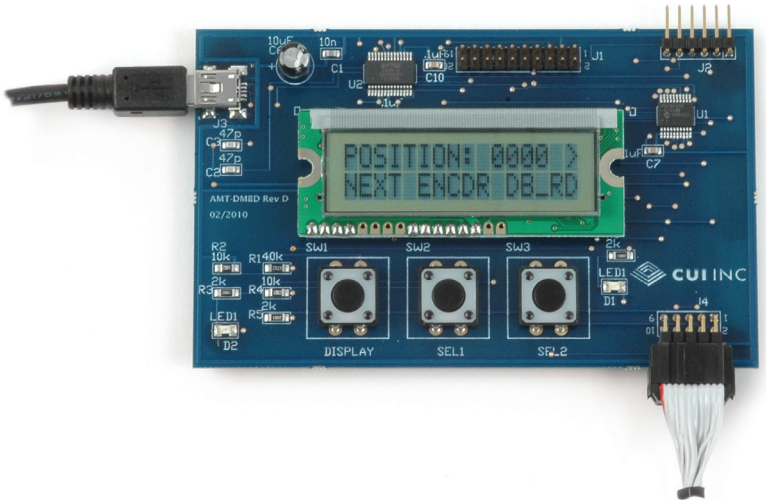
APPLICATION NOTE AN-1001

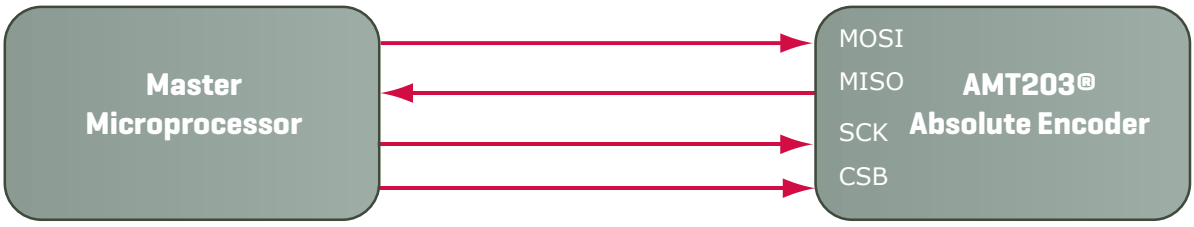
SPI Communication with AMT203 12-bit Absolute Encoder



AMT MODULAR encoders

INTRODUCTION

This application note is designed to provide guidelines on how to properly interface with the AMT 203 Absolute Encoder. The AMT 203 is a 12 bit absolute encoder that operates over the synchronous serial communication known as Serial Peripheral Interface (SPI) Bus. Explanation of the data timing and example code are given to provide the user further detail on how to properly implement the AMT 203 into a particular application.



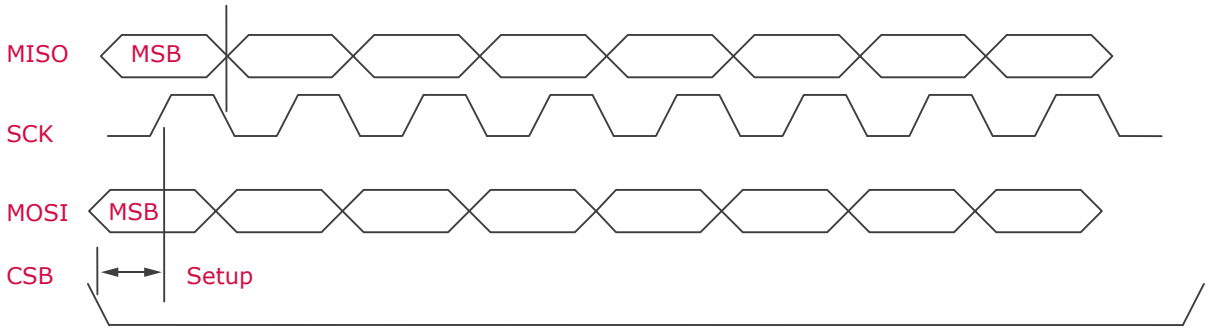
SPI BUS

The SPI or Serial Peripheral Interface Bus is a standard interface promoted by Motorola and Microchip among others. It consists of 4 signals.

MOSI: Master Out Slave In
MISO: Master In Slave Out
SCK: Serial Clock
CSB: Chip Select Bar (active low)

The SPI bus runs full duplex and transfers multiples of 8 bits in a frame. The SPI type is the most common (CPOL=0, CPHA=0), also known as Microwire. Data is captured on the rising edge of SCK and the output data is changed after the falling edge of SCK.

SINGLE SPI FRAME



Serial Peripheral Interface Bus (SPI) with CPOL=0, CPHA=0

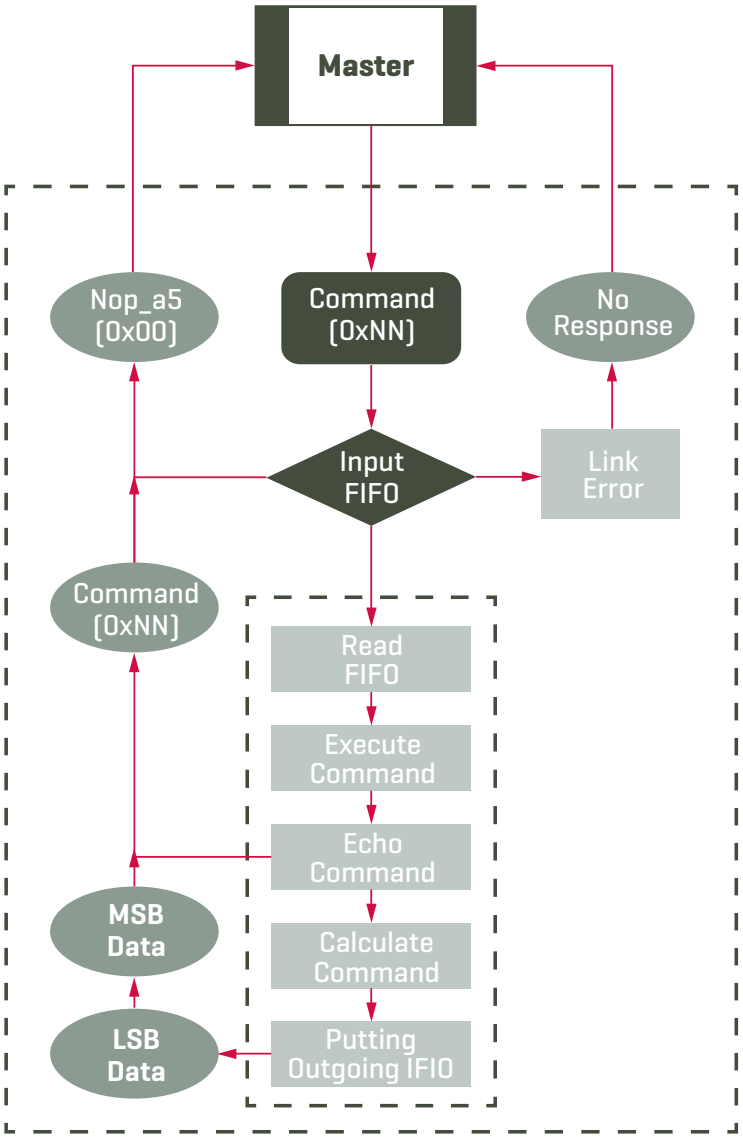
SPI FRAME SPECIFICATION

The SPI frames are all 1 byte (8 bits) in length. A hardware SPI slave port is used, from a 20 Mhz Microchip PIC16F690 processor. SPI is defined as full duplex in that a byte is received from the slave every time a byte is sent from the master.

The host (master) sends data on the MOSI line, and the encoder (slave) sends data on the MISO line. In SPI the MSB is shifted in first and is the leftmost bit shown in the documentation. The CSB line gates the transfer and there may be several CSB lines which are used to select among slave devices.

The SPI block implemented in the PIC16F690 only contains a single buffer so data must be read out before writing more data to be sent. After the master sends a frame to the slave there is a 5 µs gap during which the CSB line is high. During this time the incoming data is read out, transmit data is written and the received data is saved. The encoder uses the SPI interrupt for data transfer and FIFO structures for incoming and outgoing data.

When a "rd_pos" command is received, the byte is saved by the interrupt routine



in the incoming FIFO. On returning from the interrupt the processor reads the FIFO, executes the command, echos the command, performs calculations and puts the position data in the outgoing FIFO. Since there are FIFO buffers, several commands may be overlapped (each FIFO is 16 bytes in length).

The host (master) must read the encoder to get the data. It does this by sending "nop_a5" (No Operation command: 0x00). If the data is not ready to be read, the encoder responds with "a5". When the data is ready the encoder echos the command "rd_pos" (0x10) and the two bytes of data follow in the next two bytes that are read from the encoder.

The encoder will always respond "a5" to a "nop_a5" command (0x00) or to an unknown command. If the encoder does not respond "a5" it is an indication there is something wrong with the communication link. In this case is important to make sure the MISO and MOSI lines are not reversed.

ENCODER PROTOCOL AND TIMING CONSIDERATIONS

The Encoder uses a Microchip PIC16F690 to operate with a high speed SPI link in full duplex slave mode. The Microchip SSP (Synchronous Serial Port) block is used for this. This is common the industry so should be familiar to designers.

The SSP does not have double buffering, so there is a recovery time required by the slave between SPI commands. The recovery time is the time taken to respond to the interrupt, read the SPI data, and transfer data from a fifo to the SPI output buffer. This time is determined by the processor frequency: using a 20 Mhz clock we see a requirement of 5 µs for arbitrary commands, 3 µs for fast mode reads. ***Polling faster than this will cause a malfunction due to overflow in the SSP block and data response will no longer be valid.***

For normal operations such as position reads, there is a delay from the command until the read is done and the data is ready to be read out. During this delay time if the HOST sends a command it will receive the wait sequence (0xA5). The only command that should be sent during this time is 0x00, which is the "nop_a5" command to poll the current status of the output buffer.

COMMAND SEQUENCE

An example of the "rd_pos" sequence is as follows (rd_pos = 0x10 is used in this example):

1. The host issues the command, 0x10. The data read in at this time will be 0xa5 or 0x00 since this is the first SPI transfer.
2. The host waits a minimum of 5 µs then sends a "nop_a5" command: 0x00.
3. The response to the "nop_a5" is either 0xa5 or an echo of the command, 0x10.
    a. If it is 0xa5, it will go back to step 2.
    b. Otherwise it will go to step 4.
4. The host waits a minimum of 5 µs then sends "nop_a5", the data read is the high byte of the position.
5. The host waits a minimum of 5 µs then sends "nop_a5", the data read is the low byte of the position.
6. The host waits a minimum of 5 µs before sending another SPI command.

Thus, to read the position, the host issues rd_pos (0x10 or 0001,0000), receiving one or more wait sequences (0xA5) then a reflected rd_pos (0x10), then the MSB data followed by the LSB data.

COMMAND DESCRIPTIONS

Command 0x00: nop_a5

This "No Operation" command is used to read data from the encoder. The expected response is "a5" if there is nothing in the outgoing FIFO.

* This command is useful to verify the communication link is correct.

Command 0x10: rd_pos

This command causes a read of the current position. The internal position register is adjusted for the zero offset and direction setting when it is read out.

The response is as follows:

1. one or more 0xa5 bytes
2. command echo (0x10)
3. high byte position
4. low byte position

Command 0x70: set_zero_point

This command sets the current position to zero and saves this setting in the EEPROM. The host should send nop_a5 repeatedly after sending this command, the response will be 0xa5 while update is proceeding and (0x80) eeprom_wr is the response when update is finished.

Note: depending on the setting of comm_pos in the eeprom at locations 0xFD and 0xFE, the encoder will use these values to offset the current position calculation. Unless adjusting to a specific offset, verify that these EERPOM locations are set to 00.

**The encoder must be power-cycled after this command in order for the new zero position to be used in the position calculation.*

Command 0x80,<byte_address>,<data>: eeprom_wr

This command causes the data to be written to the address given in <byte_address>. The address can be 0x00 to 0xff for 256 bytes of data. The host should send nop_a5 repeatedly after sending this command, the response will be 0xa5 while update is proceeding and eeprom_wr is the response when update is finished.

**Certain areas are used by the system, so the user should confine data storage to the lower 128 bytes indicated.*

Command 0x90,<byte_address>,0x01: eeprom_rd

This command causes the data in eeprom at the given address to be read and put in the output fifo.

The sequence is as follows:

1. issue read command, receive 0xa5
2. issue NOP, receive 0xa5 or 0x90 (echo of read command)
3. repeat step 2 if it is 0xa5, go to step 4 if it is 0x90
4. issue NOP and receive data byte

Sample PIC Code for Reading the Position

To illustrate the use of these commands, an assembler program for the Microchip® PIC16F690 was written. This is the microcontroller used on the AMT-DMBD Demo Board and the program was tested on that platform.

The algorithm to read the position from the encoder follows the "rd_pos" sequence above:

```
do_rd_pos ; access SPI and read the position
 movlw 0x04 ; error counter err_ctr for initial 00 byte from encoder
 movwf err_ctr
 bcf main_status,1 ; clear the error flag
 movlw 0x10 ; rd_pos command
 goto send_byte ; send this byte and enter nop_a5_send loop

 ; send the command then repeat sending nop_a5 until data is ready
nop_a5_send movlw 0x00 ; load nop_a5 byte to send
send_byte call send_spi_byte ; send the byte
 call buzzlp_5us ; wait at least 5 us
 call get_spi_data ;
 movlw 0x10 ; if 0x10 received get the data
```

```
 xorwf spi_rcv_data,w
 btfsc status,z ;
     goto rd_pos_seen ;
 movlw 0xa5 ; if 0xa5 received send nop_a5 again
 xorwf spi_rcv_data,w
 btfsc status,z ;
 goto nop_a5_send ;
 decfsz err_ctr ; if it is not 0xa5 or 0x10 then flush the command
 goto nop_a5_send
 bsf main_status,1 ; error: disconnected
 return

rd_pos_seen ; ok continue and get the data

 ; bcf porta,2
 ; now read the postion data, first the hi byte is read
 movlw 0x00 ; send nop_a5 for SPI transfer
 call send_spi_byte ; get pos_hi
 call buzzlp_5us
 call get_spi_data
 movf spi_rcv_data,w
 movwf pos_hi ; save pos_hi

 ; send another nop_a5 to get the lo byte
 movlw 0x00
 call send_spi_byte ; get pos_lo
 call buzzlp_5us
 call get_spi_data
 movf spi_rcv_data,w
 movwf pos_lo ; save pos_lo
 return
```

This routine in C would be as follows:

```
int do_rd_pos() {
  err_ctr=4;
main_status_1 = 0;
cmd_to_send=0x10;
rd_pos_seen=false;
while ((main_status_1==0)&(!rd_pos_seen)){
  send_spi_byte(cmd_to_send));
  wait_us(5);
  get_spi_data(spi_rcv_data);
  if (spi_rcv_sata==0x10) rd_pos_seen=1;
  elsif (spi_rcv_data!=0xa5) {
    err_ctr--;
    if (err_ctr==0) main_status_1=1;
  }
  cmd_to_send=0x00;
}
if (main_status_1) return 0; /* error exit */
send_spi_byte(0x00);
wait 5 us;
get_spi_data(pos_hi);

send_spi_byte(0x00);
wait 5 us;
get_spi_data(pos_lo);
return 1;
}
```

Using Sample Code

The full assembler program is included in a zip file db690_asm.zip and can be downloaded at amtencoder.com which can then be assembled with the Microchip MPASM Assembler. It will produce a hex file which can be loaded into the AMT-DMBD Demo Board using a PICKit 2™ Flash Programmer.

To assemble with db690_asm.zip

1. unzip db690_asm.zip to your desktop
2. double-click on the file: do_asm.cmd in the unzipped file db690_asm

*The script assumes some paths that may need to be changed, the executable mpasmwin is assumed to be located in: "C:\Program Files (x86)\Microchip\MPASM Suite\mpasmwin"
**The hex file is copied to a desktop folder after assembly: %HOMEDRIVE%%HOMEPATH%\Desktop\hex_files\db690_appnote

Logic Analyzer Command Sequence Examples

The following section details how the command sequences should look using a logic analyzer. The details were created using a TLA5201B logic analyzer and the AMT-DMBD Demo Board to help visualize the command sequences used with the AMT 203 Absolute Encoder.

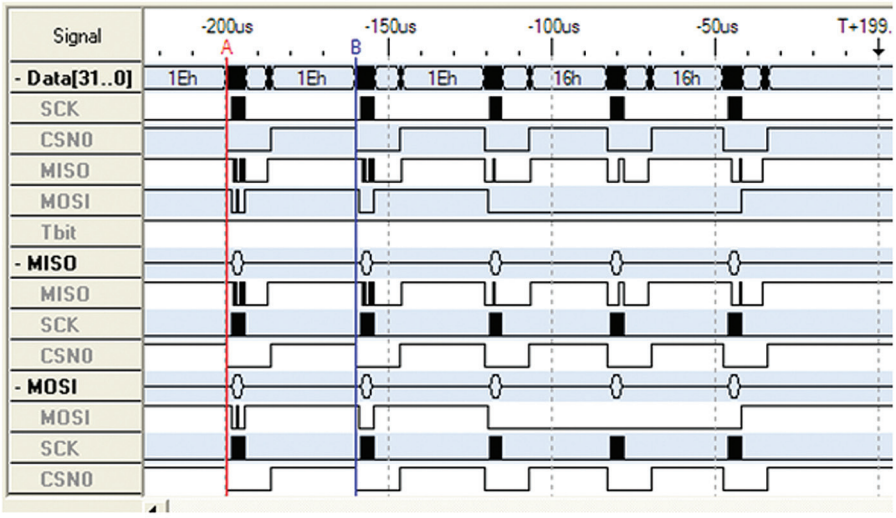


Figure 1 shows the POS_RD command sequence when the AMT-DMBD Demo Board is used.

**Note the timing does not require using a 5 µs wait loop since the execution loop is longer than that.*

This sequence consists of a POS_RD command (0x10), one or more NOP_A5 (0x00) (until the command reply is seen (0x10 reply)) then two more NOP_A5 (0x00) sequences to read in the high and low position bytes.

The delay between commands is about 39 µs, which is on the order of the internal cycle time of the encoder, so we should expect additional wait states to be inserted. An example with additional wait states is given below, in the subsection titled "Example with multiple NOP_A5 commands".

If we zoom in on the sequence in Figure 1, we can see the individual command/response cycles.

Figure 2 shows the POS_RD command being sent.

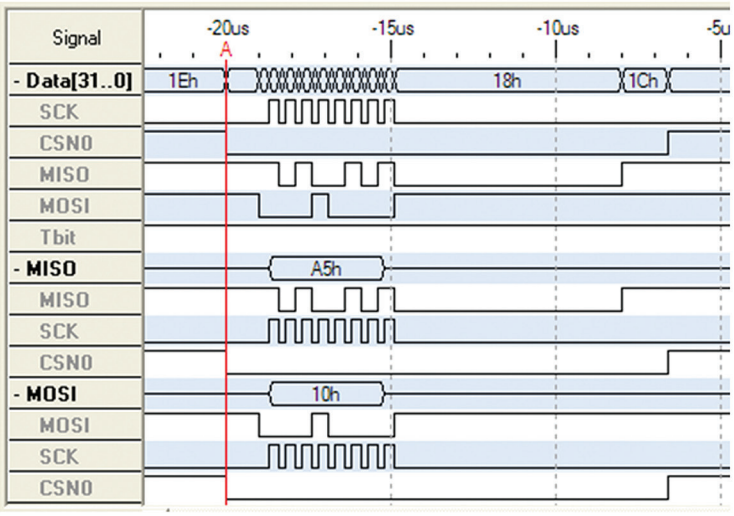


Figure 2: POS_RD command.

The command (0x10) is sent on MOSI. The slave buffer is read as the command is sent – notice that it results in a 0xA5 response. This indicates the slave is connected but did not receive the command yet. When the last bit of the command is sent, the slave must fill the output buffer immediately and there is no time to look at the incoming command, so the slave will always fill with a 0xA5 if the command has not been processed and there is a SPI transfer.

The CSN0 signal is used to select the encoder. It is set low just before the assembler program writes to the SPI buffer. It is set high by the Interrupt Service routine just before the data is read out of the incoming buffer. The interrupt preamble is about 6 us, and interrupt service is about 2 µs, so the 8 µs delay to set CSN0 high is as expected.

After sending the POS_RD command, there are one or more NOP_A5 commands depending on encoder and transmission timing. The encoder is repeatedly acquiring and calculating positions so when a POS_RD command arrives it may have to wait for completion of another operation. The 0xA5 reply is interpreted as a "wait" signal when a command is pending. Figure 3 demonstrates an NOP_A5 (0x00) command with a 0xA5 response.

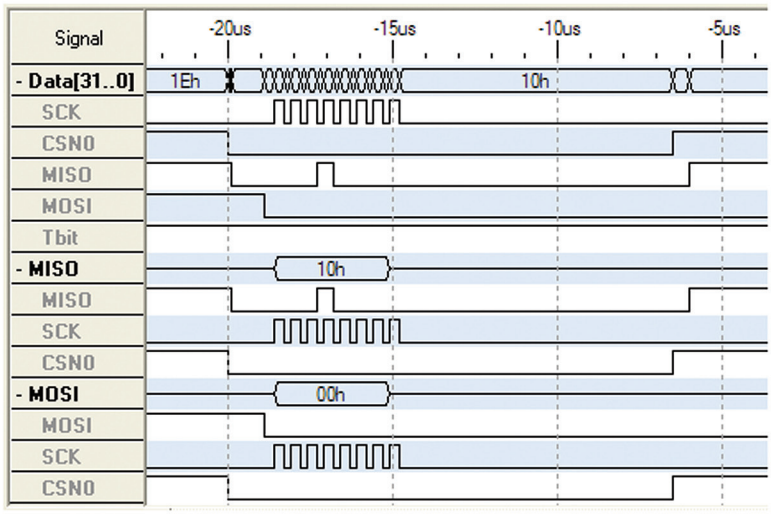


Figure 3: NOP_A5 command sent, 0xA5 received.

As previously mentioned, there will be one or more of these 0xA5 responses before the command echo. Below, figure 4 shows the command echo.

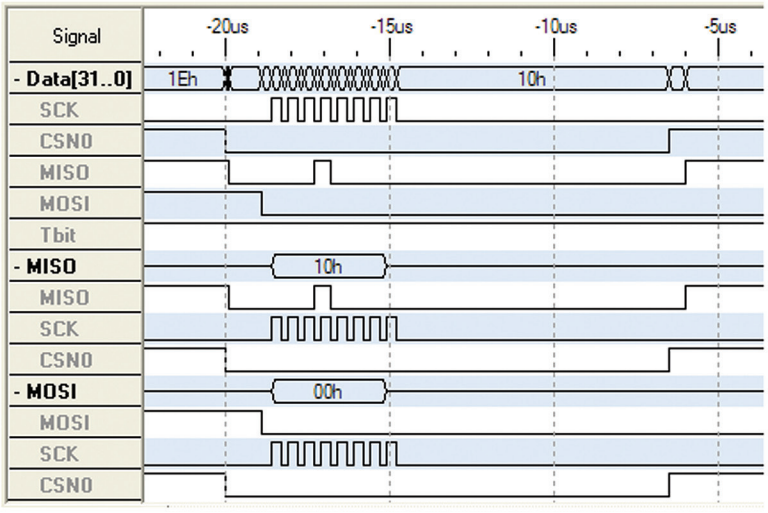


Figure 4: Echo of POS_RD command.

When the position data is ready the encoder responds to the NOP_A5 command with an echo of the pending command. In this case, the echo is POS_RD (0x10). This means the next two bytes read are the high and low position bytes. Figure 5 shows the position 0x07CA being read.

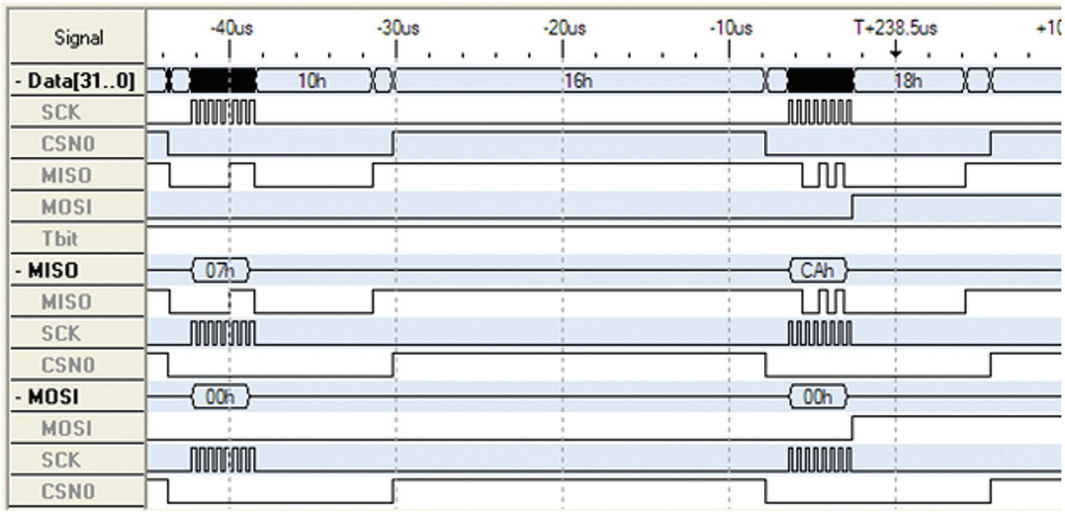


Figure 5: Position high and low byte transfer.

The encoder has a command buffer, so it is possible to send another POS_RD command before finishing the current command. It is up to the user to monitor the buffer and commands being sent to ensure data remains valid.

Example with multiple NOP_A5 commands

The example in Figure 6 below illustrates the worst-case delay based on the inherent ~39 µs cycle time. There are two extra 0xA5 responses for a total of 7 cycles for this POS_RD sequence:

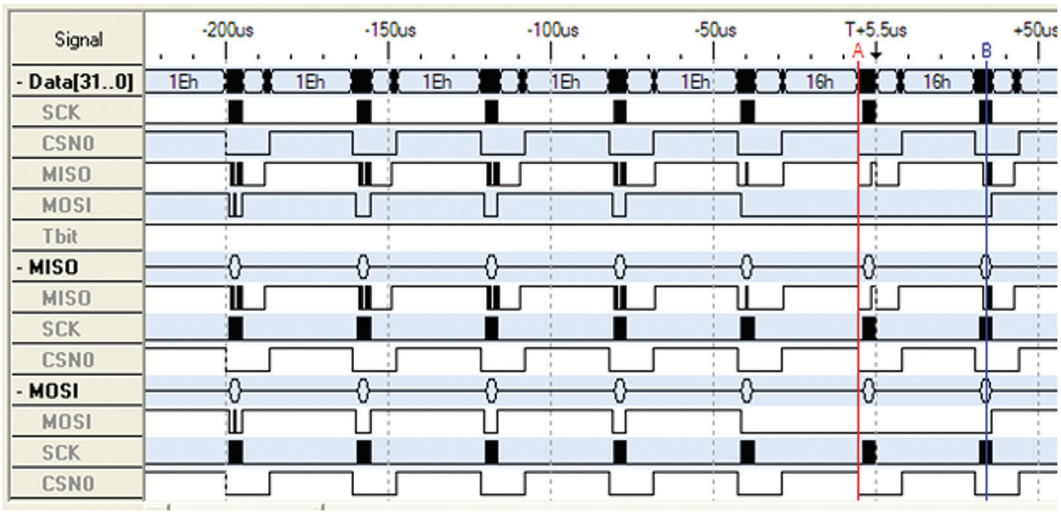


Figure 6: Worst case for example timing.

Since the encoder is not synchronized to the host, the encoder may be doing internal read operations and calculations which delay processing the incoming command. In this example the command was received and the response was not ready within an 80 µs window internal to the encoder.

To prove this is the case, a 100% predictable sequence can be attained by adding in a delay slightly larger than this uncertainty interval. We end up with a constant 5 cycle sequence as shown in Figure 7.

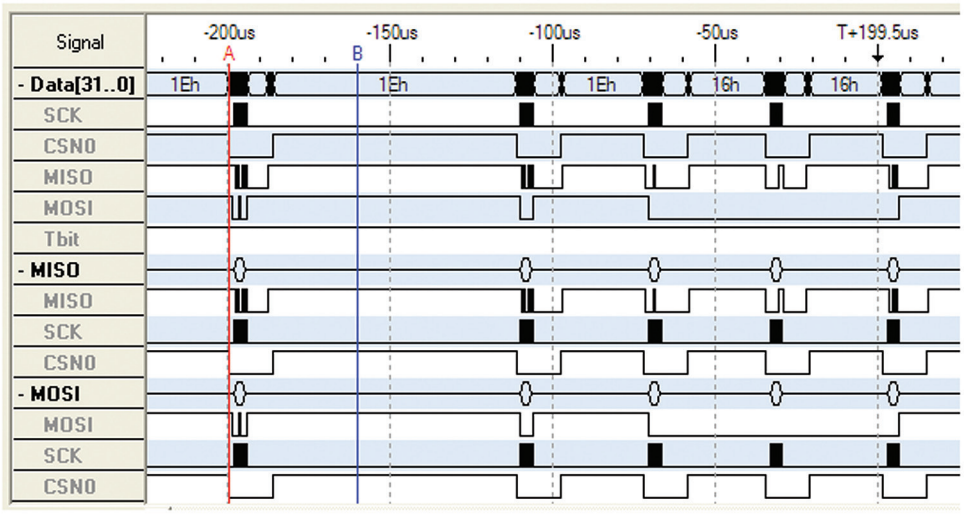


Figure 7: Consistent 5 cycle sequence.

Notice this does not save time overall, it only makes it more predictable. It does save the intervening interrupt response times that would be required in Figure 6.

High Speed considerations

For high speed applications there is a "fast read mode" that can be used. This mode is only recommended for extreme cases. In this mode, every byte transfer consists of the low position register without any direction or offset calculations. In this mode the host is responsible for maintaining direction, the high 4 bit position register, and the offset.

As an alternative the quadrature signals may be used with a counter to track the position after reading SPI position values and calibrating the host.

CUI INC®

Headquarters
20050 SW 112th Ave.
Tualatin, OR 97062
800.275.4899

Fax 503.612.2383
cui.com
techsupport@cui.com