International Journal of Electrical and Electronics Engineering Studies
Vol.5, No.1, pp. 1-34, December 2019
Published by ***ECRTD-UK***
ISSN 2056-581X(Print), ISSN 2056-5828(Online)

PIC32MZ CUSTOM SPI MASTER COMMUNICATION API LIBRARY

Abhishek N. Patel
Research and Development Department, Horiba Instruments Inc., 20 Knightsbridge Road, Piscataway, New Jersey, United States

ABSTRACT: *This paper demonstrates how to make SPI (Serial Peripheral Interface) master communication library routines and used those routines to communicate data to and from the PIC32MZ/PIC32 MCUs. Unlike the UART, SPI communication is a synchronous: the master device on an SPI bus creates a separate clock signal that dictates the timing of communication. The device do not have to be configured in advanced to share the same bit rate, and any clock frequency can be used which is within the capabilities of the chips. The SPI bus is a full-duplex bus, which allows communication to flow to and from the master device simultaneously at rate of up to 10Mbps. Typical applications include SD cards, motor controller, SRAM, LCD and sensors. Unfortunately, the microchip PICXC32 compiler does not gives us the SPI master library APIs to call from firmware application layer. This implementation requires PIC32MZ/PIC32 MCU based hardware and microchip MPLAB-X IDE, tool chain and Harmony framework.*

KEYWORDS: C, PIC32MZ, PIC32, Firmware, Embedded System, SPI, i2C

INTRODUCTION

SPI (Serial Peripheral Interface) is an interface bus commonly used for communication with flash memory, sensors, real-time clocks (RTCs), analog-to-digital converters, motor drier and more. The Serial Peripheral Interface (SPI) bus developed by Motorola to provide full-duplex synchronous serial communication between master and slave devices.

The SPI is four wire-based full duplex communication protocol these wire generally known as MOSI (master out slave in), MISO (master in slave out), SCL (a serial clock that produces by the master) and SS (slave select line that use to select specific slave during the communication).

SPI follows the master and slave architecture and communication, which always started by the master. In addition, it is a synchronous communication protocol because master and slave share the clock. SPI supports only multi-slave does not support multi-master and slave will be select by the SS (slave select) signal. In SPI during the communication, data shifted out from the master and shifted into the slave vice-versa through the shift register.

International Journal of Electrical and Electronics Engineering Studies
Vol.5, No.1, pp. 1-34, December 2019
Published by ***ECRTD-UK***
ISSN 2056-581X(Print), ISSN 2056-5828(Online)

SPI Detail

SPI is a master-slave architecture, and an SPI bus has one master device and one or more slaves. A minimal SPI bus consists of three wires (in addition to GND): Master Output Slave Input (MOSI), carrying data from the master to the slave(s); Master Input Slave Output (MISO), carrying data from the slave(s) to the master; and the master's clock output, which clocks the data transfers, one bit per clock pulse. Each SPI device on the bus correspondingly has four pins: Serial Data out (SDO), Serial Data In (SDI), Slave Select (SS) and System Clock (SCK). The MOSI line is connected to the master's SDO pin and the slaves' SDI pins, and the MISO line is connected to the master's SDI pin and the slaves' SDO pins. All slaves' SCK pins are inputs, connected to the master's SCK output.

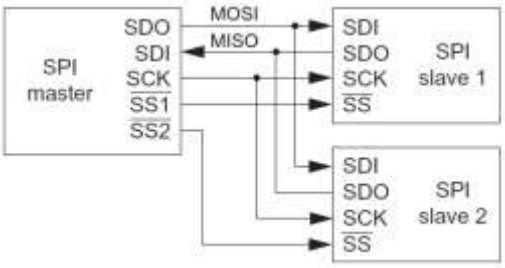


Figure 1: 4-wire SPI bus configuration with multiple slaves

The PIC32MZ consists three SPI peripherals can be either a master or a slave and typically it acting as the master.

In case of more than one slave on the bus, then the master controls which slave is active by using an active-low slave select (SS) line, one per slave. Only one slave-select line can have a low signal at a time and the low line indicates which slave is active. Unselected slaves must let their SDO output float at high impedance, effectively disconnected from the MISO line, and they should ignore data on the MOSI line. In Figure 1: 4-wire SPI bus configuration with multiple slaves, there are two slaves, and therefore two output slave-select lines from the master SS1 and SS2, and one slave-select input line for each slave. Thus, adding more slaves to the bus means adding more wires. The SPI master connected to two slave devices and each arrow indicates data direction. At a time, only one slave select line can be active (low).

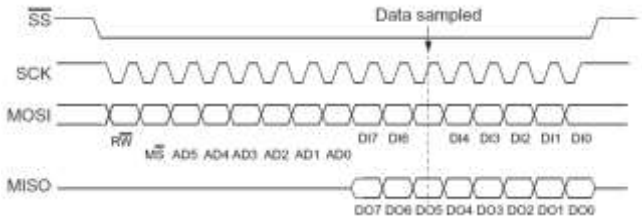


Figure 2: Timing for PIC32MZ to L6470H SPI data transaction.

ISSN 2056-581X(Print), ISSN 2056-5828(Online)

The above figure demonstrate a timing for an SPI transaction with a slave L6470H micro stepping motor controller. The L6470H selected when SS driven low. On the falling, edge of the master's SCK, both the master (controlling
MOSI with its SDO pin) and the slave (controlling MISO with its SDO pin) transition to the next bit of their signal. On the rising edge of SCK, the devices read the data, the master reading MISO with its SDI pin and the slave reading MOSI with its SDI pin. In this example, the master first sends 8 bits, the first of which (RW) determines whether it will be reading data from the L6470H or writing data to it. The bits AD0 to AD5 determine the address of the L6470H register that the master is accessing. If the operation is a read from the L6470H, the L6470H puts 8 bits DO0 to
DO7 on the MISO line; otherwise the master sends 8 bits on the MOSI line to write to the L6470H. Therefore, the master initiates all communication by creating a square wave on the clock line. Reading from and writing to the slave occur simultaneously, one bit per clock pulse. Thr transfers occur in groups of 8, 16, or 32 bits, depending on the slave.

NOTE: Above is a description of the basic SPI communication and the PIC32MZ SPI module has other modes of operations, which is not cover in detail here. For example, a framing mode allows for streaming data from supported devices. By default, the SPI peripheral has only a single input and single output buffer. The PIC32 also has an enhanced buffer mode, which provides FIFOs for queuing data waiting to be transfer or processed.

PIC32MZ SPI Status and Control Registers

Each of the three SPI peripherals, SPI2 to SPI4, has four pins associated with it: SCKx, SDIx, SDOx, and SSx, where x is 2 to 4. When the SPI peripheral is a slave, it can be configure so that it only receives and transmits data when the input SSx is low (e.g., when there is more than one slave on the bus). When the SPI peripheral is a master, it can be configure to drive the SSx automatically when communicating with a single slave. If there are multiple slaves on the bus, however, other digital outputs possibly used to control the multiple slave select pins of the slaves.

Each SPI peripheral uses four SFRs, SPIxCON, SPIxSTAT, SPIxBUF, and SPIxBRG, x = 2 to 4. As the Each SPI module consists of the following Special Function Registers (SFRs):

- SPIxCON: SPI Control Register
- SPIxSTAT: SPI Status Register
- SPIxBUF: SPI Buffer Register
- SPIxBRG: SPI Baud Rate Register

Many of the settings related to the alternative operation modes that we do not discuss. SFRs and fields that we omit may be safely left at their default values for typical applications. All default bits are zero, except for one read-only bit in SPIxSTAT.

SPIxCON:

This register contains the main control options for the SPI peripheral.

SPIxCON<28> or SPIxCONbits.MSSEN: Master slave select enable. If set, the SPI master will assert (drive low) the slave select pin SSx prior to sending an 8-, 16-, or 32-bit transmission and de-assert (drive high) after sending the data. Some devices require you to toggle the slave select after a complete multi-word transaction; in this case, you should clear SPIxCONbits.MSSEN to zero (the default) and use any digital output as the slave select.

SPIxCON<15> or SPIxCONbits.ON: Set to enable the SPI peripheral.

SPIxCON<11:10> or SPIxCONbits: MODE32 (bit 11) and SPIxCONbits.MODE16
(bit 10): Determines the communication width.
0b1X (SPIxCONbits.MODE32 = 1): 32 bits of data sent per transfer.
0b01 (SPIxCONbits.MODE32 = 0 and SPIxCONbits.MODE16 = 1): 16 bits of data sent per transfer.
0b00 (SPIxCONbits.MODE32 = SPIxCONbits.MODE16 = 0): 8 bits of data sent per transfer.

SPIxCON<9> or SPIxCONbits.SMP: determines when, relative to the clock pulses, the master samples input data. Should be set to match the slave device's specifications.
1 Sample at end of a clock pulse.
0 Sample in the middle of a clock pulse.

SPIxCON<8> or SPIxCONbits.CKE: The clock signal can be configure as either active high or active low by setting SPIxCONbits.CKP (see below). This bit, SPIxCONbits.CKE, determines whether the master changes the current output bit on the edge when the clock transitions from active to idle or idle to active (see SPIxCONbits.CKP, below). You should choose this bit based on what the slave device expects.
1 The output data changes when the clock transitions from active to idle.
0 The output data changes when the clock transitions from idle to active.

SPIxCON<7> or SPIxCONbits.SSEN: This slave select enable bit determines whether the SSx pin is use in slave mode.
1 TheSSx pin must be low for this slave to be selected on the SPI bus.
0 TheSSx pin is not used, and is available to be used with another peripheral.

SPIxCON<6> or SPIxCONbits.CKP: The clock signal can be configure as being active high or active low. Chosen in conjunction with SPIxCONbits.CKE, above, this setting should match the expectations of the slave device.
1 The clock is idle when high, active when low.
0 The clock is idle when low, active when high.

SPIxCON<5> or SPIxCONbits.MSTEN: Master enable. Usually the PIC32 operates as the master, meaning that it controls the clock and hence when and how fast data is transferred, in which case this bit should be set to 1. To use the SPI peripheral as a slave, this bit should be cleared to 0.
1 The SPI peripheral is the master.
0 The SPI peripheral is the slave.

SPIxSTAT The status of the SPI peripheral.

SPIxSTAT<11> or SPIxSTATbits.SPIBUSY: When set, indicates that the SPI peripheral is busy transferring data. You should not access the SPI buffer SPIxBUF when the peripheral is busy.

SPIxSTAT<6> or SPIxSTATbits.SPIROV: Set to indicate that an overflow has occurred, which happens when the receive buffer is full and another data word is received. This bit should be clear in software. The SPI peripheral can only receive data when this bit is clear.

SPIxSTAT<1> or SPIxSTATbits.SPITXBF: SPI transmit buffer full. Set by hardware when you write to SPIxBUF. Cleared by hardware after the data you wrote is transferred into the transmit buffer SPIxTXB, a non-memory-mapped buffer. When this bit clear, you can write to SPIxBUF.

SPIxSTAT<0> or SPIxSTATbits.SPIRXBF: SPI receive buffer full. Set by hardware when data is receive into the SPI receive buffer indicating that SPIxBUF can be read. Cleared when you read the data via SPIxBUF.
SPIxBUF Used to both read and write data over SPI. When, as the master, you write to SPIxBUF, the data is actually stored in a transmit buffer SPIxTXB, and the SPI peripheral generates a clock signal and sends the data over the SDOx pin. Meanwhile, in response to the clock signal, the slave sends data to the SDIx pin, where it is stored in a receive buffer SPIxRXB, which you do not have direct access to. To access this received data, you do a read from SPIxBUF. Therefore, perhaps unintuitive, after executing the C code

```
SPI1BUF = data1;
data2 = SPI1BUF;
```

data1 and data2 will not be identical! data1 is sent data, and data2 is received data. To avoid buffer overflow errors, every time you write to SPIxBUF you should also read from SPIxBUF, even if you do not need the data.

Additionally, since the slave can only send data when it receives a clock signal, which is only generated by the master when you write data, as a master you must write to SPIxBUF before getting new data from the slave.

$$\text{SPIxBRG} = \frac{F_{PB}}{2F_{sck}} - 1$$

SPIxBRG Determines the SPI clock frequency. Only the lowest 12 bits are used. To calculate the appropriate value for SPIxBRG use the tables provided in the Reference Manual or the following formula:

Where,

FPB is the peripheral bus clock frequency
Fsck is the desired clock frequency.

SPI can operate at relatively high frequencies, in the MHz range. The master dictates the clock frequency and the slave reads the clock; therefore, a slave device never needs to configure a clock frequency.

The interrupt vector for SPIx is _SPI_x_VECTOR, where x is 2 to 4. An interrupt can be generated by an SPI fault, SPI RX conditions, and SPI TX conditions. For SPI2, the interrupt flag status bits are IFS1bits.SPI2EIF (error), IFS1bits.SPI2RXIF (RX), and IFS1bits.SPI2TXIF (TX); the enable control bits are IEC1bits.SPI2EIE (error),

IEC1bits.SPI2RXIE (RX), and IEC1bits.SPI2TXIE (TX); and the priority and sub priority bits are IPC7bits.SPI2IP and IPC7bits.SPI2IS. For SPI3 and SPI4, the bits are named similarly, replacing SPI2 with SPIx (x = 3 or 4), and with SPI3's flag status bits in IFS0, enable control bits in IEC0, and priority in IPC6; and SPI4's in IFS1, IEC1, and IPC8.

Detail of PIC32MZ SPI master API/routines.

The 144-pin PIC32MZ device has three SPI peripherals, which control through four SFRs. Many of the fields in these SFRs initiate an "event" on the SPI bus. All default bits are zero, except for one read-only bit in SPIxSTAT. In the SFRs below, x refers to the SPI peripheral number, 2, 3 or 4. The following

registers are used:

- SPIxCON: SPIx Control Register – Used to set up SPI peripheral "x"
- SPIxSTAT: SPIx Status Register – Contains the status of the SPI peripheral "x"
- SPIxBUF: SPIx Buffer Register - Used to both read and write data buffer over SPI peripheral "x"
- SPIxBRG: SPIx Baud Rate Register – Used for setting the speed of SPI peripheral "x"

For example to configure SPI peripheral "2" as a master, we need to consider SPI2Con, SPI2STAT, SPI2BRG and SPI2BUF registers. This means the two physical pins we will connect to are SDI2 (on port RE5), SDO2 (on port RC4), SCK2 (on port RG6) and SS2 (on port RA5).

Configure the speed of the SPI peripheral 2 via SPI2BRG register

As per the PIC32MZ product page we can run up to 50MHz, the lower 12 bits of SPIxBRG register determine the SPI clock frequency. To set the clock frequency to 50MHz and for compute the value of SPIxBRG, use the formula:

$$\text{SPIxBRG} = \frac{F_{PB}}{2F_{sck}} - 1$$

Where,
FPB = 100MHz is the peripheral bus clock frequency
Fsck = 50MHz is the desired clock frequency.

As result of this, set the SPIxBRG equal to "0" for 50MHz SPI clock frequency.

Implement the code to do that:

```
// Setup the SPI Master Clock Frequency
SPI2BRG = 0;   // SPI2BRG = ((Fpb/(2*Fsck)) – 1)
               // Fsck is the freq (50MHz here)
                    // Fpb is Peripheral clock freq (100 MHz)
```

SPI peripheral 2 Buffer Modes

There are two SPI buffering modes: Standard and Enhanced.

STANDARD BUFFER MODE:
The SPI Data Receive/Transmit Buffer (SPI2BUF) register is actually two separate internal registers: the Transmit Buffer (SPI2TXB) and the Receive Buffer (SPI2RXB). These two unidirectional registers share the SFR address of SPI2BUF. When a complete byte/word received, it transferred from SPISR to SPIRXB and the SPIRBF flag is set. If the software reads the SPI2BUF buffer, the SPIRBF bit cleared. As the software writes to SPI2BUF, the data is loaded into the SPITXB bit and the SPITBF bit set by hardware. As the data transmitted out of SPISR, the SPITBF flag cleared. The SPI module double-buffers transmit/receive operations and allow continuous data transfers in the background. Transmission and reception occur simultaneously in the SPISR bit.

ENHANCED BUFFER MODE:
The Enhanced Buffer Enable (ENHBUF) bit in the SPI Control (SPI2CON) register can be set to enable the Enhanced Buffer mode. In Enhanced Buffer mode, multi-element FIFO buffers are used for the transmit buffer (SPI2TXB) and the receive buffer (SPI2RXB). SPI2BUF provides access to both the receive and transmit FIFOs and the data transmission and reception in the SPISR buffer in this mode is identical to that in Standard Buffer mode. The FIFO depth depends on the data width chosen by the Word/Half-Word Byte Communication Select (MODE) bits in the SPI Control (SPI2CON) register. If the MODE field selects 32-bit data lengths, the FIFO is 4 deep, if MODE selects 16-bit data lengths, the FIFO is 8 deep, or if MODE selects 8-bit data lengths the FIFO is 16 deep. The SPITBF status bit is set when all of the elements in the transmit FIFO buffer are full and is cleared if one or more of those elements are empty. The SPIRBF status bit is set when all of the elements in the receive FIFO buffer are full and is cleared if the SPI2BUF buffer is read by the software. The SPITBE status bit is set if all the elements in the transmit FIFO buffer are empty and is cleared otherwise. The SPIRBE bit is set if all of the elements in the receive FIFO buffer are empty and is cleared otherwise. The Shift Register Empty (SRMT) bit is valid only in Enhanced Buffer mode and is set when the shift register is empty and cleared otherwise. There is no underrun or overflow protection against reading an empty receive FIFO element or writing a full transmit FIFO element. However, the SPISR bit provides Transmit Underrun (SPITUR) and Receive Overflow (SPIROV) status bit, which can be monitored along with the other status bits. The Receive Buffer Element Count (RXBUFELM) bits in the SPI Status (SPI2STAT) register indicate the number of unread elements in the receive FIFO. The Transmit Buffer Element Count (TXBUFELM) bits in the SPI Status (SPI2STAT) register indicate the number of elements not transmitted in the transmit FIFO.

The Enhanced Buffer mode enables a 16-byte deep FIFO (First in First Out) buffer, which enables us to load up 16 bytes of data before the sending even begins. In practice, this saves a huge amount of time and gives us faster transfer rates because the SPI peripheral has less time when it is doing nothing and just waiting for us to give it data to send.

Create SPI Peripheral 2master Initialization API

The implementation of the SPI2 master contains functions roughly corresponding to the primitives discussed earlier. Each function executes the primitive command and setup the SPI2 port as a master in 8-bit mode at 50MHz using the standard polarities devices expect.

The Library API routine code:

```
// Initialize SPI2 Master
void SPI2_Master_init()
{
    SPI2CONbits.ON = 0; // Turn off SPI2 before configuring
    SPI2CONbits.MSTEN = 1; // Enable Master mode
    SPI2CONbits.CKP = 1; // Clock signal is active low, idle state is high
    SPI2CONbits.CKE = 0; // Data is shifted out/in on transition from idle (high) state to active (low) state
    SPI2CONbits.SMP = 1; // Input data is sampled at the end of the clock signal
    SPI2CONbits.MODE16 = 0; // Do not use 16-bit mode
    SPI2CONbits.MODE32 = 0; // Do not use 32-bit mode (combines with the above line to activate 8-bit mode)
```

```
    SPI2BRG = 0; // Set Baud Rate Generator to 0
    SPI2CONbits.ENHBUF = 0; // Disables Enhanced Buffer mode
    // clear any existing SPI2 Interrupts.
    IEC4bits.SPI2EIE = 0;
    IEC4bits.SPI2RXIE = 0;
    IEC4bits.SPI2TXIE = 0;
    // clear any existing SPI2 events
    IFS4bits.SPI2EIF = 0;
    IFS4bits.SPI2RXIF = 0;
    IFS4bits.SPI2TXIF = 0;
    // Clear the Read Buffer over flow.
    SPI2STATbits.SPIROV = 0;
   // Configuration is done, turn on SPI2 peripheral
   SPI2CONbits.ON = 1;
return;
}
```

Create SPI master Send and Receive data API

Only the one bit of data transferred in each direction per clock cycle. When the slave writes to its SPI buffer, the data did not actually sent until the master generates a clock signal.
Both master and slave use the clock to send and receive the data. As sending and receiving happen simultaneously, the master should always read from SPI2BUF after writing to SPI2BUF.

```
// Send/Receive one byte data to the slave.
BYTE SPI2_Master_Send_ReceiveData( BYTE Data )
{
    BYTE bSPI_ReadBack; // This is temp. buffer for clearing out the SPI buffer.

    /* Make sure the receive buffer is empty be for starting the tx/rx */
    if ( SPI2STATbits.SPIRBF )
    {
        bSPI_ReadBack = (BYTE)SPI2BUF;
    }

    SPI2BUF = Data; // Copy the transmit data for write operation.

    // Check and clear read buffer overflow flag inorder to get ready before
    // sending data and receive ack for future send data.

if(SPI2STATbits.SPIROV){SPI2STATbits.SPIROV=0;}

    // wait untill the write operation completed and received ACK for it.
    // The SPIBUSY flag which return 1 if SPI is
    // transfering a data and return 0 on transfer complete or IDLE.
```

```
    while(SPI2STATbits.SPIBUSY)
    {;}

    // To handle the compiler errors.
    if (bSPI_ReadBack)
    {
        bSPI_ReadBack = 0;
    }

    return (BYTE)SPI2BUF;
}
```

NOTE:
Before calling this routine, the CS (Chip Select) signal for SPI2 needs to be set and after calling the CS signal for SPI2 needs to be clear out.

SPI Master Lib API real world usages to interface with Micro Stepping motor controller.

This section describe application of SPI master library APIs to interface with Micro Stepping motor controller via SPI communication bus from PIC32MZ MCU.

The L6480 device, realized in analog mixed signal technology, is an advanced fully integrated solution suitable for driving two-phase bipolar stepper motors with micro stepping.

It integrates a dual full bridge gate driver for N-channel MOSFET power stages with embedded non-dissipative overcurrent protection. Thanks to a unique voltage mode-driving mode that compensates for BEMF, bus voltage and motor winding variations, the micro stepping of a true 1/128-step resolution achieved. The digital control core can generate user defined motion profiles with acceleration, deceler-ation, speed or target position easily programmed through a dedicated register set. All application commands and data registers, including those used to set analog values (i.e. current protection trip point, dead time, PWM frequency, etc.) sent through a standard 5-Mbit/s SPI. A very rich set of protections

(thermal, low bus voltage, overcurrent and motor stall) makes the L6480 device "bullet proof", as required by the most demanding motor control applications.

4.1 L6480 Micro Stepping motor controller device configuration

The descriptions of L6480 micro stepping motor controller pins are as below:

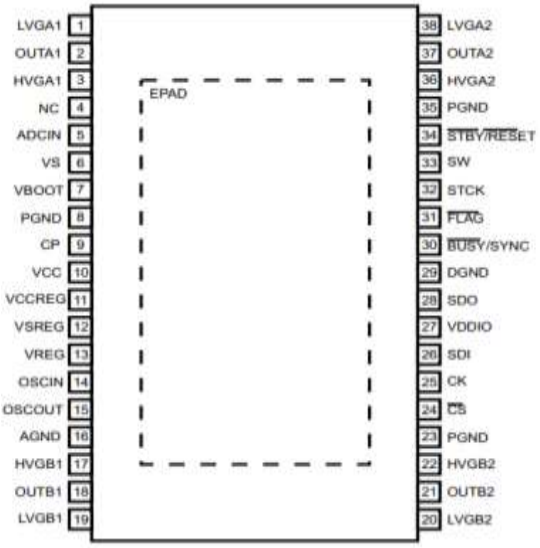


Figure 3: L6480 Pins top view

ISSN 2056-581X(Print), ISSN 2056-5828(Online)

Pin Details:

No.	Name	Type	Function
11	VCCREG	Power supply	Internal V_{REG} voltage regulator supply voltage
13	VREG	Power supply	Logic supply voltage
27	VDD	Power supply	Logic interface supply voltage
12	VSREG	Power supply	Internal V_{CC} voltage regulator supply voltage
10	VCC	Power supply	Gate driver supply voltage
14	OSCIN	Analog input	Oscillator pin1. To connect an external oscillator or clock source.
15	OSCOUT	Analog output	Oscillator pin2. To connect an external oscillator. When the internal oscillator is used, this pin can supply a 2/4/8/16 MHz clock.
9	CP	Output	Charge pump oscillator output
7	VBOOT	Power supply	Bootstrap voltage needed for driving the high-side power DMOS of both bridges (A and B).
5	ADCIN	Analog input	Internal analog to digital converter input
6	VS	Power supply	Motor voltage
3	HVGA1	Power output	High-side half-bridge A1 gate driver output.
36	HVGA2	Power output	High-side half-bridge A2 gate driver output
17	HVGB1	Power output	High-side half-bridge B1 gate driver output
22	HVGB2	Power output	High-side half-bridge B2 gate driver output
1	LVGA1	Power output	Low-side half-bridge A1 gate driver output
38	LVGA2	Power output	Low-side half-bridge A2 gate driver output
19	LVGB1	Power output	Low-side half-bridge B1 gate driver output
20	LVGB2	Power output	Low-side half-bridge B2 gate driver output
8, 23, 35	PGND	Ground	Power ground pins. They must be connected to other ground pins.
2	OUTA1	Power input	Full bridge A output 1
37	OUTA2	Power input	Full bridge A output 2
18	OUTB1	Power input	Full bridge B output 1
21	OUTB2	Power input	Full bridge B output 2
16	AGND	Ground	Analog ground. It must be connected to other ground pins.
33	SW	Logical input	External switch input pin
29	DGND	Ground	Digital ground. It must be connected to other ground pins
28	SDO	Logical output	Data output pin for serial interface
26	SDI	Logical input	Data input pin for serial interface
25	CK	Logical input	Serial interface clock
24	CS	Logical input	Chip select input pin for serial interface
30	$\overline{\text{BUSY}}$/SYNC	Open drain output	By default, the $\overline{\text{BUSY}}$ /SYNC pin is forced low when the device is performing a command. The pin can be programmed in order to generate a synchronization signal.
31	FLAG	Open drain output	Status flag pin. An internal open drain transistor can pull the pin to GND when a programmed alarm condition occurs (step loss, OCD, thermal pre-warning or shutdown, UVLO, wrong command, non-performable command).
34	STBY RESET	Logical input	Standby and reset pin. LOW logic level puts the device in Standby mode and reset logic. If not used, should be connected to V_{REG}.
32	STCK	Logical input	Step-clock input
EPAD	Exposed pad	Ground	Exposed pad. It must be connected to other ground pins.

Figure 4: L6480 Pin details

Motor Controller Commands overview

The L6480 can accept different types of commands:

- constant speed commands (Run, GoUntil, ReleaseSW)
- Absolute positioning commands (GoTo, GoTo_DIR, GoHome, GoMark)
- Motion commands (Move)
- Stop commands (SoftStop, HardStop, SoftHiz, HardHiz).

1) Constant speed commands

A constant speed command produces a motion in order to reach and maintain a user defined target speed starting from the programmed minimum speed (set in the MIN_SPEED register) and with the programmed acceleration/deceleration value (set in the ACC and DEC registers). A new constant speed command can requested anytime.(refer this figure)

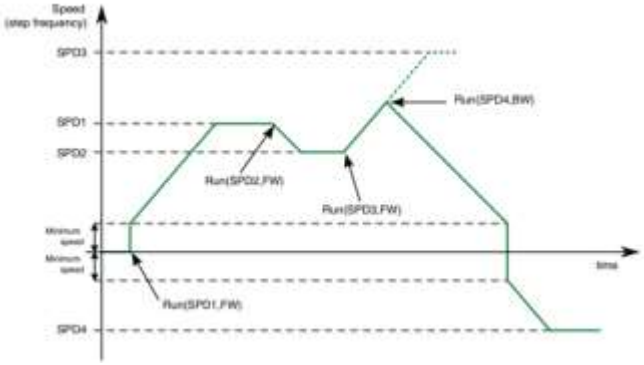
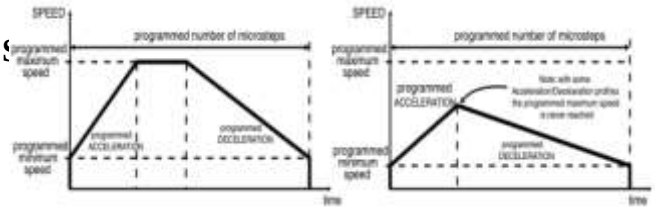


Figure 5: Constant S



2) Positioning commands

An absolute positioning command produces a motion in order to reach a user-defined position that sent to the device together with the command. The position can be reached performing the minimum path (minimum physical distance) or forcing a direction (see below figure 6). Performed motor motion is compliant to programmed speed profile boundaries (acceleration, deceleration, minimum and maximum speed). Note that with some speed profiles or positioning commands, the deceleration phase can start before the maximum speed reached.

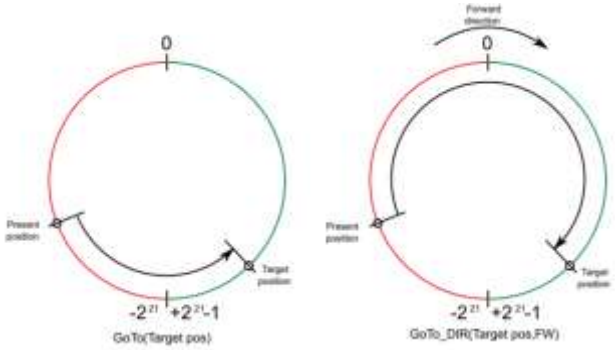


Figure 6: Positioning Commands

3) Motion commands

Motion commands produce a motion in order to perform a user-defined number of micro steps in a user-defined direction that sent to the device together with the command (see Figure 10). Performed motor motion is compliant to programmed speed profile boundaries (acceleration, deceleration, minimum and maximum speed). Note that with some speed profiles or motion commands, the deceleration phase can start before the maximum speed reached.

4) Stop commands

A stop command forces the motor to stop. Stop commands can sent anytime. The SoftStop command causes the motor to decelerate with a programmed deceleration value until MIN_SPEED value reached and then stops the motor keeping the rotor position (a holding torque is applied). The HardStop command stops the motor instantly, ignoring deceleration constraints and keeping the rotor position, (a holding torque is applied). The SoftHiZ command causes the motor to decelerate with a programmed deceleration value until the MIN_SPEED value is reached and then forces the bridges into high impedance state (no holding torque is present). The HardHiZ command instantly forces the bridges into high impedance state (no holding torque is present).

5) Step-clock mode

In Step-clock mode the motor motion defined by the step-clock signal applied to the STCK pin. At each step-clock rising edge, the motor is moved one microstep in the programmed direction and absolute position is consequently updated. When the system is in Step-clock mode the SCK_MOD flag in the STATUS register is raised, the SPEED register is set to zero and motor status is considered stopped regardless of the STCK signal frequency (the MOT_STATUS parameter in the STATUS register equal to "00").

6) GoUntil and ReleaseSW commands

In most applications, the power-up position of the stepper motor is undefined, so an initialization algorithm driving the motor to a known position is necessary. The GoUntil and ReleaseSW commands47 can be used in combination with external switch input (see Section 6.14 on page 31) to easily initialize the motor position. The GoUntil command makes the motor run at target constant speed until the SW input forced low (falling edge). When this event occurs, one of the following actions can be performed: • ABS_POS register is set to zero (home position) and the motor decelerates to zero speed (as a SoftStop command) • ABS_POS register value is stored in the MARK register and the motor decelerates to zero speed (as a SoftStop command). If the SW_MODE bit of the CONFIG register is set to '0', the motor does not decelerate but it immediately stops (as a HardStop command).

The ReleaseSW command makes the motor run at a programmed minimum speed until the SW input drives high (rising edge). When this event occurs, one of the following actions can performed:

- ABS_POS register is set to zero (home position) and the motor immediately stops (as a HardStop command)
- ABS_POS register value is stored in the MARK register and the motor immediately stops (as a HardStop command).

If the programmed minimum speed is less than 5 step/s, the motor is driven at 5 step/s

4.3. Code for PIC32MZ and L6480H communication over SPI bus.

This sample code design to configure, control and validate the micro stepping motor controller from PIC32MZ using SPI master library API. It set, read and move steps, which requested from PIC32MZ using SPI Master Library API over SPI bus interface (such as a spectroscopy/forensic/medical instrument configuration and control various components like Monochromator/Mirrors/Filters via Micro stepping motor controller for performing an experiments).

```
#include <string.h>
#include <stdio.h>
#include <GenericTypeDefs.h>
#include <p32mz2048efm144.h>
// Lib files.
#include "SPI_Master_API.h" // SPI Master API lib.
// L6480 Commands
// Send the Nothing is performed command.
void Send_NOP()
{
    BYTE bCmd = CMD_NOP;

    (void)SPIMPolPut( bCmd );

     delay_ms(1);

    return;
}
```

ISSN 2056-581X(Print), ISSN 2056-5828(Online)

```
//The GoUntil command produces a motion at SPD speed imposing a forward (DIR = '1') or

//a reverse (DIR = '0') direction. When an external switch turn-on event occurs,

//the ABS_POS register is reset (if ACT = '0') or the ABS_POS register value is

//copied into the MARK register (if ACT = '1'); the system then performs a SoftStop command.

void GoUntil( BOOL blACT, BOOL blDIR, double dSpeed, int iObjNum)

{

    BYTE bCmd = CMD_GO_UNTIL;

    const BYTE DIR_BIT = 0x01;

    const BYTE ACT_BIT = 0x08; //          ms us       ns

    const double dTick_250ns = 250.0 / 1000.0 / 1000.0 / 1000.0; //lint !e834

    double dTwoPowN28 = (double) pow((float) 2.0, (float) - 28.0);


    DWORD dwSpeed = (DWORD) ((dSpeed * dTick_250ns / dTwoPowN28) + 0.5);


    BYTE bVal2 = (BYTE) (dwSpeed / 0x10000);

    BYTE bVal1 = (BYTE) (dwSpeed / 0x100);

    BYTE bVal0 = (BYTE) dwSpeed;


    if (blDIR)
```

```
    {
        bCmd |= DIR_BIT;
    }
    if (blACT)
    {
        bCmd |= ACT_BIT;
    }

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);

    return;
}

// Executing the Run command request.
void Run( BOOL blDIR, double dwSpeed )
{
    BYTE bCmd = CMD_RUN;
    const BYTE DIR_BIT = 0x01; //          ms
us       ns
    const double dTick_250ns = 250.0 / 1000.0 /
1000.0 / 1000.0; //lint !e834
    double dTwoPowN28 = (double) pow((float)
2.0, (float) - 28.0);
```

```
    DWORD dwSpeed = (DWORD) ((dSpeed * dTick_250ns / dTwoPowN28) + 0.5);

    BYTE bVal2 = (BYTE) (dwSpeed / 0x10000);
    BYTE bVal1 = (BYTE) (dwSpeed / 0x100);
    BYTE bVal0 = (BYTE) dwSpeed;

    if (blDIR)
    {
        bCmd |= DIR_BIT;
    }

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);

    return;
}

// The ReleaseSW command produces a motion at minimum speed imposing a forward
// (DIR = '1') or reverse (DIR = '0') rotation. When SW is released (opened) the ABS_POS
// register is reset (ACT = '0') or the ABS_POS register value is copied into the MARK register
```

```
// (ACT = '1'); the system then performs a Hard-Stop command.
void Release_SW( BOOL blACT, BOOL blDIR)
{
    BYTE bCmd = CMD_GO_UNTIL;
    const BYTE DIR_BIT = 0x01;
    const BYTE ACT_BIT = 0x08;
    Send_NOP(nSPI, iObjNum);

    if (blDIR)
    {
        bCmd |= DIR_BIT;
    }
    if (blACT)
    {
        bCmd |= ACT_BIT;
    }

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);

    delay_ms(1);

    return;
```

```
}


// The GoTo command produces a motion to ABS_POS absolute position through the shortest
// path. The ABS_POS value is always in agree-ment with the selected step mode; the
// parameter value unit is equal to the selected step mode (full, half, quarter, etc.).
void GoTo(long lABSPos)
{
    BYTE bCmd = CMD_GOTO;
    DWORD dwABS_StepPos;
    BYTE bVal2;
    BYTE bVal1;
    BYTE bVal0;

    //Verify the specified absolute step position is within the valid range
    if (lABSPos > ABS_STEP_POS_MAX)
    {
        lABSPos = ABS_STEP_POS_MAX;
    }
    else if (lABSPos < ABS_STEP_POS_MIN)
    {
        lABSPos = ABS_STEP_POS_MIN;
    }
```

```
    dwABS_StepPos = (DWORD) lABSPos;
    bVal2 = (BYTE) (dwABS_StepPos / 0x10000);
    bVal1 = (BYTE) (dwABS_StepPos / 0x100);
    bVal0 = (BYTE) dwABS_StepPos;

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);

    return;
}

// The GoTo_DIR command produces a motion to ABS_POS absolute position
// imposing a forward (DIR = '1') or a reverse (DIR = '0') rotation.
void GoTo_DIR(BOOL blDIR, long lABSPos)
{
    BYTE bCmd = CMD_GOTO_DIR;
    DWORD dwABS_StepPos;
    BYTE bVal2;
    BYTE bVal1;
    BYTE bVal0;

    //Verify the specified absolute step position is
```

```
within the valid range
    if (lABSPos > ABS_STEP_POS_MAX)
    {
        lABSPos = ABS_STEP_POS_MAX;
    }
    else if (lABSPos < ABS_STEP_POS_MIN)
    {
        lABSPos = ABS_STEP_POS_MIN;
    }

    if (blDIR)
    {
       bCmd |= DIR_BIT;
    }

    dwABS_StepPos = (DWORD) lABSPos;
    bVal2 = (BYTE) (dwABS_StepPos / 0x10000);
    bVal1 = (BYTE) (dwABS_StepPos / 0x100);
    bVal0 = (BYTE) dwABS_StepPos;

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);
```

```
    return;
}

//The GoHome command produces a motion to the HOME position
//(zero position) via the shortest path.
void Go_Home()
{
    BYTE bCmd = CMD_GO_HOME;
    DWORD dwHome_Pos;
    BYTE bVal2;
    BYTE bVal1;
    BYTE bVal0;
    long lHomePos = 0;

    dwHome_Pos = (DWORD) lHomePos;
    bVal2 = (BYTE) (dwHome_Pos / 0x10000);
    bVal1 = (BYTE) (dwHome_Pos / 0x100);
    bVal0 = (BYTE) dwHome_Pos;

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);

    delay_ms(5);
```

```
    return;
}

// The move command produces a motion of N_STEP microsteps;
// the direction is selected by the DIR bit ('1' for-ward or '0' reverse).
void Move(BOOL blDIR, double nStep)
{
    BYTE bCmd = CMD_MOVE;
    const BYTE DIR_BIT = 0x01;

    BYTE bVal2 = (BYTE)( nStep / 0x10000 );
    BYTE bVal1 = (BYTE)( nStep / 0x100 );
    BYTE bVal0 = (BYTE)nStep;

    if (blDIR)
    {
        bCmd |= DIR_BIT;
    }

    (void) SPIMasterPolPut_Get(bCmd);
    (void) SPIMasterPolPut_Get(bVal2);
    (void) SPIMasterPolPut_Get(bVal1);
    (void) SPIMasterPolPut_Get(bVal0);
```

```
    return;
}

// The SoftStop command causes an immediate deceleration to
// zero speed and a consequent motor stop; the deceleration
// value used is the one stored in the DEC register.
void SoftStop()
{
    (void) SPIMasterPolPut_Get((BYTE) CMD_SOFT_STOP);

    return;
}

// The HardStop command causes an immediate motor
// stop with infinite deceleration.
void HardStop(BYTE nSPI, int iObjNum)
{
    (void) SPIMasterPolPut_Get((BYTE) CMD_HARD_STOP);

    return;
}
```

```
// The SoftHiZ command disables the power bridges (high impedance state)
// after a deceleration to zero; the deceleration value used is the one
// stored in the DEC register.
void SoftHiZ(BYTE nSPI, int iObjNum)
{
    (void) SPIMasterPolPut_Get((BYTE) CMD_SOFT_HI_Z);

    return;
}

// The HardHiZ command immediately disables the power bridges (high
// impedance state) and raises the HiZ flag.
void HardHiZ(BYTE nSPI, int iObjNum)
{
    (void) SPIMasterPolPut_Get((BYTE) CMD_HARD_HI_Z);

    return;
}

// The ABS_POS register contains the current motor absolute position
```

```
// in agreement with the selected step mode; the stored value unit
// is equal to the selected step mode (full, half,quarter, etc.).
// To Set and Get ABS position.
long GetABS_POS()
{
    long lABS_POS;
    BYTE bCmd = ( CMD_GET_PARAM | ABSOLUTE_POSITION_REG );
    uDataTypes uSPI_Data;
    const long lSIGN_BIT = 0x200000;

    uSPI_Data.l = 0;
    uSPI_Data.b[0] = 0;
    uSPI_Data.b[1] = 0;
    uSPI_Data.b[2] = 0;
    uSPI_Data.b[3] = 0;

    GetMotorCtrlParam(bCmd, &uSPI_Data.b[0], 3);

    lABS_POS = uSPI_Data.l;



    // Convert from 2's complement
    if (lSIGN_BIT & lABS_POS)
```

```
    {
        lABS_POS = ~lABS_POS + 1;
        lABS_POS &= 0x3fffff;
        lABS_POS *= -1;
    }

    return lABS_POS;
}

void SetABS_POS(WORD wUniqueID, long lABS_POS, int iObjNum)

{
    //long lABS_POS;
    BYTE bCmd = (CMD_SET_PARAM | ABSOLUTE_POSITION_REG);
    uDataTypes uSPI_Data;
    //const long lSIGN_BIT = 0x200000;

    uSPI_Data.l = lABS_POS;

    SetMotorCtrlParam(bCmd, &uSPI_Data.b[0], 3);

    return;
}
```

```
void GetMotorCtrlParam(BYTE bCmd, BYTE * bpData, BYTE bSize)
{
    BYTE bTest_Read;

    (void) SPIMasterPolPut_Get(bCmd);
    BYTE bData[3];

    while (bSize)
    { //do we need to pump the data through ?
        bpData[ --bSize ] = SPIMaster-PolPut_Get(CMD_NOP);
        bData[bSize] = bpData[ bSize ];
    }

    if (bData[0] == 0x00)
    {
        bTest_Read = bData[0];
    }
    else
    {
        bTest_Read = bData[1];
    }

    // To handle the compiler errors.
```

```
    if (bTest_Read == 0)
    {
        bSize = 0;
    }

    return;
}

void SetMotorCtrlParam( BYTE nSPI, BYTE bCmd, const BYTE * bpData, BYTE bSize, int iObjNum)
{
    (void) SPIMasterPolPut_Get(bCmd);

    while (bSize)
    {

        (void) SPIMaster-PolPut_Get(bpData[ --bSize ]);
    }

    return;
}
```

CONCLUSION

These costume SPI master Library APIs can be useful for any PIC32 embedded system based project.

There could be a numerous condition in real world applications where we need this SPI Master Library APIs to communicate with flash memory, sensors, real-time clocks (RTCs), analog-to-digital converters, motor drier and more via SPI communication bus from PIC32 MCU. These SPI Master Library APIs implemented successfully using PIC32MZ MCU hardware and MPLAB software stack that includes MPLAB X IDE, Tool chain and Harmony framework. The above example of Micro stepping Motor Controller L6480 from PIC32MZ demonstrated usages of SPI master library APIs.

References

1) **"**Embedded Computing and Mechatronics with the PIC32 Microcontroller 1st Edition" by Kevin Lynch (Author), Nicholas Marchuk (Author), Matthew Elwin (Author)
2) Data Sheet of "PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Family " by Microchip.
3) Abhishek N. Patel, PIC32MZ Milliseconds and Microseconds Delay Routines, Electrical and Electronic Engineering, Vol. 9 No. 2, 2019, pp. 41-44. doi: 10.5923/j.eee.20190902.03.
4) Abhishek N. Patel, PIC32MZ Custom I2C Master Communication API Library, Electrical and Electronic Engineering, Vol. 9 No. 2, 2019, pp. 45-52. doi: 10.5923/j.eee.20190902.04.
5) L6480 Micro stepping motor controller with motion engine and SPI by ST "https://www.st.com/resource/en/datasheet/l6480.pdf#page=44&zoom=100,0,596"
6) SPI tutorial by Corelis https://www.corelis.com/education/tutorials/spi-tutorial/
7) SPI Communication between PIC32s by "nxr" (Neuroscience and Robotics Lab).
8) SPI guide by sparkfun
9) Introduction to SPI interface by Analog Dialogue