

Data Manipulation and Basic Settings for Xtrinsic MMA865xFC Accelerometers

by: Fengyi Li, Applications Engineer

1 Introduction

This document shows you how to configure the MMA865xFC devices to extract and manipulate the acceleration data. These devices have been designed to be compatible with a shared memory map. An example MMA865xFC Driver Code is available. It is briefly referred to in this application note, to demonstrate the device configuration in C.

The MMA8652FC device has all of the embedded features of this device family, including 8 different sample rates, 32 different cutoff frequencies for the high-pass filter, 3 dynamic ranges and 4 oversampling modes. It also has a 32-sample FIFO for collecting and storing data, which is the most efficient way to access the data for minimizing the I²C transactions. The FIFO can collect the regular low-pass filtered data, as well as the data from the high-pass filter. The manipulation of the data into different formats is also important for algorithm development and for display.

Contents

1	Introduction	1
2	MMA8652, 3FC 3-axis Accelerometer	2
3	Changing Modes of the MMA8652,3FC	3
4	Changing Dynamic Range	4
5	Setting the Data Rate	7
6	Setting the Oversampling Mode	8
7	Setting the High-Pass Filter Cutoff Frequency	11
8	12-bit or 10-bit Data Streaming and Data Conversions	13
9	8-bit XYZ Data Streaming and Conversions	25
10	Polling Data vs. Interrupts	28
11	Using the 32-Sample FIFO	32
12	MMA865x Driver Quick Start	34

1.1 Summary

- There is a Standby mode that responds to I²C communication but doesn't allow for updated data. There are also three dynamic ranges: 2g, 4g, and 8g, which can be used to observe the changes in sensitivity and the full acceleration range while the device is active.
- An example of how to set the data rate is shown. There are 8 different data rates, ranging from 1.56 Hz to 800 Hz.
- An example of how to set the High-Pass Filter Cutoff Frequency is given. The high-pass filtered output data is affected by the filter cutoff frequency settings.
- There are four different oversampling modes that can be set.
- An example and the format conversions for manipulating 12/10/8-bit data, converting 2's complement hex data to two different formats, which include formatting to signed integer (counts) and signed decimal fractions (in g's) for high-pass filtered data or for low-pass filtered data.
- An example of how to set up the device to poll the data or to configure an interrupt service routine is shown.
- An example of how to configure the FIFO to store and flush data.
- There is a driver available that will run on the MMA8652,3FC Sensor Toolbox Demo Board, which provides an example in CodeWarrior for everything discussed in this application note. The driver runs in RealTerm or HyperTerminal and can be used to capture and log data in different formats.

2 MMA8652, 3FC 3-axis Accelerometer

The MMA8652,3FC has a selectable dynamic range of $\pm 2g$, $\pm 4g$, $\pm 8g$. The device has 8 different output data rates, selectable high-pass filter cutoff frequencies, and in some cases high-pass filtered output data available. The resolution of the data and the embedded features is dependant on the specific device.

Note: The MMA8652,3FC has a memory map similar to the MMA8451,2,3Q.

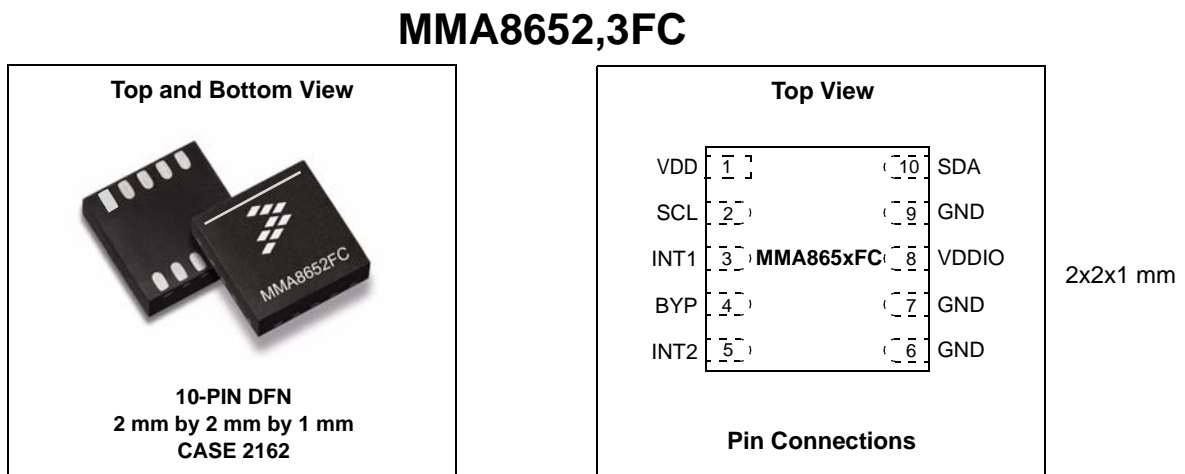


Figure 1. MMA8652,3FC consumer 3-axis accelerometer (2x2x1 mm)

2.1 Output data, sample rates and dynamic ranges for both devices

2.1.1 MMA8652FC

- 12-bit data
 - **2g** (1024 counts/g = 1 mg/LSB)
 - **4g** (512 counts/g = 2 mg /LSB)
 - **8g** (256 counts/g = 3.9 mg/LSB)
- 8-bit data
 - **2g** (64 counts/g = 15.6 mg/LSB)
 - **4g** (32 counts/g = 31.25 mg/LSB)
 - **8g** (16 counts/g = 62.5 mg/LSB)
- Embedded 32-sample FIFO

2.1.2 MMA8653FC

(Note: no HPF data)

- 10-bit data
 - **2g** (256 counts/g = 3.9 mg/LSB)
 - **4g** (128 counts/g = 7.8 mg/LSB)
 - **8g** (64 counts/g = 15.6 mg/LSB)
- 8-bit data
 - **2g** (64 counts/g = 15.6 mg/LSB)
 - **4g** (32 counts/g = 31.25 mg/LSB)
 - **8g** (16 counts/g = 62.5 mg/LSB)

3 Changing Modes of the MMA8652,3FC

The device can be in either Standby mode or Active mode.

- Most (although not all) changes to the registers must be done while the accelerometer is in Standby mode.
- Data is only acquired and updated while in Active mode.

To change the device mode, configure the ACTIVE bit in CTRL_REG1 register (0x2A):

- To enter Standby mode, clear the ACTIVE bit.
- To enter Active mode, set the ACTIVE bit.

Table 1. 0x2A CTRL_REG1 register (read/write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	FREAD	ACTIVE

Example 1. Going into Standby and Active modes

```
void MMA865x_Standby (void)
{
byte n;
/*
** Put the sensor into Standby Mode by clearing
** the Active bit of the System Control 1 Register
**
*/
IIC_RegWrite(CTRL_REG1, (IIC_RegRead(CTRL_REG1) & ~ ACTIVE_MASK));
}

void MMA865x_Active ()
{
/*
** Put the sensor into Active Mode by setting the
** Active bit of the System Control 1 Register
**
*/
IIC_RegWrite(CTRL_REG1, (IIC_RegRead(CTRL_REG1) | ACTIVE_MASK));
}
```

4 Changing Dynamic Range

There are also 3 different dynamic ranges that can be set (2g, 4g, 8g).

- The dynamic range can only be changed while in Standby mode.
- To control the dynamic range, configure the FS[1:0] bits in the XYZ_DATA_CFG register (0x0E).

Table 2. 0x0E XYZ_DATA_CFG register (read/write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
—	00	0	HPF_Out	0	0	FS1	FS0

Table 3. Full-scale selection

FS1	FS0	g Range
0	0	±2g
0	1	±4g
1	0	±8g
1	1	Reserved

4.1 2g Active mode

To enter 2g Active mode:

1. Enter Standby mode.

2. Change the FS bits to 00.
3. Enter Active mode.

Example 2. Entering 2G Active mode

```
/*
**Put the part in Standby Mode
*/
MMA865x_Standby();
/*
**Write the 2g dynamic range value into register 0x0E
*/
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) & ~FS_MASK));
/*
**Put the part back into the Active Mode
*/
MMA865x_Active();
```

4.2 4g Active mode

To enter 4g Active mode:

1. Enter Standby mode.
2. Change the FS bits to 01.
3. Enter Active mode.

Example 3. Entering 4G Active mode

```
/*
**Put the part in Standby Mode
*/
MMA865x_Standby();
/*
**Write the 4g dynamic range value into register 0x0E
*/
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) & ~FS_MASK));
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) | FULL_SCALE_4G));
/*
**Put the part back into the Active Mode
*/
MMA865x_Active();
```

4.3 8g Active mode

To enter 8g Active mode,

1. Enter Standby mode.
2. Change the FS bits to 10.
3. Enter Active mode.

Example 4. Entering 8G Active mode

```
/*
**Put the part in Standby Mode
*/
MMA865x_Standby();
/*
**Write the 8g dynamic range value into register 0x0E
*/
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) & ~FS_MASK));
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) | FULL_SCALE_8G));
/*
**Put the part back into the Active Mode
*/
MMA865x_Active();
```

5 Setting the Data Rate

The active mode Output Data Rate (ODR) and Sleep Mode Data Rate are programmable via other control bits in the CTRL_REG1 register (0x2A), as shown in Table 4, “0x2A CTRL_REG1 register (read/write),” on page 7. Unless Sleep mode is enabled, the Active mode data rate is the data rate that will always be enabled.

Table 5, “Output data rates,” on page 7 shows how the DR2:DR0 bits affect the ODR. These are the active mode data rates available. The default data rate is DR = 000, 800 Hz.

Table 4. 0x2A CTRL_REG1 register (read/write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	FREAD	ACTIVE

Table 5. Output data rates

DR2	DR1	DR0	Output Data Rate (ODR) (Hz)	Time Between Data Samples (ms)
0	0	0	800	1.25
0	0	1	400	2.5
0	1	0	200	5
0	1	1	100	10
1	0	0	50	20
1	0	1	12.5	80
1	1	0	6.25	160
1	1	1	1.563	640

Table 6. SLEEP mode rates

ASLP_RATE1	ASLP_RATE0	Frequency (Hz)	
0	0	50	Note: When the device is in Auto-SLEEP mode, the system ODR and the data rate for all the system functional blocks are overridden by the data rate set by the ASLP_RATE field.
0	1	12.5	
1	0	6.25	
1	1	1.56	

Example 5. Adjusting the Output Data Rate (ODR)

```

/*
** Adjust the desired Output Data Rate value as needed.
*/
DataRateValue = 3 << 3;
/*
** Put the device into Standby Mode
*/

```

```

MMA865x_Standby();
/*
** Write in the Data Rate value into Ctrl Reg 1
*/
IIC_RegWrite(CTRL_REG1,IIC_RegRead(CTRL_REG1) & ~DR_MASK);
IIC_RegWrite(CTRL_REG1, IIC_RegRead(CTRL_REG1)| DataRateValue);
/*
** Put the device into the active mode again.
*/
MMA865x_Active();

```

6 Setting the Oversampling Mode

There are four different oversampling modes:

- Normal mode
- A low noise + power mode
- A high-resolution mode
- A low-power mode

The oversampling modes are changed using CTRL_REG2 System Control 2 register (0x2B).

Table 7. 0x2B CTRL_REG2 register (read/write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ST	RST	0	SMODS1	SMODS0	SLPE	MODS1	MODS0

The difference between the oversampling modes is in the internal averaging of the sampled data. The following table shows the number of samples used in the averaging for the different Output Data Rates, which is called the OS Ratio (oversampling ratio).

Table 8. Oversampling modes with OS ratio per ODR

ODR (Hz)	Mode			
	Normal	Low Noise Low Power	High Resolution	Low Power
	OS Ratio	OS Ratio	OS Ratio	OS Ratio
1.56	128	32	1024	16
6.25	32	8	256	4
12.5	16	4	128	2
50	4	4	32	2
100	4	4	16	2
200	4	4	8	2
400	4	4	4	2
800	2	2	2	2

The following code examples show how to change to different oversampling modes.

Example 6. Changing to normal oversampling mode

```
/** Oversampling Mode: Normal MODS=00 */
/** Put the device into Standby Mode */
MMA865x_Standby();

/** Clear the Mode bits in CTRL_REG2 */
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) &
~MODS_MASK));

/** Put the device into Active Mode */
MMA865x_Active();
```

Example 7. Changing to low noise, low power oversampling mode

```
/** Oversampling Mode: Low Noise Low Power MODS = 01 */
/** Put the device into Standby Mode */
MMA865x_Standby();

/** Clear the Mode bits in CTRL_REG2 */
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) &
~MODS_MASK));

/** Set the MODS bits to 01 for Low Noise Low Power Mode */
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) |
MODS0_MASK));

/** Put the device into Active Mode */
MMA865x_Active();
```

Example 8. Changing to high resolution oversampling mode

```
/** Oversampling Mode: HI RESOLUTION MODS =10 */
/** Put the device into Standby Mode */
MMA865x_Standby();

/** Clear the Mode bits in CTRL_REG2 */
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) &
~MODS_MASK));

/** Set the MODS bits to 10 for Hi Resolution Mode */
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) |
MODS1_MASK));
```

```
/** Put the device into Active Mode */  
MMA846x_Active();
```

Example 9. Changing to low power oversampling mode

```
/** Oversampling Mode: LOW POWER MODS = 11 */  
/** Put the device into Standby Mode */  
MMA865x_Standby();  
  
/** Clear the Mode bits in CTRL_REG2 */  
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) &  
~MODS_MASK));  
  
/** Set the MODS bits to 11 for Low Power Mode */  
IIC_RegWrite(SlaveAddressIIC, CTRL_REG2, (IIC_RegRead(SlaveAddressIIC, CTRL_REG2) |  
MODS_MASK));  
  
/** Put the device into Active Mode */  
MMA865x_Active();
```

7 Setting the High-Pass Filter Cutoff Frequency

The HP_FILTER_CUTOFF register (at 0x0F) sets the high-pass cutoff frequency, F_c , for the data. The output of this filter is provided in the Output Data Registers (0x01 to 0x06).

NOTE

The high-pass filtered output data is available for the MMA8652FC only. The MMA8653FC has the internal high-pass filter for the embedded functions but does not have access to the output data. The available cutoff frequencies change depending upon the set Output Data Rate.

Table 9. 0x0F HP_FILTER_CUTOFF: High-pass filter register (read/write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	Pulse_HPF_Byp	Pulse_LPF_EN	0	0	SEL1	SEL0

Table 10 presents the different cutoff frequencies for the high-pass filter based on the different set data rates.

NOTE

The cutoff frequencies *change* based on output data rate and on the oversampling mode.

Table 10. HP_FILTER_CUTOFF setting options

SEL1	SEL0	800 Hz	400 Hz	200 Hz	100 Hz	50 Hz	12.5 Hz	6.25 Hz	1.56 Hz
Oversampling Mode = Normal									
0	0	16 Hz	16 Hz	8 Hz	4 Hz	2 Hz	2 Hz	2 Hz	2 Hz
0	1	8 Hz	8 Hz	4 Hz	2 Hz	1 Hz	1 Hz	1 Hz	1 Hz
1	0	4 Hz	4 Hz	2 Hz	1 Hz	0.5 Hz	0.5 Hz	0.5 Hz	0.5 Hz
1	1	2 Hz	2 Hz	1 Hz	0.5 Hz	0.25 Hz	0.25 Hz	0.25 Hz	0.25 Hz
Oversampling Mode = Low Noise Low Power									
0	0	16 Hz	16 Hz	8 Hz	4 Hz	2 Hz	0.5 Hz	0.5 Hz	0.5 Hz
0	1	8 Hz	8 Hz	4 Hz	2 Hz	1 Hz	0.25 Hz	0.25 Hz	0.25 Hz
1	0	4 Hz	4 Hz	2 Hz	1 Hz	0.5 Hz	0.125 Hz	0.125 Hz	0.125 Hz
1	1	2 Hz	2 Hz	1 Hz	0.5 Hz	0.25 Hz	0.063 Hz	0.063 Hz	0.063 Hz
Oversampling Mode = High Resolution									
0	0	16 Hz	16 Hz	16 Hz	16 Hz	16 Hz	16 Hz	16 Hz	16 Hz
0	1	8 Hz	8 Hz	8 Hz	8 Hz	8 Hz	8 Hz	8 Hz	8 Hz
1	0	4 Hz	4 Hz	4 Hz	4 Hz	4 Hz	4 Hz	4 Hz	4 Hz
1	1	2 Hz	2 Hz	2 Hz	2 Hz	2 Hz	2 Hz	2 Hz	2 Hz

Table 10. HP_FILTER_CUTOFF setting options (Continued)

SEL1	SEL0	800 Hz	400 Hz	200 Hz	100 Hz	50 Hz	12.5 Hz	6.25 Hz	1.56 Hz
Oversampling Mode = Low Power									
0	0	16 Hz	8 Hz	4 Hz	2 Hz	1 Hz	0.25 Hz	0.25 Hz	0.25 Hz
0	1	8 Hz	4 Hz	2 Hz	1 Hz	0.5 Hz	0.125 Hz	0.125 Hz	0.125 Hz
1	0	4 Hz	2 Hz	1 Hz	0.5 Hz	0.25 Hz	0.063 Hz	0.063 Hz	0.063 Hz
1	1	2 Hz	1 Hz	0.5 Hz	0.25 Hz	0.125 Hz	0.031 Hz	0.031 Hz	0.031 Hz

To set the cutoff frequency, a value from 0x00 to 0x03 must be chosen for the SEL[1:0] bits, as per Table 10. In order to make this change, the sensor must be in Standby mode before writing to the HP_FILTER_CUTOFF register.

Consider an example where the device is operating in 2g Active mode with a 400 Hz ODR and the default Fc of 16 Hz. The following code demonstrates how to change Fc to 4 Hz without making any other configuration modifications. This is done by changing the SEL bits to 10, (as per Table 10).

Example 10. Changing only Fc

```

/** Select the desired cutoff frequency */
CutOffValue=2;

/** Put the device in Standby Mode */
MMA865x_Standby();

/** Write in the cutoff value */
IIC_RegWrite(HP_FILTER_CUTOFF_REG, (IIC_RegRead(HP_FILTER_CUTOFF_REG)& ~SEL_MASK);
IIC_RegWrite(HP_FILTER_CUTOFF_REG, (IIC_RegRead(HP_FILTER_CUTOFF_REG) | CutOffValue);

/* ** Put the device back in the active mode */
MMA865x_Active();

```

7.1 High-pass filtered data or low-pass filtered data

Registers 0x01 – 0x06 store the X, Y, Z data. The device can be configured to produce high-pass filtered data or low-pass filtered data, by setting or clearing the HPF_OUT bit in XYZ_DATA_CFG register (0x0E). The next code example shows how to set the HPF_OUT bit.

Example 11. Selecting high-pass or low-pass filtered data

```

/** Put the device in Standby Mode */
MMA865x_Standby();

/** Set the HPF_OUT Bit to enable the HPF Data Out */
IIC_RegWrite(XYZ_DATA_CFG_REG, (IIC_RegRead(XYZ_DATA_CFG_REG) | HPF_OUT_MASK));

/** Put the device back into Active Mode */
MMA845x_Active();

```

8 12-bit or 10-bit Data Streaming and Data Conversions

The MMA8652FC has 12-bit XYZ data and the MMA8653 has 10-bit data. This section is an overview of how to manipulate the data to continuously burst out 12-bit data in different data formats from the MCU. The examples are shown for the 12-bit data, but you can understand what changes would be made for the 10-bit data. The driver code has all the functions for all data formats available.

The event flag can be monitored by reading the STATUS register (0x00). This can be done by using either a polling or interrupt technique, which is discussed later in [Section 10, “Polling Data vs. Interrupts,” on page 28](#).

Reading the STATUS register does not clear the STATUS register; *reading the data* clears the STATUS register.

Table 11. 0x00 STATUS: Data status registers (read-only)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ZYXOW	ZOW	YOW	XOW	ZYXDR	ZDR	YDR	XDR

The ZYXDR flag is set whenever there is new data available in any axis. The following code example monitors this flag and, upon the detection of new data, reads the 12/10-bit XYZ data into an array (`value[]`) in RAM with a single, multi-byte I²C access. These values are then copied into 16-bit variables prior to further processing.

Example 12. Using ZYXDR flag to detect new data

```
/** Poll the ZYXDR status bit and wait for it to set */
RegisterFlag.Byte = IIC_RegRead(STATUS_00_REG);

if (RegisterFlag.ZYXDR_BIT == 1)
{
    /** Read 12/10-bit XYZ results using a 6 byte IIC access */
    IIC_RegReadN(OUT_X_MSB_REG, 6, &value[0]);

    /** Copy and save each result as a 16-bit left-justified value */
    x_value.Byte.hi = value[0];
    x_value.Byte.lo = value[1];
    y_value.Byte.hi = value[2];
    y_value.Byte.lo = value[3];
    z_value.Byte.hi = value[4];
    z_value.Byte.lo = value[5];
}
```

The corresponding 16-bit results in left-justified format (2’s complement numbers) are provided, as an example of the register and variable formats for the X-axis result below:

Table 12. 0x00 x_value.Byte.hi: X_MSB register MMA8652FC (read-only)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XD11	XD10	XD9	XD8	XD7	XD6	XD5	XD4

Table 13. 0x01 x_value.Byte.lo: X_LSB register MMA8652FC (read-only)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XD3	XD2	XD1	XD0	0	0	0	0

Table 14. x_value 16-bit 2's complement result

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XD11	XD10	XD9	XD8	XD7	XD6	XD5	XD4	XD3	XD2	XD1	XD0	0	0	0	0

8.1 Converting 12-bit 2's complement hex number to signed integer (counts)

Converting a *signed value* into *counts* implies that the 2's complement hex number is converted to an integer number with a + or – sign. For example,

$$0xABC0 = -836$$

$$0x1510 = +337$$

The sign of the result is easy to determine, by checking the high byte of the value. If the high byte of the value is greater than 0x7F, then the value is a negative number and needs to be transformed (by performing a 2's complement conversion). This involves executing a 1's complement (i.e., switch all 1's to 0's and all 0's to 1's), followed by adding 1 to the result.

The code below performs this conversion. It also adds the additional output formatting step of replacing each leading zero digit with a space character, which is done by passing 0xF0 to SCI_NibbOut(). Upon close examination, it is seen that this routine will add 0x30 to 0xF0, resulting in a value of 0x120, which gets truncated to 0x20 (the ASCII space character).

Example 13. Converting a signed value into counts

```
void SCI_s12dec_Out (tword data) {
    SCI_sDec_Out (data, 4);
}

/*****\
* SCI output signed N-bit left-justified integer as decimal
\*****/
void SCI_sDec_Out (tword data, unsigned char shift)
{
    byte a, b, c, d;
    word r;

    /* ** Determine sign and output */
```

```

if (data.Byte.hi > 0x7F)
{
    SCI_CharOut ('-');
    data.Word = ~data.Word + 1;
}
else
{
    SCI_CharOut ('+');
}

/* ** Calculate */
a = (byte)((data.Word >> shift) / 1000);
r = (data.Word >> shift) % 1000;
b = (byte)(r / 100);
r %= 100;
c = (byte)(r / 10);
d = (byte)(r % 10);

/* ** Format */
if (a == 0)
{
    a = 0xF0;
    if (b == 0)
    {
        b = 0xF0;
        if (c == 0)
        {
            c = 0xF0;
        }
    }
}

/* ** Output result */
SCI_NibbOut (a);
SCI_NibbOut (b);
SCI_NibbOut (c);
SCI_NibbOut (d);
}

```

8.2 Converting 12-bit 2's complement hex number to signed decimal fraction (in g's)

Based on different microcontroller architecture, the conversation from 12/10 bit 2's complement hex number to signed decimal factions is different.

- A 32-bit architecture MCU has enough bit length to calculate the mg-values following one simple formula.
- Conversion on the 16-bit and 8-bit architecture require multiple steps: calculating the integer and the fractions separately, and adding them together.

8.2.1 Conversion using a 32-bit architecture MCU

When using a 32-bit architecture MCU, the formula below converts the 12-bit 2's complement to a fractional g-value. The 32-bit MCU has the 32-bit arithmetic unit that can contain the calculation result without truncating it. In the cases where a 32-bit output variable can be arranged via software, this formula can also be considered.

$$\text{Result} = (1000 \times \text{data.DWord} + 512) \gg 10$$

This formula is based on the MMA8652FC device property, with a 2-g accelerometer dynamic scale and 12-bit digital resolution, where 1g = 1000 mg = 1024 counts. For example,

$$0x7FF0 \text{ count} = +1023 \text{ mg}$$

$$0xFFF0 \text{ count} = -1 \text{ mg}$$

$$0xF000 \text{ count} = -1024 \text{ mg}$$

Data.DWord is the signed accelerometer value extended to 32 bits. The next formula can be used to extend the data to any processor word length. To start the extension, the left-aligned 12-bit data should first be right-aligned, followed by a 4-bit right-shifting operation. Its signed bit should next be examined (compared), to identify if this value is negative. If the signed bit is negative, then the negative sign is extended by minus 0x4000.

Example 14. Extend a signed number to the microcontroller ALU length

```
Data.DWord = Data.Word >> 4;  
if (Data.DWord >= 0x1000)  
    Data.DWord -= 0x2000;
```

A 10-bit right shift operation is used to accomplish the division by 1024. The fractions greater than or equal to 0.5 (if the division operation is chosen) would have been truncated by the shifting operation. To compensate for this drawback that the shifting operation brings (and to get a more accurate rounded number), 512 is added *before shifting*. This addition allows the fractions greater or equal to 0.5 be rounded up to an integer after the shift.

Example 15. Fractional conversion for 32-bit architecture MCU

```
void SCI_s12frac_Out (tword data)  
{  
    dword result;
```

```

byte a, b, c, d;
word r;

data.DWord = data.Word >> 4;
if (data.DWord >= 0x1000)
    data.DWord = data.DWord - 0x2000;
result = (1000 * data.DWord + 512) >> 10;

/** Determine sign and output */
if (result > 0x80000000)
{
    SCI_CharOut ('-');
    result = ~result + 1;
}
else
{
    SCI_CharOut ('+');
}

/** Convert mantissa value to 4 decimal places */
r = result % 1000;
a = (byte)(result / 1000);
b = (byte)(r / 100);
r %= 100;
c = (byte)(r / 10);
d = (byte)(r%10);

/** Output mantissa */
SCI_NibbOut (a);
SCI_NibbOut (b);
SCI_NibbOut (c);
SCI_NibbOut (d);
SCI_CharOut ('mg')
}

```

The MMA8652x device, when run at different dynamic scales, has its own formula. Refer to [Table 15](#) for its corresponding formula.

Table 15. Convert MMA865xFC 2's complement hex number to signed decimal fractions (in g's)

Device	Full Scale Value	counts/g	Formula
MMA8652FC	±2g	1024	Result = (1000 x data.DWord + 512) >> 10
MMA8652FC	±4g	512	Result = (1000 x data.DWord + 256) >> 9
MMA8652FC	±8g	256	Result = (1000 x data.DWord + 128) >> 8
MMA8653FC	±2g	256	Result = (1000 x data.DWord + 128) >> 8
MMA8653FC	±4g	128	Result = (1000 x data.DWord + 64) >> 7
MMA8653FC	±8g	64	Result = (1000 x data.DWord + 32) >> 6

8.2.2 Conversion using a 16-bit / 8-bit architecture MCU

Converting to a signed value into g's requires performing the same operations as shown previously, with the added step of resolving the integer and fractional portions of the value. The scale of the accelerometer's Active mode (either 2g, 4g or 8g) determines the location of the inferred radix point separating these segments, and thereby the overall sensitivity of the result. In all cases, the most significant bit (bit 11) represents the sign of the result (either positive or negative).

- **In 2g Active mode**, 1g = 1024 counts.
Therefore, bit 10 is the only bit that will contribute to an integer value of either 0, 1. $2^{12} = 4096$. Bits 9 – 0 will be fractional values.
- **In 4g Active mode**, 1g = 512 counts.
Therefore, bits 10 and 9 will contribute to an integer value of 0, 1, 2, or 3. Bits 8 – 0 will be fractional values.
- **In 8g Active mode**, 1g = 256 counts.
Therefore, bits 10, 9 and 8 will contribute to an integer value of 0, 1, 2, 3, 4, 5, 6, and 7. Bits 7 – 0 will be fractional values.

Table 16. Full-scale value with corresponding integer bits and fraction bits

Full Scale Value	Counts/g	Sign Bit	Integer Bits	Fraction Bits
2g	1024	11	10 ($2^{10} = 1024$)	0 – 9
4g	512	11	9 ($2^9 = 512$), 8 ($2^8 = 256$)	0 – 8
8g	256	11	10 ($2^{10} = 1024$), 9 ($2^9 = 512$), 8 ($2^8 = 256$)	0 – 7

8.2.3 2g Active mode

Adjusting the data into 16-bit left-justified format, the implied radix point of a result when operating in 2g Active mode is between word format bits 12 and 11, as can be seen in Table 17. In the table:

- The “MMA8652FC 12b” row shows where the 12 bits of the result are placed in this format.
- The “MSB/LSB” row indicates which result register was the source of the data, either the most-significant byte (“M”) or the least-significant byte (“L”).
- The “Integer/Fraction” row shows that bit 15 is the sign bit (“±”), while the single integer bit is located at bit 14 (“I”) from the word format.

Table 17. 2g Active mode 12-bit data conversion to decimal fraction number

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	11	10	9	8	7	6	5	4	3	2	1	0	x	x	x	x
Integer/Fraction	±	I	F	F	F	F	F	F	F	F	F	F	x	x	x	x
MSB/LSB	M	M	M	M	M	M	M	M	L	L	L	L	0	0	0	0

Once the sign and integer of the result have been determined, the result is logically shifted to the left by 2 binary locations, leaving only the fraction portion of the result, as can be seen in [Table 18](#).

Table 18. 2g Active mode 12-bit data in word format after left-shift, to eliminate integer and sign bits

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	9	8	7	6	5	4	3	2	1	0	x	x	x	x	x	x
Integer/Fraction	F	F	F	F	F	F	F	F	F	F	x	x	x	x	x	x
Fraction Bits	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	x	x	x	x	x	x
MSB/LSB	M	M	M	M	M	M	L	L	L	L	0	0	0	0	0	0

This leaves 6 MSB bits and 4 LSB bits after shifting left by 2. Therefore there are 10 bits for the fraction portion in 2g Active mode. The 2g Active mode has the highest number of bits for the fraction portion with 12 bits, because it has the highest sensitivity. In [Table 19](#), the decimal value is rounded to the fourth decimal place because the final fraction number will have 4 significant digits. This should be sufficient.

The values shown in [Table 19](#) are translated here into C macros, for use in the code example at the end of this section.

Table 19. 2g Active mode fraction values

	2g Mode	Calculation (1024 counts/g)	Rounded to 4 th Decimal Place	Integer Number
2^{-1}	$2^9 = 512$	$512/1024 = 0.5$	0.5000	5000
2^{-2}	$2^8 = 256$	$256/1024 = 0.25$	0.2500	2500
2^{-3}	$2^7 = 128$	$128/1024 = 0.125$	0.1250	1250
2^{-4}	$2^6 = 64$	$64/1024 = 0.0625$	0.0625	625
2^{-5}	$2^5 = 32$	$32/1024 = 0.03125$	0.0313	313
2^{-6}	$2^4 = 16$	$16/1024 = 0.015625$	0.0156	156
2^{-7}	$2^3 = 8$	$8/1024 = 0.0078125$	0.0078	78
2^{-8}	$2^2 = 4$	$4/1024 = 0.00390625$	0.0039	39
2^{-9}	$2^1 = 2$	$2/1024 = 0.001953125$	0.0020	20
2^{-10}	$2^0 = 1$	$1/1024 = 0.0009765625$	0.0010	10

Example 16. C macros

```
#define FRAC_2d1          5000
#define FRAC_2d2          2500
#define FRAC_2d3          1250
#define FRAC_2d4           625
#define FRAC_2d5           313
#define FRAC_2d6           156
#define FRAC_2d7            78
#define FRAC_2d8            39
#define FRAC_2d9            20
#define FRAC_2d10         10
```

For each of the 10 fraction bits, if the value of the bit is set then the corresponding decimal value will be added to the total. For example, if bits 6, 4 and 2 are set, then the total will be $(625 + 156 + 39 = 820)$, which corresponds to 0.0820g.

The highest fractional value occurs when all fraction bits are set $(5000 + 2500 + 1250 + 625 + 313 + 156 + 78 + 39 + 20 + 10 = 9990)$, which corresponds to 0.9990g. In 2g Active mode, the resolution is 0.977 mg $(1/1024)$. Calculating out the fraction to 4 significant digits gives a resolution of 1.0 mg for 2g Active mode.

8.2.4 4g Active mode

In 4g Active mode, there are 2 integer bits and 9 fraction bits (as shown in [Table 20](#)).

Table 20. 4g Active mode 12-bit data conversion to decimal fraction number

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	11	10	9	8	7	6	5	4	3	2	1	0	x	x	x	x
Integer/Fraction	±	I	I	F	F	F	F	F	F	F	F	F	x	x	x	x
MSB/LSB	M	M	M	M	M	M	M	M	L	L	L	L	0	0	0	0

In this case, logically shifting the sample to the left by 3 binary locations leaves the fractional portion of the result, as shown in [Table 21](#). [Table 21](#) shows the 9 bits of the fraction, with the corresponding decimal values for each bit shown in [Table 22](#).

Table 21. 4g Active Mode 12-bit in word format after left-shift, to eliminate integer and sign bits

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	8	7	6	5	4	3	2	1	0	x	x	x	x	x	x	x
Integer/Fraction	F	F	F	F	F	F	F	F	F	x	x	x	x	x	x	x
Fraction Bits	-1	-2	-3	-4	-5	-6	-7	-8	-9	x	x	x	x	x	x	x
MSB/LSB	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0

Table 22. 4g Active mode fraction values

	4g Mode	Calculation (512 counts/g)	Rounded to 4 th Decimal Place	Integer Number
2^{-1}	$2^8 = 256$	$256/512 = 0.5$	0.5000	5000
2^{-2}	$2^7 = 128$	$128/512 = 0.25$	0.2500	2500
2^{-3}	$2^6 = 64$	$64/512 = 0.125$	0.1250	1250
2^{-4}	$2^5 = 32$	$32/512 = 0.0625$	0.0625	625
2^{-5}	$2^4 = 16$	$16/512 = 0.03125$	0.0313	313
2^{-6}	$2^3 = 8$	$8/512 = 0.015625$	0.0156	156
2^{-7}	$2^2 = 4$	$4/512 = 0.0078125$	0.0078	78
2^{-8}	$2^1 = 2$	$2/512 = 0.00390625$	0.0039	39
2^{-9}	$2^0 = 1$	$1/512 = 0.00195312$	0.0020	20

For each of the 9 fraction bits, if the value of the bit is set, then the corresponding decimal value will be added to the total. For example, if bits 6, 4 and 2 are set, then the total will be $(1250 + 313 + 78 = 1641)$, which corresponds to 0.1641g.

The highest fractional value occurs when all fraction bits are set $(5000 + 2500 + 1250 + 625 + 313 + 156 + 78 + 39 + 20 = 9980)$, which corresponds to 0.9980g. The resolution in 4g Active mode is 1.953 mg. Calculating the fractional value to 4 significant digits results in a resolution of 2.0 mg.

8.2.5 8g Active mode

In 8g Active mode, there are 3 integer bits, leaving 8 bits for the fraction (as per [Table 23](#)).

Table 23. 8g Active mode 12-bit data conversion to decimal fraction number

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	11	10	9	8	7	6	5	4	3	2	1	0	x	x	x	x
Integer/Fraction	±	I	I	I	F	F	F	F	F	F	F	F	x	x	x	x
MSB/LSB	M	M	M	M	M	M	M	M	L	L	L	L	0	0	0	0

The fractional portion of the result can be extracted, by logically shifting the sample to the left by 4 binary locations. This result is shown in [Table 24](#), with [Table 25](#) providing the corresponding decimal values.

Table 24. 8g Active mode 12-bit in word format after left-shift, to eliminate integer and sign bits

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652FC 12b	7	6	5	4	3	2	1	0	x	x	x	x	x	x	x	x
Integer/Fraction	F	F	F	F	F	F	F	F	x	x	x	x	x	x	x	x
Fraction Bits	-1	-2	-3	-4	-5	-6	-7	-8	x	x	x	x	x	x	x	x
MSB/LSB	M	M	M	M	L	L	L	L	0	0	0	0	0	0	0	0

Table 25. 8g Active mode fraction values

	8g Mode	Calculation (256 counts/g)	Rounded to 4 th Decimal Place	Integer Number
2^{-1}	$2^7 = 128$	$128/256 = 0.5$	0.5000	5000
2^{-2}	$2^6 = 64$	$64/256 = 0.25$	0.2500	2500
2^{-3}	$2^5 = 32$	$32/256 = 0.125$	0.1250	1250
2^{-4}	$2^4 = 16$	$16/256 = 0.0625$	0.0625	625
2^{-5}	$2^3 = 8$	$8/256 = 0.03125$	0.0313	313
2^{-6}	$2^2 = 4$	$4/256 = 0.01563$	0.0156	156
2^{-7}	$2^1 = 2$	$2/256 = 0.007812$	0.0078	78
2^{-8}	$2^0 = 1$	$1/256 = 0.003906$	0.0039	39

For each of the 8 fraction bits, if the value of the bit is set, then the corresponding decimal value will be added to the total. For example, if bit 7, 4 and 2 are set, then the total will be $(5000 + 625 + 156 = 5781)$, which corresponds to 0.5781g.

The highest fractional value occurs when all fraction bits are set $(5000 + 2500 + 1250 + 625 + 313 + 156 + 78 + 39 = 9961)$, which corresponds to 0.9961g. The resolution in 8g Active mode is 3.92 mg.

Calculating the fractional value to four significant digits results in 3.9 mg resolution.

Next is the code example that performs the conversion of a 12-bit signed 2's complement value into a signed decimal fraction (displayed in g's). This routine can also be used to convert 12-bit, 10-bit or 8-bit data. The extra unused bits will remain zeros.

Example 17. Converting 12-bit signed 2's complement to a signed decimal fraction

```
void SCI_s12frac_Out (tword data)
{
    BIT_FIELD value;
    word result;
    byte a, b, c, d;
    word r;

    /*** Determine sign and output */
    if (data.Byte.hi > 0x7F)
    {
        SCI_CharOut ('-');

        data.Word &= 0xFFFF0;
        data.Word = ~data.Word + 1;
    }
    else
    {
        SCI_CharOut ('+');
    }

    /*** Determine integer value and output */
    if (full_scale == FULL_SCALE_2G)
    {
        SCI_NibbOut((data.Byte.hi & 0x40) >>6);
        data.Word = data.Word <<2;
    }
    else if (full_scale == FULL_SCALE_4G)
    {
        SCI_NibbOut((data.Byte.hi & 0x60) >>5);
        data.Word = data.Word <<3;
    }
    else
    {
        SCI_NibbOut((data.Byte.hi & 0x70) >>4);
        data.Word = data.Word <<4;
    }

    SCI_CharOut ('.');

    /*** Determine mantissa value */
    result = 0;
    value.Byte = data.Byte.hi;
    if (value.Bit._7 == 1)
        result += FRAC_2d1;
    if (value.Bit._6 == 1)
        result += FRAC_2d2;
}
```

```
if (value.Bit._5 == 1)
    result += FRAC_2d3;
if (value.Bit._4 == 1)
    result += FRAC_2d4;
//
data.Word = data.Word <<4;
value.Byte = data.Byte.hi;
//
if (value.Bit._7 == 1)
    result += FRAC_2d5;
if (value.Bit._6 == 1)
    result += FRAC_2d6;
if (value.Bit._5 == 1)
    result += FRAC_2d7;
if (value.Bit._4 == 1)
    result += FRAC_2d8;
//
if (full_scale != FULL_SCALE_8G)
{
    if (value.Bit._3 == 1)
        result += FRAC_2d9;
}
if (full_scale == FULL_SCALE_2G)
{
    if (value.Bit._2 == 1)
        result += FRAC_2d10;
}
}
```


9 8-bit XYZ Data Streaming and Conversions

The MMA8652,3FC can provide 8-bit XYZ high-pass filtered data (MMA8652FC only) or low-pass filtered data. To read out 8-bit data, the F_READ bit in Register 0x2A must be set.

As was shown with 12-bit data, the ZYXDR flag in the STATUS register should be monitored.

The next code example shows how to read the 8-bit sample data by polling the STATUS register located at 0x00. Note that the samples are saved in 16-bit left-justified format, so that the data conversion subroutines previously described, can be reused here.

Example 18. Reading 8-bit sample data by polling the STATUS register

```

/** Poll the ZYXDR status bit and wait for it to set */
RegisterFlag.Byte = IIC_RegRead(STATUS_00_REG);

if (RegisterFlag.ZYXDR_BIT == 1)
{
    /** Read 8-bit XYZ results using a 3 byte IIC access */
    IIC_RegReadN(OUT_X_MSB_REG, 3, &value[0]);

    /** Copy and save each result as a 16-bit left-justified value */
    x_value.Byte.hi = value[0];
    x_value.Byte.lo = 0;
    y_value.Byte.hi = value[1];
    y_value.Byte.lo = 0;
    z_value.Byte.hi = value[2];
    z_value.Byte.lo = 0;
}

```

The 8-bit values can be converted to the various formats described previously for the 12-bit samples, using the same conversion subroutines, if that the data is formatted appropriately. This is easily done by simply copying the sample into the upper 8-bits and “zero-filling” the lower byte, as shown in the code examples above. For a 12-bit X-axis sample, the result of this procedure is shown below.

Table 26. 0x01 OUT_X_MSB_REG: X_MSB register (read-only)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XD11	XD10	XD9	XD8	XD7	XD6	XD5	XD4

Table 27. x_value 16-bit 2’s complement result

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XD11	XD10	XD9	XD8	XD7	XD6	XD5	XD4	0	0	0	0	0	0	0	0

For more about applicable data conversion formats, see [Section 8, “12-bit or 10-bit Data Streaming and Data Conversions,”](#) on page 13. Specific 8-bit versions of the previous routines are provided in the following sections.

- For the MMA8652FC, the MSB bits are XD11 to XD4.
- For the MMA8653FC, the MSB bits are XD9 through XD2.

9.1 Converting 8-bit 2's complement hex number to signed integer number

Converting a signed value into counts implies that the 2's complement hex number is converted to an integer number with a + or – sign. For example,

0xAB = -85
0x54 = +84

This conversion is similar to the one shown previously, with the added step of converting an 8-bit binary value into a decimal result (which could contain up to 3 digits, i.e., 0x7F = +127). The code below performs this conversion. It also adds the additional output formatting step of replacing each leading zero digit with a space character, as described in [Section 8.1, “Converting 12-bit 2's complement hex number to signed integer \(counts\),”](#) on page 14.

Example 19. Converting an 8-bit binary value into a decimal number

```
void SCI_s8dec_Out (byte data)
{
    byte a, b, c;
    word r;

    /*** Determine sign and output **/
    if (data > 0x7F)
    {
        SCI_CharOut ('-');
        data = ~data + 1;
    }
    else
    {
        SCI_CharOut ('+');
    }

    /*** Calculate decimal equivalence:
    **   a = hundreds
    **   b = tens
    **   c = ones */
    a = (byte)(data / 100);
    r = (data) % 100;
    b = (byte)(r / 10);
    c = (byte)(r % 10);

    /*** Format adjustment for leading zeros **/
    if (a == 0)
    {
        a = 0xF0;
        if (b == 0)
        {
            b = 0xF0;

```

```

}
}

/** Output result */
SCI_NibbOut (a);
SCI_NibbOut (b);
SCI_NibbOut (c);
}

```

9.2 Converting 8-bit 2's complement hex number to signed decimal fraction (in g's)

Converting 8-bit data to a signed value (in g's) is done in the same manner as for 12-bit data, see [Section 8.2, "Converting 12-bit 2's complement hex number to signed decimal fraction \(in g's\),"](#) on page 16. Therefore, only the details regarding the use of 2g Active mode are described here, in [Section 9.2.1, "2g Active mode,"](#) on page 27.

The scale of the accelerometer's Active mode (either 2g, 4g or 8g) determines the location of the inferred radix point separating these segments, and thereby the overall sensitivity of the result. In all cases, the most significant bit (bit 7) represents the sign of the result (either positive or negative).

- **In 2g Active mode**, 1g = 64 counts.
Therefore, bit 6 is the only bit that will contribute to an integer value of either 0, 1. $2^6 = 64$.
- **In 4g Active mode**, 1g = 32 counts.
Therefore, bits 6 and 5 will contribute to an integer value of 0, 1, 2, or 3.
- **In 8g Active mode**, 1g = 16 counts.
Therefore, bits 6, 5 and 4 will contribute to an integer value of 0, 1, 2, 3, 4, 5, 6, and 7.

Table 28. Full-scale value with corresponding integer bits and fraction bits

Full-Scale Value	Counts/g	Sign Bit	Integer Bits	Fraction Bits
2g	64	7	6 ($2^6 = 64$)	0 – 5
4g	32	7	6 ($2^6 = 64$), 5 ($2^5 = 32$)	0 – 4
8g	16	7	6 ($2^6 = 64$), 5 ($2^5 = 32$), 4 ($2^4 = 16$)	0 – 3

9.2.1 2g Active mode

The subroutine provided in [Section 8, "12-bit or 10-bit Data Streaming and Data Conversions,"](#) on page 13 can be used to convert 8-bit data, if the data has been adjusted into 16-bit left-justified format, as shown in [Table 29](#).

Table 29. 2g Active mode 8-bit data conversion to decimal fraction number

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652,3FC 8b	7	6	5	4	3	2	1	0	x	x	x	x	x	x	x	x
Integer/Fraction	±	I	F	F	F	F	F	F	x	x	x	x	x	x	x	x
MSB/LSB	M	M	M	M	M	M	M	M	0	0	0	0	0	0	0	0

Performing a logical shift to the left by 2 binary locations will provide the result's fractional portion, as shown in [Table 30](#).

Table 30. 2g Active mode 8-bit data in word format after left-shift, to eliminate integer and sign bits

Word Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMA8652,3FC 8b	5	4	3	2	1	0	X	x	x	x	x	x	x	x	x	x
Integer/Fraction	F	F	F	F	F	F	X	x	x	x	x	x	x	x	x	x
Fraction Bits	-1	-2	-3	-4	-5	-6	X	x	x	x	x	x	x	x	x	x
MSB/LSB	M	M	M	M	M	M	0	0	0	0	0	0	0	0	0	0

The decimal values of the 6 bits of the fractional portion are shown in [Table 31](#). These values are rounded to the 4th decimal place, because the final fraction number will have 4 significant digits.

Table 31. 2g Active mode fraction values

	2g Mode	Calculation (64 counts/g)	Rounded to 4 th Decimal Place	Integer Number
2^{-1}	$2^5 = 32$	$32/64 = 0.5$	0.5000	5000
2^{-2}	$2^4 = 16$	$16/64 = 0.25$	0.2500	2500
2^{-3}	$2^3 = 8$	$8/64 = 0.125$	0.1250	1250
2^{-4}	$2^2 = 4$	$4/64 = 0.0625$	0.0625	625
2^{-5}	$2^1 = 2$	$2/64 = 0.03125$	0.0313	313
2^{-6}	$2^0 = 1$	$1/64 = 0.015625$	0.0156	156

For each of the 6 fraction bits, if the value of the bit is set, then the corresponding decimal value will be added to the total. The highest fractional value occurs when all fraction bits are set ($5000 + 2500 + 1250 + 625 + 313 + 156 = 9844$), which corresponds to 0.9844g. In 2g Active mode, the resolution is 15.625mg. Calculating out the fraction to 4 significant digits gives a resolution of 15.6mg for 2g Active mode.

Using the same methodology shown here for 2g Active mode, similar calculations and conversions can be performed for the 4g and 8g Active modes.

10 Polling Data vs. Interrupts

The data can be polled continuously or it can be set up to a hardware interrupt or exception to the MCU each time new data is ready. Depending on the circumstances one might be more desirable than the other although polling typically is less efficient.

10.1 Polling data

Polling requires less configuration of the device and is very simple to implement. However, the MCU must poll the sensor at a rate that is faster than the Output Data Rate. Otherwise, if the polling is too slow, the data samples can be missed. The MCU can detect this condition, by checking the overwrite flags in the STATUS register (ZYGOW, ZOW, YOW, and XOW).

The code examples provided so far in this document have primarily described the polling technique. As a summary, here is a more complete example of the basic code (specific to the operation of the MMA8652,3FC) required to continuously poll 12-bit or 10-bit XYZ data.

Example 20. Continuously polling for 12-bit or 10-bit XYZ data

```
/** Go to the Standby Mode */
MMA865xFC_Standby();

/** Clear the F_Read bit to ensure both MSB's and LSB's are indexed */
IIC_RegWrite(CTRL_REG1, (IIC_RegRead(CTRL_REG1) & ~FREAD_MASK));

/** Go back to Active Mode */
MMA865xFC_Active();

/** Using a basic control loop, continuously poll the sensor */
for (;;)
{
    /** Poll the ZYXDR status bit and wait for it to set */
    RegisterFlag.Byte = IIC_RegRead(STATUS_00_REG);

    if (RegisterFlag.ZYXDR_BIT == 1)
    {
        /** Read 12/10-bit XYZ results using a 6 byte IIC access */
        IIC_RegReadN(OUT_X_MSB_REG, 6, &value[0]);

        /** Copy and save each result as a 16-bit left-justified value */
        x_value.Byte.hi = value[0];
        x_value.Byte.lo = value[1];
        y_value.Byte.hi = value[2];
        y_value.Byte.lo = value[3];
        z_value.Byte.hi = value[4];
        z_value.Byte.lo = value[5];

        /** Go process the XYZ data */
        GoProcessXYZ(&value[0]);
    }

    /** Perform other necessary operations */
}
}
```

10.2 Reading data using an interrupt routine

Streaming data via hardware interrupts is more efficient than polling, because the MCU only interfaces with the accelerometer when it has new data. The data is read only when new data is available. If the data is not read every time there is a new sample, this will be indicated by the overwrite register flags.

To configure the interrupt pin:

- Put the device into Standby mode.
- Choose the interrupt polarity, and a pull-push or open drain option, using the Interrupt Control register CTRL_REG3 (0x2C).
- Enable the interrupt source, using the Interrupt Enable register CTRL_REG4 (0x2D).
- Choose the pin that interrupts route to, using the Interrupt Configuration register CTRL_REG5 (0x2E).
- Put the device back into the Active mode.

The next example shows the register settings to configure the device to generate an interrupt upon each new data ready. The MCU's Interrupt Service Routine (ISR) shown below responds by reading the 12-bit or 10-bit XYZ data, and then setting a software flag indicating the arrival of new data. It is considered good practice to keep ISRs as fast as possible, so the actual processing of this data is not done here. Note the similarities to the polling method. Accessing 8-bit data can also be performed similarly.

Example 21. Generating an interrupt when new accelerometer data is ready

```

/** Go to the Standby Mode */
MMA865xFC_Standby();

/** Clear the F_Read bit to ensure both MSB's and LSB's are indexed */
IIC_RegWrite(CTRL_REG1, (IIC_RegRead(CTRL_REG1) & ~FREAD_MASK);

/** Configure the INT pins for Open Drain and Active Low */
IIC_RegWrite(CTRL_REG3, PP_OD_MASK);

/** Enable the Data Ready Interrupt and route it to INT1 */
IIC_RegWrite(CTRL_REG4, INT_EN_DRDY_MASK);
IIC_RegWrite(CTRL_REG5, INT_CFG_DRDY_MASK);

/** Go back to Active Mode */
MMA865xFC_Active();

/** Perform other necessary operations */

\*****\
* MMA8652,3FC Interrupt Service Routine
\*****/
interrupt void isr_MMA8652FC (void)
{

    /** Clear the MCU's interrupt flag */
    CLEAR_MMA8652FC_INTERRUPT;

    /** Go read the Interrupt Source Register */
    RegisterFlag.Byte = IIC_RegRead(INT_SOURCE_REG);
    if (RegisterFlag.SRC_DRDY_BIT == 1)
    {

        /** Read 12 or 10-bit XYZ results using a 6 byte IIC access */
        IIC_RegReadN(OUT_X_MSB_REG, 6, &value[0]);
    }
}

```

```
    /*** Copy and save each result as a 16-bit left-justified value */
    x_value.Byte.hi = value[0];
    x_value.Byte.lo = value[1];
    y_value.Byte.hi = value[2];
    y_value.Byte.lo = value[3];
    z_value.Byte.hi = value[4];
    z_value.Byte.lo = value[5];

    /*** Indicate that new data exists to be processed */
    NEW_DATA = TRUE;
}
}
```

11 Using the 32-Sample FIFO

The most efficient way to access data (particularly for data logging) is to use the internal 32-sample FIFO buffer. This minimizes the number of I²C transactions.

For more about how to configure the FIFO, see MMA845x application notes AN4073. The FIFO can be configured in different modes:

- Circular Buffer mode: In Circular Buffer mode, the device discards the oldest data when overflowed.
- Stop and Overflow mode
- Trigger mode
- Disabled mode

This configuration can be set with the F_SETUP FIF setup register (0x09).

NOTE

The MMA8652FC device has this FIFO feature; the MMA8653FC device does not have this FIFO feature.

Configure the FIFO example:

- Circular Buffer mode F_MODE = 01
- Set the Watermark
- Set the FIFO Interrupt
- Route the FIFO Interrupt to INT2
- Set the Interrupt Pins for Open Drain Active Low

Example 22. Using the FIFO

```
/** Go to Standby Mode */
MMA865xFC_Standby();

/** Set F_Mode to Trigger, Set the Watermark Value into the F_SETUP register */
IIC_RegWrite(F_SETUP_REG, F_MODE_TRIGGER + WATERMARK_VAL);

/** Enable the FIFO Interrupt and Set it to INT2 */
IIC_RegWrite(CTRL_REG4, INT_EN_FIFO_MASK);
IIC_RegWrite(CTRL_REG5, ~INT_CFG_FIFO_MASK);

/** Configure the INT pins for Open Drain and Active Low */
IIC_RegWrite(CTRL_REG3, PP_OD_MASK);

/** Go to Active Mode */
MMA865xFC_Active();

/*****\
* MMA8652FC Interrupt Service Routine for the FIFO
\*****/
interrupt void isr_MMA8652FC (void)
```



```

{
  /*** Clear the MCU's interrupt flag */
  CLEAR_MMA8652FC_INTERRUPT;

  /*** Go read the Interrupt Source Register */
  RegisterFlag.Byte = IIC_RegRead(INT_SOURCE_REG);
  if (RegisterFlag.SRC_FIFO_BIT == 1)
  {
    /*** Read 12-bit XYZ results using a multi-read IIC access */
    IIC_RegReadN(OUT_X_MSB_REG, WATERMARK_VAL*6, &value[0]);

    /*** Read FIFO STATUS register to clear the FIFO interrupt bit */
    RegisterFlag.Byte = IIC_RegRead(F_SETUP_REG);

    /*** Copy and save each result as a 16-bit left-justified value */
    x_value.Byte.hi = value[0];
    x_value.Byte.lo = value[1];
    y_value.Byte.hi = value[2];
    y_value.Byte.lo = value[3];
    z_value.Byte.hi = value[4];
    z_value.Byte.lo = value[5];

    /*** Indicate that new data exists to be processed */
    NEW_DATA = TRUE;
  }
}

```

12 MMA865x Driver Quick Start

This section describes the MMA865x Driver and how to use it, in conjunction with the Windows HyperTerminal program.

Table 32. Tools

Hardware	Use the MMA8491FC Sensor Toolbox Kit (LFSTBEB8491 or RDMMA8491).
Communication tool	Please use Windows HyperTerminal or any other communication tool.
Programming tool	To program the driver to the Sensor Toolbox board or to make any modifications to the program, use CodeWarrior v6.3. Download this tool at: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-MICROCONTROLLERS

12.1 Overview

This driver program enables you to quickly use the MMA865xFC devices via the I²C bus. The driver is also a reference program that you can use. The driver program is:

- based on the MMA865xFC Sensor toolbox (STB) hardware kit.
- intended to be downloaded onto the MC9S08QE8 microcontroller (on the baseboard).

NOTE

The Sensor toolbox kit will NOT be recognized by the STB software after programming.

You can use any Terminal emulator program to control the device operation from a computer. In this application note, we use Windows HyperTerminal. A list of commands is available in the next section.

This driver provides the essential APIs to set up MMA865xFC devices, read/write registers, stream data in data polling, interrupt, interrupt with FIFO modes, and display data in different formats.

12.2 Commands list

Table 33. MMA865xFC driver commands (in HyperTerminal)

Command	Function	Echo	Notes/ Comments
RR xx	Read Register	RR xx = nn	
RW xx = nn	Write Register	Fail/Success	Device reads back the register value written to confirm successful write.
RH n	Read High-Pass Filter	OverSample = xx ODR = xx HP = xx Mode = xx	High-Pass Filter 0 – 3 (cutoff frequency), 4 = Off. Example of response with RH 2 command: "OverSample = Normal ODR = 800 Hz HP = 4 Hz Mode = 2g"
RF	Report Current Configuration	OverSample = xx ODR = xx HP = xx Mode = xx	Example of response with RF command: "OverSample = Normal ODR = 800 Hz HP = 4 Hz Mode = 2g"

Table 33. MMA865xFC driver commands (in HyperTerminal) (Continued)

Command	Function	Echo	Notes/ Comments
CN (Normal Data) or CH (High-Pass Filter Data)	Read XYZ as signed counts	X = nn Y = nn Z = nn	Example of response: X = -250 Y = -26 Z = + 968
GN (Normal Data) or GH (High-Pass Filter Data)	Read XYZ as signed g's	X = nng Y = nng Z = nng	Example of response: X = -0.1992g Y = -0.0577g Z = +0.9473g
S aa: (aa: CN = counts Normal, CH = counts HPF, GN = g's Normal, GH = g's HPF)	Stream XYZ, polling	X = nn Y = nn Z = nn, repeated	Example of response: X = -250 Y = -126 Z = + 968, repeated.
I aa n (aa: CN=counts Normal, CH = counts HPF, GN = g's Normal, GH = g's HPF n: 1 = INT1; 2 = INT2)	Stream XYZ, interrupts	X = nn Y = nn Z = nn, repeated	Example of response: X = -250 Y = -126 Z = + 968, repeated.
F aa ww: (aa: CN=counts Normal, CH = counts HPF, GN = g's Normal, GH = g's HPF, ww : Watermark = 1 to 31)	Stream XYZ, FIFO	FIFO Watermark Samples= x group= xx X = xx Y = xx Z = xx	Example of response: FIFO Watermark Samples = 3 group = 3B, repeated. X = -214 Y = -39 Z = +968 X = -216 Y = -40 Z = +967 X = -12 Y = -37 Z = +970

12.3 Quick start procedure

12.3.1 Initial set-up

1. Connect the USB cable to power the STB board.
2. Make sure the SW1 is on. The power LED CR1 should be lit.
3. Connect the programmer to J1 on the USB communication board of the STB kit.
4. Using CodeWarrior v6.3, open the MMA865xFC driver program.
5. Program the driver to the MC9S08QE8 Microcontroller.

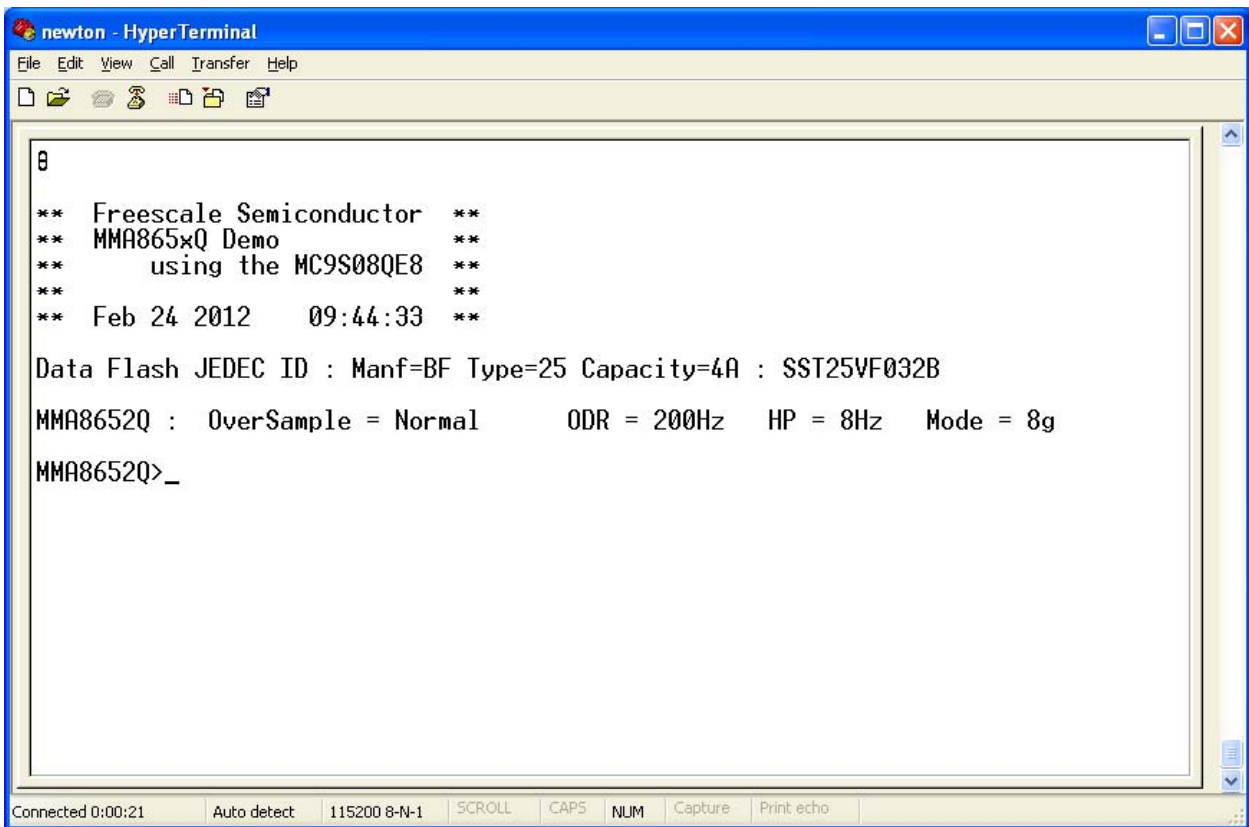
NOTE

The Sensor toolbox kit will NOT be recognized by the STB software after programming.

6. Disconnect the programmer.
7. Turn off, then turn on SW1, to allow the part to restart.

12.3.2 Run

1. Run HyperTerminal.
2. Configure the HyperTerminal port:
 - Bits Per Second: **115200**
 - Data Bits: **8**
 - Parity: **None**
 - Stop Bits: **1**
 - Flow Control: **None**
3. Enter any key to bring up the initial prompt, as shown below. This menu header only appears once when the STB board is first powered on. The MMA865xFC device is recognized and its details are listed.



The screenshot shows a HyperTerminal window titled "newton - HyperTerminal". The window contains the following text:

```
␣  
  
** Freescale Semiconductor **  
** MMA865xQ Demo **  
** using the MC9S08QE8 **  
** Feb 24 2012 09:44:33 **  
  
Data Flash JEDEC ID : Manf=BF Type=25 Capacity=4A : SST25VF032B  
MMA8652Q : OverSample = Normal ODR = 200Hz HP = 8Hz Mode = 8g  
MMA8652Q>_
```

The status bar at the bottom of the window shows: "Connected 0:00:21 Auto detect 115200 8-N-1 SCROLL CAPS NUM Capture Print echo".

Figure 2. HyperTerminal screen at power-up

4. Enter: "?" will bring up the following menu. Menu commands are explained in [Table 33](#), "MMA865xFC driver commands (in HyperTerminal)," on page 34.

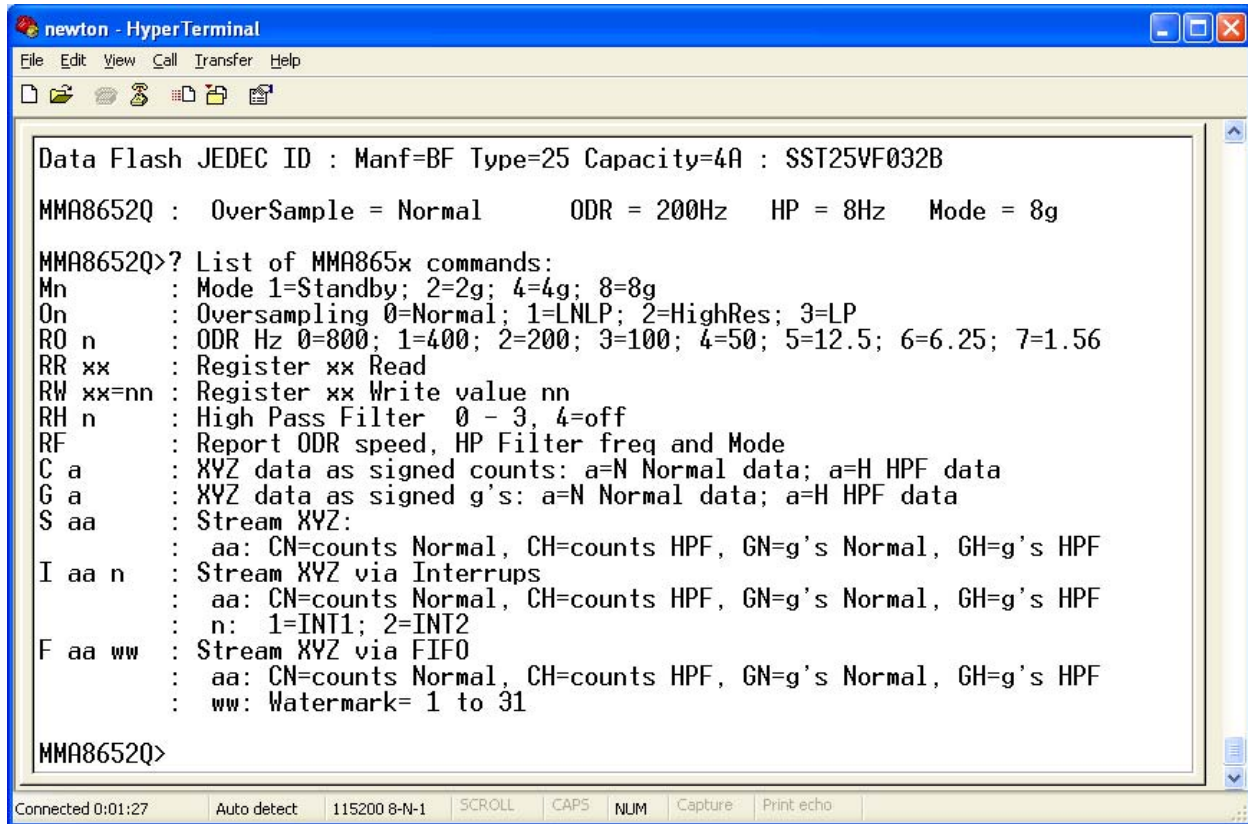


Figure 3. HyperTerminal menu

5. Type in the command to set up the device sampling conditions or to start sampling.

NOTE

The commands are not case sensitive.

6. Press any key to exit out of the data streaming operation.

12.3.3 Acquire data

1. In HyperTerminal, select Transfer 'Capture Text'.
2. Save the file to a known location and select Start.

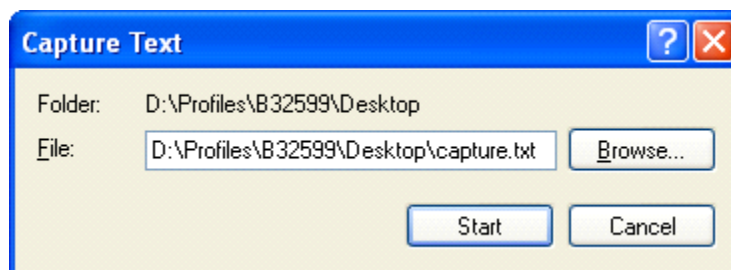


Figure 4. HyperTerminal Capture Text window

3. Select one of the stream commands. The data will now log.
4. When a desired amount of data has been collected, select Transfer 'Capture Text ' Stop
5. Open Excel.
6. Open the log file.

NOTE

If the file does not appear, select the 'Files Of Type' dropdown menu and select 'All Files (*.*)'.

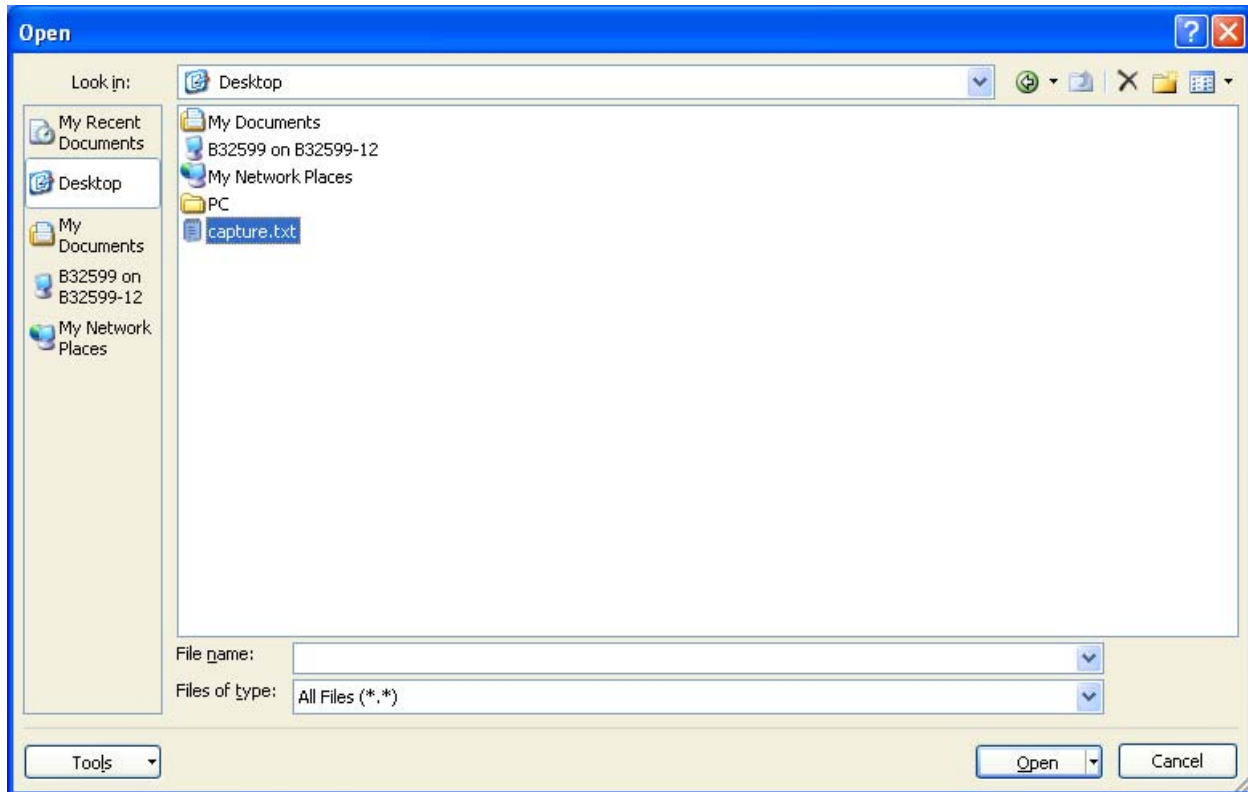


Figure 5. Select 'Files of Type' to be 'All Files'

7. The Text Import Wizard will appear; select the Delimited option, then select Next.
8. Select the Space option; in 'Other' type in "=". Select Finish
9. The data set will have 6 columns: X, X Data, Y, Y Data, Z, and Z Data.

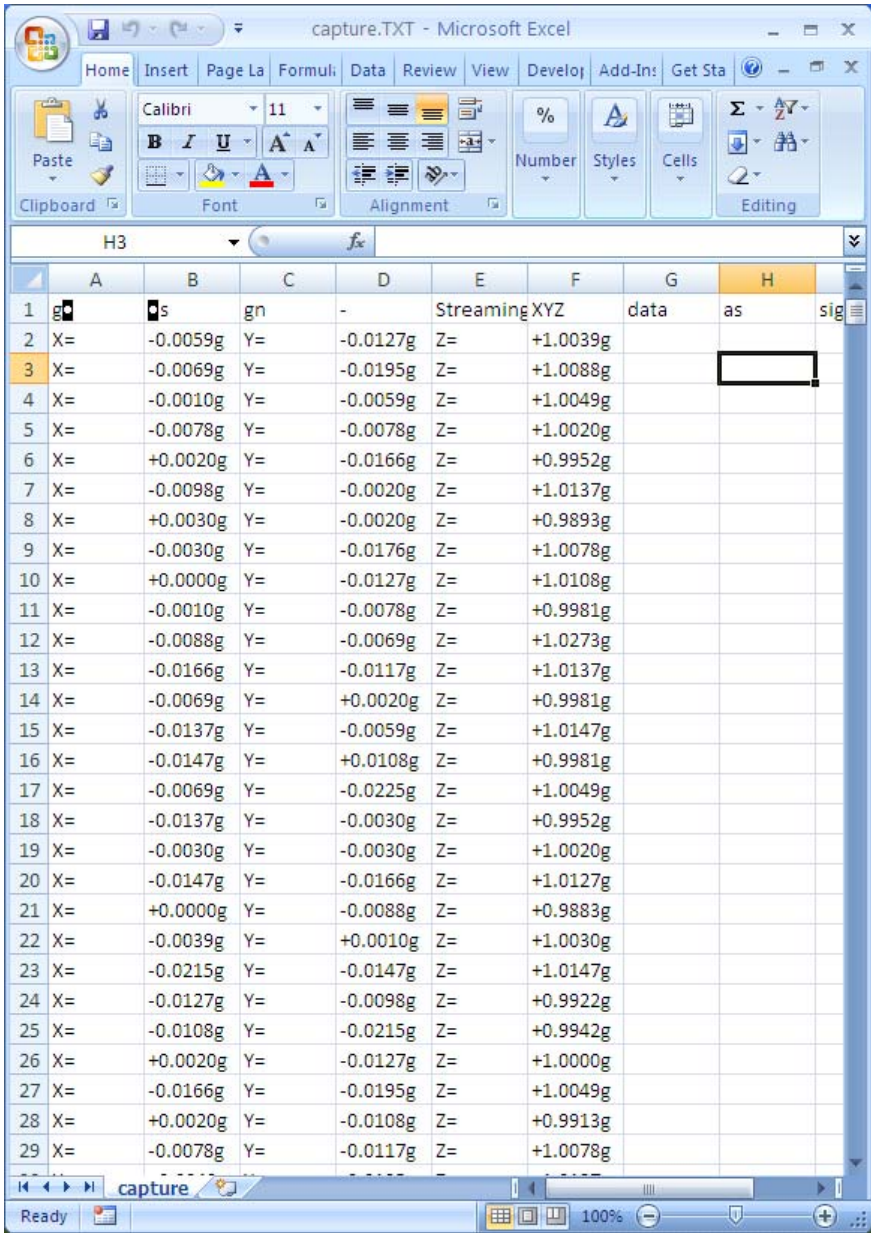


Figure 6. Imported sample data in Excel sheet

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, Altivec, and CodeWarrior, are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Xtrinsic is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 2012 Freescale Semiconductor, Inc.

Document Number: AN4083

Rev 0
08/2012

