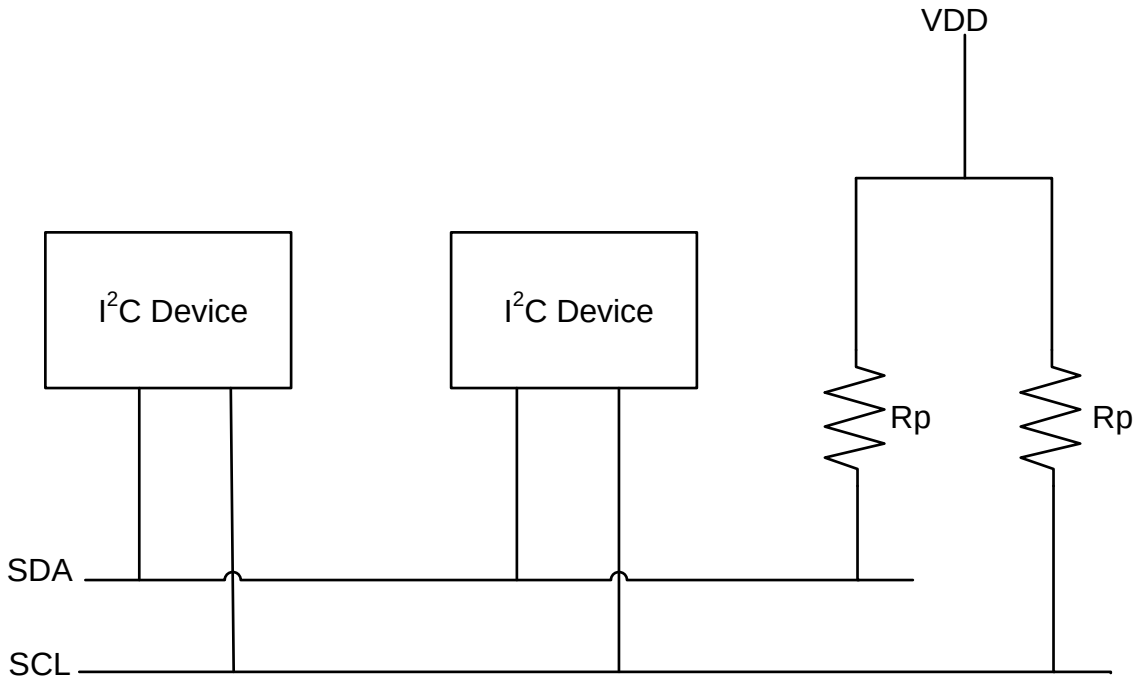

Figure 5. I²C Resistors

For more information on how to calculate these values please refer to the Philips I²C-Bus Specification version 2.1, section 16.1.

What are the RC values that affect the value of the pullup resistors?

Many factors can be considered in choosing the correct value for a pullup resistor for the I^2C clock and data signals. There is an RC time constant on the bus, where the pullup resistor and bus capacitance are the R and C values. Also entering into the equation are the bus clock rate, and the high-logic voltage level. Lower voltage interfaces, 3.3V versus 5.0V, can run faster with the same value pullup resistor. Bus capacitance is influenced by the type of devices on the bus as well as the physical length of the bus and the type of PC board, ground planes etc.

The calculations can quickly become very complicated, but there are a few basic ideas that will help determine if the resistor is suitable. In order create robust signaling on the I^2C bus, the rise time of the clock and data signals should be a small portion (say 10-20%) of the total clock / data period. I^2C is a fully asynchronous logic interface, where one of the signals happens to be named CLK, but the behavior is more like a data valid, enable, or strobe. It is important in all asynchronous designs to make sure that the data path and the enable path do not line up. The data signal should be stable (either high or low), then the clock signal floats high then is driven low. The data line normally only changes while the clock is low. If the data changes when clock is high, then that indicates a start (high to low transition) or stop (low to high transition) condition. Here is an example waveform.

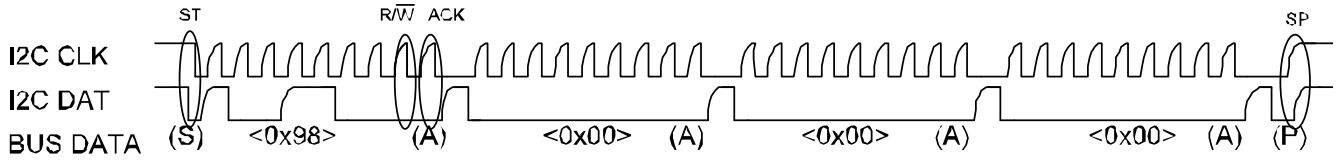

Figure 6. An example of an I^2C Timing Waveform

Some useful rules of thumb:

- 4.7 kΩ resistor is a good starting point, and most used for DC-100 kHz-400 kHz operation.
- For slave devices capable of running in the 1-5 MHz range, a 1 kΩ resistor may be required.
- For extremely long I^2C lines, for many slave devices, where the bus capacitance is in the 200 pF or more range, a 1 kΩ resistor may be required.
- Resistors lower than 1 kΩ are not recommended due to the current required to work against a low-resistance load.

What is the maximum speed of the I^2C?

All Freescale sensors support an I^2C bus speed of 400 kHz. Faster speeds are possible for some sensors; please refer to the sensor's data sheet for further information. The I^2C bus should only run as fast as the slowest device on the bus.

What is Clock Stretching?

When the master is reading from the slave, it's the slave that places the data on the SDA line, but it's the master that controls the clock. What if the slave is not ready to send the data? The sensor on the slave device will need to go to an interrupt routine, save its working registers, find out what address the master wants to read from, get the data and place it in its transmission register. This can take many µs to happen, meanwhile, the master is blissfully sending out clock pulses on the SCL line that the slave cannot respond to.

The I^2C protocol provides a solution to this: the slave is allowed to hold the SCL line low. This is called clock stretching. When the slave gets the read command from the master, it holds the clock line low. The sensor then gets the requested data, places it in the transmission register and releases the clock line, allowing the pullup resistor to finally pull it high. Well behaved master's will issue the first clock pulse of the read by making SCL high and then check to see if it really has gone high. If it's still low then it's the slave that holding it low and the master should wait until it goes high before continuing. Luckily, the hardware I^2C ports on most MCUs will handle this automatically.

I^2C is not reading the correct data from the expected address

Please see “Explanation of Dummy Read” on page 2.

4 Sample Driver Code for Freescale Microprocessors

The sample code was written for the demo DEMOSTB board which utilizes the Freescale MC9S08QE8 processor.

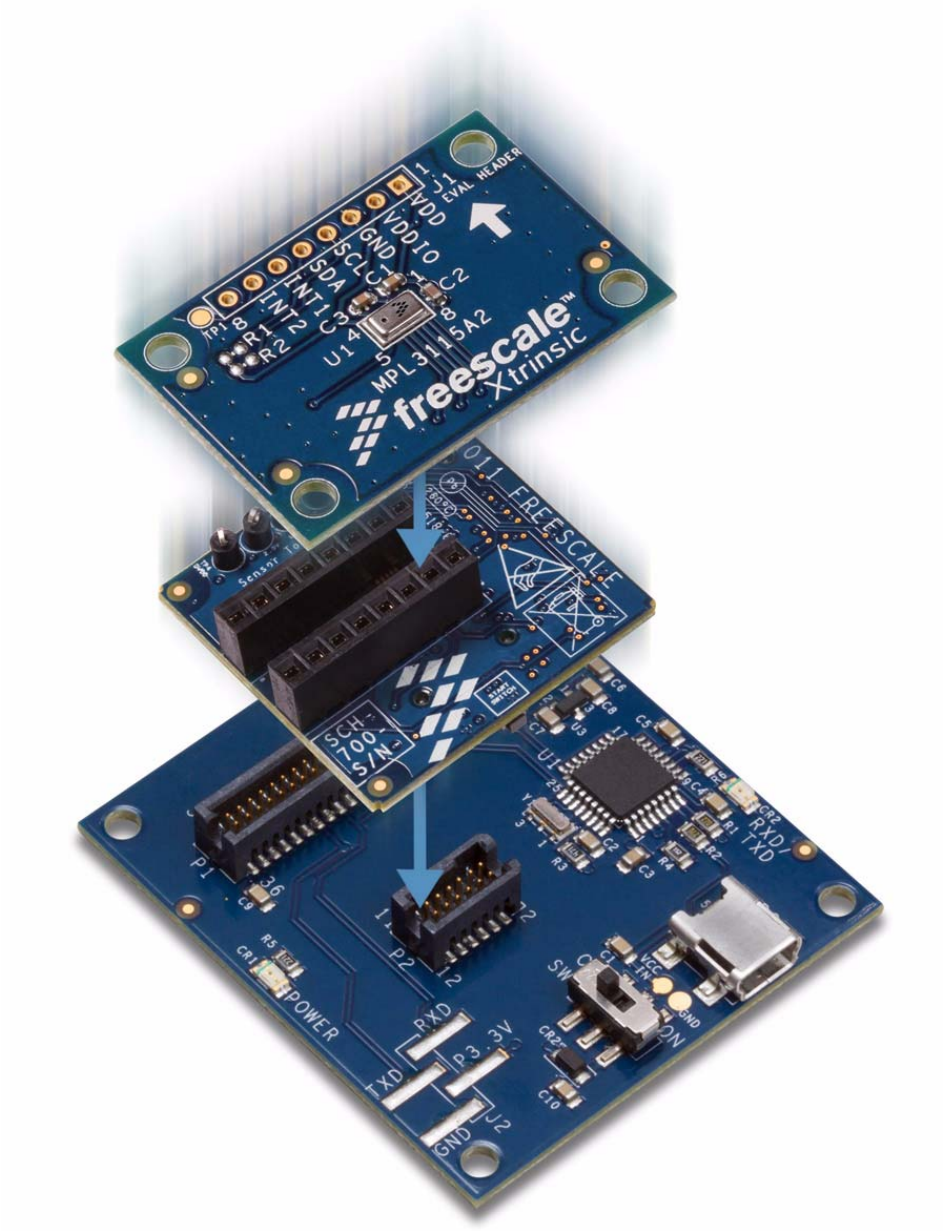

Figure 7. MPL3115A2 daughter board, LFSTBEBMPL3115A2 evaluation board and the LSTBUS interface board

The following acronyms, defined in the MC9S08QE8 Reference Manual, are used in the sample code. These correspond to bits within the IIC control registers within the MCU:

Table 2. Acronym Definitions

Acronym	Definition
IICC_MST	Master Mode Select
IICS_BUSY	Bus Busy
IICC_RSTA	Repeat Start
IICS_TCF	Transfer Complete Flag
IICS_IICIF	IIC Interrupt Flag
IICS_RXAK	Receive Acknowledge
IICC_TX	Transmit Mode Select
IICC_TXAK	Transmit Acknowledge
IICD	Data Register

4.1 IIC_Start

```
/*******************************************************\
* Initiate IIC Start Condition
\*******************************************************/
void IIC_Start(void)
{
  IICC_MST = 1;
  timeout = 0;
  while ((!IICS_BUSY) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x01;
} //*** Wait until BUSY=1
```

4.2 IIC_Stop

```
/*******************************************************\
* Initiate IIC Stop Condition
\*******************************************************/
void IIC_Stop(void)
{
  IICC_MST = 0;
  timeout = 0;
  while ( (IICS_BUSY) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x02;
} //*** Wait until BUSY=0
```

4.3 IIC_RepeatStart

```
/*******************************************************\
* Initiate IIC Repeat Start Condition
\*******************************************************/
void IIC_RepeatStart(void)
{
  IICC_RSTA = 1;
  timeout = 0;
  while ((!IICS_BUSY) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x04;
} //*** Wait until BUSY=1
```

4.4 IIC_CycleWrite

```
/*******************************************************\
* IIC Cycle Write
\*******************************************************/
void IIC_CycleWrite(byte bout)
{
  timeout = 0;
  while ((!IICS_TCF) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x08;
  IICD = bout;
  timeout = 0;
  while ((!IICS_IICIF) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x10;
  IICS_IICIF = 1;
  if (IICS_RXAK)
    error |= 0x20;
}
```

4.5 IIC_CycleRead

```
/*******************************************************\
* IIC Cycle Read
\*******************************************************/
byte IIC_CycleRead(byte byteLeft)
{
  byte bread;
  timeout = 0;
  while ((!IICS_TCF) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error|=0x08;
  IICC_TX = 0;
  IICC_TXAK = byteLeft <= 1 ? 1 : 0;  //Set NACK when reading the last byte
```

```
  bread = IICD;
  timeout = 0;
  while ((!IICS_IICIF) && (timeout<1000))

  while ((!IICS_TCF) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error|=0x08;
  IICC_TX = 0;
  IICC_TXAK = byteLeft <= 1 ? 1 : 0;  //Set NACK when reading the last byte
  bread = IICD;
  timeout = 0;
  while ((!IICS_IICIF) && (timeout<1000))
     timeout++;
   if (timeout) >= 1000)
     error |= 0x10;
   IICS_IICIF=1;
   return bread;
```

4.6 IIC_StopRead

```
/*********************************************************\
* Initiate IIC Stop Condition on Read
\*********************************************************/
byte IIC_StopRead(void)
{
  IICC_MST = 0;
  timeout = 0;
  while ( (IICS_BUSY) && (timeout<1000))
    timeout++;
  if (timeout >= 1000)
    error |= 0x02;
} //*** Wait until BUSY=0
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

AN4481
Rev 0.1
07/2012

freescale™