Freescale Semiconductor
Application Note

Document Number: AN4296
Rev 1, 02/2013

Data Manipulation and the Basic Settings of Xtrinsic MMA8491Q Accelerometer

by: Fengyi Li, Applications Engineer

This document shows you how to configure the MMA8491Q to extract and manipulate acceleration data. The MMA8491Q is a straightforward implementation for taking acceleration samples, and enables you to take samples at any time or to stream data at any chosen sampling rate, particularly at an extremely low speed. You can drive this device with one GPIO pin and one timer from the host microcontroller. The manipulation of the data into different formats is important for algorithm development and for display.

This application note accompanies the MMA8491 Driver Code, and explains:

- Configuring the MMA8491Q as a tilt sensor
- Polling XYZ accelerometer data
- Streaming data
- Setting the sampling rate
- Setting when the acceleration data is read
- Changing the accelerometer data format (hex to counts or to decimal numbers)
- Installing/using the MMA8491 driver

Contents

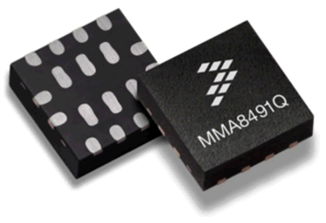
1 Introduction

The MMA8491Q device is an industrial, low voltage, low power 3-axis multifunctional accelerometer and tilt sensor with digital output, housed in a 3 mm × 3 mm × 1 mm QFN package (0.65 mm pitch). The device can accommodate two accelerometer configurations:

- As an easy-to-implement 45° tilt sensor (out of the box), using one line output per axis.
- As a digital (I^2C) output accelerometer with I^2C serial communication ports, 14-bit ±8 g raw data can be read from the device with a 1 mg/LSB sensitivity.

Features

- Sensor g-range: ±8 g
- Fixed oversampling rate: 2
- Digital resolution: 14-bit
- Resolution: 1 mg/LSB



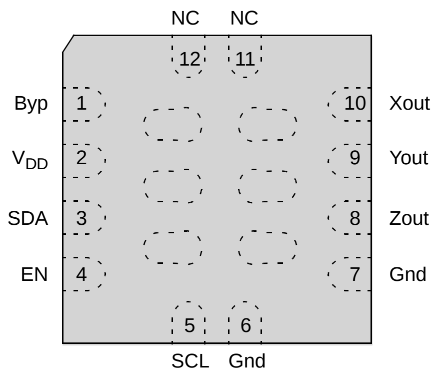


Figure 1. Pins of MMA8491Q

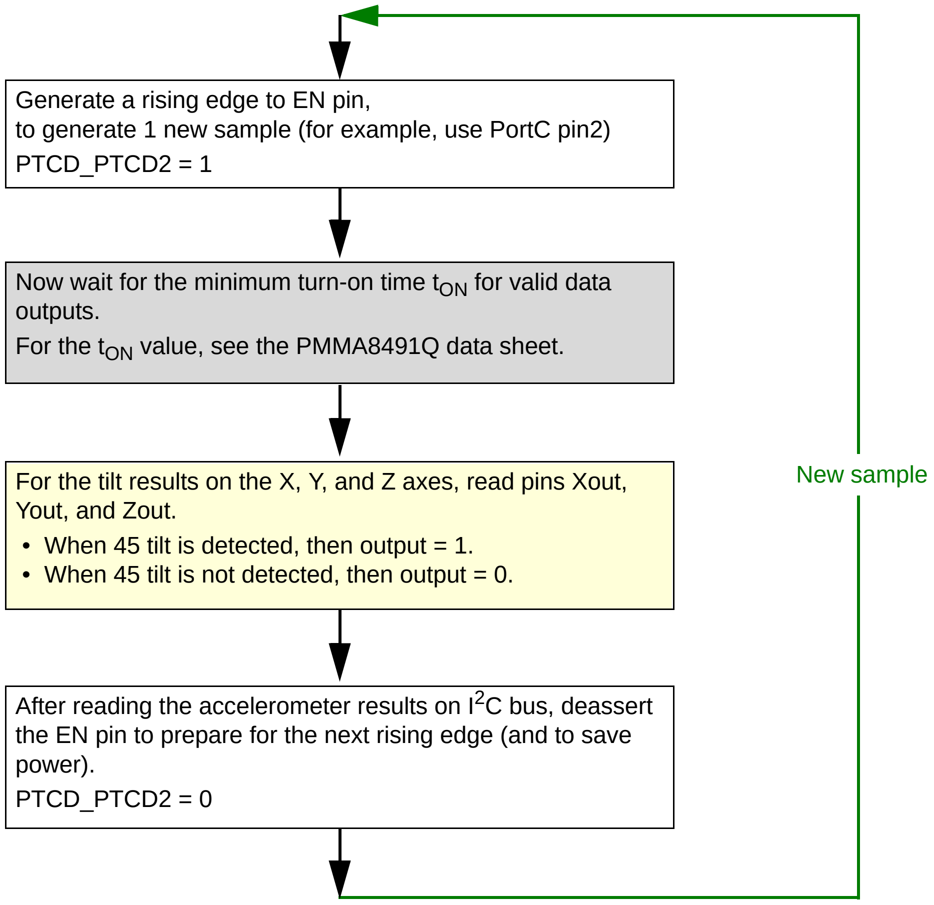
There is a driver available that runs on the MMA8491 Sensor Toolbox kit, which provides an example in CodeWarrior for everything discussed in this application note. The driver may be used with RealTerm or HyperTerminal, to capture and log data in different formats.

1.1 MMA8491Q data sampling operations

1. The accelerometer is turned on at the rising edge of the EN pin, and acquires one sample for each of the three axes.
2. Samples are acquired, converted, and compensated for zero-g offset and gain errors, and then compared to an internal threshold value of 0.688 g and stored.
3. If the absolute value of any of the X, Y, Z axis samples exceeds this threshold, then the corresponding outputs on these axes drive logic Highs, and vice versa. (If the absolute value of any of the X, Y, Z axis samples are equal to or lower than this threshold, then the corresponding outputs on these axes drive logic lows.) These signals can be used as tilt event triggers to MCUs, and indicate that new data is ready.
4. Valid accelerometer data is stored into Data Registers. To determine whether the new sample data is ready, read Status Register 0x00.
5. To take a sample, pull the EN pin High.
   To acquire the next sample, deassert the EN pin and then reassert the EN pin. This toggle of the EN pin can be easily done by using a GPIO pin (or a timer output) from the host microcontroller.

NOTE

- Send a rising edge to the EN pin to take one sample.
- The EN pin should not be asserted before VDD reaches 1.65 V.



Generate a rising edge to EN pin,
to generate 1 new sample (for example, use PortC pin2)
PTCD_PTCD2 = 1
Now wait for the minimum turn-on time t_{ON} for valid data outputs.
For the t_{ON} value, see the PMMA8491Q data sheet.
For the tilt results on the X, Y, and Z axes, read pins Xout, Yout, and Zout.
• When 45 tilt is detected, then output = 1.
• When 45 tilt is not detected, then output = 0.
After reading the accelerometer results on I^2C bus, deassert the EN pin to prepare for the next rising edge (and to save power).
PTCD_PTCD2 = 0
New sample

Figure 2. Sampling accelerometer data using MMA8491Q

2 Configuring the MMA8491Q

2.1 Tilt sensor

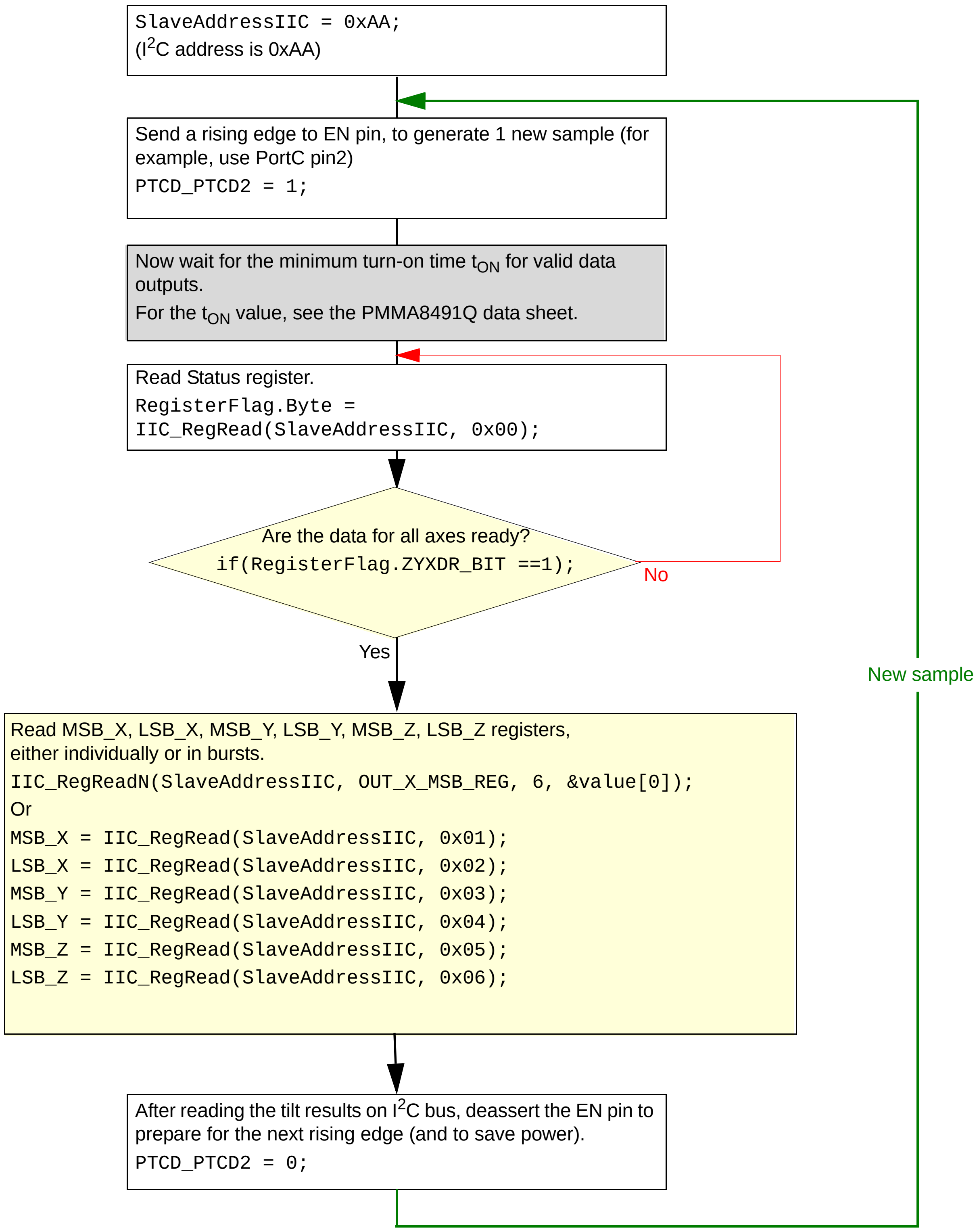
MMA8491Q is an out-of-the-box 45° tilt sensor. The corresponding tilt sensing results are output on the XOUT, YOUT, and ZOUT pins. These pins are set to 1 when the tilt greater than 45° is sensed.

Assert the EN pin to acquire a new tilt result. The tilt sensing flow is the same as in Figure 2. This toggle of the EN pin can be easily done by using a software controller GPIO pin (or a timer output) from the host microcontroller. Both cases are described in Section 3, "Streaming Data," on page 8.

2.2 Additional accelerometer XYZ data polling

The MMA8491Q accelerometer data output can be read via the I^2C communication bus, using an I^2C address of 0x55. The MMA8491Q data can be polled whenever the device is in (should be) ACTIVE mode (EN pin remains high after a sample is taken. For details about operational modes, see the data sheet.). It does not have a dedicated data ready pin (like other general purpose accelerometers do).

However, users might find this case interesting. They can use XOUT, YOUT, ZOUT pins as interrupts, to trigger a data reading right after the tilting events. To quickly read out the accelerometer data via I^2C protocol, follow the procedure in Figure 3.


SlaveAddressIIC = 0xAA;
(I^{2}C address is 0xAA)

Send a rising edge to EN pin, to generate 1 new sample (for example, use PortC pin2)

```
PTCD_PTCD2 = 1;
```

Now wait for the minimum turn-on time t_{ON} for valid data outputs.
For the t_{ON} value, see the PMMA8491Q data sheet.

Read Status register.

```
RegisterFlag.Byte =
IIC_RegRead(SlaveAddressIIC, 0x00);
```

Are the data for all axes ready?

```
if(RegisterFlag.ZYXDR_BIT ==1);
```

Read MSB_X, LSB_X, MSB_Y, LSB_Y, MSB_Z, LSB_Z registers, either individually or in bursts.

```
IIC_RegReadN(SlaveAddressIIC, OUT_X_MSB_REG, 6, &value[0]);
```

Or

```
MSB_X = IIC_RegRead(SlaveAddressIIC, 0x01);
LSB_X = IIC_RegRead(SlaveAddressIIC, 0x02);
MSB_Y = IIC_RegRead(SlaveAddressIIC, 0x03);
LSB_Y = IIC_RegRead(SlaveAddressIIC, 0x04);
MSB_Z = IIC_RegRead(SlaveAddressIIC, 0x05);
LSB_Z = IIC_RegRead(SlaveAddressIIC, 0x06);
```

After reading the tilt results on I^{2}C bus, deassert the EN pin to prepare for the next rising edge (and to save power).

```
PTCD_PTCD2 = 0;
```

Figure 3. Reading data quickly over I^{2}C

To determine if the data is ready, read the status bit XYZDR (XYZ data ready) bit:

- When new data is ready, the data ready bits are set to 1.
- After the corresponding data registers are read, the data ready bits are updated to 0.

Address offset	Register	Bit							
		7	6	5	4	3	2	1	0
0x00	STATUS	0	0	0	0	ZYXDR	ZDR	YDR	XDR

Table 1. STATUS register

Field	Description
ZYXDR	X, Y, Z-axis new Data Ready ZYXDR signals that a new sample for all channels is available. • ZYXDR is cleared when the high-bytes of the acceleration data (OUT_X_MSB, OUT_Y_MSB, OUT_Z_MSB) of all channels are read. 0 No new set of data ready (default) 1 A new set of data is ready
ZDR	Z-axis new Data Available ZDR is set whenever a new acceleration sample related to the Z-axis is generated. • ZDR is cleared any time that the OUT_Z_MSB register is read. 0 No new Z-axis data is ready (default) 1 A new Z-axis data is ready
YDR	Y-axis new Data Available YDR is set whenever a new acceleration sample related to the Y-axis is generated. • YDR is cleared any time that the OUT_Y_MSB register is read. 0 No new Y-axis data ready (default) 1 A new Y-axis data is ready
XDR	X-axis new Data Available XDR is set whenever a new acceleration sample related to the X-axis is generated. • XDR is cleared any time that the OUT_X_MSB register is read. 0 No new X-axis data ready (default) 1 A new X-axis data is ready

After reading the axes data, you should clear the EN pin for two good reasons:

- To prepare for the EN pin rising edge
- To preserve the power

Depending on your application, you can either clear the EN pin, or de-assert the power and EN pins. For more information about saving power, see the MMA8491Q data sheet.

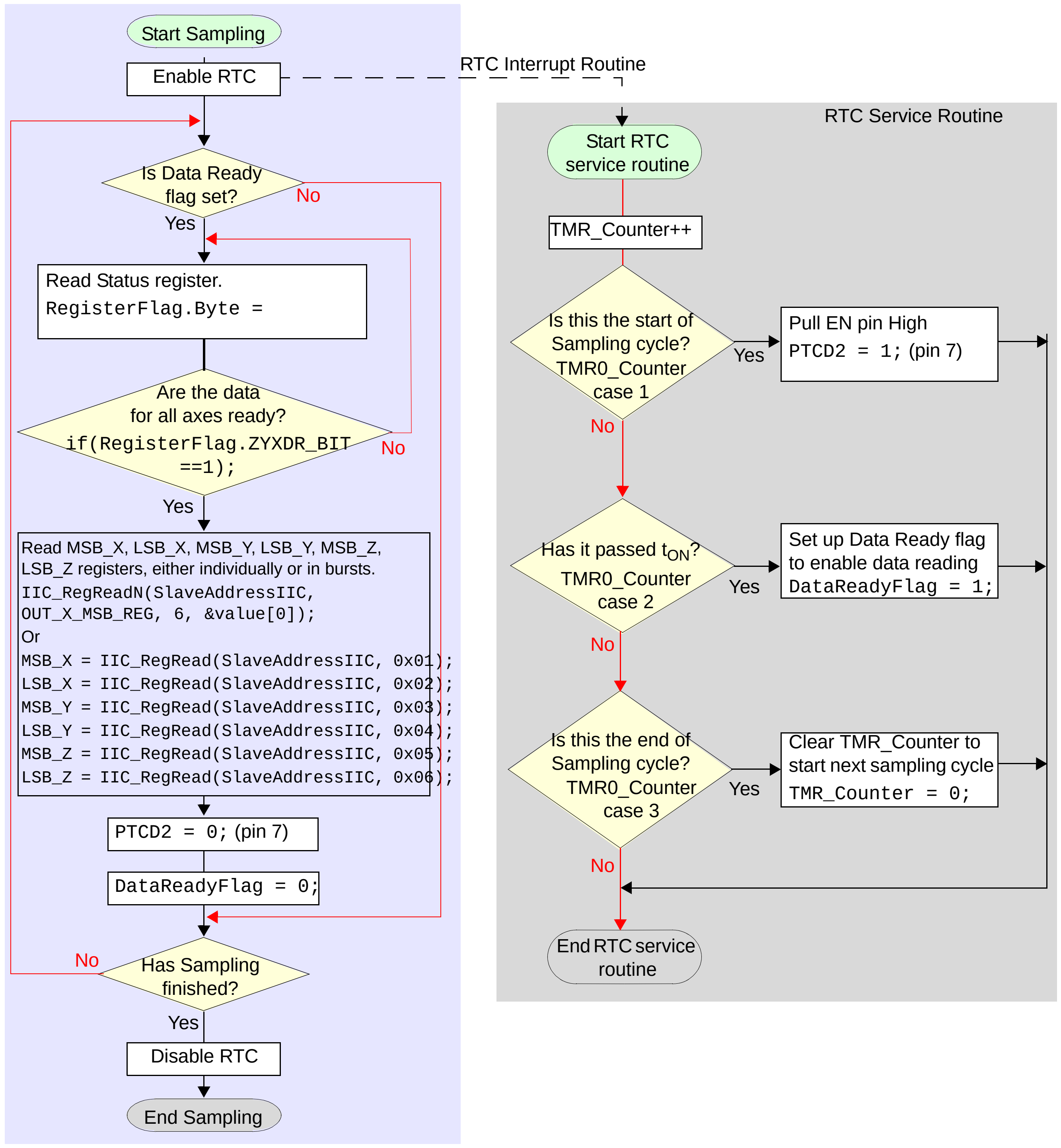
Example 1. MMA8491Q data polling code in main.c

```
for(;;)
{
   /*  Use the microcontroller GPIO PortC2 pin to drive MMA8491Q EN pin */
   PTCD_PTCD2 = 1; // pull EN pin HIGH
   /* Poll data via I2C if data is ready */
   RegisterFlag.Byte = IIC_RegRead(SlaveAddressIIC, STATUS_00_REG);
   if (RegisterFlag.ZYXDR_BIT == 1)
   {
        /* Read the  XYZ sample data*/
         IIC_RegReadN(SlaveAddressIIC, OUT_X_MSB_REG, 6, &value[0]);

        /*  Or read individual registers */
        // MSB_X = IIC_RegRead(SlaveAddressIIC, 0x01);
        // LSB_X = IIC_RegRead(SlaveAddressIIC, 0x02);
        // MSB_Y = IIC_RegRead(SlaveAddressIIC, 0x03);
        // LSB_Y = IIC_RegRead(SlaveAddressIIC, 0x04);
        // MSB_Z = IIC_RegRead(SlaveAddressIIC, 0x05);
        // LSB_Z = IIC_RegRead(SlaveAddressIIC, 0x06);
   }
   /*  Use the microcontroller GPIO PortC2 pin to drive MMA8491Q EN pin */
   PTCD_PTCD2 = 0; // pull EN pin LOW
}
```

3 Streaming Data

In this section, a suggested flow diagram is presented. It uses a real-time clock (RTC) to time the MMA8491 device's data streaming events: assert the EN pin to start to sample, wait for data register to be ready to read, and set the sampling rate. The flow is the expansion of Figure 3 and shown in Figure 4.



Start Sampling
Enable RTC
RTC Interrupt Routine
Is Data Ready flag set?
No
Yes
Read Status register.
RegisterFlag.Byte =
Are the data for all axes ready?
if(RegisterFlag.ZYXDR_BIT ==1);
No
Yes
Read MSB_X, LSB_X, MSB_Y, LSB_Y, MSB_Z, LSB_Z registers, either individually or in bursts.
IIC_RegReadN(SlaveAddressIIC, OUT_X_MSB_REG, 6, &value[0]);
Or
MSB_X = IIC_RegRead(SlaveAddressIIC, 0x01);
LSB_X = IIC_RegRead(SlaveAddressIIC, 0x02);
MSB_Y = IIC_RegRead(SlaveAddressIIC, 0x03);
LSB_Y = IIC_RegRead(SlaveAddressIIC, 0x04);
MSB_Z = IIC_RegRead(SlaveAddressIIC, 0x05);
LSB_Z = IIC_RegRead(SlaveAddressIIC, 0x06);
PTCD2 = 0; (pin 7)
DataReadyFlag = 0;
No
Has Sampling finished?
Yes
Disable RTC
End Sampling
RTC Service Routine
Start RTC service routine
TMR_Counter++
Is this the start of Sampling cycle? TMR0_Counter case 1
Yes
Pull EN pin High
PTCD2 = 1; (pin 7)
No
Has it passed t_{ON}? TMR0_Counter case 2
Yes
Set up Data Ready flag to enable data reading
DataReadyFlag = 1;
No
Is this the end of Sampling cycle? TMR0_Counter case 3
Yes
Clear TMR_Counter to start next sampling cycle
TMR_Counter = 0;
No
End RTC service routine

Figure 4. Sampling MMA8491Q using the RTC timer (for sampling rate and wait time)

3.1 Stream data

You can stream data from the MMA8491Q device by controlling the RTC module:

- Streaming starts when the RTC module is enabled and starts to tick.
- Streaming finishes when the RTC module is disabled.

MCU pulls the device EN pin HIGH to take samples at the set sampling rate.

Example 2. RTC module initialization variable definition in system.h

```
#define RTC_ENABLED                   (RTCSC_RTIE = 1)
#define RTC_DISABLED                  (RTCSC_RTIE = 0)
```

Example 3. RTC disable control in main.c

```
// When DATA stream is completed or not started, RTC is disabled
if (XYZ_STREAM == FALSE)
{
         RTC_DISABLED;
}
```

Example 4. RTC enable control in mma8491_terminal.c

```
//When user command 'S' is read in microcontroller, RTC is enabled
case 'S':
          XYZ_STREAM = TRUE;
          sample_num_dec = sample_num;
          PROMPT_WAIT = TRUE;
          SCISendString("Start to take samples. \r\n");
          Tmr_counter = 0;
          RTC_ENABLED;
          break;
```

3.2 Setting the sample rate

To stream MMA8491Q data at a fixed rate, you can adopt a timer (by employing a timer on the host microcontroller). In this scheme, a control variable (TMR_Counter) is used alone with the RTC to record the lapse time, TMR_Counter should only be updated by the RTC service routine. The TMR Counter increases by one on every RTC interrupt, and this number is then compared to a preset value. The sampling period is a multiple of the RTC interrupt period and this preset value. The TMR_Counter should be cleared at the beginning and end of each sampling cycle.

The sampling period is determined by:

Sampling period = RTC interrupt period × preset_sampling_rate_counter_value

3.3 Setting when the acceleration data is read

Understand that the acceleration data is valid *after the turn-on time* (t_{ON}) and that any I^2C transition values are not valid beforehand (*before the turn-on time has elapsed*). You can use this property to time the Data Register reading, which can help you eliminate unnecessary I^2C transactions on the bus and save total system power consumption.

To ensure a valid I^2C reading value, the data reading time delay should be set larger than t_{ON}. If the timer to track the data reading time is the same timer as the timer used to set the sampling rate, then the control variable TMR_Counter can be used.

The data read period is determined by:

Data ready period = RTC interrupt period × preset_data_ready_counter_value

When the TMR_Counter is equal to the preset counter preset_data_ready_counter_value, the control variable DataReadyFlag is set, which signals to the main program that it (the main program) should conduct an I^2C reading transaction. In this scheme (shown in Figure 4), only one I^2C transaction is allowed. After the I^2C transaction is conducted, the DataReadyFlag variable is cleared.

Example 5. RTC module initialization variable definition in system.h

```
/***************************************************************************
**   Real Time Clock Interrupt (RTC)
**
**   0x000C  KBISC     KBI Interrupt Status and Control Register
**   0x000D  KBIPE     KBI Interrupt Pin Select Register
**   0x000E  KBIES     KBI Interrupt Edge Select Register
**   0x000F  IRQSC
*/

//  internal clock 16 kHz,  8 prescale to 250 us per tick
#define init_RTCSC     0b01000001;
#define init_RTCMOD    0b00000001;    // mod by 2 to scale the tick to 500 us
#define init_RTC_period    0x64;  // set sample rate at 1 sec / sample
#define CLEAR_RTC_INTERRUPT        (RTCSC_RTIF = 1)
```

Example 6. Sample rate and XYZ data ready delay setting in main.c

```
/*********************************************************\
* Real timer service routine
\*********************************************************/
interrupt void isr_RTC(void)
{
 /*
```

```
  *   RTC timer is set to trigger every 500 us
  *       Set EN = 1 at 0 us
  *       Set read I2C data flag POLL_ACTIVE at 500 us, clear EN afterwards
  *       Mark Error if missing data
  */
  Tmr_counter ++;              // Used to test turn on time
  switch (Tmr_counter)
  {
    case 1:    // start sampling
      PTCD_PTCD2 = 1;
      break;
    case 3:      // Data Ready Delay
      if (POLL_ACTIVE == TRUE)
        POLL_ERROR = TRUE;
      else
        POLL_ACTIVE = TRUE;
      break;
    default: break;
  }
  if (Tmr_counter > (RTC_period - 1)) // Sample rate variable: RTC_period
      Tmr_counter = 0;

  CLEAR_RTC_INTERRUPT;              // Clear RTC Interrupt
}
```

4 Converting 14-bit data

The accelerometer signal is directly converted and stored as 14-bit data to registers when it is ready, after the device turn-on time t_{ON}. The data stays in the registers as long as VDD and EN pins stay HIGH. You can use I^2C communication to read the data registers (0x01 – 0x06).

Table 2. Data registers map(1)(2)

Name	Type	Register Address	Auto-Increment Address(3)	Default	Comment
STATUS	R	0x00	0x01	0x00	Read time status
OUT_X_MSB	R	0x01	0x02	Output	[7:0] are the 8 MSBs of the 14-bit sample
OUT_X_LSB	R	0x02	0x03	Output	[7:2] are the 6 LSB of 14-bit sample
OUT_Y_MSB	R	0x03	0x04	Output	[7:0] are 8 MSBs of the 14-bit sample
OUT_Y_LSB	R	0x04	0x05	Output	[7:2] are the 6 LSB of 14-bit sample
OUT_Z_MSB	R	0x05	0x06	Output	[7:0] are the 8 MSBs of the 14-bit sample
OUT_Z_LSB	R	0x06	**0x00**	Output	[7:2] are the 6 LSB of 14-bit sample

1. Register contents are preserved when EN pin is set high after sampling.
2. Register contents are reset when EN pin is set low.
3. Auto-increment is the I^2C feature that automatically updates the I^2C read address after each read.
   Auto-increment addresses that are not a simple increment are highlighted in **bold**.
   The auto-increment addressing is only enabled when device registers are read using I^2C burst read mode; therefore the internal storage of the auto-increment address is cleared whenever a stop bit is detected.

Each axis has 2 data registers, MSB and LSB. The corresponding 14-bit result is in left-justified format (2's complement numbers). See Table 3.

Table 3. Register bit map

Address Offset	Name		7	6	5	4	3	2	1	0
0x00	STATUS	R	0	0	0	0	ZYXDR	ZDR	YDR	XDR
0x01	OUT_X_MSB	R	XD[13:6]							
0x02	OUT_X_LSB	R	XD[5:0]						0	0
0x03	OUT_Y_MSB	R	YD[13:6]							
0x04	OUT_Y_LSB	R	YD[5:0]						0	0
0x05	OUT_Z_MSB	R	ZD[13:6]							
0x06	OUT_Z_LSB	R	ZD[5:0]						0	0

4.1 Converting the 14-bit 2's complement hex number to a signed integer (counts)

The data stored in the data registers are 14-bit 2's complement numbers. Converting this data to a signed value (in counts) means that the 2's complement hex number is converted to an integer number (with a + or – sign). For example:

0xABCC = –5389

0x544 = +337

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

Located in the driver program, 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(). Looking more closely, you can see that this routine will add 0x30 to 0xF0, resulting in a value of 0x120, which gets truncated to 0x20 (the ASCII space character).

Example 7. SCI_s14dec_Out function in sci.c

```
void SCI_s14dec_Out (tword data)
{
  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 >>2) / 1000);
  r = (data.Word >>2) % 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);
}
```

4.2 Converting the 14-bit 2’s complement hex number to a signed decimal fraction (in g’s)

Based on different microcontroller architecture, the conversation of a 14-bit 2’s complement hex number to a signed decimal fraction is different. The 32-bit architecture MCU has enough bit length to calculate the mg-values following just one simple formula. Conversion of a 16-bit number using a lower bit-length microcontroller architecture requires multiple steps: calculate the integer and the fractions separately, and then add them together.

4.2.1 Conversion using a 32-bit MCU

The formula below converts the 14-bit 2’s complement to a fractional g-value, using a 32-bit architecture MCU. The 32-bit MCU has a 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.

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

The above formula is based on a MMA8491Q device property. The device has an 8-g accelerometer scale and a 14-bit digital resolution, where 1 g = 1000 mg = 1024 counts.

For example:

0x7FFC count = +1023 mg

0xFFFC count = –1 mg

0xF000 count = –1024 mg

Data.DWord is the signed accelerometer value extended to 32 bits. The formula below can be used to extend the data to any processor word length. To start the extension, the left-aligned 14-bit data must first be right-aligned (by a 2-bit right shifting operation). Next, the data’s signed bit will be compared, to identify if this sign value is negative. If the data’s sign value is negative, then the negative sign is extended by minus 0x4000.

Example 8. Extend a signed number to a microcontroller ALU length

```
Data.DWord = Data.Word >> 2;
If (Data.DWord >= 0x2000)
   Data.DWord -= 0x4000;
```

A 10-bit right shift operation is used to accomplish the division by 1024. If the division operation is chosen, the fractions *that are greater than or equal to 0.5* are 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 to be rounded up to an integer after the shift.

Example 9. Fractional conversion for a 32-bit architecture MCU

```
void SCI_s14frac_Out (tword data)
{
   dword result;
   byte a, b, c, d;
   word r;

   data.DWord = data.Word >> 2;
   if (data.DWord >= 0x2000)
      data.DWord = data.DWord - 0x4000;
   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')
}
```

4.2.2 Conversion using a non-32-bit MCU (8-bit, 16-bit)

This section describes how to do the fractional conversion when using MCUs that use an 8-bit or 16-bit (less than 32-bit) architecture.

Converting to a signed value into g's requires performing the same operations as shown previously (Section 4.2.1, "Conversion using a 32-bit MCU"), with an added step of resolving the integer and fractional portions of the value. The scale of the accelerometer's Active Mode (i.e., either 2 g, 4 g, or 8 g) determines the location of the inferred radix point separating these segments, and thereby the overall sensitivity of the result.

For the MMA8491Q:

- The most significant bit (bit 13) represents the sign of the result (either positive or negative).
- MMA8491Q has only one scale (8 g, where 1 g = 1024 counts). Therefore, bits 12, 11, and 10 will contribute to an integer value of 0, 1, 2, 3, 4, 5, 6, and 7; bits 9 through 0 will be fractional values.

Table 4. 8-g scale value with corresponding integer and fraction bits

Full Scale Value	Conts/g	Sign Bit	Integer Bits	Fraction Bits
8g	1024	13	12 (2^{12} = 4096) 11 (2^{11} = 2048) 10 (2^{10} = 1024)	0 through 9

The fractional portion of the result can be extracted, by logically shifting the sample to the left by 4 binary locations. Table 6 shows this result; Table 7 provides the corresponding decimal values.

Table 5. 8-g full scale 14-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
MMA8491Q (14-bit)	13	12	11	10	9	8	7	6	5	4	3	2	1	0	x	x
Integer/Fraction	±	I	I	I	F	F	F	F	F	F	F	F	F	F	x	x
MSB/LSB	M	M	M	M	M	M	M	M	L	L	L	L	L	L	0	0

Table 6. 8-g full scale 14-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
MMA8491Q (14-bit)	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	L	L	L	L	L	L	0	0	0	0	0	0

Table 7. 8-g mode fraction values

	8g Mode	Calculation 256 counts/g	Rounded to 4th Decimal Place	Integer
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 = 0.01563	0.0156	156
2^{-7}	$2^3 = 8$	8/1024 = 0.007812	0.0078	78
2^{-8}	$2^2 = 4$	4/1024 = 0.003906	0.0039	39
2^{-9}	$2^1 = 2$	2/1024 = 0.001953	0.0020	20
2^{-10}	$2^0 = 1$	1/1024 = 0.000976	0.0010	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 bit 9, 6, and 4 are set, then the total will be (5000 + 625 + 156 = 5781), which corresponds to 0.5781 g:

- The highest fractional value occurs when all fraction bits are set (5000 + 2500 + 1250 + 625 + 313 + 156 + 78 + 39 + 20 + 10 = 9991), which corresponds to 0.9991g.
- The resolution in full scale 8 g Active Mode is 0.977 mg. Adding the fractional value to 4 significant digits results in 1-mg resolution.

Example 10 shows a code example that performs the conversion of a 14-bit signed 2's complement value into a signed decimal fraction displayed in g's. This routine (with modification) can also be used to convert 12-bit, 10-bit or 8-bit data, with 2 g or 4 g full scales. The extra unused bits will remain zeros.

Example 10. Fractional conversion SCI_s14frac_Out in sci.c

```
void SCI_s14frac_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 &= 0xFFFC;
    data.Word = ~data.Word + 1;
  }
  else
  {
```

```
  SCI_CharOut ('+');
}
/*
**  Determine integer value and output
*/
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 (value.Bit._3 ==1)
  result += FRAC_2d9;
if (value.Bit._2 ==1)
  result += FRAC_2d10;

if(value.Bit._1 ==1)
result += FRAC_2d11;
if(value.Bit._0 ==1)
result += FRAC_2d12;

/*
**  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 ('g');
}
```

5 Using the MMA8491 HyperTerminal Driver

This section shows how to use the MMA8491 Driver (which is preinstalled) with the Windows HyperTerminal program. Table 8 lists the tools you will need.

Table 8. Tools

For	Use
Hardware	MMA8491Q Sensor Toolbox Kit (LFSTBEB8491 or RDMMA8491). See http://www.freescale.com/webapp/sps/site/overview.jsp?code=SNSTOOLBOX&fsrch=1&sr=1
Software	Windows HyperTerminal or any other communication tool
Programming	To program the driver to Sensor Toolbox board or to make any modifications to the program, use Code Warrior v 6.3. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-MICROCONTROLLERS

5.1 Overview

The MMA8491 driver program enables you to quickly access the MMA8491Q data and tilt sensing result via an I^2C bus. TheMMA8491 driver is also a reference program that you can use. This program is based on the MMA8491Q Sensor Toolbox (STB) kit. The driver is downloadable to the MC9S08QE8 microcontroller on the baseboard.

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 for you to choose from.

This driver configures the MMA8491Q device to take one sample per second. To trigger the sampling, the driver uses the MCU’s GPIO pin C2. The rising edge of the GPIO pin is timed with an RTC (real-time clock). The RTC runs using a 500-us step. The accelerometer data is read 1 ms after the EN pin is asserted, and is then displayed on the HyperTerminal screen. The EN pin is turned off after the read, regardless of the data display.

5.2 Procedures

5.2.1 Set up equipment and apply power

1. Connect the USB cable, to power the STB board.
2. Make sure that SW1 is on.
   The power LED CR1should be lit.

5.2.2 Download HyperTerminal driver (if necessary)

1. Download the MMA8491Q HyperTerminal program from the MMA8491Q product site.
   The driver can be found on the “software & tools” tab, with the name of AN4296SW.
   http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MMA8491Q

NOTE

The MMA8491Q Sensor Toolbox kit comes pre-programmed with HyperTerminal diver.

2. Connect the programmer to the J1 connector on the USB communication board of the STB kit.
3. Open the MMA8491Q driver program (using Code Warrior 6.3).
4. Program the driver to the MC9S08QE8 microcontroller.
5. Disconnect the programmer.
6. Turn off SW1, then turn on SW1, to allow the part to restart.

NOTE

You can program the HyperTerminal driver into other Sensor Toolbox kits. If you do, then the Sensor Toolbox kits will NOT be recognized by the STB software afterwards.

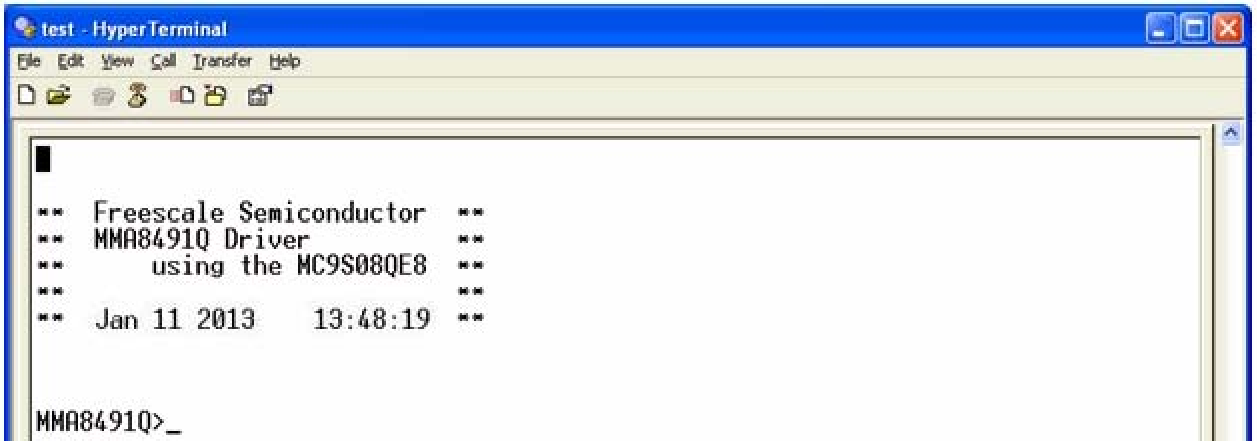
5.2.3 Run communications program, start sampling

1. Run HyperTerminal.
2. Configure HyperTerminal port with these settings:

Table 9. HyperTerminal settings

Parameter	Value
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 (Figure 5). This menu header only appears when the STB board is first powered on.



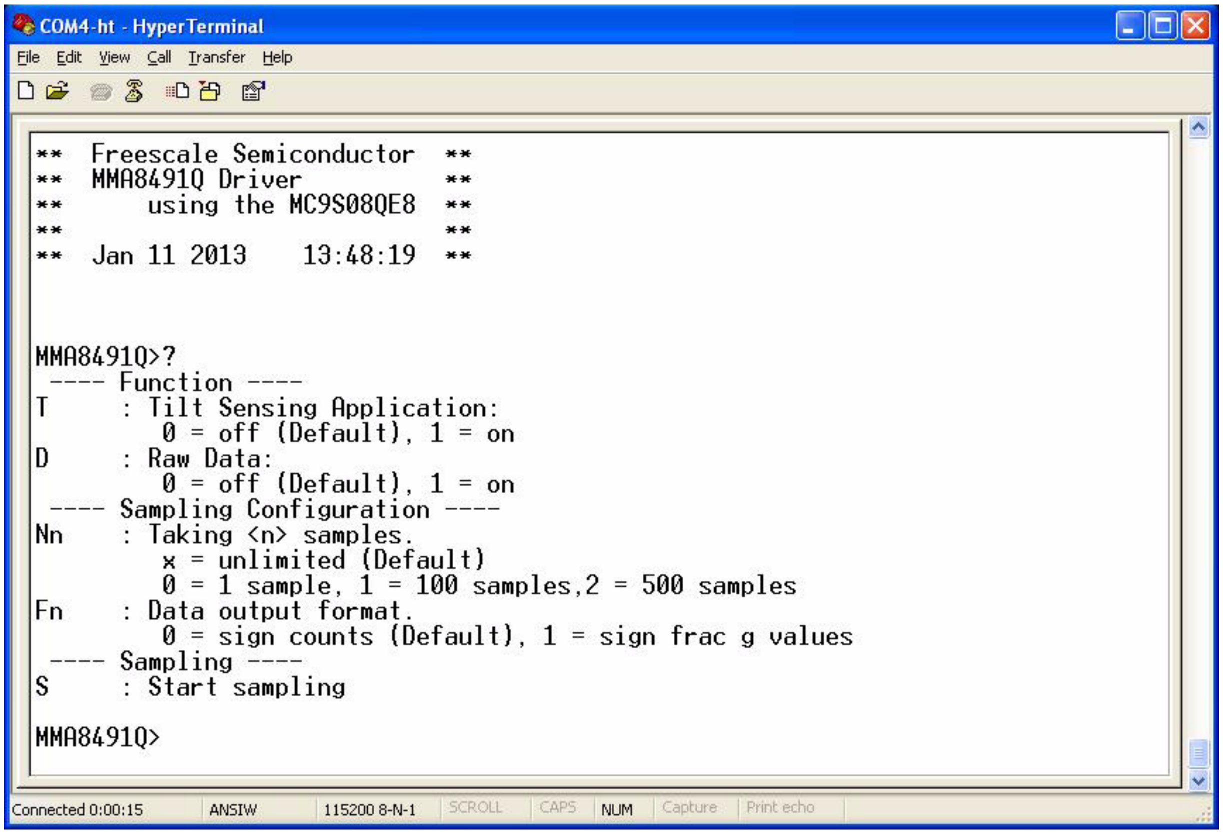
```
█

**  Freescale Semiconductor   **
**  MMA8491Q Driver           **
**      using the MC9S08QE8   **
**                            **
**  Jan 11 2013    13:48:19   **


MMA8491Q>_
```

Figure 5. HyperTerminal at power up

4. Entering “?” will bring up the menu shown in Figure 6.
   Table 10 lists the HyperTerminal menu commands.



```
**  Freescale Semiconductor  **
**  MMA8491Q Driver          **
**      using the MC9S08QE8  **
**                           **
**  Jan 11 2013    13:48:19  **


MMA8491Q>?
 ---- Function ----
T      : Tilt Sensing Application:
          0 = off (Default), 1 = on
D      : Raw Data:
          0 = off (Default), 1 = on
 ---- Sampling Configuration ----
Nn     : Taking <n> samples.
          x = unlimited (Default)
          0 = 1 sample, 1 = 100 samples,2 = 500 samples
Fn     : Data output format.
          0 = sign counts (Default), 1 = sign frac g values
 ---- Sampling ----
S      : Start sampling

MMA8491Q>
```

Figure 6. HyperTerminal menu

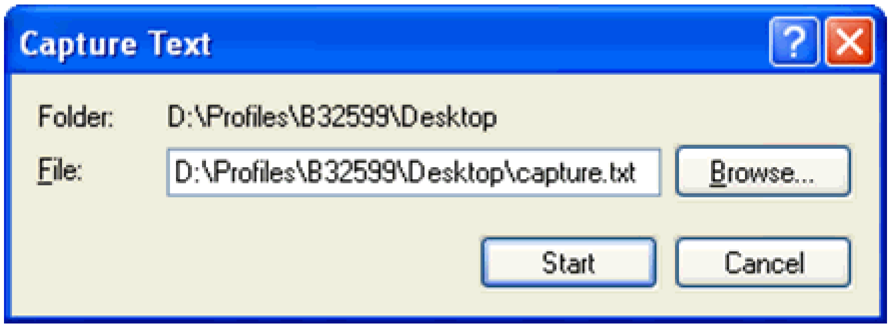
Table 10. HyperTerminal menu commands

Command	Function	Notes
T*n*	Tilt Sensing Selector	• N=0 Tilt sensing function turned off • N=1 Tilt sensing function turned on
D*n*	Data output Selector	• N=0 Data output turned off • N=1 Data output turned on
N*n*	Stream data Sample number	• N=0 1 sample • N=1 100 samples • N=2 500 samples • N=3 unlimited samples
F*n*	Read XYZ as signed counts	• N=0, read XYZ as signed counts Example of response: X= -250 Y= -126 Z= +968 • N=1, read XYZ as signed g’s Example of response: X= -0.1992g Y= -0.0577g Z= +0.9473g
S	Start sampling	

5. Enter the commands to configure the device sampling conditions or to start sampling. Note that the commands are not case-sensitive.
6. Press any key to exit out of the data streaming.

5.2.4 Acquire and save data

1. In HyperTerminal, select **Transfer→Capture Text**.
2. Save the file to a known location, and select **Start**.
   The data log will start.



Folder: D:\Profiles\B32599\Desktop
File: D:\Profiles\B32599\Desktop\capture.txt

Figure 7. Capture text dialog in HyperTerminal

3. Select one of the commands.
4. When a satisfactory amount of data has been collected, select **Transfer→Capture Text→Stop**.
5. Open Excel.
6. Open the log file.
   Note that if the file does not appear, then select the **Files Of Type** drop-down menu, and select **All Files (*.*)**.

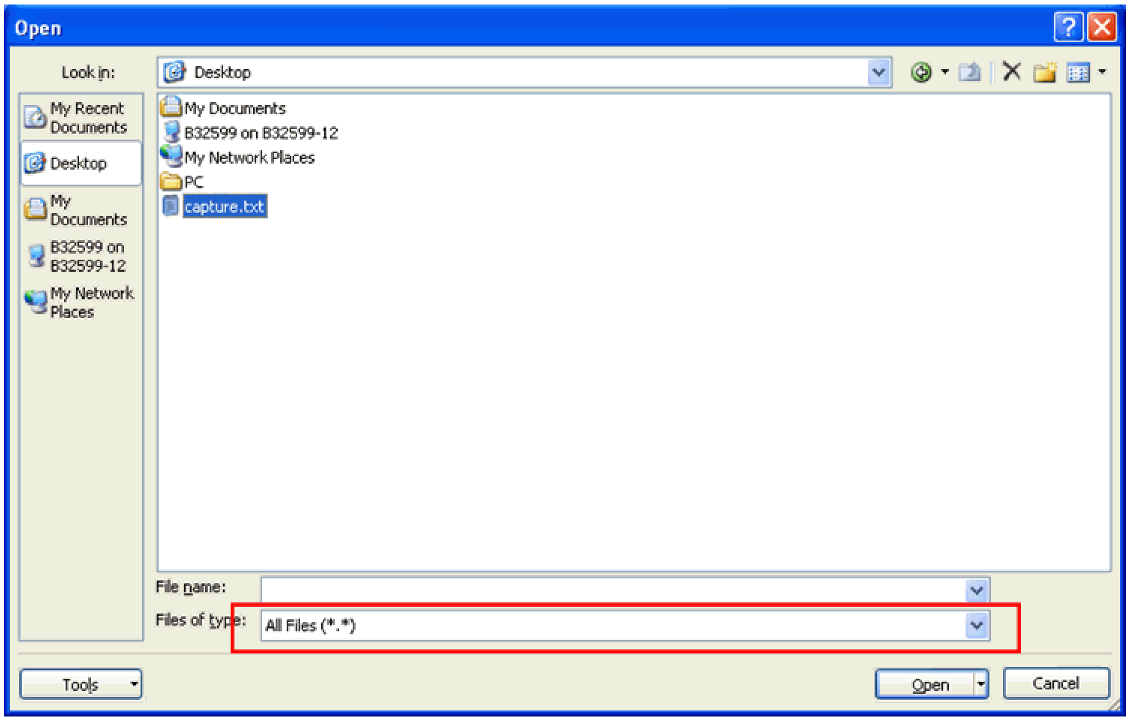
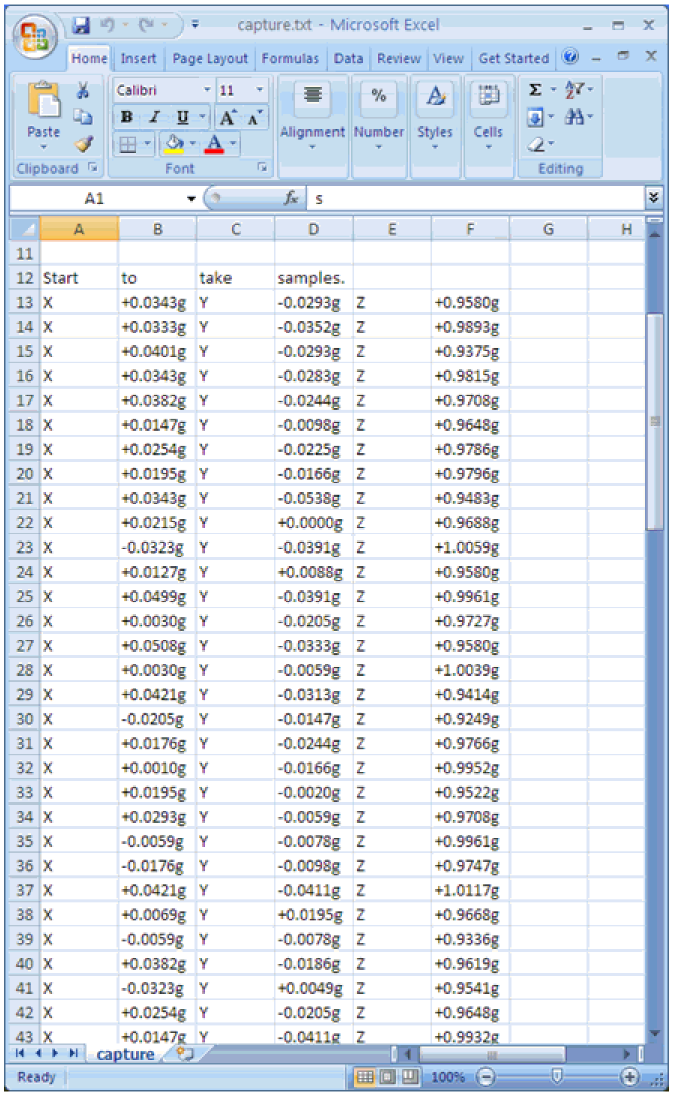


Figure 8. Select "Files of Type" to be "All Files"

7. The Text Import Wizard should now appear; select the **Delimited** option and then select **Next**.
8. Depends on the output data, choose delimiters accordingly. For example, when convert g's values, Select the **Space** option, and in **Other** type "=". Next select **Finish**. The data set will have 6 columns: X, X Data, Y, Y Data, Z, Z Data. See Figure 9.



	A	B	C	D	E	F	G	H
11								
12	Start	to	take	samples.				
13	X	+0.0343g	Y	-0.0293g	Z	+0.9580g		
14	X	+0.0333g	Y	-0.0352g	Z	+0.9893g		
15	X	+0.0401g	Y	-0.0293g	Z	+0.9375g		
16	X	+0.0343g	Y	-0.0283g	Z	+0.9815g		
17	X	+0.0382g	Y	-0.0244g	Z	+0.9708g		
18	X	+0.0147g	Y	-0.0098g	Z	+0.9648g		
19	X	+0.0254g	Y	-0.0225g	Z	+0.9786g		
20	X	+0.0195g	Y	-0.0166g	Z	+0.9796g		
21	X	+0.0343g	Y	-0.0538g	Z	+0.9483g		
22	X	+0.0215g	Y	+0.0000g	Z	+0.9688g		
23	X	-0.0323g	Y	-0.0391g	Z	+1.0059g		
24	X	+0.0127g	Y	+0.0088g	Z	+0.9580g		
25	X	+0.0499g	Y	-0.0391g	Z	+0.9961g		
26	X	+0.0030g	Y	-0.0205g	Z	+0.9727g		
27	X	+0.0508g	Y	-0.0333g	Z	+0.9580g		
28	X	+0.0030g	Y	-0.0059g	Z	+1.0039g		
29	X	+0.0421g	Y	-0.0313g	Z	+0.9414g		
30	X	-0.0205g	Y	-0.0147g	Z	+0.9249g		
31	X	+0.0176g	Y	-0.0244g	Z	+0.9766g		
32	X	+0.0010g	Y	-0.0166g	Z	+0.9952g		
33	X	+0.0195g	Y	-0.0020g	Z	+0.9522g		
34	X	+0.0293g	Y	-0.0059g	Z	+0.9708g		
35	X	-0.0059g	Y	-0.0078g	Z	+0.9961g		
36	X	-0.0176g	Y	-0.0098g	Z	+0.9747g		
37	X	+0.0421g	Y	-0.0411g	Z	+1.0117g		
38	X	+0.0069g	Y	+0.0195g	Z	+0.9668g		
39	X	-0.0059g	Y	-0.0078g	Z	+0.9336g		
40	X	+0.0382g	Y	-0.0186g	Z	+0.9619g		
41	X	-0.0323g	Y	+0.0049g	Z	+0.9541g		
42	X	+0.0254g	Y	-0.0205g	Z	+0.9648g		
43	X	+0.0147g	Y	-0.0411g	Z	+0.9932g		

Figure 9. Imported sample data in Excel file

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4296
Rev 1
02/2013