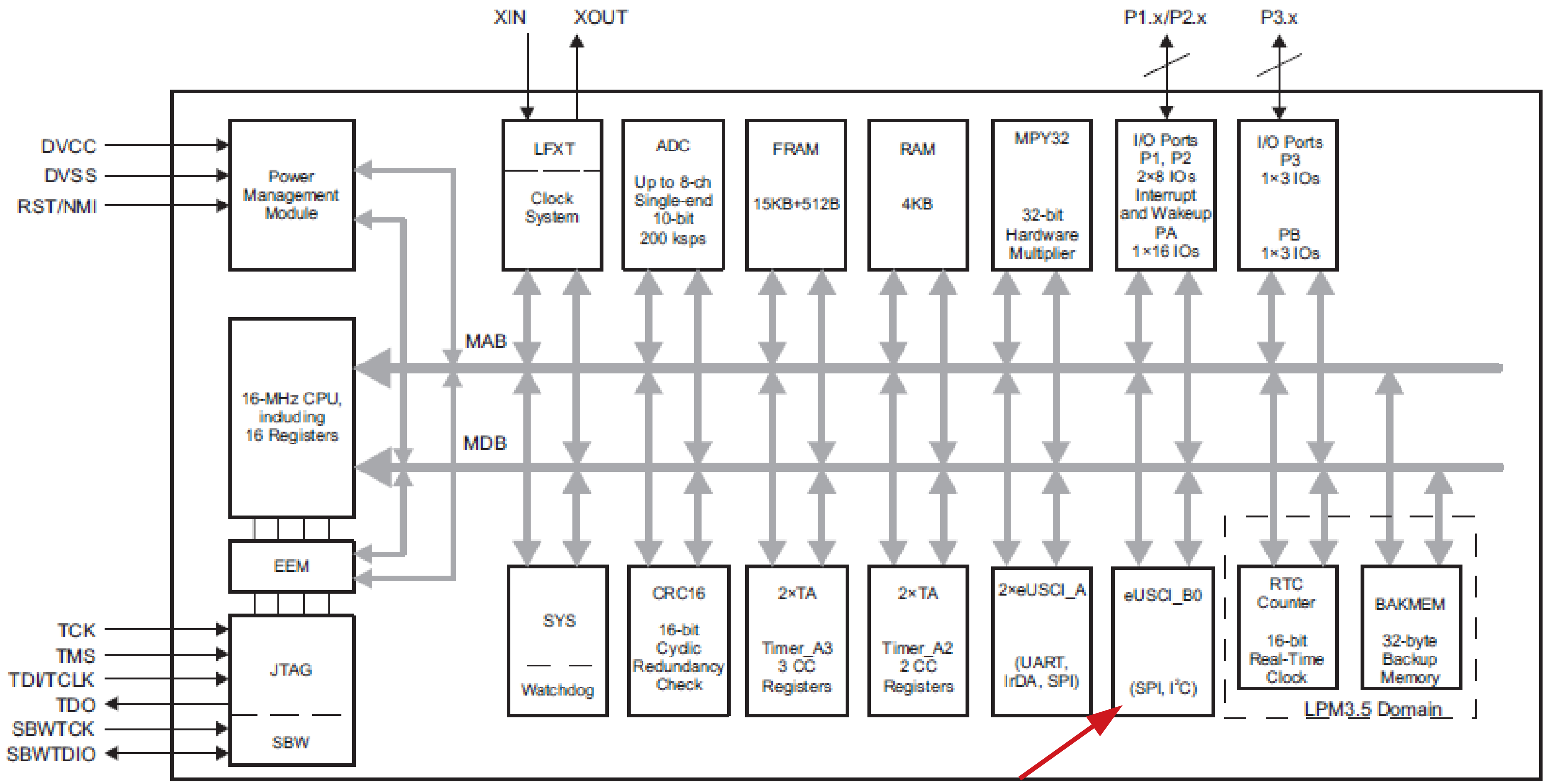
SPI – Universal Serial Communication Interface SPI Mode



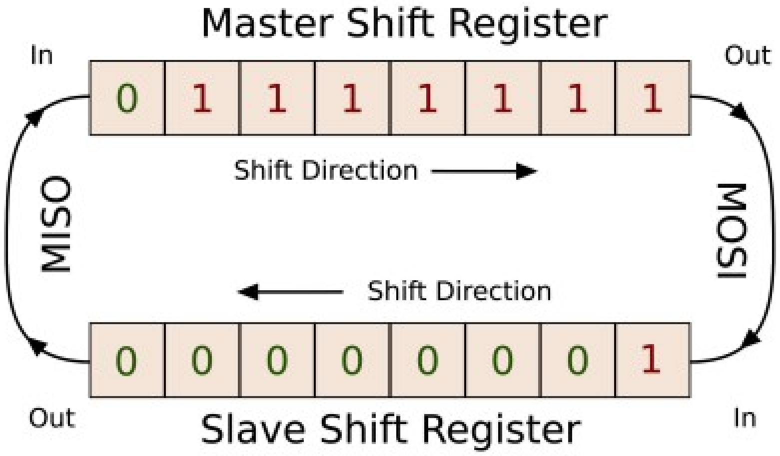
XIN
XOUT
P1.x/P2.x
P3.x
DVCC
DVSS
RST/NMI
Power Management Module
LFXT
Clock System
ADC
Up to 8-ch Single-end 10-bit 200 ksps
FRAM
15KB+512B
RAM
4KB
MPY32
32-bit Hardware Multiplier
I/O Ports P1, P2 2×8 IOs Interrupt and Wakeup PA 1×16 IOs
I/O Ports P3 1×3 IOs
PB 1×3 IOs
MAB
MDB
16-MHz CPU, including 16 Registers
EEM
TCK
TMS
TDI/TCLK
TDO
SBWTCK
SBWTDIO
JTAG
SBW
SYS
Watchdog
CRC16
16-bit Cyclic Redundancy Check
2×TA
Timer_A3 3 CC Registers
2×TA
Timer_A2 2 CC Registers
2×eUSCI_A
(UART, IrDA, SPI)
eUSCI_B0
(SPI, I²C)
RTC Counter
16-bit Real-Time Clock
BAKMEM
32-byte Backup Memory
LPM3.5 Domain

Serial Peripheral Interface (SPI) is not really a protocol, but more of a general idea. It’s the bare-minimum way to transfer a lot of data between two chips as quickly as possible,

WHAT IS SPI?
The core idea of SPI is that each device has a shift-register that it can use to send or receive a byte of data.

These two shift registers are connected together in a ring, the output of one going to the input of the other and vice-versa.

One device, the master,controls the common clock signal that makes sure that each register shifts one bit in just exactly as the other is shifting one bit out (and vice-versa). It’s hard to get simpler than that.

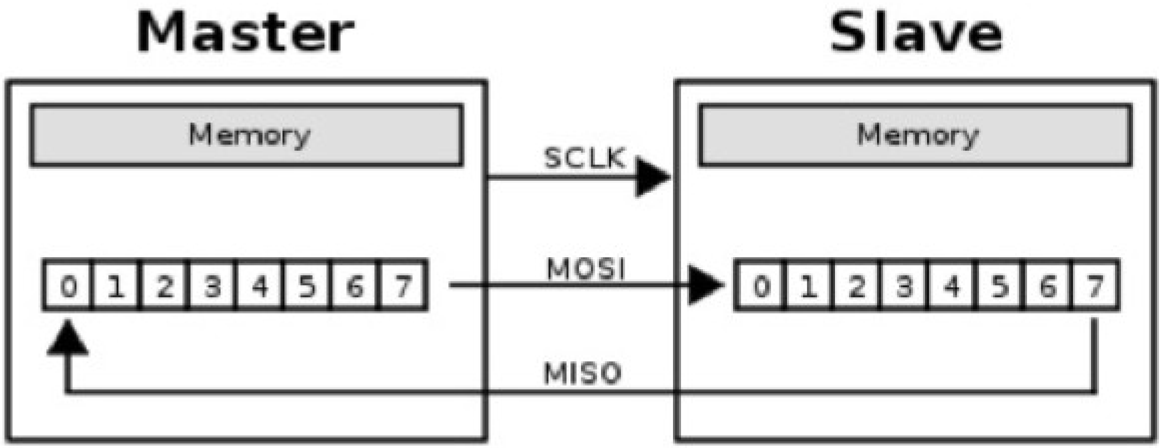


https://hackaday.com/2016/07/01/what-could-go-wrong-spi/

It’s this simplicity that makes SPI fast. While asynchronous serial communications can run in the hundred-of-thousands of bits per second, SPI is usually good for ten megabits per second or more.

You often see asynchronous serial between man and machine, because people are fairly slow. But between machine and machine, it’s going to be SPI or I2C.

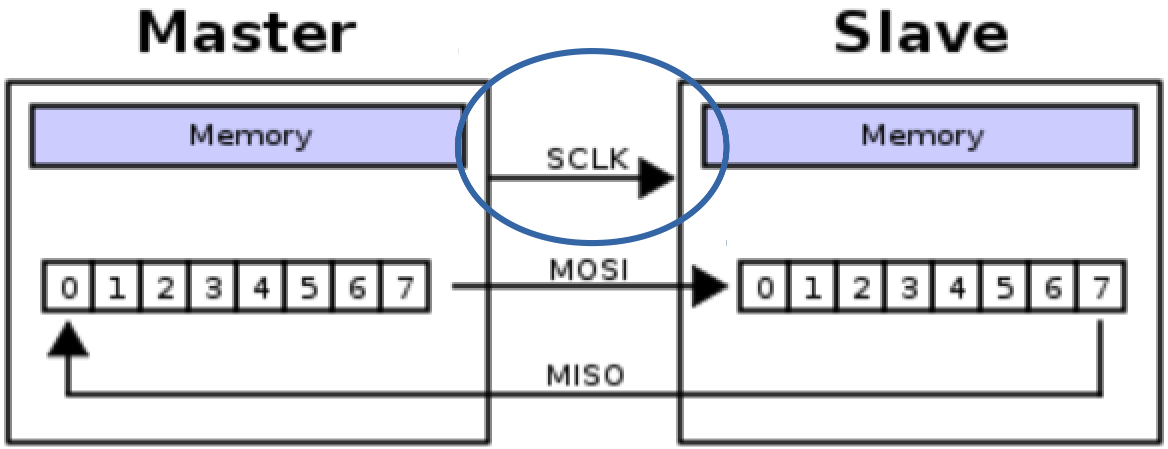
Turning this pair of shift registers into a full-blown data bus involves a couple more wires



SPI is used to talk to a variety of peripherals, such as

- Sensors: temperature, pressure, ADC, touchscreens, video game controllers
- Control devices: audio codecs, digital potentiometers, DAC
- Camera lenses: Canon EF lens mount
- Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11, handheld video games
- Memory: flash and EEPROM
- Real-time clocks
- LCD, sometimes even for managing image data
- Any MMC or SD card (including SDIO variant[6])

For high-performance systems, FPGAs sometimes use SPI to interface as a slave to a host, as a master to sensors, or for flash memory used to bootstrap if they are SRAM-based.



A typical hardware setup using two shift registers to form an inter-chip circular buffer

To begin communication, the master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. The master then selects the slave device with a logic level 0 on the chip select line. If a waiting period is required, such as for an analog-to-digital conversion, the master must wait for at least that period of time before issuing clock cycles.

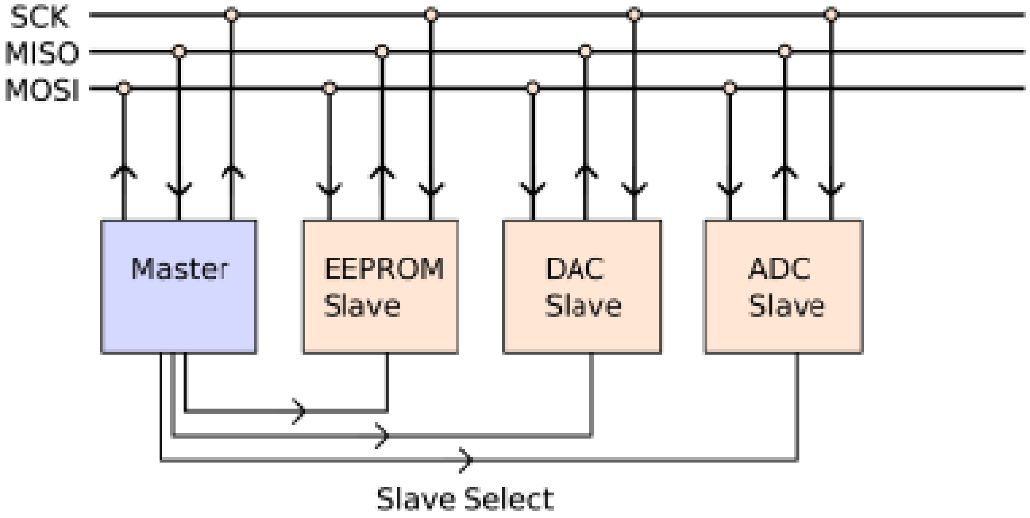
During each SPI clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended.

The master controls the clock (CLK or SCK) line, that's shared among all of the devices on the bus. Instead of a simple ring as drawn above, the master's shift register is effectively in a ring with each of the slave devices, and the lines making up this ring are labelled MISO ("master-in, slave-out") and MOSI ("master-out, slave-in") depending on the direction of data flow.

Since all of the rings are shared, each slave has an additional dedicated line that tells it when to attach and detach from the bus.

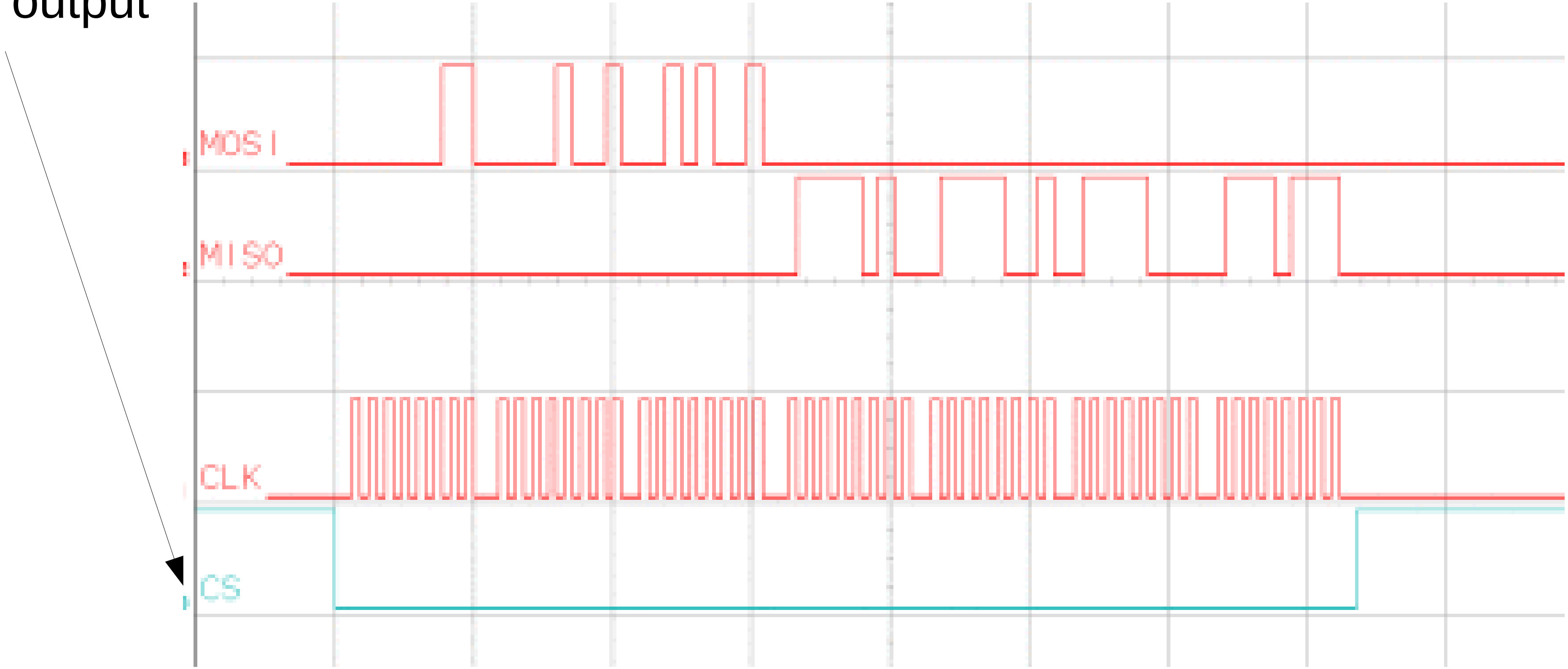
Each slave has a slave-select (SS or sometimes called chip-select CS) line, and when it's high, the slave disconnects its MISO line, and ignores what comes in over MOSI.

When the individual SS line is pulled low, the slave engages. Note that the master is responsible for keeping one and only one SS lineactive low at any given time.

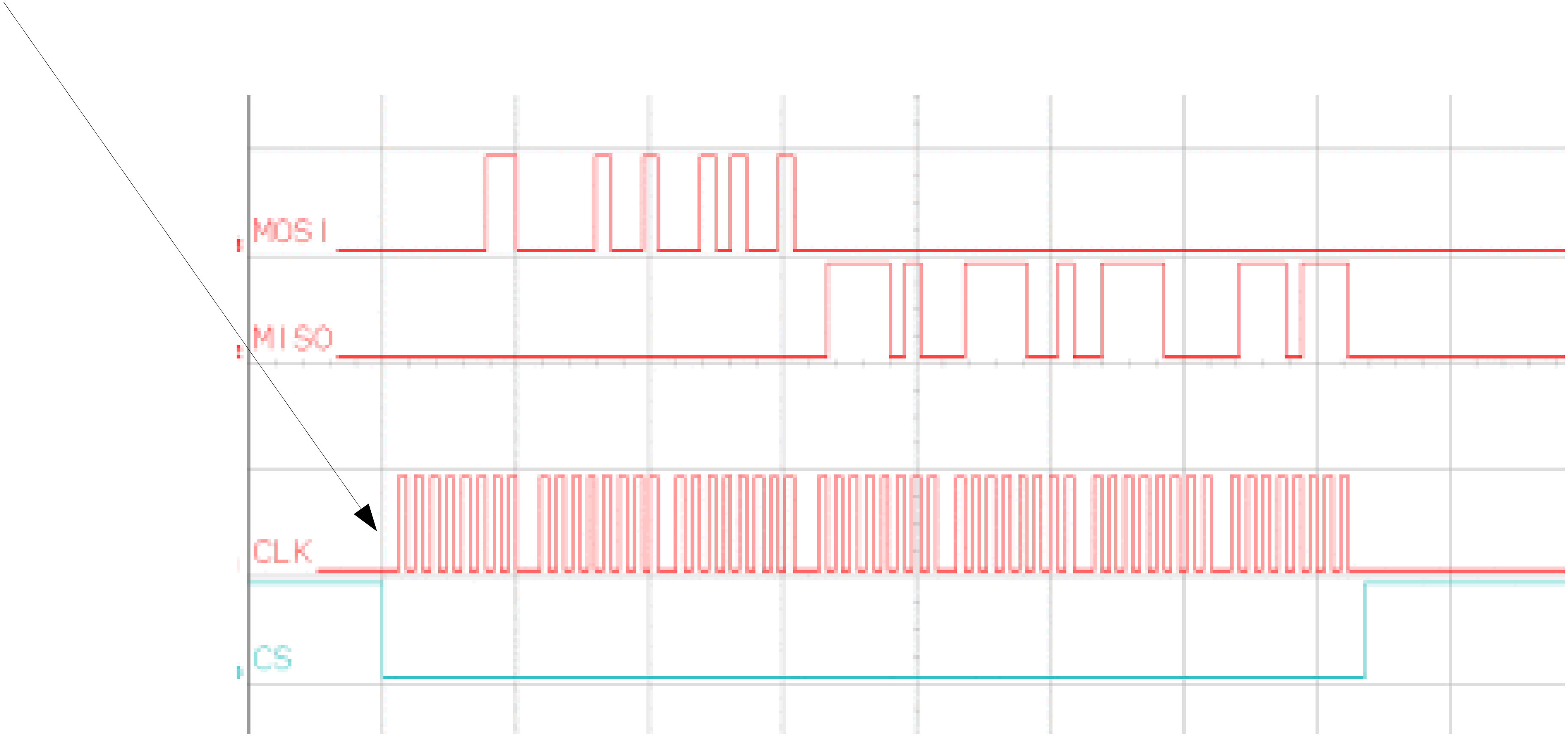


Typical SPI Communication:

1. The master pulls the slave's personal slave-select line low, at which point the slave wakes up, starts listening, and connects to the MISO line. Depending on the phase both chips may also set up their first bit of output



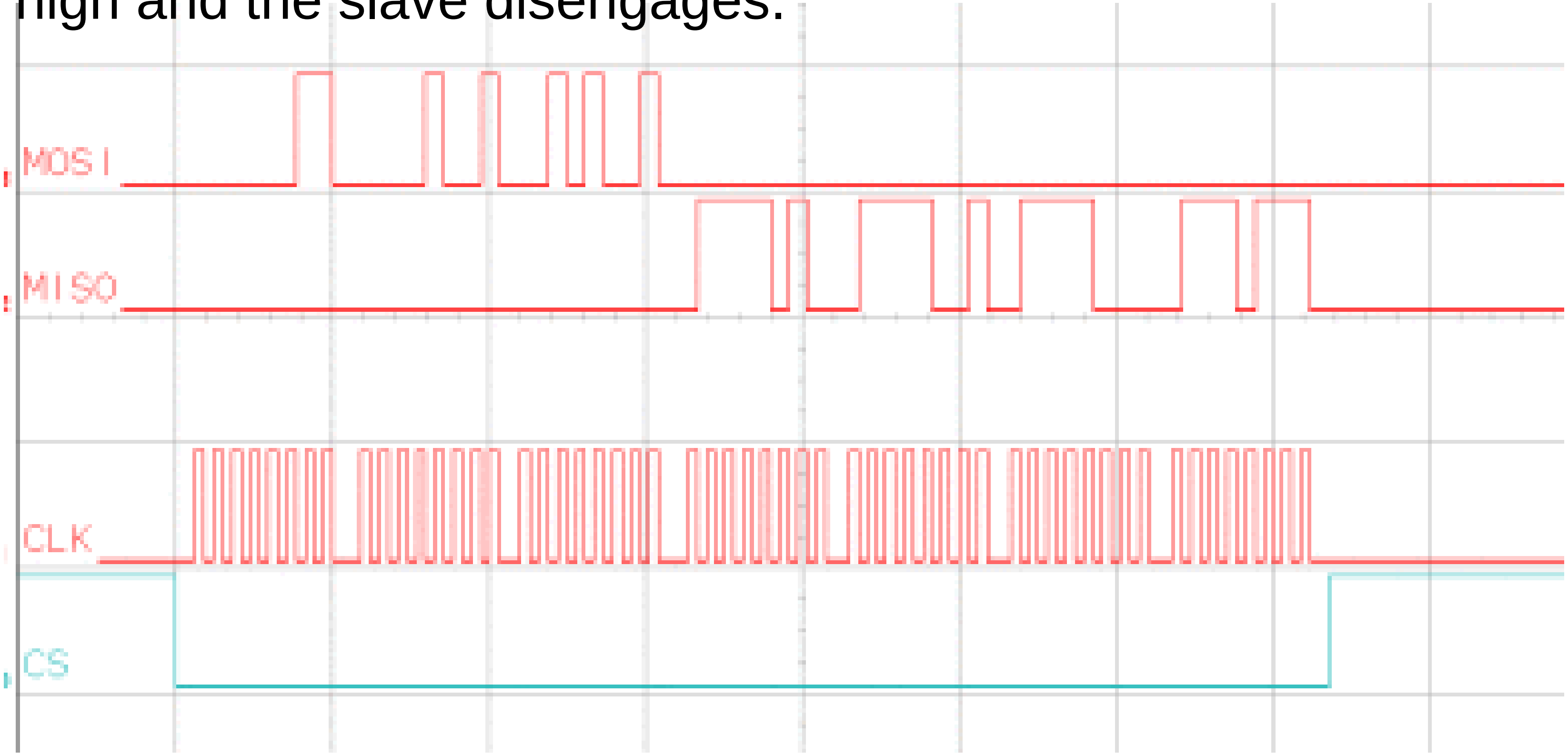
2. The master sends the first clock pulse and the first bit of data moves from master to slave (along MOSI) and from slave to master (along MISO).



3. The master keeps cycling the clock, bits are traded, and after eight bits, both sides read in the received data and queue up the next byte for transmission.

3. The master keeps cycling the clock, bits are traded, and after eight bits, both sides read in the received data and queue up the next byte for transmission.

4. After a number of bytes are traded this way, the master again drives the SS line high and the slave disengages.



SPEED
Because SPI is clocked, and the slave-select line delimits a conversation, there’s not much that can go wrong in syncronize the two devices.

Not much, except when the master talks too fast for the slave to follow. The good news? This is easy to debug.

For debugging purposes, there’s nothing to lose by going slow. Nearly every chip that can handle SPI data at 10 MHz can handle it at 100 kHz as well.

On the other hand, due to all sorts of real-world issues with voltages propagating from one side of a wire to another and the chip’s ability to push current into the wire to overcome its parasitic capacitance, the maximum speed at which your system can run is variable.

For really high SPI speeds (say, 10 MHz and above?) your system design may be the limiting factor.

Find Arduino code:

Convert Loop to Timer functions

Replace calls to library which is not interrupt able or locally coded

Use RT_ADC3 as work horse – 10 interrupts per second – replace Loop:

Analyze library functions used

Original Arduino Code – library functions used

```
//Add the SPI library so we can communicate with the ADXL345 sensor
//  Step 1
// Original Code for Arduino - will compile in Arduino or Energia IDE
// works and tested on Arduino UNO R3  HW

#include <SPI.h>

//https://www.sparkfun.com/tutorials/240
//http://forum.arduino.cc/index.php/topic,159313.0.html

//Assign the Chip Select signal to pin 8.
int CS=8;

//This is a list of some of the registers available on the ADXL345.
//To learn more about these and the rest of the registers on the ADXL345, read the datasheet!
char POWER_CTL = 0x2D;  //Power Control Register
char DATA_FORMAT = 0x31;
char DATAX0 = 0x32; //X-Axis Data 0
char DATAX1 = 0x33; //X-Axis Data 1
char DATAY0 = 0x34; //Y-Axis Data 0
char DATAY1 = 0x35; //Y-Axis Data 1
char DATAZ0 = 0x36; //Z-Axis Data 0
char DATAZ1 = 0x37; //Z-Axis Data 1

//This buffer will hold values read from the ADXL345 registers.
unsigned char values[10];
//These variables will be used to hold the x,y and z axis accelerometer values.
int x,y,z;
```

SetUp

```
void setup(){
  //Initiate an SPI communication instance.
  SPI.begin();
  //Configure the SPI connection for the ADXL345.
  SPI.setDataMode(SPI_MODE3);
  //Create a serial connection to display the data on the terminal.
  Serial.begin(9600);

  //Set up the Chip Select pin to be an output from the Arduino.
  pinMode(CS, OUTPUT);
  //Before communication starts, the Chip Select pin needs to be set high.
  digitalWrite(CS, HIGH);

  //Put the ADXL345 into +/- 4G range by writing the value 0x01 to the DATA_FORMAT register.
  writeRegister(DATA_FORMAT, 0x01);
  //Put the ADXL345 into Measurement Mode by writing 0x08 to the POWER_CTL register.
  writeRegister(POWER_CTL, 0x08);  //Measurement mode
}
```

SPI functions used:
- begin()
- setDataMode()

GPIO pin used for Chip Select
- pinMode(CS, OUTPUT)
- digitalWrite(CS, HIGH/LOW)

Replace loop: with TimerA0_A3 CCR0 interrupts (10/second)

```
void loop(){
  //Reading 6 bytes of data starting at register DATAX0 will retrieve the
  // x,y and z acceleration values from the ADXL345.
  //The results of the read operation will get stored to the values[] buffer.
  readRegister(DATAX0, 6, values);

  //The ADXL345 gives 10-bit acceleration values, but they are stored
  // as bytes (8-bits). To get the full value, two bytes must be combined for each axis.
  //The X value is stored in values[0] and values[1].
  x = ((int)values[1]<<8)|(int)values[0];
  //The Y value is stored in values[2] and values[3].
  y = ((int)values[3]<<8)|(int)values[2];
  //The Z value is stored in values[4] and values[5].
  z = ((int)values[5]<<8)|(int)values[4];

  //Print the results to the terminal.
  Serial.print(x, DEC);
  Serial.print(',');
  Serial.print(y, DEC);
  Serial.print(',');
  Serial.println(z, DEC);
  delay(10);
}
```

Serial.print is replaced with sprintf() and UARTPrint() functions

```
//This function will write a value to a register on the ADXL345.
//Parameters:
//  char registerAddress - The register to write a value to
//  char value - The value to be written to the specified register.
void writeRegister(char registerAddress, char value){
  //Set Chip Select pin low to signal the beginning of an SPI packet.
  digitalWrite(CS, LOW);
  //Transfer the register address over SPI.
  SPI.transfer(registerAddress);
  //Transfer the desired register value over SPI.
  SPI.transfer(value);
  //Set the Chip Select pin high to signal the end of an SPI packet.
  digitalWrite(CS, HIGH);
}
```

digitalWrite changes CS output pin HIGH/LOW

SPI.transfer(value) both input/output the SPI transfers

```
//This function will read a certain number of registers starting from
// a specified address and store their values in a buffer.
//Parameters:
//  char registerAddress - The register addresse to start the read sequence from.
//  int numBytes - The number of registers that should be read.
//  char * values - A pointer to a buffer where the results of the operation should be stored.
void readRegister(char registerAddress, int numBytes, unsigned char * values){
  //Since a read operation, the most significant bit of the register address should be set.
  char address = 0x80 | registerAddress;
  //If we're doing a multi-byte read, bit 6 needs to be set as well.
  if(numBytes > 1)address = address | 0x40;

  //Set the Chip select pin low to start an SPI packet.
  digitalWrite(CS, LOW);
  //Transfer the starting register address that needs to be read.
  SPI.transfer(address);
  //Continue to read registers until the number specified,
  //storing the results to the input buffer.
  for(int i=0; i<numBytes; i++){
    values[i] = SPI.transfer(0x00);
  }
  //Set the Chips Select pin high to end the SPI packet.
  digitalWrite(CS, HIGH);
}
```

SAME THING:
digitalWrite changes CS output pin HIGH/LOW

SPI.transfer(value) both input/output the SPI transfers

SPI is always two directions.

When you send, you also receive, and to receive, you have to send.

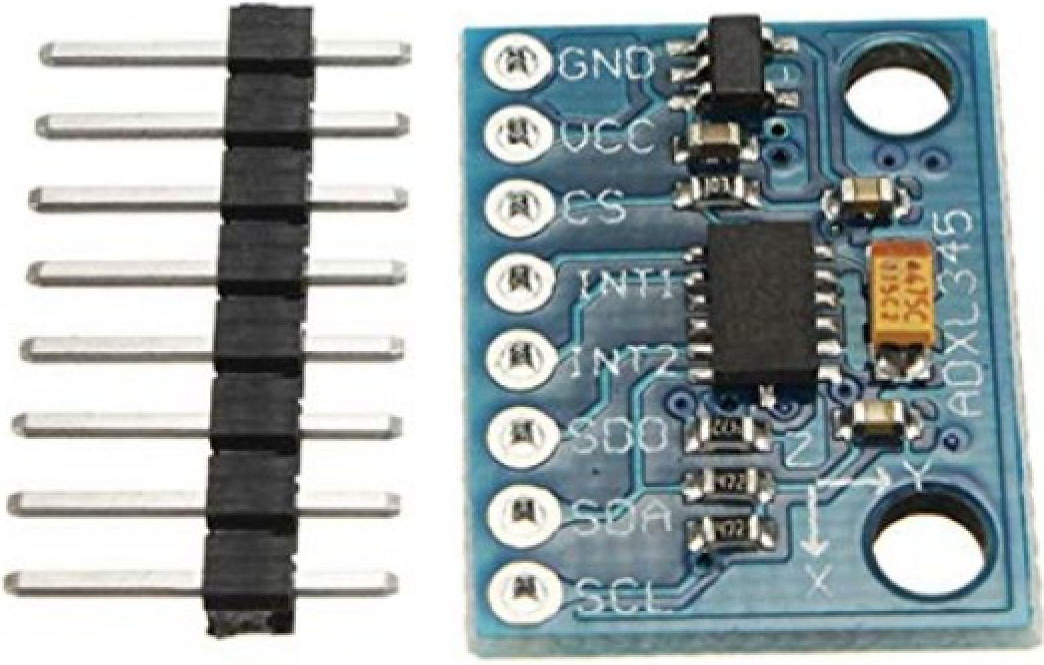
In your code, when you send the command, you'll receive a dummy answer while you're sending the command. While the bits of the commands are sent (and the slave hasn't received the command), the SPI hardware is already 'receiving' bits simultaneously.

So once the command byte has been sent, a byte has been received too,
which you'll need to discard.

THEN you send one or two dummy bytes and while they're sent, you're receiving the 8 or 16 bit answer.

My code does not use TXISR to interrupt, because this two-way transfer is for 6 bytes only. Code in the next Module can show way to start one byte and enter low power mode to wait for that byte to be sent.

So the MSP430 routines here just poll until the TX is complete and then sends/receives the next 'transfer' byte


GND
VCC
CS
INT1
INT2
SDO
SDA
SCL
ADXL345
Z
Y
X

sketch_SPI_ADXL345.ino

```
/************************************************************************************
 * This is morph of RT_AD3 to SPI ADXL345 input and print
 *  replaces Arduino Loop:  with 10 clock interrupts per second
 *   to process 'loop' tasks.
 *
 *
 *  20181002  H. Watson
*   Arduino Pin      ADXL345     MSP430FR2433
*   pin 13 SCK  -> SCL  ..........  P2.4
*   pin 12 MISO -> SDO  ..........  P2.5
*   pin 11 MOSI -> SDA  ..........  P2.6
*   pin 8   CS    -> CS  .......... P2.1
*
* //https://www.sparkfun.com/tutorials/240
*  //http://forum.arduino.cc/index.php/topic,159313.0.html
*  robo_maniac
*
 *
 * 1. create 10 Hz timer interrupt
 * 2. Get values from ADXL345 for x,y,z axes
 * 3. add TxISR to print out string with axis values
 * 4. add sprintf value to generate ouput string from axis values
  *
 *
 *
 *  H. Watson  20181029
 * /
```

```
//  ACLK = REFOCLK = 32kHz, MCLK = SMCLK = default DCODIV = 1MHz.
//
//                MSP430FR2433
//             -----------------
//         /|\|                 |
//          | |             P2.4|-->SCL / SCK
//          --|RST          P2.5|-->SDO / MISO
//            |             P2.6|-->SDA / MOSI
//            |             P2.1|-->CS
//            |                 |
//            |             P1.0|-->RED LED
//
//
```

```c
//******************************************************************************/
#include <msp430.h>


int putchar(int TxByte);  // output char
void UARTPutString(const char* strptr); // begin output of string
void UARTSetup (void);
unsigned char SPI_Transfer ( unsigned char tempB );
void readRegister(char registerAddress, int numBytes, unsigned char * values);
void writeRegister(char registerAddress, char value);


const char* TxPtr ;
char OutStr[50];  // buffer to hold output string
unsigned char Count;

/* SPI SETUP Control values  */
//This is a list of some of the registers available on the ADXL345.
//To learn more about these and the rest of the registers on the ADXL345, read the datasheet!
char POWER_CTL = 0x2D;  //Power Control Register
char DATA_FORMAT = 0x31;
char DATAX0 = 0x32; //X-Axis Data 0
char DATAX1 = 0x33; //X-Axis Data 1
char DATAY0 = 0x34; //Y-Axis Data 0
char DATAY1 = 0x35; //Y-Axis Data 1
char DATAZ0 = 0x36; //Z-Axis Data 0
char DATAZ1 = 0x37; //Z-Axis Data 1
//This buffer will hold values read from the ADXL345 registers.
unsigned char values[10];
//These variables will be used to hold the x,y and z axis accelerometer values.
int x,y,z;

```

SetUp

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                               // Stop watchdog timer

    // Disable the GPIO power-on default high-impedance mode to activate
    // previously configured port settings
    PM5CTL0 &= ~LOCKLPM5;

    // Configure GPIO Setup
    // RED LED
    P1DIR |= BIT0;                                          // Set P1.0 as output
    P1OUT |= BIT0;                                          // P1.0 high

//Configure the SPI connection for the ADXL345.
//SPI.setDataMode(SPI_MODE3);
// set the port pins
/*      ADXL345     MSP430FR433
  -> SCL  ..........  P2.4
  -> SDO  ..........  P2.5
  -> SDA  ..........  P2.6
  -> CS  .......... P2.1
  */
    P2DIR |= BIT4 | BIT5 | BIT6 | BIT1;
    P2SEL0 |= BIT4 | BIT5 | BIT6;
```

SPI SetUp

```
// SPI setup eUSCI_A1 used in SPI.h
UCA1CTLW0 |= UCSWRST;                 // **Put state machine in reset**
UCA1CTLW0 |= UCMST|UCSYNC|UCCKPL|UCMSB| UCMODE_0;   // 3-pin, 8-bit SPI master
                                      // Clock polarity high, MSB
UCA1CTLW0 |= UCSSEL__SMCLK;           // SMCLK
UCA1BR0 = 0x01;                       // /2,fBitClock = fBRCLK/(UCBRx+1).
UCA1BR1 = 0;                          //
UCA1MCTLW = 0;                        // No modulation
UCA1CTLW0 &= ~UCSWRST;                // **Initialize USCI state machine**

P2OUT |= BIT1;             //CS HIGH
//Put the ADXL345 into +/- 4G range by writing the value 0x01 to the DATA_FORMAT register.
writeRegister(DATA_FORMAT, 0x01);
//Put the ADXL345 into Measurement Mode by writing 0x08 to the POWER_CTL register.
writeRegister(POWER_CTL, 0x08);  //Measurement mode
```

Timer setup - 10/second

```
    UARTSetup();        // set BAUD rate

    // Timer0_A3 Setup  ISR 10/second:
    TA0CCTL0 |= CCIE;                           // TACCR0 interrupt enabled
    TA0EX0 |= TAIDEX_3;                         // SMCLK/8/4 = 31250 Hz
    TA0CCR0 = 3125;                            // 10 per second
    TA0CTL = TASSEL_2 | MC_1 | ID_3;                   // SMCLK/8 = 125K , UP mode



  // go to standby
    __bis_SR_register(LPM0_bits | GIE);
}
```

Timer ISR – This is where Loop Tasks go

```c
// Timer A0 interrupt service routine
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    P1OUT ^= BIT0;
    // print ASCII alphabet 10 char/second
    if(!(UCA0IE & UCTXIE))
    { // if flag is clear, means last string output is done
      // GET SPI VALUES
      //Reading 6 bytes of data starting at register DATAX0 will retrieve the
      //x,y and z acceleration values from the ADXL345.
        //The results of the read operation will get stored to the values[] buffer.
        readRegister(DATAX0, 6, values);

        //The ADXL345 gives 10-bit acceleration values,
        //but they are stored as bytes (8-bits).
        //To get the full value, two bytes must be combined for each axis.
        //The X value is stored in values[0] and values[1].
        x = ((int)values[1]<<8)|(int)values[0];
        //The Y value is stored in values[2] and values[3].
        y = ((int)values[3]<<8)|(int)values[2];
        //The Z value is stored in values[4] and values[5].
        z = ((int)values[5]<<8)|(int)values[4];

      //sprintf(OutStr,"The value of Count is %d \n",Count++);
      sprintf(OutStr,"%d,%d,%d\n",x, y, z);
      UARTPutString(OutStr); // begin output of string
    }
}
```

UART Print the string

```
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(UCA0IV)
  {
    case USCI_NONE: break;
    case USCI_UART_UCRXIFG:
      while(!(UCA0IFG&UCTXIFG));
      UCA0TXBUF = UCA0RXBUF;
      __no_operation();
      break;
    case USCI_UART_UCTXIFG:
        // load char value
       // unsigned char testVal=*TxPtr++;
        if(!(*TxPtr))  // if zero, then stop
        {
           UCA0IE &= ~UCTXIE;  // turn off interrupt
        }
        else
        {
           UCA0TXBUF = *TxPtr++ ;
        }
      break;
    case USCI_UART_UCSTTIFG: break;
    case USCI_UART_UCTXCPTIFG: break;
    default: break;
  }
}
```

UART SetUp and UARTPutString

```
void UARTSetup (void)
{
  // Configure UART pins
  P1SEL0 |= BIT4 | BIT5;                      // set 2-UART pin as second function
  // Configure UART
  UCA0CTLW0 |= UCSWRST;               // reset UART
  UCA0CTLW0 |= UCSSEL__SMCLK;         // use SMCLK input
    UCA0BR0 = 104;                   // 1MHz SMCLK/9600 BAUD
    UCA0MCTLW = 0x1100; //           // remainder of Baud Rate
  UCA0CTLW0 &= ~UCSWRST;

}

void UARTPutString(const char* strptr) // begin output of string
{
  // load TxBuf with first char then enable interrupt
    TxPtr = strptr;
    UCA0TXBUF = *TxPtr++;  //load first, assume at least one char in buffer
    UCA0IE |= UCTXIE;   // interrupt when transmitted  - ISR turns off when done
}
```

SPI_Transfer – Polled method

```
// NO SPI ISR, this is polled output/input function
// output data with polled SPI communication
unsigned char SPI_Transfer ( unsigned char tempB )
{
    UCA1TXBUF  =  tempB;   //  Send    0xAA    over    SPI to  Slave
     while (UCA1STATW & UCBUSY);  //wait until done (receiving whole byte)
    return (UCA1RXBUF);  //send back input value
}
```

Chip Select and writeRegister for ADXL345 SPI

```
//  SPI Communication functions
//This function will write a value to a register on the ADXL345.
//Parameters:
//  char registerAddress - The register to write a value to
//  char value - The value to be written to the specified register.
void writeRegister(char registerAddress, char value){
  //Set Chip Select pin low to signal the beginning of an SPI packet.
  //  digitalWrite(CS, LOW);
  P2OUT &= ~BIT1;                //CS LOW
  //Transfer the register address over SPI.
  SPI_Transfer(registerAddress);
  //Transfer the desired register value over SPI.
  SPI_Transfer(value);
  //Set the Chip Select pin high to signal the end of an SPI packet.
  //  digitalWrite(CS, HIGH);
  P2OUT |= BIT1;                 //CS HIGH
}
```

Chip Select and readRegister for ADXL345 SPI

```
//This function will read a certain number of registers starting from a specified
//address and store their values in a buffer.
//Parameters:
//  char registerAddress - The register addresse to start the read sequence from.
//  int numBytes - The number of registers that should be read.
//  char * values - A pointer to a buffer where the results of the operation should be stored.
void readRegister(char registerAddress, int numBytes, unsigned char * values){
  //Since read operation, the most significant bit of the register address should be set.
  char address = 0x80 | registerAddress;
  //If we're doing a multi-byte read, bit 6 needs to be set as well.
  if(numBytes > 1)address = address | 0x40;

  //Set the Chip select pin low to start an SPI packet.
  //digitalWrite(CS, LOW);
  P2OUT &= ~BIT1;         //CS LOW
  //Transfer the starting register address that needs to be read.
  SPI_Transfer(address);
  //Continue to read registers until the number specified,
  //storing the results to the input buffer.
  for(int i=0; i<numBytes; i++){
    values[i] = SPI_Transfer(0x00);
  }
  //Set the Chips Select pin high to end the SPI packet.
  //digitalWrite(CS, HIGH);
  P2OUT |= BIT1;              //CS HIGH
}
```