These first three pages are taken from the MPU-9250 data sheet.

See Page 4 for the start of the timing diagrams

1.2 Purpose and Scope

This document provides a description, specifications, and design related information on the MPU-9250 MotionTracking device. The device is housed in a small 3x3x1mm QFN package.

Specifications are subject to change without notice. Final specifications will be updated based upon characterization of production silicon. For references to register map and descriptions of individual registers, please refer to the MPU-9250 Register Map and Register Descriptions document.

1.3 Product Overview

MPU-9250 is a multi-chip module (MCM) consisting of two dies integrated into a single QFN package. One die houses the 3-Axis gyroscope and the 3-Axis accelerometer. The other die houses the AK8963 3-Axis magnetometer from Asahi Kasei Microdevices Corporation. Hence, the MPU-9250 is a 9-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor™ (DMP) all in a small 3x3x1mm package available as a pin-compatible upgrade from the MPU-6515. With its dedicated I^{2}C sensor bus, the MPU-9250 directly provides complete 9-axis MotionFusion™ output. The MPU-9250 MotionTracking device, with its 9-axis integration, on-chip MotionFusion™, and run-time calibration firmware, enables manufacturers to eliminate the costly and complex selection, qualification, and system level integration of discrete devices, guaranteeing optimal motion performance for consumers. MPU-9250 is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I^{2}C port.

These are the settings we will be using

MPU-9250 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs, three 16-bit ADCs for digitizing the accelerometer outputs, and three 16-bit ADCs for digitizing the magnetometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of [✓ ±250], ±500, ±1000, and ±2000°/sec (dps), a user-programmable accelerometer full-scale range of ±2*g*, [✓ ±4*g*], ±8*g*, and ±16*g*, and a magnetometer full-scale range of ±4800µT.

Other industry-leading features include programmable digital filters, a precision clock with 1% drift from -40°C to 85°C, an embedded temperature sensor, and programmable interrupts. The device features I^{2}C and SPI serial interfaces, a VDD operating range of 2.4V to 3.6V, and a separate digital IO supply, VDDIO from 1.71V to VDD.

Communication with all registers of the device is performed using either I^{2}C at 400kHz or SPI at 1MHz. For applications requiring faster communications, the sensor and interrupt registers may be read using SPI at 20MHz.

By leveraging its patented and volume-proven CMOS-MEMS fabrication platform, which integrates MEMS wafers with companion CMOS electronics through wafer-level bonding, InvenSense has driven the package size down to a footprint and thickness of 3x3x1mm, to provide a very small yet high performance low cost package. The device provides high robustness by supporting 10,000*g* shock reliability.

1.4 Applications

- Location based services, points of interest, and dead reckoning
- Handset and portable gaming
- Motion-based game controllers
- 3D remote controls for Internet connected DTVs and set top boxes, 3D mice
- Wearable sensors for health, fitness and sports

Accel_X, Y, Z
16 bit integer
$32767 \updownarrow -32768 = 4g \updownarrow -4g$

GyroX, Y, Z
16 bit integer
$32767 \updownarrow -32768 = +250°/s \updownarrow -250°/s$

9 Assembly

This section provides general guidelines for assembling InvenSense Micro Electro-Mechanical Systems (MEMS) devices packaged in quad flat no-lead package (QFN) surface mount integrated circuits.

9.1 Orientation of Axes

The diagram below shows the orientation of the axes of sensitivity and the polarity of rotation. Note the pin 1 identifier (•) in the figure.

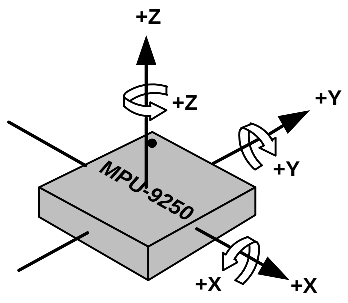


Figure 4. Orientation of Axes of Sensitivity and Polarity of Rotation for Accelerometer and Gyroscope

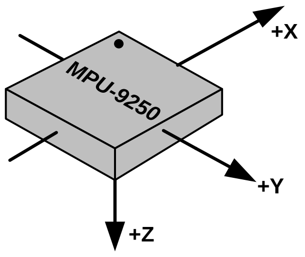


Figure 5. Orientation of Axes of Sensitivity for Compass

9.2 Package Dimensions

24 Lead QFN (3x3x1) mm NiPdAu Lead-frame finish

7.5 SPI Interface

SPI is a 4-wire synchronous serial interface that uses two control lines and two data lines. The MPU-9250 always operates as a Slave device during standard Master-Slave SPI operation.

With respect to the Master, the Serial Clock output (SCLK), the Serial Data Output (SDO) and the Serial Data Input (SDI) are shared among the Slave devices. Each SPI slave device requires its own Chip Select (CS) line from the master.

CS goes low (active) at the start of transmission and goes back high (inactive) at the end. Only one CS line is active at a time, ensuring that only one slave is selected at any given time. The CS lines of the non-selected slave devices are held high, causing their SDO lines to remain in a high-impedance (high-z) state so that they do not interfere with any active devices.

SPI Operational Features

1. Data is delivered MSB first and LSB last
2. Data is latched on the rising edge of SCLK
3. Data should be transitioned on the falling edge of SCLK
4. The maximum frequency of SCLK is 1MHz
5. SPI read and write operations are completed in 16 or more clock cycles (two or more bytes). The first byte contains the SPI Address, and the following byte(s) contain(s) the SPI data. The first bit of the first byte contains the Read/Write bit and indicates the Read (1) or Write (0) operation. The following 7 bits contain the Register Address. In cases of multiple-byte Read/Writes, data is two or more bytes:

SPI Address format

MSB							**LSB**
R/W	A6	A5	A4	A3	A2	A1	A0

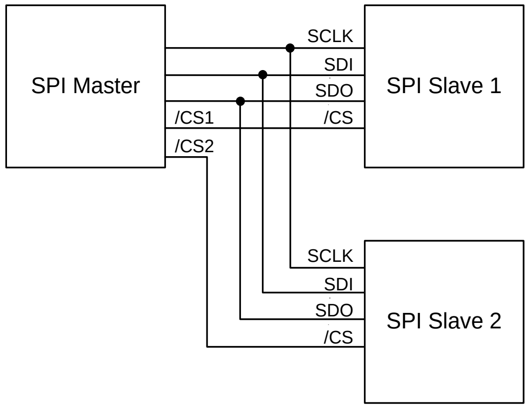
Read/Write bit

1 = Read
0 = Write

SPI Data format

MSB							**LSB**
D7	D6	D5	D4	D3	D2	D1	D0

6. Supports Single or Burst Read/Writes.



Typical SPI Master / Slave Configuration

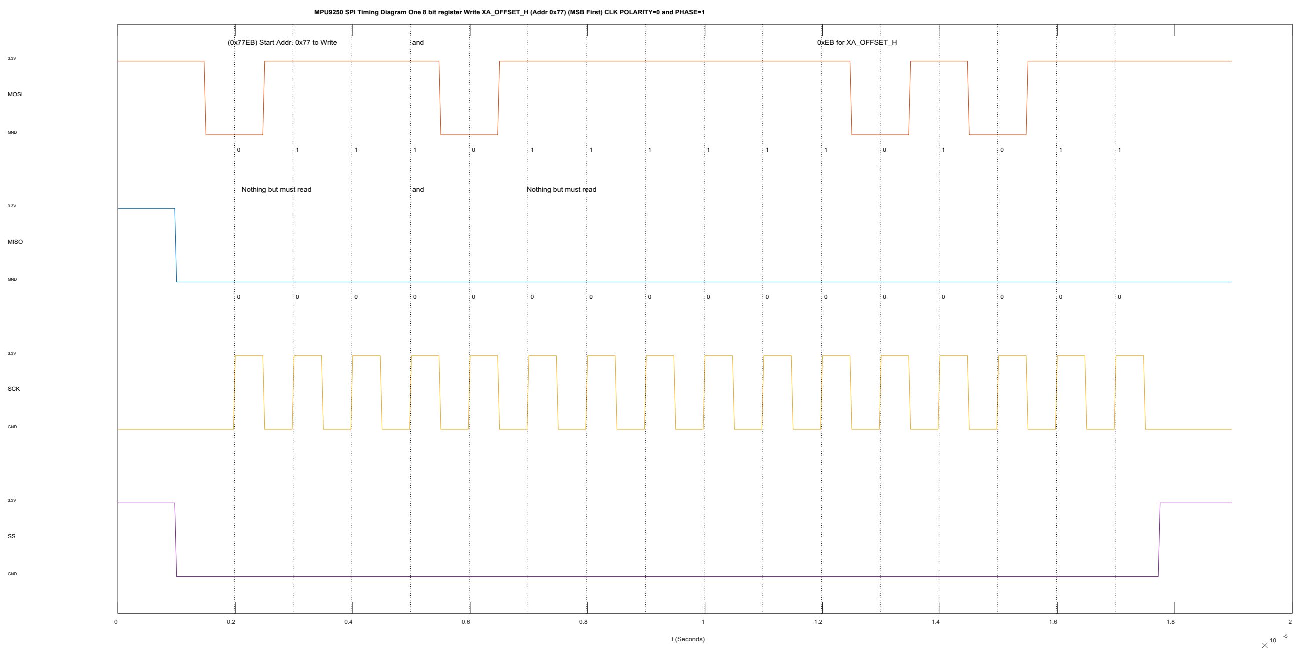
MPU-9250 SPI Timing Diagrams and Programming Tips

SPI Interface, Timing Diagrams for Write and Read.

This is an addendum to the MPU-9250's Datasheet and Register Map documents. The MPU-9250 Datasheet has a very brief discussion of how to use SPI to communicate with the registers of the MPU-9250. After debugging the SPI interface with the MPU-9250, I understand why they thought it did not merit a large discussion, because the protocol is very straight forward. But for a person first programming the MPU-9250, like myself years ago, it seems to be a daunting task due to the large number of programmable registers in the MPU-9250. So hopefully this SPI guide will help you understand the SPI interface to the MPU-9250.

First we will start with writing to a single register in the MPU-9250. The minimum number of bits needed to either write or read to/from the MPU-9250 is 16 bits.

Example Writing to One Register



To understand the SPI write protocol study the timing diagram above and read through the below steps. *Open the MPU-9250 Register Map and find that the register to set the MSB of the X accelerometer's offset (XA_OFFSET_H) is located at the address 0x77. Note that this example writing to XA_OFFSET_H is the same for writing to any register inside the MPU-9250.*

1. SPIB needs to be setup in its initializations in main() to transmit 16 bits at a time. `SpibRegs.SPICCR.bit.SPICHAR = 0xF;` Each write to the TX FIFO through `SpibRegs.SPITXBUF` will transmit 16 bits.
2. SS\ must be pulled low by the SPI Master.
3. 0x77EB is written to `SpibRegs.SPITXBUF = 0x77EB;` The 0x77 is the register's address and the 0xEB is the data to be written to the register. This data is transmitted to the MPU-9250 on the MOSI pin (see above). *If you needed to write a different value to XA_OFFSET_H the 0x77 would remain the same and only the 0xEB would change to your new value.*
4. After this write to the TX FIFO, wait for the transmission to complete which is the same thing as waiting for one 16 bit value to be received and placed in the RX FIFO. If this write is performed in main(), or a function called in main(), you would need

to wait in a while loop until one value is in the RX FIFO `while(SpibRegs.SPIFFRX.bit.RXFFST !=1);` If outside of main() you would wait for the SPIB RX interrupt to happen.

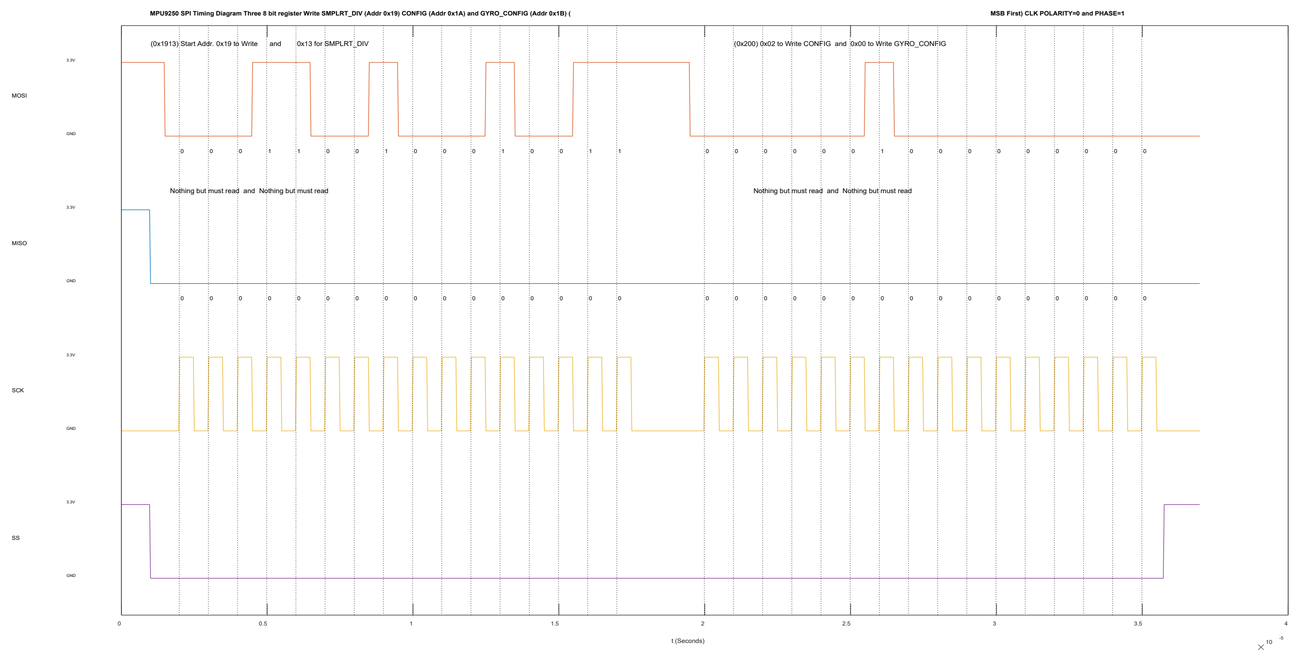
5. During the transmission of 0x77EB on MOSI, data is received on the MISO pin even though the timing diagram above shows that "Nothing" (zeros) is sent back from the slave (MPU-9250). So by the end of transmitting the 16 bits, there will be one 16 bit "garbage" value on the RX FIFO that you will need to read to clear the RX FIFO.
6. SS\ must be pulled high by the SPI Master.

```
// Code
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x7700 | 0x00EB);
while(SpibRegs.SPIFFRX.bit.RXFFST !=1);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;  // Read value even though not needed

DELAY_US(10);  // Delay a bit before issuing another MPU-9250 SPI transfer
```

Example Writing to Three (or Five or Seven, etc.) Registers

Note: with this method of communicating to the MPU-9250 16 bits at a time there is no way to send values an even number of registers. So if you wanted to write to 10 registers, you would use this method to write to 9 registers and then use the method above to write to just one register.


MPU9250 SPI Timing Diagram Three 8 bit register Write SMPLRT_DIV (Addr 0x19) CONFIG (Addr 0x1A) and GYRO_CONFIG (Addr 0x1B) (
MSB First) CLK POLARITY=0 and PHASE=1
(0x1913) Start Addr. 0x19 to Write and 0x13 for SMPLRT_DIV
(0x200) 0x02 to Write CONFIG and 0x00 to Write GYRO_CONFIG
Nothing but must read and Nothing but must read
Nothing but must read and Nothing but must read
MOSI
MISO
SCK
SS
3.3V
GND
t (Seconds)
× 10^-5

To understand the SPI write protocol of multiple 16 bit values, study the timing diagram above and read through the below steps. *Open the MPU-9250 Register Map and find that the register (SMPLRT_DIV) is located at the address 0x19 and the registers CONFIG and GYRO_CONFIG are next in line at 0x1A and 0x1B.*

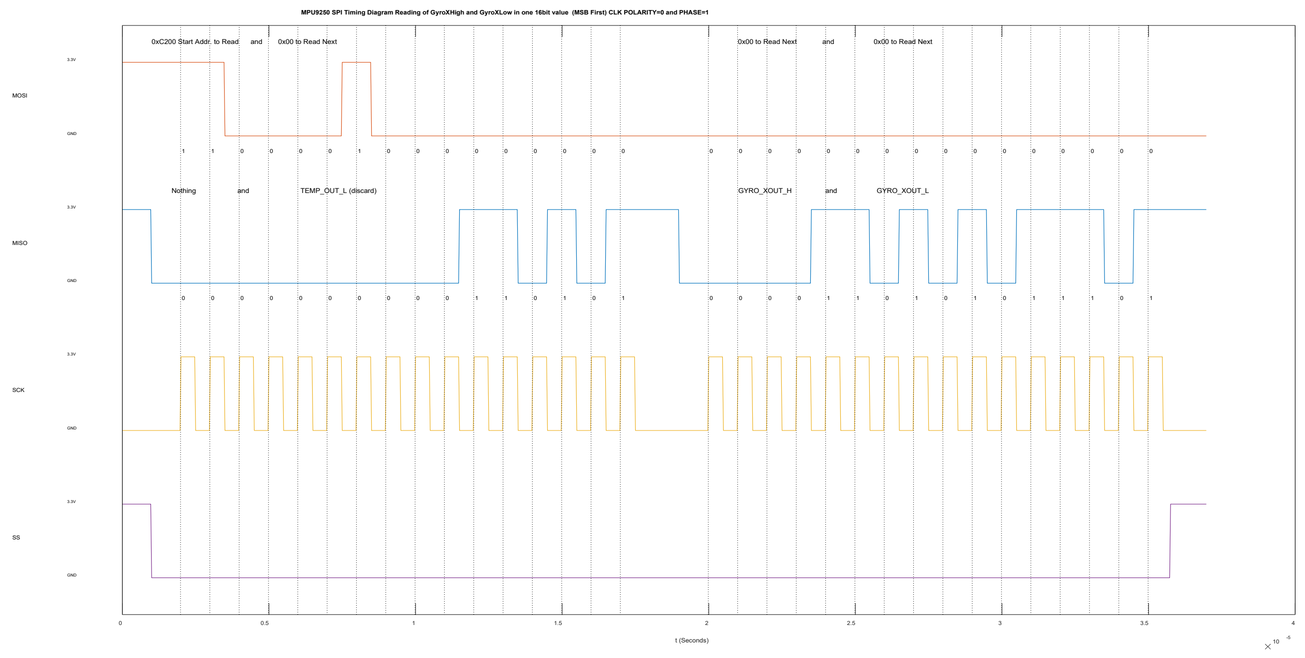
1. SPIB needs to be setup in its initializations in main() to transmit 16 bits at a time. `SpibRegs.SPICCR.bit.SPICHAR = 0xF;` Each write to the TX FIFO through `SpibRegs.SPITXBUF` will transmit 16 bits.
2. SS\ must be pulled low by the SPI Master.

3. 0x1913 then 0x0200 is written to SpibRegs.SPITXBUF one right after the other. SpibRegs.SPITXBUF = 0x1913; SpibRegs.SPITXBUF = 0x200; The 0x19 is the register's address and the 0x13 is the data to be written to the SMPLRT_DIV register. Of the second 16 bits sent, the upper 0x02 is written to the CONFIG (0x1A) register and lower 0x00 is written to the GYRO_CONFIG (0x1B) register. If you wanted to also write to the next two registers 0x1C and 0x1D you could write a third 16 bits with the upper 8 bits written to the register at 0x1C and the low 8 bits written to the register at 0x1D. A fourth 16 bits be written if you want to write to the next two registers and a fifth for the next two, etc.
4. After these writes to the TX FIFO, wait for the transmissions to complete which is the same thing as waiting for the same number of receives to complete. If this write is performed in main(), or a function called in main(), wait in a while loop until the same number of 16 bit values transmitted are in the RX FIFO while(SpibRegs.SPIFFRX.bit.RXFFST !=2 (or 3 or 4, etc)); If outside of main() you would wait for the SPIB RX interrupt to happen.
5. During the transmission of 2 16 bit values (or 3 or 4, etc.) on MOSI, data is received on the MISO pin even though the timing diagram above shows that "Nothing" (zeros) is sent back from the slave (MPU-9250). So by the end of transmitting the multiple 16 bits, there will be the same number of 16 bit "garbage" values on the RX FIFO that you will need to read to clear the RX FIFO.
6. SS\ must be pulled high by the SPI Master.

```
// Code
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPITXBUF = (0x1900 | 0x0013);
SpibRegs.SPITXBUF = (0x0200 | 0x0000);
while(SpibRegs.SPIFFRX.bit.RXFFST !=2);
GpioDataRegs.GPCSET.bit.GPIO66 = 1;
temp = SpibRegs.SPIRXBUF;  // Read value even though not needed
temp = SpibRegs.SPIRXBUF;  // Read second value even though not needed

DELAY_US(10);  // Delay a bit before issuing another MPU-9250 SPI transfer
```

Example Reading from one 16 bit sensor pair (8bit MSB then 8bit LSB) Registers


MPU9250 SPI Timing Diagram Reading of GyroXHigh and GyroXLow in one 16bit value (MSB First) CLK POLARITY=0 and PHASE=1
0xC200 Start Addr. to Read and 0x00 to Read Next
0x00 to Read Next and 0x00 to Read Next
MOSI
3.3V
GND
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Nothing and TEMP_OUT_L (discard)
GYRO_XOUT_H and GYRO_XOUT_L
MISO
0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1
0 0 0 0 1 1 0 1 0 1 0 1 1 1 0 1
SCK
SS
0
0.5
1
1.5
2
2.5
3
3.5
4
t (Seconds)
× 10^-5

Most, if not all, the SPI readings you will need to perform with the MPU-9250 are readings of the three gyro values the three accelerometer and the three magnetometer readings. Notice that each of these sensor readings are a 16 bit value that are made up of an 8 bit H (High, MSB) byte and an 8 bit L (Low, LSB) byte. I am calling this a sensor pair. What would be perfect is if when a 16 bit value is read from the MPU-9250 it has the full 16 bit reading of the sensor so that no post processing of combining two 8 bits is necessary. To do this the protocol reads one 8 bit register that is not needed and discarded.

To understand the SPI read protocol, study the timing diagram above and read through the below steps. *Open the MPU-9250 Register Map and find that the register (TEMP_OUT_L) is located at the address 0x42 and the registers GYRO_XOUT_H and GYRO_XOUT_L are next in line at 0x43 and 0x44.*

1. SPIB needs to be setup in its initializations in main() to transmit 16 bits at a time. `SpibRegs.SPICCR.bit.SPICHAR = 0xF;` Each write to the TX FIFO through `SpibRegs.SPITXBUF` will transmit 16 bits.
2. SS\ must be pulled low by the SPI Master.
3. 0xC200 then 0x0000 are written to `SpibRegs.SPITXBUF` one right after the other. `SpibRegs.SPITXBUF = 0xC200; SpibRegs.SPITXBUF = 0x0;` Take a look at the MPU-9250's register map and notice that the highest register address is 0x7E. The MPU-9250 uses 7 bits to specify what address is being written to or read from. The most significant eighth bit is use to tell the MPU-9250 if the command is a write command (eighth bit = 0) or a read command (eighth bit = 1). This is why this command first transmits 0xC2 (0x80 + 0x42) requesting a read starting at register 0x42. Finishing the command 0xC200 written to TXBUF, the bottom 0x00 is just anything you wish to send because something must be sent to receive bits back from the slave. The second 16 bit value written to SPITXBUF is just 0x0. Again this could be anything as the MPU-9250 ignores this value but it must be sent in order that another 16 bits can be read form the MPU-9250.
4. While these two values, 0xC2 and 0x0, are being transmitted to the MPU-9250, the MPU-9250 sees that the first bit sent is a 1 so it knows this is a read command. While the SPI transmits the 0xC2 value, nothing (all zeros) is sent back on the MISO pin. But when the remaining 0x00 of the 0xC200 command is sent, the value in register 0x42 is transmitted back to the SPI master.

The second 16 bit value written is 0x0000. While this value is being transmitted the full value of GYRO_X (both its High byte and Low byte) is read by the SPI Master.

5. After these writes to the TX FIFO, your program must wait for the transmissions to complete which is the same thing as waiting for the same number of receives and those 16 bit values placed in the RX FIFO. Set RXFFIL to 2 and when two 16 bit values have been received the SPIB RX interrupt is called.
6. Inside the SPIB RX interrupt function, read both 16bit values from the RX FIFO. The first value is discarded because it only has half of the temperature's 16 bit value. The second value you read from the RX FIFO has the full 16 bit reading from the X axis gyro.
7. SS\ must be pulled high by the SPI Master.

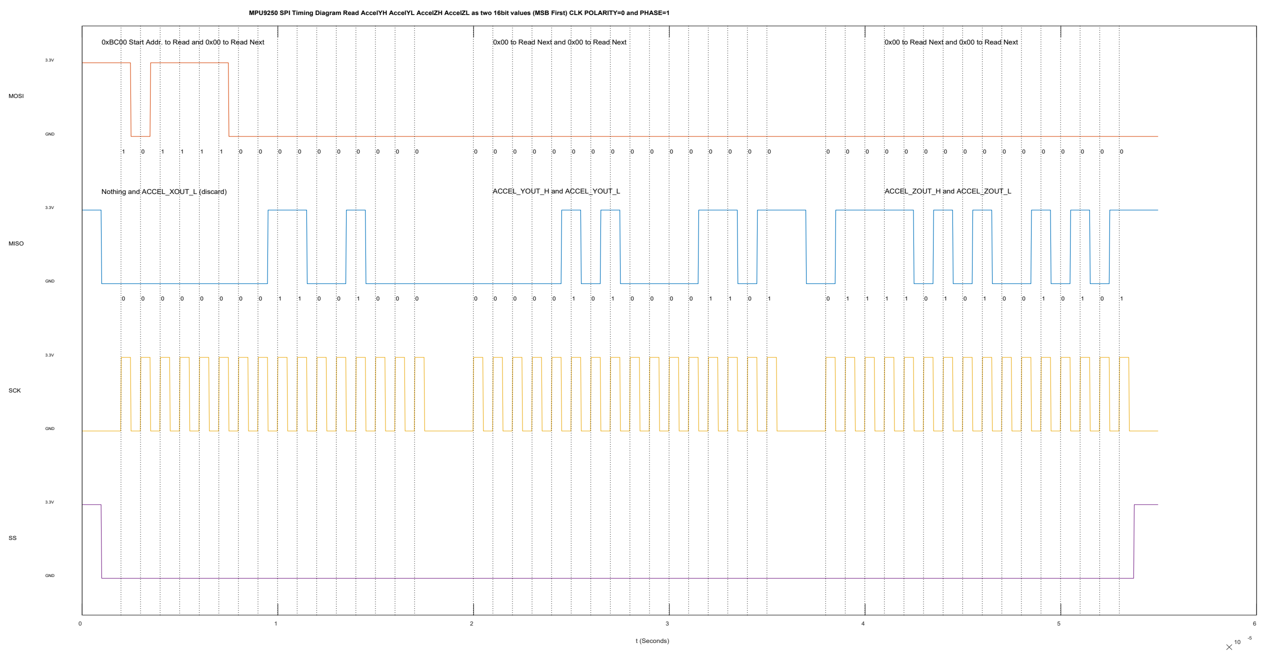
```
// Code inside CPU Timer 0
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPIFFRX.bit.RXFFIL = 2;
SpibRegs.SPITXBUF = ((0x8000) | (0x4200));
SpibRegs.SPITXBUF = 0;
```

```
// Code inside SPIB Interrupt Service Routine
int16_t dummy= 0;
int16_t gyroXraw = 0;
float gyroXreading = 0;
__interrupt void SPIB_isr(void){
   GpioDataRegs.GPCSET.bit.GPIO66 = 1;
   dummy = SpibRegs.SPIRXBUF;
   gyroXraw = SpibRegs.SPIRXBUF;

   gyroXreading  = gyroXraw*250.0/32767.0;

   SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1;  // Clear Overflow flag
   SpibRegs.SPIFFRX.bit.RXFFINTCLR=1;  // Clear Interrupt flag
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;
}
```

Example Reading from two (three, four, etc.) 16 bit sensor pairs (8bit MSB then 8bit LSB) Registers



This example continues the previous example that just read one 16 bit sensor reading. See this previous example for a bit more explanation. This example reads two 16 bit sensor readings but after reading this explanation you should be able to see how additional readings could be read by sending more 0x0000 commands to continue the transmission. This example reads two sensor pairs that make up ACCEL_Y and ACCEL_Z each 16 bit values.

To understand the SPI read protocol, study the timing diagram above and read through the below steps. *Open the MPU-9250 Register Map and find that the register (ACCEL_XOUT_L) is located at the address 0x3C and the registers ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL_ZOUT_H, and ACCEL_ZOUT_L are next in line at 0x3D, 0x3E, 0x3F and 0x40.*

1. SPIB needs to be setup in its initializations in main() to transmit 16 bits at a time. `SpibRegs.SPICCR.bit.SPICHAR = 0xF;` Each write to the TX FIFO through `SpibRegs.SPITXBUF` will transmit 16 bits.
2. SS\ must be pulled low by the SPI Master.
3. 0xBC00 then 0x0000 then 0x0000 are written to `SpibRegs.SPITXBUF` one right after the other. `SpibRegs.SPITXBUF = 0xBC00; SpibRegs.SPITXBUF = 0x0; SpibRegs.SPITXBUF = 0x0;` Writing 0xBC00 tells the MPU-9250 to start transmitting the value at address 0x3C to the SPI Master on the MISO pin. While 0xBC is being transmitted to the MPU-9250 nothing (zeros) is sent back to the master but when the second 8 zeros are sent to the MPU-9250 it sends the value stored at address 0x3C. Then when the next 16 zeros are sent, the value stored at 0x3D and then the value stored at 0x3E are sent to the SPI Master. When the second 16 zeros are sent, the values stored at 0x3F and 0x40 are sent to the SPI Master. For this example that is all the transmissions but if you did want to read the values at 0x41 and 0x42 you could write out another 16 zeros and continue this until you have read all the needed registers.
4. This example had the SPI Master send three 16 bit values to the MPU-9250. So in return the MPU-9250 will transmit back to the master three 16 bit values. The first 16 bit value has no needed information so it should be discarded. The second value is the 16 bit ACCEL_Y reading and the third value is the ACCEL_Z reading.

5. After these writes to the TX FIFO, your program must wait for the transmissions to complete which is the same thing as waiting for the same number of receives and those 16 bit values placed in the RX FIFO. Set RXFFIL to 3 and when three 16 bit values have been received the SPIB RX interrupt is called.
6. Inside the SPIB RX interrupt read the three 16bit values from the RX FIFO. The first value is discarded because it only has half of the temperature's 16 bit value. The second value read from the RX FIFO has the full 16 bit reading from the Y axis accelerometer. The third value read from the RX FIFO has the full 16 bit reading from the Z axis accelerometer.
7. SS\ must be pulled high by the SPI Master.

```
// Code inside CPU Timer 0
GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;
SpibRegs.SPIFFRX.bit.RXFFIL = 3;
SpibRegs.SPITXBUF = ((0x8000) | (0x3C00));
SpibRegs.SPITXBUF = 0;
SpibRegs.SPITXBUF = 0;

// Code inside SPIB Interrupt Service Routine
int16_t dummy= 0;
int16_t accelYraw = 0;
int16_t accelZraw = 0;
float accelYreading = 0;
float accelZreading = 0;
__interrupt void SPIB_isr(void){
   GpioDataRegs.GPCSET.bit.GPIO66 = 1;
   dummy = SpibRegs.SPIRXBUF;
   accelYraw = SpibRegs.SPIRXBUF;
   accelZraw = SpibRegs.SPIRXBUF;

   accelYreading = accelYraw*4.0/32767.0;
   accelZreading = accelZraw*4.0/32767.0;

   SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1;  // Clear Overflow flag
   SpibRegs.SPIFFRX.bit.RXFFINTCLR=1;  // Clear Interrupt flag
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;
}
```