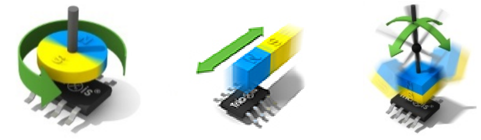
MLX90363 Getting Started Guide

Abstract

This guide was written with the goal of describing the basic setup and troubleshooting steps required for integrating the MLX90363 Triaxis™ device into either a rotary, linear, or joystick application.



Applicable Applications

All rotary, linear, and joystick applications using the MLX90363 in either the single or dual die package.

Recommended Tools and Resources

MLX90363 Datasheet
Oscilloscope or SPI->PC adaptor with SPI decoding
MLX90363 demo board and mbed board or PTC-04 with SPI daughterboard

Convention, Acronyms, and Abbreviations

Number Format and Text

Values in hexadecimal are shown with a preceding “0x”. For example: 0x12 is decimal 18.
Binary values are shown with a preceding “0b”. For example: 0b10010 is decimal 18.

Textual descriptions are shown in this format
`Code examples are shown in this format`
Warnings and cautions are shown in this format

Acronyms

MOSI = Master Out, Slave In
MISO = Master In, Slave Out
CLK = Serial Clock
SS = Slave Select (also called chip select). Active low.
NTT = Nothing To Transmit

MSb = Most Significant Bit
LSb = Least Significant Bit
MSB = Most Significant Byte
LSB = Least Significant Byte

Table of Contents

Axis Definition

The axes of the MLX90363 are defined in the datasheet and are shown below.

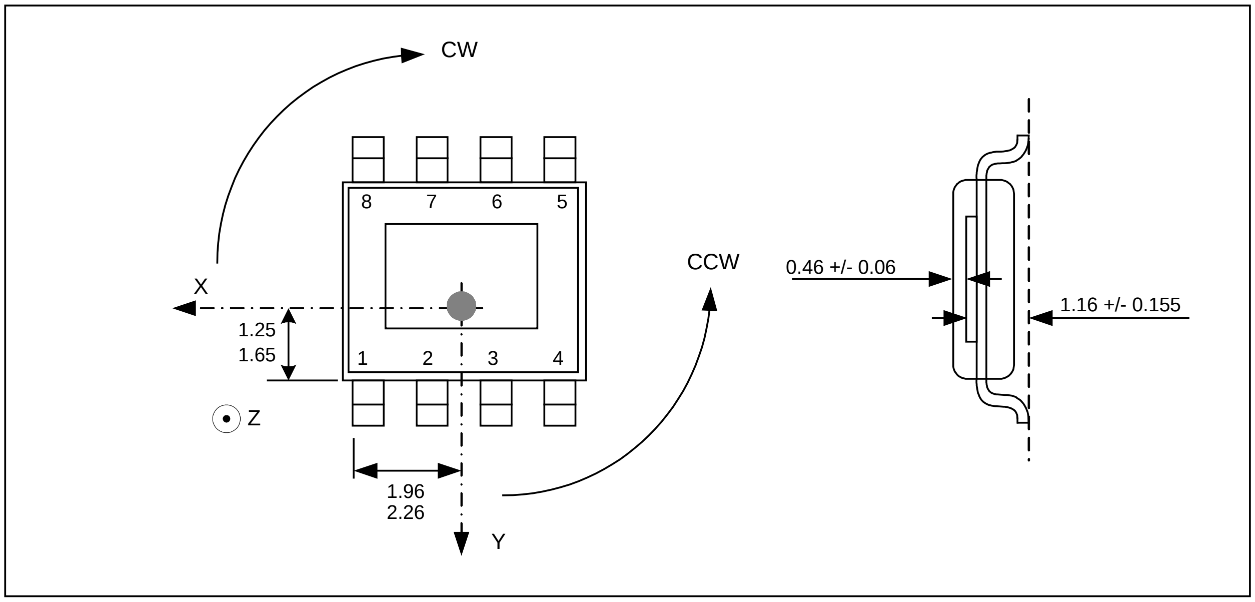


Figure 1: MLX90363 SOIC8 Axis Definitions

SPI Message Notation

The SPI message format used in this document is shown below. The row numbers indicate the word number while the column numbers indicate the bit number. When particular bits are not relevant the cell is shaded gray and these values will typically be set to zero (0) in the MOSI and MISO messages. Refer to the datasheet for more details.

#	7	6	5	4	3	2	1	0	#	7	6	5	4	3	2	1	0
1								(3)	0	(2)							(1)
3									2								(4)
5									4								
7	CRC [7:0]								6	Marker		Opcode or Counter					

(1): Bit[0] of Byte[0] (2): Bit[7] of Byte[0] (3): Bit[0] of Byte[1] (4): Bit[0] of Byte[2]

Marker determines what type of message is being requested or returned.
Marker 0x00: Alpha message
Marker 0x01: Alpha/Beta message
Marker 0x02: XYZ message
Marker 0x03: Command message (eg: Memory Write, Error, NTT)

The Opcode is the unique message identifier in the event of Marker 0x03 messages. In the event of a Marker 0x00, 0x01 and 0x02 message it is a rolling counter that increments with every GET message.

CRC is the unique CRC value computed based on the contents of the SPI message. For more details please refer to the datasheet.

Code Samples and Hardware Setup

The code samples given in this document are written for an mbed microcontroller development board, specifically the LPC11U24 evaluation board, using a single die MLX90363 (P/N MLX90363EDC-ABB-000). The examples make use of the virtual serial port on the mbed to output human readable data. The hardware block diagram for this setup is shown below for a single die connection.

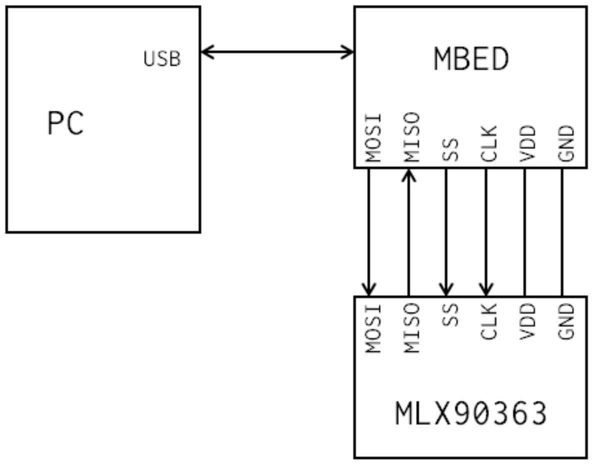
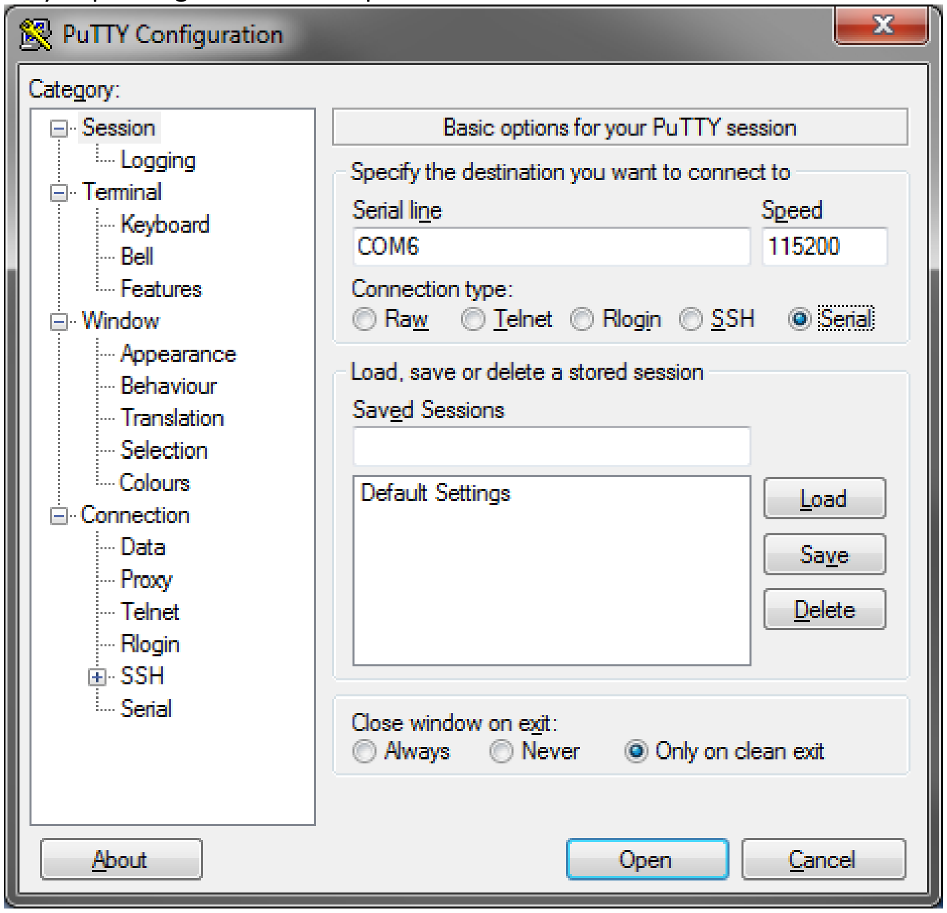


Figure 2: PC, mbed, MLX90363 block diagram

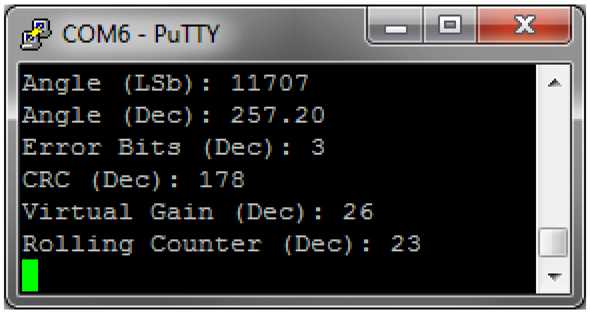
It is recommended to use the application diagram in the datasheet when selecting the components that surround the MLX90363. This typically requires a capacitor on the V_{DEC} pin and a capacitor on V_{DD} and a pullup resistor on the SS pin. Additional components, such as small series resistors, may be recommended for long distances on MISO, MOSI, and CLK. Please refer to the datasheet for the latest recommended circuit configuration.

Data output is performed via the mbed serial port. It is necessary to install a serial driver after which point any terminal program can be used to monitor the serial data stream. The image below shows the output from the rotary and linear applications in a PuTTY window. PuTTY is a free terminal program that can be used to monitor the serial port data. The code examples use a baud rate of 115200 and the COM port will vary depending on which USB port is used.



Serial line: COM6
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

Figure 3: PuTTY Setup Screen



```
Angle (LSb): 11707
Angle (Dec): 257.20
Error Bits (Dec): 3
CRC (Dec): 178
Virtual Gain (Dec): 26
Rolling Counter (Dec): 23
```

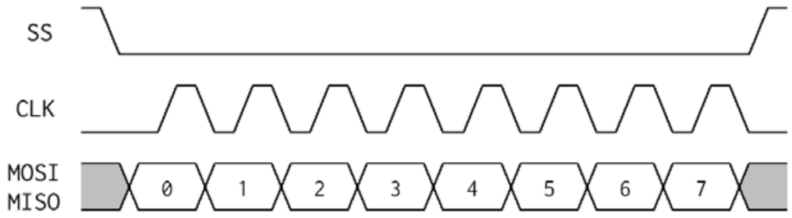
Figure 4: Serial Data Output Example

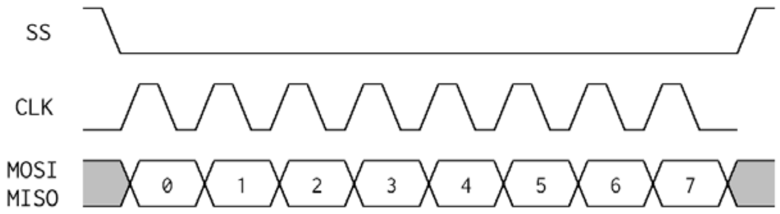
Caution: The code samples given in this guide are intended to enable basic functionality and do not implement the recommended startup sequence or diagnostic checks. It is strongly recommended that the user create additional procedures to implement these functions.

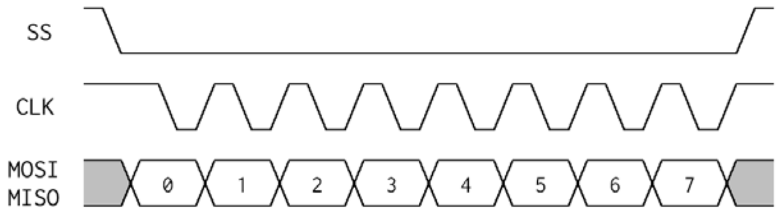
SPI Bus Protocol

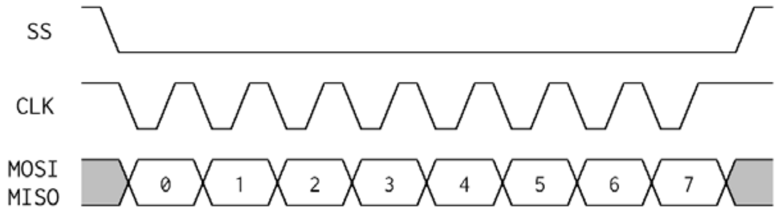
One initial hurdle with starting work with the MLX90363 is configuring the SPI bus correctly. Particularly important is the clock polarity (inactive level low or high) and phase (transfer data on falling or rising edge). These are grouped into modes according to the SPI specification and are shown below.

Parameters	Signal Diagram	Use with MLX90363?

MODE 0
CPOL=0
CPHA=0
		✗

MODE 1
CPOL=0
CPHA=1
		✓

MODE 2
CPOL=1
CPHA=0
		✗

MODE 3
CPOL=1
CPHA=1
		✗

Mode 1 is the only mode that the MLX90363 is able to use. If the wrong mode is selected then the MLX90363 will not be able to interpret the message correctly.

Note that the data and bus will all appear valid from a signal integrity standpoint even if the wrong mode is selected. A useful diagnostic is to confirm whether or not the CRC code from the MLX90363 is correct. If it is not then it is likely that the clock polarity, phase, or clock frequency is incorrect. Use an oscilloscope to ensure that the SPI lines look similar to the timing diagram of Mode 1 above.

Messages begin by transmitting Byte 0 first and ending with Byte 7 (CRC). Within each byte the MSb is transmitted first and LSb last. Therefore, the first bit clocked out is Byte 0[7] and the last bit in the message is Byte 7[7].

The clock frequency should not exceed the maximum allowed by the PINFILTER EEPROM setting. It is typically advised to start with 500kHz and increase the clock frequency after ensuring the hardware setup is correct.

Key Instructions

NOP

The NOP is a useful function to investigate the SPI bus and to obtain the result of the previous command. As the output is dependent on a user-provided key the behavior of the NOP message is deterministic and provides the ability to debug the SPI communication.

#	7	6	5	4	3	2	1	0	#	7	6	5	4	3	2	1	0
1									0								
3	KEY [15:8]								2	KEY [7:0]							
5									4								
7	CRC [7:0]								6	1	1	0	1	0	0	0	0

Table 1: NOP MOSI message structure

#	7	6	5	4	3	2	1	0	#	7	6	5	4	3	2	1	0
1									0								
3	KEY ECHO [15:8]								2	KEY ECHO [7:0]							
5	INVERTED KEY ECHO [15:8]								4	INVERTED KEY ECHO [15:8]							
7	CRC [7:0]								6	1	1	0	1	0	0	0	1

Table 2: NOP MISO message structure

Note that the key can be any 16-bit number. In the examples in this application note the key is selected to be 0xAAAA. The KEY ECHO is the same as the key transmitted by the master while the INVERTED KEY ECHO is the key but bitwise inverted. For example, a key of 0xAAAA results in an inverted key echo of 0x5555.

```
Code:
0xAAAA = 0b1010101010101010
0x5555 = 0b0101010101010101
```

GET1

The GET messages are used to obtain angular measurement data from the MLX90363. In this document only GET1 will be used. The basic format is shown below.

#	7	6	5	4	3	2	1	0	#	7	6	5	4	3	2	1	0
1								RST	0								
3	TIME OUT VALUE [15:8]								2	TIME OUT VALUE [7:0]							
5									4								
7	CRC [7:0]								6	Marker		0	1	0	0	1	1

Table 3: GET1 message structure

RST is a single bit which commands the MLX90363 to reset its internal rolling counter. In following examples this will be set to 0.

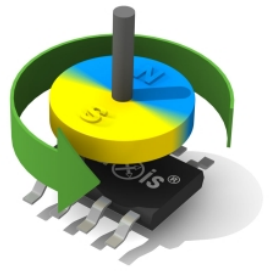
TIME OUT VALUE sets the internal time out between GET messages. In the event that the delay between the GET message and retreival of the aquired data exceeds the time out value then the MLX90363 will reply with a NTT or error message. Each LSb corresponds to 1 µs. In the following examples this will be set to 0xFFFF or approximately 65ms.

GET2 and GET3 messages are also available to request data but are beyond the scope of this note.

GET returns either a single angle, alpha, or two angles, alpha and beta, or the raw XYZ components. Alpha values are used for rotary and linear applications where the 2D formula is used. Alpha+Beta values are used for joystick applications where two angles (eg: fore/aft and left/right movement) need to be determined.

For 2D computations the angle is mapped over fourteen bits which results in a resolution of $\left(360/2^{14}\right) \cong 0.022\,\mathrm{deg/LSb}$. The maximum angle span is 360 degrees for linear and rotary applications while joystick applications are limited to the range of 0 to 180 degrees.

Rotary Application



Rotary applications are the most common and are also the easiest to implement. No EEPROM parameters must be programmed and only a GET message needs to be sent to obtain data.

Magnetic Configuration

A diametrically magnetized magnet (similar to the one shown in the above image) is normally used and recommended. While there is no restriction on size beyond that needed to fulfill the magnetic field strength requirement (20-70mT), it is advised to not make the magnet too small in diameter as errors caused by off-axis assembly errors increase as the magnet diameter shrinks. A general rule of thumb is that the magnet diameter has to be ten times the maximum off-axis error to maintain a nonlinearity error of less than 1 degree.

Suggested EEPROM Configuration

For rotary applications it is not necessary to modify the EEPROM contents from their default settings. Some settings, such as the filter, may be modified to best fit the end application.

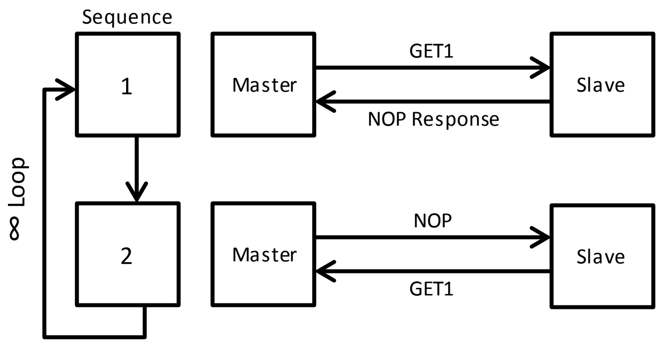
Rotary Code Example

The following code example configures the microcontroller to report out the following pieces of data:

1) Angle in LSb's
2) Angle in decimal degrees
3) The two error bits in the GET1 message
4) The CRC in decimal format
5) The virtual gain in decimal format
6) The rolling counter in decimal format

The data is collected, formatted, and output to the virtual serial port.

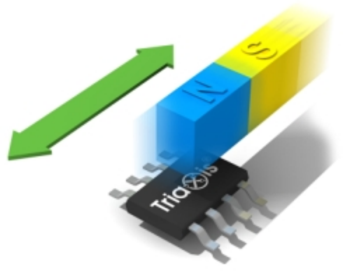
A GET1 message is sent approximately every 500ms which does not include the time required to transmit the various data over the serial port. As this is slower than the maximum time out rate of 65ms it is necessary to interlace GET1 and NOP messages to avoid receiving the NTT error.



```
/***********************************************
MLX90363 Example rotary program for mbed board
Purpose: Demonstrate getting angular data from
MLX90363 via GET1a message.

Revision: 001
Date: 1-Nov-13
Changes: Initial Revision
***********************************************/

#include "mbed.h"

SPI spi(p5, p6, p7);         //MOSI, MISO, CLK
DigitalOut ss1(p8);          //SS Pin for Die 1

Serial pc(USBTX, USBRX);            //Create a virtual serial port

int main(void) {
       char u8_spi_mode = 1; //SPI mode
       char u8_spi_bits = 8;        //number of bits per mbed write command
       char u8_spi_read_buffer[8];
       char u8_spi_write_buffer[8];
       float f32_angle_degrees;
       unsigned int u16_angle_lsb;
       char u8_error_lsb;
       char u8_rollcnt_dec;
       char u8_virtualgain_dec;
       char u8_crc_dec;

       //360 degrees is mapped to 14 bits = 360/2^14 = 0.02197
       const float f32_lsb_to_dec_degrees = 0.02197;

       //Serial port setup
       pc.baud(115200);

       //SPI Bus setup
       spi.format(u8_spi_bits, u8_spi_mode);
       spi.frequency(500000);
       ss1 = 1;    //SS to deselected state

       while (1){
              //issue GET1 message
              u8_spi_write_buffer[0] = 0x00;
              u8_spi_write_buffer[1] = 0x00;
              u8_spi_write_buffer[2] = 0xFF;
              u8_spi_write_buffer[3] = 0xFF;
              u8_spi_write_buffer[4] = 0x00;
              u8_spi_write_buffer[5] = 0x00;
              u8_spi_write_buffer[6] = 0x13;
              u8_spi_write_buffer[7] = 0xEA;

              ss1 = 0;
              for (int i = 0; i < 8; i++){
                     u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
              }
              ss1 = 1;
              wait_ms(1);

              //issue NOP message
              u8_spi_write_buffer[0] = 0x00;
              u8_spi_write_buffer[1] = 0x00;
              u8_spi_write_buffer[2] = 0xAA;
              u8_spi_write_buffer[3] = 0xAA;
```

```
			u8_spi_write_buffer[4] = 0x00;
			u8_spi_write_buffer[5] = 0x00;
			u8_spi_write_buffer[6] = 0xD0;
			u8_spi_write_buffer[7] = 0xAB;

			ss1 = 0;
			for (int i = 0; i < 8; i++){
				u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
			}
			ss1 = 1;

			//Extract and convert the angle to degrees
			//remove error bits and shift to high byte
			u16_angle_lsb = (u8_spi_read_buffer[1] & 0x3F) << 8;
			//add LSB of angle
			u16_angle_lsb = u16_angle_lsb + u8_spi_read_buffer[0];
			//convert to decimal degrees
			f32_angle_degrees = u16_angle_lsb * f32_lsb_to_dec_degrees;

			//Extract the error bits
			u8_error_lsb = u8_spi_read_buffer[1] >> 6;

			//Extract the CRC
			u8_crc_dec = u8_spi_read_buffer[7];

			//Extract the virtual gain byte
			u8_virtualgain_dec = u8_spi_read_buffer[4];

			//Extract the rolling counter
			u8_rollcnt_dec = u8_spi_read_buffer[6] & 0x3F;

			//Send results to serial port
			pc.printf("Angle (LSb): %d\r\n", u16_angle_lsb);
			pc.printf("Angle (Dec): %2.2f\r\n", f32_angle_degrees);
			pc.printf("Error Bits (Dec): %d\r\n", u8_error_lsb);
			pc.printf("CRC (Dec): %d\r\n", u8_crc_dec);
			pc.printf("Virtual Gain (Dec): %d\r\n", u8_virtualgain_dec);
			pc.printf("Rolling Counter (Dec): %d\r\n", u8_rollcnt_dec);
			wait(0.5);
		}
}
```

Linear Application



Linear applications are used when linear motion is present (eg: linear actuator) or it is not possible to mount the sensor on the axis of rotation. The method of retreiving the data is the same as a rotary application (GET1) however the sensor EEPROM must be programmed with a few values in order to enable a linear calculation.

Magnetic Configuration

The magnetic configuration described in this guide uses the setup shown in the image above. However, this is not the only configuration possible. Rotating the sensor or magnet 90 degrees is also acceptable. Arc motions are also possible with the sensor configured for linear motion but will typically have additional nonlinearity error which must be compensated in the connected microcontroller. A key point is that the linear motion utilizes a combination of X and Z or Y and Z axes unlike the rotary configuration that uses X and Y axes.

EEPROM Configuration

For this application it is necessary to modify the MapXYZ parameter to select the relevant field components. In this example axes X and Z are used which requires a MapXYZ selection of 1. It is also necessary to modify the SMISM and SEL_SMISM parameter to reduce nonlinearity error due to the sensitivity mismatch between the X and Z or Y and Z axes. A typical value is a factor of 0.7 applied to the X or Y axis.

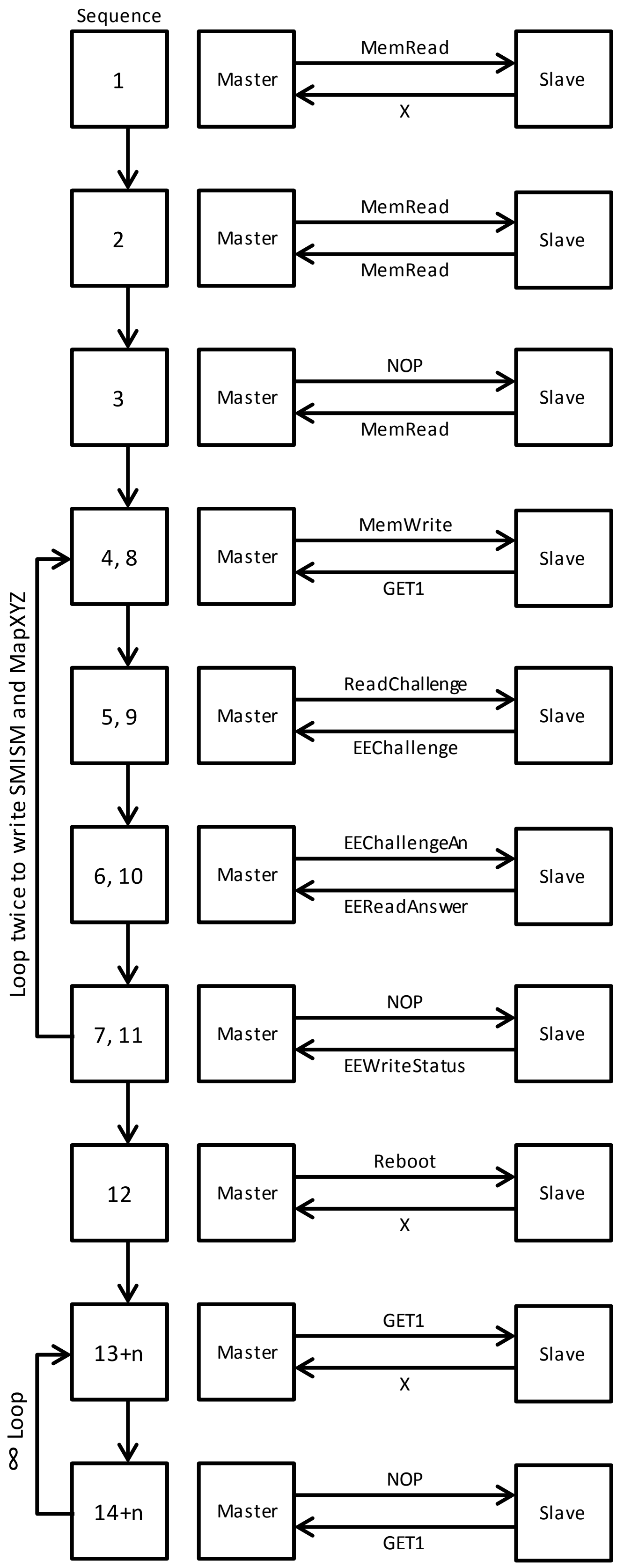
EEPROM Parameter	Address	Typical Value
SMISM	0x1032[14:0]	0x5999
SEL_SMISM	0x1032[15]	0
MapXYZ	0x102A[2:0]	1

Table 4: EEPROM Configuration for Linear Applications

Code Example

The code example below builds on the code example given for the rotary application. The ability to dynamically calculate the CRC was added along with code to read, modify, and write specific EEPROM parameters. The EEPROM only needs to be written once after which point a reboot or power cycle needs to be performed to take into account the modified parameters.

Once programmed the EEPROM parameters are stable through power cycles. Therefore, Melexis recommends that the EEPROM parameters are only programmed once and not every power cycle. This could be assured by performing the programming at in circuit testing (ICT) or by logging a value in the system microcontroller.

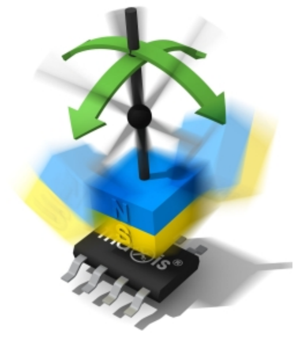

Sequence
1
Master
MemRead
X
Slave
2
Master
MemRead
MemRead
Slave
3
Master
NOP
MemRead
Slave
4, 8
Master
MemWrite
GET1
Slave
5, 9
Master
ReadChallenge
EEChallenge
Slave
6, 10
Master
EEChallengeAn
EEReadAnswer
Slave
7, 11
Master
NOP
EEWriteStatus
Slave
Loop twice to write SMISM and MapXYZ
12
Master
Reboot
X
Slave
13+n
Master
GET1
X
Slave
14+n
Master
NOP
GET1
Slave
∞ Loop

```
/***********************************************
MLX90363 Example linear program for mbed board
Purpose: Demonstrate getting linear angle data
from MLX90363 via GET1a message after
modification of SMISM and MapXYZ.

Revision: 001
Date: 1-Nov-13
Changes: Initial Revision
***********************************************/

#include "mbed.h"

SPI spi(p5, p6, p7);     //MOSI, MISO, CLK
DigitalOut ss1(p8);      //SS Pin for Die 1

Serial pc(USBTX, USBRX);    //Create a serial port

//Define and initialize CRC array, 256 bytes
char CRCArray[] = {
       0x00, 0x2F, 0x5E, 0x71, 0xBC, 0x93, 0xE2, 0xCD, 0x57, 0x78, 0x09, 0x26,
       0xEB, 0xC4, 0xB5, 0x9A, 0xAE, 0x81, 0xF0, 0xDF, 0x12, 0x3D, 0x4C, 0x63,
       0xF9, 0xD6, 0xA7, 0x88, 0x45, 0x6A, 0x1B, 0x34, 0x73, 0x5C, 0x2D, 0x02,
       0xCF, 0xE0, 0x91, 0xBE, 0x24, 0x0B, 0x7A, 0x55, 0x98, 0xB7, 0xC6, 0xE9,
       0xDD, 0xF2, 0x83, 0xAC, 0x61, 0x4E, 0x3F, 0x10, 0x8A, 0xA5, 0xD4, 0xFB,
       0x36, 0x19, 0x68, 0x47, 0xE6, 0xC9, 0xB8, 0x97, 0x5A, 0x75, 0x04, 0x2B,
       0xB1, 0x9E, 0xEF, 0xC0, 0x0D, 0x22, 0x53, 0x7C, 0x48, 0x67, 0x16, 0x39,
       0xF4, 0xDB, 0xAA, 0x85, 0x1F, 0x30, 0x41, 0x6E, 0xA3, 0x8C, 0xFD, 0xD2,
       0x95, 0xBA, 0xCB, 0xE4, 0x29, 0x06, 0x77, 0x58, 0xC2, 0xED, 0x9C, 0xB3,
       0x7E, 0x51, 0x20, 0x0F, 0x3B, 0x14, 0x65, 0x4A, 0x87, 0xA8, 0xD9, 0xF6,
       0x6C, 0x43, 0x32, 0x1D, 0xD0, 0xFF, 0x8E, 0xA1, 0xE3, 0xCC, 0xBD, 0x92,
       0x5F, 0x70, 0x01, 0x2E, 0xB4, 0x9B, 0xEA, 0xC5, 0x08, 0x27, 0x56, 0x79,
       0x4D, 0x62, 0x13, 0x3C, 0xF1, 0xDE, 0xAF, 0x80, 0x1A, 0x35, 0x44, 0x6B,
       0xA6, 0x89, 0xF8, 0xD7, 0x90, 0xBF, 0xCE, 0xE1, 0x2C, 0x03, 0x72, 0x5D,
       0xC7, 0xE8, 0x99, 0xB6, 0x7B, 0x54, 0x25, 0x0A, 0x3E, 0x11, 0x60, 0x4F,
       0x82, 0xAD, 0xDC, 0xF3, 0x69, 0x46, 0x37, 0x18, 0xD5, 0xFA, 0x8B, 0xA4,
       0x05, 0x2A, 0x5B, 0x74, 0xB9, 0x96, 0xE7, 0xC8, 0x52, 0x7D, 0x0C, 0x23,
       0xEE, 0xC1, 0xB0, 0x9F, 0xAB, 0x84, 0xF5, 0xDA, 0x17, 0x38, 0x49, 0x66,
       0xFC, 0xD3, 0xA2, 0x8D, 0x40, 0x6F, 0x1E, 0x31, 0x76, 0x59, 0x28, 0x07,
       0xCA, 0xE5, 0x94, 0xBB, 0x21, 0x0E, 0x7F, 0x50, 0x9D, 0xB2, 0xC3, 0xEC,
       0xD8, 0xF7, 0x86, 0xA9, 0x64, 0x4B, 0x3A, 0x15, 0x8F, 0xA0, 0xD1, 0xFE,
       0x33, 0x1C, 0x6D, 0x42 };

//Function ComputeCRC
//Requires: 7 bytes to be transmitted to slave device
//Returns: 1 byte corresponding to CRC code
char ComputeCRC(char Byte0, char Byte1, char Byte2, char Byte3, char Byte4, char Byte5, char
Byte6){
       char CRC = 0xFF;
       CRC = CRCArray[CRC ^ Byte0];
       CRC = CRCArray[CRC ^ Byte1];
       CRC = CRCArray[CRC ^ Byte2];
       CRC = CRCArray[CRC ^ Byte3];
       CRC = CRCArray[CRC ^ Byte4];
       CRC = CRCArray[CRC ^ Byte5];
       CRC = CRCArray[CRC ^ Byte6];
       CRC = ~CRC;
       return CRC;
}

int main(void) {
       char u8_spi_mode = 1;
       char u8_spi_bits = 8;
       char u8_spi_read_buffer[8], u8_spi_write_buffer[8];
       float f32_angle_degrees;
       int u16_angle_lsb;
       char RByte0, RByte1, RByte2, RByte3, RByte4, RByte5, RByte6, RByte7;
       char Addr_102A_LSB, Addr_102A_MSB, Addr_1032_LSB, Addr_1032_MSB;
       char u8_error_lsb, u8_rollcnt_dec, u8_virtualgain_dec, u8_crc_dec;
       char u8_pass = 0;
       const float f32_lsb_to_dec_degrees = 0.02197;

       //Serial port setup
       pc.baud(115200);

       //SPI Bus setup
       spi.format(u8_spi_bits, u8_spi_mode);
       spi.frequency(500000);
       ss1 = 1;    //Deselect MLX90363

       pc.printf("Beginning Test\r\n");

       //write MapXYZ and SMISM
       //Transmit MemRead and discard results
       //Requires: Two memory address (even only)
       //Result: Ready status bytes or invalid data on first message
       //Result: Data in memory addresses 0x102A and 0x1032 in second message
       for (int i = 0; i <= 1; i++){
              ss1 = 0;
              RByte0 = spi.write(0x2A);   //Memory location 0x102A
              RByte1 = spi.write(0x10);
              RByte2 = spi.write(0x32);   //Memory location 0x1032
              RByte3 = spi.write(0x10);
              RByte4 = spi.write(0x00);
              RByte5 = spi.write(0x00);
              RByte6 = spi.write(0xC1);
              RByte7 = spi.write(ComputeCRC(0x2A, 0x10, 0x32, 0x10, 0x00, 0x00, 0xC1));
              ss1 = 1;
              wait_ms(10);
       }

       //Set MAPXYZ to 1 to use X and Z axes, clear 3D bit
       Addr_102A_LSB = (RByte0 & 0xF0) | 0x01;
       Addr_102A_MSB = RByte1;

       //Set SMISM and SEL_SMISM
       Addr_1032_LSB = 0x99;
       Addr_1032_MSB = 0x59;

       //Transmit NOP to reset communication
       //EEPROM write should be preceded by a NOP (Ref pg 26, S14.10, Rev 002)
       ss1 = 0;
       RByte0 = spi.write(0x00);
       RByte1 = spi.write(0x00);
       RByte2 = spi.write(0xAA);
       RByte3 = spi.write(0xAA);
       RByte4 = spi.write(0x00);
       RByte5 = spi.write(0x00);
       RByte6 = spi.write(0xD0);
       RByte7 = spi.write(0xAB);
       ss1 = 1;
       wait_ms(10);

       for (int i = 0; i <= 1; i++){
              //Transmit MemWrite message
              //Requires: Key to address location (two bytes)
```

```
// Data to transmit (two bytes)
//Results: NOP Response
ss1 = 0;
if (u8_pass == 0){
       RByte0 = spi.write(0x00);
       RByte1 = spi.write(0x2A);
       RByte2 = spi.write(0x45);
       RByte3 = spi.write(0x8F);
       RByte4 = spi.write(Addr_102A_LSB);
       RByte5 = spi.write(Addr_102A_MSB);
       RByte6 = spi.write(0xC3);
       RByte7 = spi.write(ComputeCRC(0x00, 0x2A, 0x45, 0x8F,
              Addr_102A_LSB, Addr_102A_MSB, 0xC3));
}
else{
       RByte0 = spi.write(0x00);
       RByte1 = spi.write(0x32);
       RByte2 = spi.write(0x13);
       RByte3 = spi.write(0x17);
       RByte4 = spi.write(Addr_1032_LSB);
       RByte5 = spi.write(Addr_1032_MSB);
       RByte6 = spi.write(0xC3);
       RByte7 = spi.write(ComputeCRC(0x00, 0x32, 0x13, 0x17,
              Addr_1032_LSB, Addr_1032_MSB, 0xC3));
}
ss1 = 1;
wait_ms(10);

//Transmit ReadChallenge message
//Requires: Nothing
//Results: EE Challenge Key
ss1 = 0;
RByte0 = spi.write(0x00);
RByte1 = spi.write(0x00);
RByte2 = spi.write(0x00);
RByte3 = spi.write(0x00);
RByte4 = spi.write(0x00);
RByte5 = spi.write(0x00);
RByte6 = spi.write(0xCF);
RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xCF));
ss1 = 1;
wait_ms(10);

//Transmit EEChallengeAnswer message
//Requires: Key Echo, Inverted Key Echo
//Results: EEReadAnswer
ss1 = 0;
spi.write(0x00);
spi.write(0x00);
spi.write(RByte2 ^ 0x34);
spi.write(RByte3 ^ 0x12);
spi.write(~(RByte2 ^ 0x34));
spi.write(~(RByte3 ^ 0x12));
spi.write(0xC5);
spi.write(ComputeCRC(0x00, 0x00, RByte2 ^ 0x34, RByte3 ^ 0x12,
       ~(RByte2 ^ 0x34), ~(RByte3 ^ 0x12), 0xC5));
ss1 = 1;
wait_ms(40);    //mandatory delay for writing EEPROM

//Transmit NOP message
//Required to read out EEPROMWriteStatus
//Results: Error code indicating successful write or not
//Error codes: 1=Success  All others=Failure
```

```
                ss1 = 0;
                RByte0 = spi.write(0x00);
                RByte1 = spi.write(0x00);
                RByte2 = spi.write(0xAA);
                RByte3 = spi.write(0xAA);
                RByte4 = spi.write(0x00);
                RByte5 = spi.write(0x00);
                RByte6 = spi.write(0xD0);
                RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0xAA, 0xAA, 0x00, 0x00, 0xD0));
                ss1 = 1;
                pc.printf("Error code (1=success): %02X\r\n", RByte0);
                u8_pass++;
        }
        wait(10);
        //Transmit reboot command
        //Required to reset chip to use new EEPROM parameters
        ss1 = 0;
        RByte0 = spi.write(0x00);
        RByte1 = spi.write(0x00);
        RByte2 = spi.write(0x00);
        RByte3 = spi.write(0x00);
        RByte4 = spi.write(0x00);
        RByte5 = spi.write(0x00);
        RByte6 = spi.write(0xEF);
        RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xEF));
        ss1 = 1;
        wait_ms(30);
        pc.printf("Programming complete\r\n");

        while (1){
                u8_spi_write_buffer[0] = 0x00;
                u8_spi_write_buffer[1] = 0x00;
                u8_spi_write_buffer[2] = 0xFF;
                u8_spi_write_buffer[3] = 0xFF;
                u8_spi_write_buffer[4] = 0x00;
                u8_spi_write_buffer[5] = 0x00;
                u8_spi_write_buffer[6] = 0x13;
                u8_spi_write_buffer[7] = 0xEA;

                ss1 = 0;
                for (int i = 0; i < 8; i++){
                        u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
                }
                ss1 = 1;

                wait_ms(10);
                u8_spi_write_buffer[0] = 0x00;
                u8_spi_write_buffer[1] = 0x00;
                u8_spi_write_buffer[2] = 0xAA;
                u8_spi_write_buffer[3] = 0xAA;
                u8_spi_write_buffer[4] = 0x00;
                u8_spi_write_buffer[5] = 0x00;
                u8_spi_write_buffer[6] = 0xD0;
                u8_spi_write_buffer[7] = 0xAB;

                ss1 = 0;
                for (int i = 0; i < 8; i++){
                        u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
                }
                ss1 = 1;

                //Extract and convert the angle to degrees
                //remove error bits and combine two bytes into word
                u16_angle_lsb = (u8_spi_read_buffer[1] & 0x3F) << 8;
                u16_angle_lsb = u16_angle_lsb + u8_spi_read_buffer[0];

                //convert to decimal degrees
                f32_angle_degrees = u16_angle_lsb * f32_lsb_to_dec_degrees;

                //Extract the error bits
                u8_error_lsb = u8_spi_read_buffer[1] >> 6;

                //Extract the CRC
                u8_crc_dec = u8_spi_read_buffer[7];

                //Extract the virtual gain byte
                u8_virtualgain_dec = u8_spi_read_buffer[4];

                //Extract the rolling counter
                u8_rollcnt_dec = u8_spi_read_buffer[6] & 0x3F;

                //Send results to serial port
                pc.printf("Angle (LSb): %d\r\n", u16_angle_lsb);
                pc.printf("Angle (Dec): %2.2f\r\n", f32_angle_degrees);
                pc.printf("Error Bits (Dec): %d\r\n", u8_error_lsb);
                pc.printf("CRC (Dec): %d\r\n", u8_crc_dec);
                pc.printf("Virtual Gain (Dec): %d\r\n", u8_virtualgain_dec);
                pc.printf("Rolling Counter (Dec): %d\r\n", u8_rollcnt_dec);
                wait(0.5);
        }
}
```

Joystick Application



Joystick applications are used when when 3D motion is to be sensed. Typical applications include items such as joysticks, game controllers, and robotic actuators.

Magnetic Configuration

The magnetic configuration described in this guide uses the setup shown in the image above. This configuration is called the ball and socket configuration. Another application, the gimbal design, can also be used where the two axes of rotation are coplanar with the die of the MLX90363. The gimbal design offers improved linearity performance across the entire rotation range vs. the ball and socket design due to the constant air gap between the sensor and magnet over the entire range of motion.

EEPROM Configuration

For joystick applications it is necessary to modify the 3D parameter to enable the 3D calculation. It is also necessary to modify the KALPHA and KBETA parameters to reduce nonlinearity error due to the sensitivity mismatch between the X and Z or Y and Z axes, similar to the linear application. A typical value is a factor of 1.4.

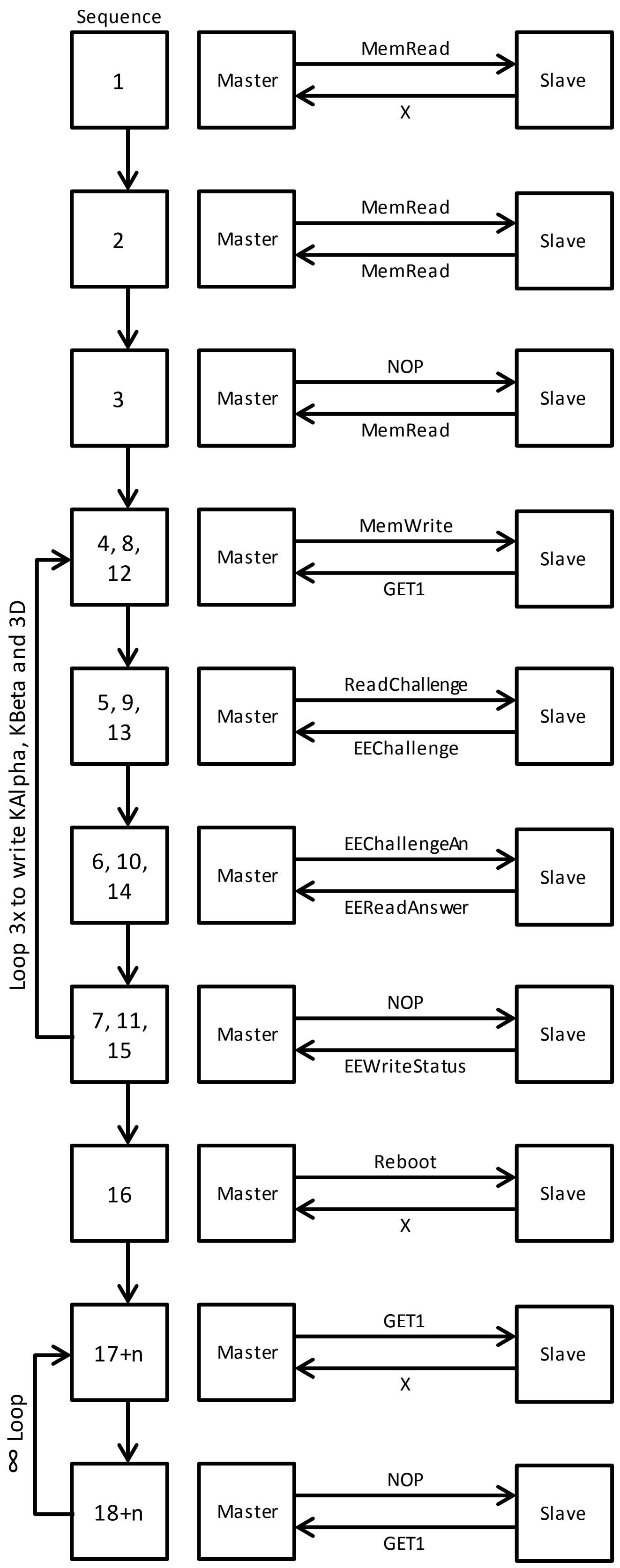
EEPROM Parameter	Address	Typical Value
KAlpha	0x1022[15:0]	0xB333
KBeta	0x1024[15:0]	0xB333
3D	0x102A[3]	1
Kt	0x1030[15:0]	1 (default)

Table 5: EEPROM Configuration for Joystick Applications

Code Example

The code example below builds on the code example given for the linear application. The basic flow remains the same with the CRC computation, modification of the EEPROM parameters, and readout of the data however the GET1 message with the Alpha + Beta marker is now used to enable the readout of both fore/aft angles and left/right angles.

Once programmed the EEPROM parameters are stable through power cycles. Therefore, Melexis recommends that the EEPROM parameters are only programmed once and not every power cycle. This could be assured by performing the programming at in circuit testing (ICT) or by logging a value in the system microcontroller.


Sequence
1
Master
MemRead
X
Slave
2
Master
MemRead
MemRead
Slave
3
Master
NOP
MemRead
Slave
4, 8, 12
Master
MemWrite
GET1
Slave
5, 9, 13
Master
ReadChallenge
EEChallenge
Slave
6, 10, 14
Master
EEChallengeAn
EEReadAnswer
Slave
7, 11, 15
Master
NOP
EEWriteStatus
Slave
Loop 3x to write KAlpha, KBeta and 3D
16
Master
Reboot
X
Slave
17+n
Master
GET1
X
Slave
18+n
Master
NOP
GET1
Slave
∞ Loop

```
/************************************************
MLX90363 Example joystick program for mbed board
Purpose: Demonstrate getting linear angle data
from MLX90363 via GET1ab message after
modification of KAlpha, KBeta, and 3D.

Revision: 001
Date: 1-Nov-13
Changes: Initial Revision
************************************************/

#include "mbed.h"

SPI spi(p5, p6, p7);                    //MOSI, MISO, CLK
DigitalOut ss1(p8);                     //SS Pin for Die 1

Serial pc(USBTX, USBRX);    //Create a serial port

//Define and initialize CRC array, 256 bytes
char CRCArray[] = {
       0x00, 0x2F, 0x5E, 0x71, 0xBC, 0x93, 0xE2, 0xCD, 0x57, 0x78, 0x09, 0x26,
       0xEB, 0xC4, 0xB5, 0x9A, 0xAE, 0x81, 0xF0, 0xDF, 0x12, 0x3D, 0x4C, 0x63,
       0xF9, 0xD6, 0xA7, 0x88, 0x45, 0x6A, 0x1B, 0x34, 0x73, 0x5C, 0x2D, 0x02,
       0xCF, 0xE0, 0x91, 0xBE, 0x24, 0x0B, 0x7A, 0x55, 0x98, 0xB7, 0xC6, 0xE9,
       0xDD, 0xF2, 0x83, 0xAC, 0x61, 0x4E, 0x3F, 0x10, 0x8A, 0xA5, 0xD4, 0xFB,
       0x36, 0x19, 0x68, 0x47, 0xE6, 0xC9, 0xB8, 0x97, 0x5A, 0x75, 0x04, 0x2B,
       0xB1, 0x9E, 0xEF, 0xC0, 0x0D, 0x22, 0x53, 0x7C, 0x48, 0x67, 0x16, 0x39,
       0xF4, 0xDB, 0xAA, 0x85, 0x1F, 0x30, 0x41, 0x6E, 0xA3, 0x8C, 0xFD, 0xD2,
       0x95, 0xBA, 0xCB, 0xE4, 0x29, 0x06, 0x77, 0x58, 0xC2, 0xED, 0x9C, 0xB3,
       0x7E, 0x51, 0x20, 0x0F, 0x3B, 0x14, 0x65, 0x4A, 0x87, 0xA8, 0xD9, 0xF6,
       0x6C, 0x43, 0x32, 0x1D, 0xD0, 0xFF, 0x8E, 0xA1, 0xE3, 0xCC, 0xBD, 0x92,
       0x5F, 0x70, 0x01, 0x2E, 0xB4, 0x9B, 0xEA, 0xC5, 0x08, 0x27, 0x56, 0x79,
       0x4D, 0x62, 0x13, 0x3C, 0xF1, 0xDE, 0xAF, 0x80, 0x1A, 0x35, 0x44, 0x6B,
       0xA6, 0x89, 0xF8, 0xD7, 0x90, 0xBF, 0xCE, 0xE1, 0x2C, 0x03, 0x72, 0x5D,
       0xC7, 0xE8, 0x99, 0xB6, 0x7B, 0x54, 0x25, 0x0A, 0x3E, 0x11, 0x60, 0x4F,
       0x82, 0xAD, 0xDC, 0xF3, 0x69, 0x46, 0x37, 0x18, 0xD5, 0xFA, 0x8B, 0xA4,
       0x05, 0x2A, 0x5B, 0x74, 0xB9, 0x96, 0xE7, 0xC8, 0x52, 0x7D, 0x0C, 0x23,
       0xEE, 0xC1, 0xB0, 0x9F, 0xAB, 0x84, 0xF5, 0xDA, 0x17, 0x38, 0x49, 0x66,
       0xFC, 0xD3, 0xA2, 0x8D, 0x40, 0x6F, 0x1E, 0x31, 0x76, 0x59, 0x28, 0x07,
       0xCA, 0xE5, 0x94, 0xBB, 0x21, 0x0E, 0x7F, 0x50, 0x9D, 0xB2, 0xC3, 0xEC,
       0xD8, 0xF7, 0x86, 0xA9, 0x64, 0x4B, 0x3A, 0x15, 0x8F, 0xA0, 0xD1, 0xFE,
       0x33, 0x1C, 0x6D, 0x42 };

//Function ComputeCRC
//Requires: 7 bytes to be transmitted to slave device
//Returns: 1 byte corresponding to CRC code
char ComputeCRC(char Byte0, char Byte1, char Byte2, char Byte3,
       char Byte4, char Byte5, char Byte6){
       char CRC = 0xFF;
       CRC = CRCArray[CRC ^ Byte0];
       CRC = CRCArray[CRC ^ Byte1];
       CRC = CRCArray[CRC ^ Byte2];
       CRC = CRCArray[CRC ^ Byte3];
       CRC = CRCArray[CRC ^ Byte4];
       CRC = CRCArray[CRC ^ Byte5];
       CRC = CRCArray[CRC ^ Byte6];
       CRC = ~CRC;
       return CRC;
}

int main(void) {
       char u8_spi_mode = 1;   //spi mode
       char u8_spi_bits = 8;   //number of bits per mbed write command
       char u8_spi_read_buffer[8];
       char u8_spi_write_buffer[8];
       float f32_alpha_angle_degrees, f32_beta_angle_degrees;
       int u16_alpha_angle_lsb, u16_beta_angle_lsb;
       char RByte0, RByte1, RByte2, RByte3, RByte4, RByte5, RByte6, RByte7;
       char Addr_102A_LSB, Addr_102A_MSB, Addr_1022_LSB, Addr_1022_MSB,
              Addr_1024_LSB, Addr_1024_MSB;
       char u8_error_lsb, u8_rollcnt_dec, u8_virtualgain_dec, u8_crc_dec;
       char u8_pass = 0;
       const float f32_lsb_to_dec_degrees = 0.02197;

       //Serial port setup
       pc.baud(115200);

       //SPI Bus setup
       spi.format(u8_spi_bits, u8_spi_mode);
       spi.frequency(500000);
       ss1 = 1;

       pc.printf("Beginning Test\r\n");

       //write MapXYZ and SMISM
       //Transmit MemRead and discard results
       //Requires: Two memory address (even only)
       //Result: Ready status bytes or invalid data on first message
       //Result: Data in memory addresses 0x102A and 0x1032 in second message
       for (int i = 0; i <= 1; i++){
              ss1 = 0;
              RByte0 = spi.write(0x2A);   //Memory location 0x102A
              RByte1 = spi.write(0x10);
              RByte2 = spi.write(0x00);   //Memory location 0x1000
              RByte3 = spi.write(0x10);
              RByte4 = spi.write(0x00);
              RByte5 = spi.write(0x00);
              RByte6 = spi.write(0xC1);
              RByte7 = spi.write(ComputeCRC(0x2A, 0x10, 0x32, 0x10, 0x00, 0x00, 0xC1));
              ss1 = 1;
              wait_ms(10);
       }

       //Set MAPXYZ to 0, Enable 3D mode
       Addr_102A_LSB = (RByte0 & 0xF0) ^ 0x08;
       Addr_102A_MSB = RByte1;

       //Set Kalpha to 1.4
       Addr_1022_LSB = 0x33;
       Addr_1022_MSB = 0xB3;

       //Set Kbeta to 1.4
       Addr_1024_LSB = 0x33;
       Addr_1024_MSB = 0xB3;

       //Transmit NOP to reset communication
       //EEPROM write should be preceded by a NOP (Ref pg 26, S14.10, Rev 002)
       ss1 = 0;
       RByte0 = spi.write(0x00);
       RByte1 = spi.write(0x00);
       RByte2 = spi.write(0xAA);
       RByte3 = spi.write(0xAA);
       RByte4 = spi.write(0x00);
       RByte5 = spi.write(0x00);
       RByte6 = spi.write(0xD0);
       RByte7 = spi.write(0xAB);
```

```
ss1 = 1;
wait_ms(10);

for (int i = 0; i <= 2; i++){
        //Define MemWrite message
        //Requires: Key to address location (two bytes)
        // Data to transmit (two bytes)
        //Results: NOP Response
        ss1 = 0;
        if (u8_pass == 0){
                RByte0 = spi.write(0x00);
                RByte1 = spi.write(0x2A);
                RByte2 = spi.write(0x45);
                RByte3 = spi.write(0x8F);
                RByte4 = spi.write(Addr_102A_LSB);
                RByte5 = spi.write(Addr_102A_MSB);
                RByte6 = spi.write(0xC3);
                RByte7 = spi.write(ComputeCRC(0x00, 0x2A, 0x45, 0x8F, Addr_102A_LSB,
                        Addr_102A_MSB, 0xC3));
        }
        else if (u8_pass == 1){
                RByte0 = spi.write(0x00);
                RByte1 = spi.write(0x22);
                RByte2 = spi.write(0xDD);
                RByte3 = spi.write(0xFB);
                RByte4 = spi.write(Addr_1022_LSB);
                RByte5 = spi.write(Addr_1022_MSB);
                RByte6 = spi.write(0xC3);
                RByte7 = spi.write(ComputeCRC(0x00, 0x22, 0xDD, 0xFB, Addr_1022_LSB,
                        Addr_1022_MSB, 0xC3));
        }
        else if (u8_pass == 2){
                RByte0 = spi.write(0x00);
                RByte1 = spi.write(0x24);
                RByte2 = spi.write(0xC9);
                RByte3 = spi.write(0x9F);
                RByte4 = spi.write(Addr_1024_LSB);
                RByte5 = spi.write(Addr_1024_MSB);
                RByte6 = spi.write(0xC3);
                RByte7 = spi.write(ComputeCRC(0x00, 0x24, 0xC9, 0x9F, Addr_1024_LSB,
                        Addr_1024_MSB, 0xC3));
        }
        ss1 = 1;
        wait_ms(10);

        //Define ReadChallenge message
        //Requires: Nothing
        //Results: EE Challenge
        ss1 = 0; //Transmit readchallenge message, keep result for xor operation
        RByte0 = spi.write(0x00);
        RByte1 = spi.write(0x00);
        RByte2 = spi.write(0x00);
        RByte3 = spi.write(0x00);
        RByte4 = spi.write(0x00);
        RByte5 = spi.write(0x00);
        RByte6 = spi.write(0xCF);
        RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xCF));
        ss1 = 1;
        wait_ms(10);

        //Define EEChallengeAnswer message
        //Requires: Key Echo, Inverted Key Echo
        // Key Echo to be XOR'd with 0x1234
        //Results: EEReadAnswer
        ss1 = 0; //Transmit readchallenge message, keep result for xor operation
        spi.write(0x00);
        spi.write(0x00);
        spi.write(RByte2 ^ 0x34);
        spi.write(RByte3 ^ 0x12);
        spi.write(~(RByte2 ^ 0x34));
        spi.write(~(RByte3 ^ 0x12));
        spi.write(0xC5);
        RByte7 = spi.write(ComputeCRC(0x00, 0x00, RByte2 ^ 0x34, RByte3 ^ 0x12,
                ~(RByte2 ^ 0x34), ~(RByte3 ^ 0x12), 0xC5));
        ss1 = 1;
        wait_ms(40); //delay for writing EEPROM

        //Define NOP message
        //Required to read out EEPROMWriteStatus
        //Results: Error code indicating successful write or not
        //Error codes:
        //1: Success All others: Failure
        ss1 = 0; //Transmit NOP message, keep result for check
        RByte0 = spi.write(0x00);
        RByte1 = spi.write(0x00);
        RByte2 = spi.write(0xAA);
        RByte3 = spi.write(0xAA);
        RByte4 = spi.write(0x00);
        RByte5 = spi.write(0x00);
        RByte6 = spi.write(0xD0);
        RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0xAA, 0xAA, 0x00, 0x00, 0xD0));
        ss1 = 1;
        u8_pass++;
        pc.printf("Error code (1=success): %02X\r\n", RByte0);
}
wait(10);

//Define reboot command
//Required to reset chip to use new EEPROM parameters
ss1 = 0;
RByte0 = spi.write(0x00);
RByte1 = spi.write(0x00);
RByte2 = spi.write(0x00);
RByte3 = spi.write(0x00);
RByte4 = spi.write(0x00);
RByte5 = spi.write(0x00);
RByte6 = spi.write(0xEF);
RByte7 = spi.write(ComputeCRC(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xEF));
ss1 = 1;
wait_ms(30);
pc.printf("Programming complete\r\n");

while (1){
        u8_spi_write_buffer[0] = 0x00;
        u8_spi_write_buffer[1] = 0x00;
        u8_spi_write_buffer[2] = 0xFF;
        u8_spi_write_buffer[3] = 0xFF;
        u8_spi_write_buffer[4] = 0x00;
        u8_spi_write_buffer[5] = 0x00;
        u8_spi_write_buffer[6] = 0x53;
        u8_spi_write_buffer[7] = 0x0C;

        ss1 = 0;
        for (int i = 0; i < 8; i++){
                u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
        }
```

```
            ss1 = 1;

            wait_ms(10);
            u8_spi_write_buffer[0] = 0x00;
            u8_spi_write_buffer[1] = 0x00;
            u8_spi_write_buffer[2] = 0xAA;
            u8_spi_write_buffer[3] = 0xAA;
            u8_spi_write_buffer[4] = 0x00;
            u8_spi_write_buffer[5] = 0x00;
            u8_spi_write_buffer[6] = 0xD0;
            u8_spi_write_buffer[7] = 0xAB;

            ss1 = 0;
            for (int i = 0; i < 8; i++){
                   u8_spi_read_buffer[i] = spi.write(u8_spi_write_buffer[i]);
            }
            ss1 = 1;

            //Extract and convert the alpha angle to degrees
            //remove error bits and shift to high byte
            u16_alpha_angle_lsb = (u8_spi_read_buffer[1] & 0x3F) << 8;
            //add LSb of angle
            u16_alpha_angle_lsb = u16_alpha_angle_lsb + u8_spi_read_buffer[0];
            //convert to decimal degrees
            f32_alpha_angle_degrees = u16_alpha_angle_lsb * f32_lsb_to_dec_degrees;

            //Extract and convert the beta angle to degrees
            //remove error bits and shift to high byte
            u16_beta_angle_lsb = (u8_spi_read_buffer[3] & 0x3F) << 8;
            //add LSb of angle
            u16_beta_angle_lsb = u16_beta_angle_lsb + u8_spi_read_buffer[2];
            //convert to decimal degrees
            f32_beta_angle_degrees = u16_beta_angle_lsb * f32_lsb_to_dec_degrees;

            //Extract the error bits
            u8_error_lsb = u8_spi_read_buffer[1] >> 6;

            //Extract the CRC
            u8_crc_dec = u8_spi_read_buffer[7];

            //Extract the virtual gain byte
            u8_virtualgain_dec = u8_spi_read_buffer[4];

            //Extract the rolling counter
            u8_rollcnt_dec = u8_spi_read_buffer[6] & 0x3F;

            //Send results to serial port
            pc.printf("Alpha Angle (LSb): %d\r\n", u16_alpha_angle_lsb);
            pc.printf("Alpha Angle (Dec): %2.2f\r\n", f32_alpha_angle_degrees);
            pc.printf("Beta Angle (LSb): %d\r\n", u16_beta_angle_lsb);
            pc.printf("Beta Angle (Dec): %2.2f\r\n", f32_beta_angle_degrees);
            pc.printf("Error Bits (Dec): %d\r\n", u8_error_lsb);
            pc.printf("CRC (Dec): %d\r\n", u8_crc_dec);
            pc.printf("Virtual Gain (Dec): %d\r\n", u8_virtualgain_dec);
            pc.printf("Rolling Counter (Dec): %d\r\n", u8_rollcnt_dec);
            wait(0.5);
       }
}
```

FAQ's and Troubleshooting

Q1. I'm using the dual-die package (TSSOP-16) and cannot communicate

On the dual die package the SCLK, MOSI, and MISO lines are typically connected together on a bus. If both of the slave select lines are pulled low (or left floating) bus contention can occur where multiple slaves attempt to transmit at the same time.

A general recommendation is to provide pullup resistors on the slave select lines of both dies. This ensures that the MLX90363 is only active when the slave select is actively driven by the output of the microcontroller.

Q2. I don't receive the ready message at startup. Instead I get an error message. What's wrong?

Failure to receive the ready message is usually caused by activity on the slave select line prior to transmitting a message. Check the slave select line with a scope and ensure that it is not going low during the initialization period.

Q3. What gain should we target? Can I use it to determine the magnetic field?

The magnetic field applied to the sensor should be within 20mT to 70mT. Exceeding the field strength can cause additional nonlinearity in the signal. Not meeting the field requirements will cause additional noise to be present on the sensor. Additionally, diagnostics can be set due to exceeding the max/min settings in the EEPROM.

Due to process variations the virtual gain between devices can vary even when the same magnetic field strength is applied. However, while Melexis does not recommend using the virtual gain to determine the field strength, a rough rule of thumb is that the virtual gain should be between 13 and 27 over the temperature range and movement span.

Q4. The PinFilter parameter is at an odd address. Memwrite says only even's are allowed. How do I modify the PinFilter parameter?

The memory write operation operates on words of data and requires even addresses as operands. PinFilter resides at memory address 0x1001[1:0] which is equivalent to 0x1000[9:8] or, phrased another way, the upper byte of the memory word residing at even memory address 0x1000.

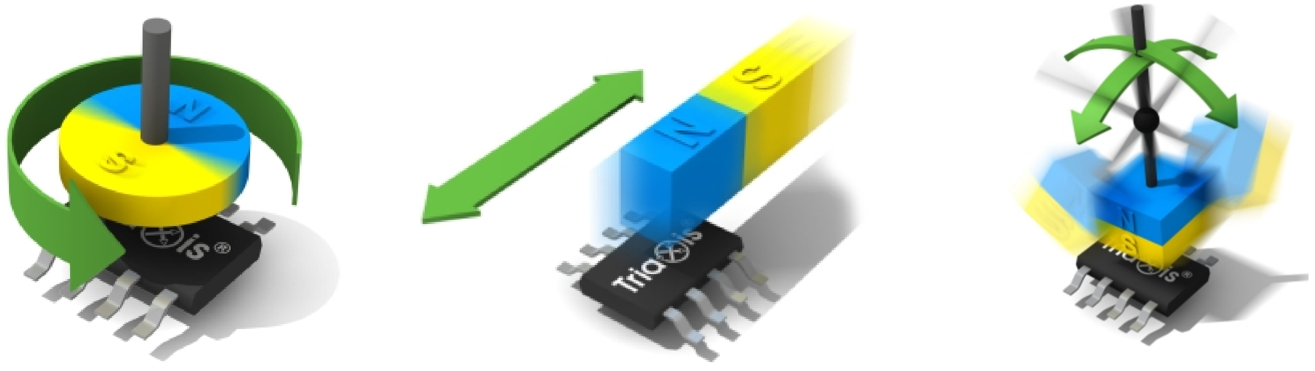
Q5. What are the logic thresholds of the part?

Voh > 65% Vdd and Vol < 35% Vdd

Q6. What is the XYZ marker used for?

The XYZ marker of the GET message allows for reading out the raw field values from the sensor. In this way the sensor can be used as a 3D magnetometer for measuring magnetic fields. The data returned by the sensor is a signed 14-bit value in two's complement form. When using this mode it is important to note that the X and Y axes have magnetic gain (≈1.4x) due to the flux concentrator in the package. This is not compensated for by the K or SMISM parameters and therefore requires external compensation in the microcontroller.

The values are also affected by the gain of the sensor therefore the gain needs to be fixed (EEPROM values GainMin = GainMax) or compensated by reading the gain value and applying a scaling factor.



For the latest version of this document, go to our website at
www.melexis.com

Or for additional information contact your regional sales office

Europe, Africa, Asia:	America:
Phone: +32 1367 0495	Phone: +1 248 306 5400
Email: sales_europe@melexis.com	Email: sales_usa@melexis.com

ISO/TS 16949 and ISO 14001 Certified