**Application Note**

Using SPI to Create a Configurable System

Author: Stefan Turkowski

Serial Peripheral Interface Protocol is used in many systems to control all kinds of subsystems. A protocol is simply a set of rules put in place to ensure that every circuit using SPI will be able to be controlled. If your code and your circuit follows these rules, you should have no trouble controlling many devices with one microcontroller. See [Wikipedia](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus) for more information.

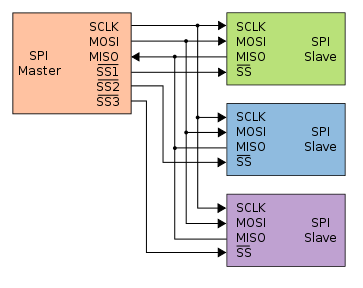
**Benefits**

* On-the-fly parameter adjustments (Such as digital potentiometers)
* DAC operation made easy
* IC control (such as Infineon BGT24 or power amplifiers)

**Links to useful Chips**

* [Single potentiometer](http://www.digikey.com/product-detail/en/AD5160BRJZ5-RL7/AD5160BRJZ5-RL7CT-ND/1718628)
* [Dual potentiometer](http://www.digikey.com/product-detail/en/AD5142BRUZ100/AD5142BRUZ100-ND/3777413)
* [Digital to Analog Converter](http://www.digikey.com/product-detail/en/MCP4921T-E%2FMS/MCP4921T-E%2FMSCT-ND/736001) (Although a DAC with 16 bits instead of 12 is recommended because will produce a cleaner triangle wave)
* [Infineon BGT24](http://www.infineon.com/dgdl/BGT24MTR11_v3.1.pdf?fileId=db3a304339dcf4b10139def491930214)

**How it Works**



**Understanding the diagram above**

* Your microcontroller is the “master” because it controls the clock and slave selects.
* Your digital potentiometers and DAC are your “slaves”. You can only tell one slave what to do at one time. You control which slave you are talking to with the slave select (chip select, SS, ect…) pins.
* Master out, slave in (MOSI) is used to transmit data from the master to the slave.
* Master in, slave out(MISO) is used to transmit data from the slave to the master. You will probably not need to use this pin though.

**Communicating**

* You must set SS low to talk to that slave.
* To talk to a slave, you may use a command like:
  + SSIDataPutNonBlocking(SSI2\_BASE, character); *This is included in TivaWare, which you should download.*
  + writeData(0x18); *A function found in our code, which sends 00011000 to the slave. It is recommended that you use this function.*
* The commands will start the clock, send data at the clock rate, and turn off the clock. The data is loaded into internal registers on each IC, so that the data is stored even after the clock is turned off. Every slave that has its SS high will ignore all data sent.
* You must set SS high once you are done talking to a slave.

**Initializing Everything**

* Initializing all the peripherals on a Tiva is the most difficult part. It must be done before any data can be sent. If you use the code that I provide, and understand which ports and pins you are using, it should not be a problem. Here are the important lines of code:
* ROM\_SysCtlPeripheralEnable(SYSCTL\_PERIPH\_SSI2); enables SSI port 2. There is also SSI port 0 and SSI port 1.
* ROM\_SysCtlClockSet(SYSCTL\_SYSDIV\_2\_5 | SYSCTL\_USE\_PLL | SYSCTL\_XTAL\_16MHZ | SYSCTL\_OSC\_MAIN); The internal clock is 200 MHz, but the highest division you can use is 2.5.
* SSIConfigSetExpClk(SSI2\_BASE, SysCtlClockGet(), SSI\_FRF\_MOTO\_MODE\_0, SSI\_MODE\_MASTER, SysCtlClockGet()/20, 8); creates the settings for SSI2. SysCtlClockGet()/2 is the fastest that the SSI can operate (at half the speed of the processor), but it is recommended to use a higher division because some SPI devices cannot operate at such a high clock frequency anyway. This SSI is set to 8 bits, but 4 and 16 can also be used. If 16 bits is used, you can use writeData(0xF5AB);

**Before You Design Your PCB**

* BREADBOARD TEST EVERYTHING. Understand how communicating with your ICs works before making final decisions.
* Make sure that your SPI traces to not go under any signals, especially RF signals. SPI generates a lot of noise, and it should be turned off when possible.
* If possible, use SPI1 and SPI2, because talking to multiple devices with one SPI line does not always work out as cleanly as it sounds.

**Caution**

* Each IC has its own requirements for SPI, including max clock rate and mode:
  + You must send data under a certain frequency for each chip. This can be changed on the fly if you need to with SSIConfigSetExpClk(SSI2\_BASE, SysCtlClockGet(), SSI\_FRF\_MOTO\_MODE\_0, SSI\_MODE\_MASTER, **SysCtlClockGet()/50**, 8);
  + SPI mode determines if the data is loaded in on the rising or falling edge of the clock. There are four different modes. Most chips appear to be mode 0, but you should definitely double check the data sheet.

**Tiva or Stellaris Code**

/\*

Triangle wave and sync pulse generator to control a VCO for FMCW radar. This code also initializes the BGT24, and set a digital potentiometer for coarse tune.

The MPC4921 DAC is used to generate a triangle wave with a period of 40ms.

For use with Tiva or Stellaris Launchpad

Stefan and Joe

\*/

#include <stdint.h>

#include <stdbool.h>

#include "inc/tm4c123gh6pm.h"

#include "inc/hw\_types.h"

#include "inc/hw\_gpio.h"

#include "inc/hw\_memmap.h"

#include "inc/hw\_nvic.h"

#include "driverlib/sysctl.h"

#include "driverlib/rom.h"

#include "driverlib/rom\_map.h"

#include "driverlib/gpio.h"

#include "driverlib/pin\_map.h"

#include "driverlib/ssi.h"

#define NUM\_SSI\_DATA 1

uint32\_t pui32DataTx[NUM\_SSI\_DATA];

uint32\_t pui32DataRx[NUM\_SSI\_DATA];

uint32\_t ui32Index;

void writeCommand(uint8\_t c) //Not needed in this code, but might be useful later.

{

while(ROM\_SSIDataGetNonBlocking(SSI2\_BASE, &pui32DataRx[0]))

{

}

// We want to set DC low because we are trying to write a command

//ROM\_GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_6, 0); NOT NEEDED

// We only need to put(). We don't need to get() because we are not

// looking for information from the Display

SSIDataPutNonBlocking(SSI2\_BASE, c);

while(SSIBusy(SSI2\_BASE)) //Hopefully this step will not be needed when integrated into a code running with interrupts

{

}

}

void writeData(uint8\_t c)

{

while(ROM\_SSIDataGetNonBlocking(SSI2\_BASE, &pui32DataRx[0]))

{

}

// If we want to set DC (not direct current) high because we are trying to write data

// DC is a GPIO: We are choosing PB6

//ROM\_GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_6, GPIO\_PIN\_6); Not needed

// We only need to put(). We don't need to get() because we are not

// looking for information from the OLED

SSIDataPut(SSI2\_BASE, c); //Does the whole SPI process EXCEPT Slave Select low

while(SSIBusy(SSI2\_BASE)) //Hopefully this step will not be needed when integrated into a code running with interrupts

{

}

}

char data = 0;// A byte is an 8-bit number

unsigned int outputValue = 0;// A word is a 16-bit number

void setup()

{

ROM\_SysCtlPeripheralEnable(SYSCTL\_PERIPH\_SSI2); //for CLK and data

ROM\_SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOC); //enable slave select ports for DAC and Infineon

GPIOPinTypeGPIOOutput(GPIO\_PORTC\_BASE, GPIO\_PIN\_6); //Enable DAC SS pin as GPIO

GPIOPinTypeGPIOOutput(GPIO\_PORTC\_BASE, GPIO\_PIN\_4); //Enable Infineon SS pin as GPIO

GPIOPinTypeGPIOOutput(GPIO\_PORTC\_BASE, GPIO\_PIN\_5); //Enable Digital Pot SS pin as GPIO

ROM\_SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOB); //Still enabling

GPIOPinConfigure(GPIO\_PB7\_SSI2TX); //MOSI transmit data

//GPIOPinConfigure(GPIO\_PB5\_SSI2FSS); //Predefined SS not needed

GPIOPinTypeSSI(GPIO\_PORTB\_BASE, GPIO\_PIN\_4 | GPIO\_PIN\_7); // Configure SSI Pins

ROM\_SysCtlClockSet(SYSCTL\_SYSDIV\_2\_5 | SYSCTL\_USE\_PLL | SYSCTL\_XTAL\_16MHZ |

SYSCTL\_OSC\_MAIN); //80 MHz

SSIConfigSetExpClk(SSI2\_BASE, SysCtlClockGet(), SSI\_FRF\_MOTO\_MODE\_0, SSI\_MODE\_MASTER, SysCtlClockGet()/20, 8); //Running SSI clock at 40 MHz. Should be too fast...

GPIOPinConfigure(GPIO\_PB4\_SSI2CLK); //Clock COnfigure

ROM\_SSIEnable(SSI2\_BASE);

//GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_5, 0); //Set predefined SS to 0 (if we used it)

ROM\_SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOF); //Enable onboard LED port

//ROM\_GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_1); //Enable onboard LED pins for testing

//ROM\_GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_2);

//ROM\_GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

//ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_5, 0); //SS low for Pot

//writeData(0x00); //For Pot

//ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_5, GPIO\_PIN\_5); //SS High for Pot

//Turning on Infineon

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_4, 0); //SS low

writeData(0x00); // Send the upper byte for BGT24

writeData(0x18); // Send the lower byte for BGT24

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_4, GPIO\_PIN\_4); //SS High

}

int highByte(int x)

{

return (0xff00 & x)>>8;

}

int lowByte(int x)

{

return (0x00ff & x);

}

int main(void)

{

setup();

for(;;)

{

int a = 0;

// Rising edge of the triangle wave

while (a <= 4080)

{

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_2, GPIO\_PIN\_2); //Turn LED on for testing Purposes

outputValue = a;

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_6, 0); //SS low

data = highByte(outputValue); // Take the upper byte

data = 0x0F & data;

data = 0x30 | data;

writeData(data); // Send the upper byte

data =lowByte(outputValue); // Shift in the 8 lower bits

writeData(data); // Send the lower byte

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_6, GPIO\_PIN\_6); //SS High

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_2, 0); //Turn LED off for testing Purposes

a = a+64;

}

// Falling edge of the triangle wave, very similar to the above

int b = 4080;

while (b >= 64)

{

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_2, GPIO\_PIN\_2); //Turn LED on for testing Purposes

outputValue = b;

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_6, 0); //SS LOW

data =highByte(outputValue);

data = 0x0F & data;

data = 0x30 | data;

writeData(data);

data =lowByte(outputValue);

writeData(data);

ROM\_GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_6, GPIO\_PIN\_6); //SS HIGH

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_2, 1); //Turn LED on for testing Purposes

b = b-64;

}

}

}