**Getting Started with NI SPI API**

**Contents**

Introduction  
Requirements  
API Overview  
Using the API  
Debugging  
Customizing the API to Add 2 SPI Data Lines  
Customizing the API to Have a Busy or Ready Signal

# Introduction

The NI SPI API allows you to quickly get SPI communication up and running on your NI System on Module. This document will walk you through the process of setting up the SPI Engine and configuring it to work with your SPI module. It will also cover common troubleshooting techniques and ways you can further customize this API to work with other SPI configurations.

# Requirements

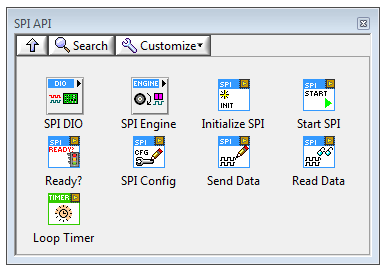
Software

* LabVIEW Full Development System
* LabVIEW Real Time
* LabVIEW FPGA

Hardware

* NI System on Module
* NI 9651
* Target SPI Module

**API Overview**

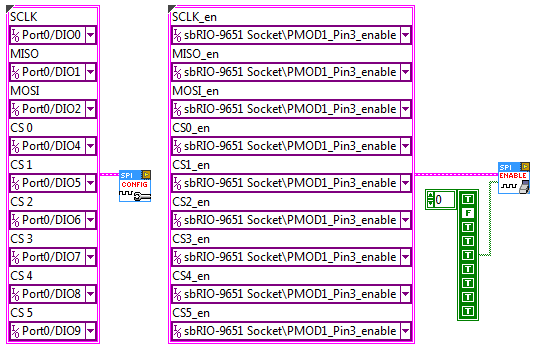


There are different components of the SPI API to control the communication between the FPGA and SPI module. The core component of the SPI API that controls the digital communication between the hardware components is the SPI Engine. The SPI Engine can be configured to use specific FPGA DIO lines to act as the MISO, MOSI, CS, and CLK lines for the SPI communication. The VIs in the SPI DIO sub-palette are used to configure which DIO lines will be used by the SPI Engine, as well as read and write to these DIO lines.

Alongside the SPI Engine, there are VIs available in the palette that are used to queue up commands to be sent out on the SPI bus. The SPI Engine uses FIFOs to receive these commands to write SPI messages out on the bus and then return messages read back from the SPI Slave(s).

* Select DIO lines

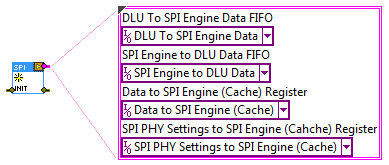
When first using the API, the initial step before starting the SPI communication is to setup the digital I/O lines that will be used for the data transmission. The Digital Config VI from the SPI DIO sub-palette can be used to select the lines. Placing this VI and creating a control for the ‘SPI DIO Lines’ input will allow specific DIO lines to be assigned to the different required SPI I/O lines, as shown on the left below.



* Configure enable lines

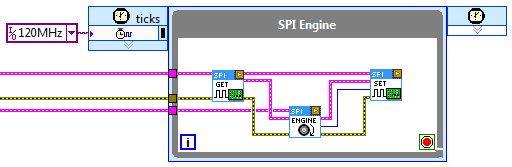
If you are using FPGA hardware that requires the DIO lines to either be enabled for output or disabled for input, such as the sbRIO-9651 used in the SPI API examples, you can use the Enable Digital VI from the same palette to set these enable states, shown on the right above.

* Initialize Memory Items



The Initialize VI sets up all of the memory items that the SPI Engine and Command Queuing VIs will use to communicate between one another.

* Setup SPI Engine

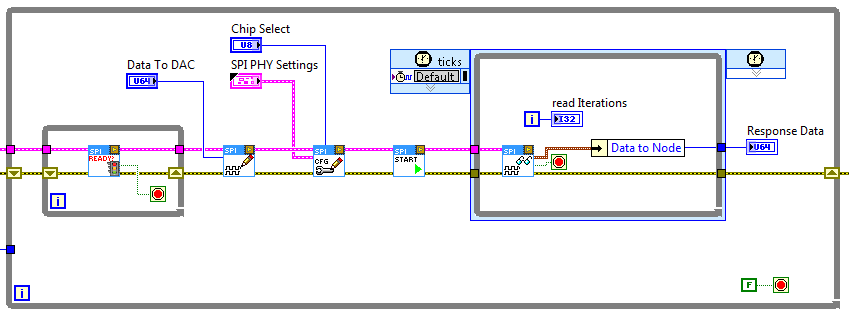


The SPI Engine runs within a Single-Cycle Timed Loop to read and write data to the DIO lines defined in the previously mentioned VIs. It uses the memory items created by the Initialize VI to both receive commands queued up in other loops (discussed in later section), receive data that needs to be written onto the SPI bus, and return data read from Slaves on the bus.

The first VI encountered in the SPI Engine above, the Read Digital Lines VI, reads the current state of the input SPI lines (MISO line) to pass into the SPI Engine. The next VI, the SPI Engine VI, is the actual engine that processes commands and controls the DIO lines for output. Finally, the last VI writes data to the output digital lines (MOSI, CS, and CLK) to communicate commands to Slave SPI devices.

* Setup Command Queuing loop

With the SPI Engine in place, the next step in using the API is to setup the Command Queuing loop where commands can be sent to the SPI Engine via the memory items created by the Initialize VI. In this loop we can choose to write out specifically formatted messages to our Slave SPI devices, and then read their raw responses.



The following discusses the purpose of each VI in the above image.

Ready.png The Ready VI checks to ensure that there is enough room to queue up new commands to the SPI Engine.

SPI Write.png The SPI Write VI takes a data set from the user that will be sent to the SPI Engine for transmitting on the MOSI line.

SPI Config.png The SPI Config VI takes configuration data from the user that describes how the SPI Engine should transmit the data provided by the SPI Write VI. The configuration options define the length of the data in bits to be transmitted, the Chip Select line to use, etc.

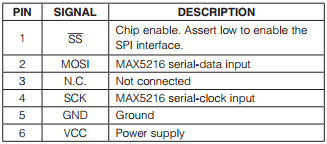
Start.png The SPI Start VI takes the information provided by the SPI Write and SPI Config VIs and transfers it to the SPI Engine.

SPI Read.png The SPI Read VI polls the SPI Engine to read the MISO response from the selected SPI Slave.

# Using the API

The SPI API is easy to configure for any SPI module that you would like to communicate with. This walk through will use the MAX5216 DAC as an example. The example code included with the API is based off this same chip. You will need to follow this general procedure for any SPI chip you may use.

The first step is to plug the SPI module into one of the SPI sockets on your sbRIO. For this example, we will have the MAX5216 plugged into PMOD1 on the sbRIO. After this, you must configure the Chip Select DIO Lines, Enable Lines, and Enable array used in the SPI session based on the information contained in the user manual. From the manual for the [MAX5216PMB1 Peripheral Module](http://datasheets.maximintegrated.com/en/ds/MAX5216PMB1.pdf) we can find the following table which shows how to configure these settings:



The constants below are populated based on the table above. SCLK (abbreviated as SCK in the chart above) is pin 4 on the PMOD connector. Since we are using connector PMOD1, we will set the SCLK setting for the Chip Select DIO Lines cluster to correspond to Pin 4 on that connector. Since SCLK is a clock output, we select the output version of that pin. We must also select the Enable Line that will allow us to use that pin. Set SCLK\_en to the enable pin for Pin 4 of PMOD1 in order to do this. Last, we must set the first element of the Enable array to True. This defines that particular pin as an output.

Continue in a similar manner for the other pins.

MISO – Since the DAC is an output only device, there is no input. This pin is N.C. (no connect) in the chart above. We still setup the PMOD lines for this in the image below to demonstrate how to do so, even though it is not necessary in this case. Set MISO to PMOD1 Pin3 In. We must also set the enable line for PMOD1 Pin3 and set element two of the Enable array to false, defining this pin as an output pin.

MOSI – Set the Chip select DIO Line set to PMOD1 Pin2, Enable Line MISO\_en to PMOD1 Pin2, and Enable array element 3 to false.

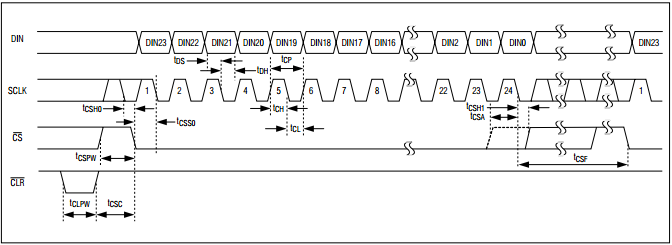
Chip Select (Abbreviated as SS above, CS below) – Set the Chip select DIO Line set to PMOD1 Pin1, Enable Line CS0\_en to PMOD1 Pin1, and Enable array element 4 to false.

CS 1 – CS 5 – These other pins are available for use when more than one SPI chips are in use. For this example it is not important what they are wired to, as only one chip is in use.

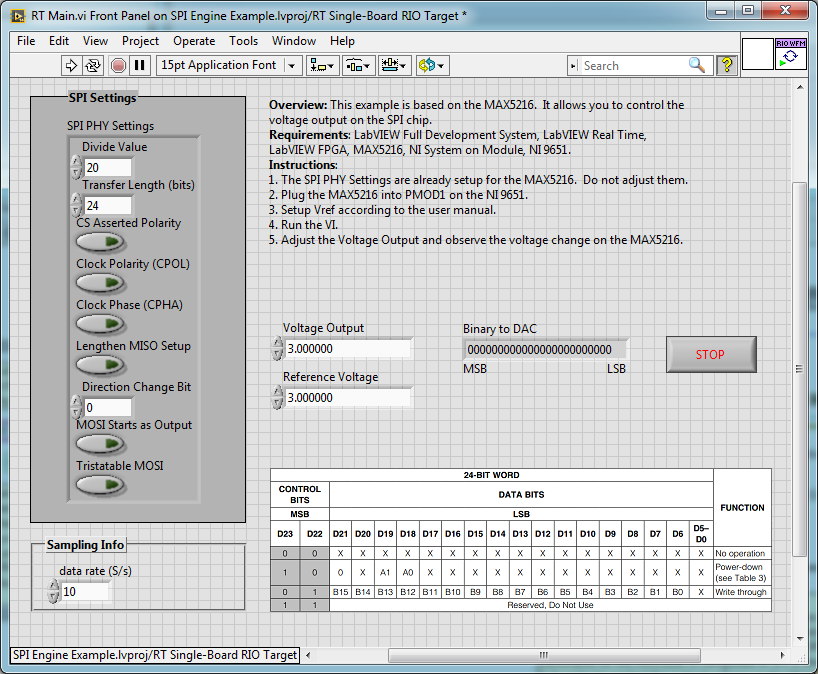
GND and Vcc – These correspond to pins 5 and 6 on the connector already, and do not need to be defined in the cluster constant below.



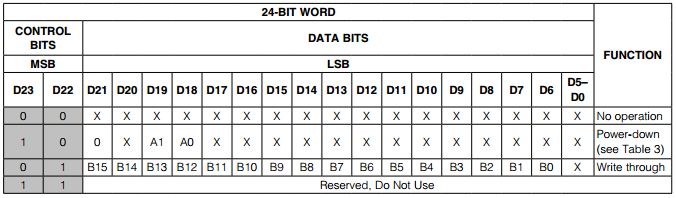
Giving these constants to the Config SPI Digital Lines.vi and the Enable SPI Digital Lines(CLIP).vi within FPGA Main.vi of the example program, is all that is necessary to configure the SPI session. The other information we need in order to properly control this device is contained within the [MAX5216 Data Sheet](http://datasheets.maximintegrated.com/en/ds/MAX5214-MAX5216.pdf). In particular, what we need to know is how the SPI data packet is formed. On page 4 of the user manual we will find this image:



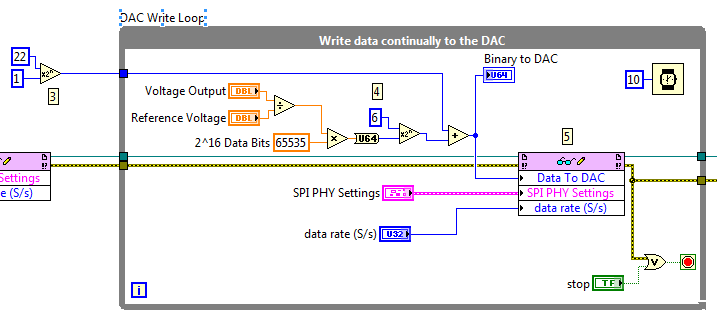
From this we can tell that teach time the chip select line is asserted, 24 bits of data are read into the SPI chip from the digital input lines (Din, or MOSI). We can also tell that the clock polarity is non-inverted, as the rising edge of the clock aligns with the other signals. The SPI PHY Settings control passes this information into the SPI engine each iteration of the data transmission loop. In order to allow maximum flexibility while testing and setting up code, this information is configured as a control. This allows the real time VI to set this information during run time instead of requiring an FPGA recompile. Open RT Main.vi and enter the settings in the front panel as shown below:



Next we need to know the structure of the commands that the SPI engine will send out to the SPI chip. This information is contained for the MAX5216 in the truth table on page 13 of the data sheet seen below (it is also shown in the front panel above for convenience):



Since the only operation we intend to perform with this chip is a simple output, we only need to implement the write through operation shown as the third option in the table. The code below demonstrates how to dynamically send any voltage (within the output range of the MAX5216) to this chip using the write through command:



Each write command contains a 1 in the 22nd bit position. This is calculated at the beginning of the code, and added to each output. The output voltage is then calculated as Vout/(Vref\*65535) (I.E. Desired voltage/(some known constant input voltage \*2^16 data bits). This formula is defined in the MAX5216 data sheet. Since the last 6 bits of the 24 bit data packet are not used, the desired output voltage is bit shifted by 6 before the control code bit is added in. This data is then sent to the FPGA.

# Debugging

Simulation VI

Scoping the lines