EECS 452 Lab 7:
SPI, I2S on C5515 and DE2-70

In this lab you will work more with the SPI and I2S protocols. Specifically, you will learn how to send data between the PMODs and the C5515 and how to send data between C5515 and DE2-70 devices. Finally, you will implement an application involving the C5515, the DE2-70, and real time audio processing.

Note: In this lab and in general, when using communication protocols it is often necessary to set the binary state of configuration registers that decide multiple communication settings. For example, the I2SCTRL registers are 16 bits long and decide enable, mono/stereo, data delay, DSP/I2S format, etc. A best practice is to #defines for each of these configurations and then finally take the logical OR of them. This improves readability and helps when debugging or changing parameters later on.

1. Introduction to SPI

The SPI (Serial Peripheral Interface) is 4-pin port that a master device and a slave device can send data across.

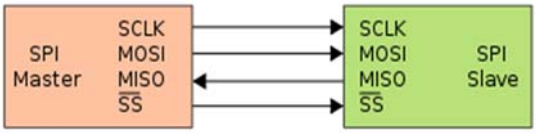

Figure 1: SPI port / signal reference on master and slave device

The SPI bus specifies four logic signals:

- SCLK: serial clock (output from master)
- MOSI: master output, slave input (output from master)
- MISO: master input, slave output (output from slave)
- SS: slave select (active low, output from master)

C5515

In the first part of this lab the C5515 stick will act as the master device and the DE2-70 will act as the slave device. Communication between master and slave is bi-directional. The key differences between master and slave are that the master is responsible for setting the clock for communication and that multiple slaves connect to a single master.

We need to go through some initializations to configure the C5515 to act as master device.

- Enable the SCLK output
- Set the SCLK frequency
- Configure slave chip select polarity
- Set character length
- Set interrupts

Step 1: Enable the SCLK output

First we need to enable the C5515 SCLK output. The SPI Clock Control Register (SPICCR) contains the clock enable that we can set to 1. (See Page 25 of SPI Manual)

Step 2: Set the SCLK frequency

Next we need to set the SCLK frequency. The C5515 SPI module contains a programmable clock divider that cuts the SPI Input Clock by some fraction and outputs the reduced frequency clock as the SCLK. The SPI Input Clock passed to the module is the same clock the C5515 CPU runs on of 100 MHz.

To ensure the SPI module communicates properly we need the SPI Input Clock frequency to be at least four greater than the SCLK frequency. (Refer to Section 2.5 on pages 12-14 in the SPI Manual to find out why)

To set the SCLK frequency, we need to configure the SPI Clock Divider Register (SPICDR) to set the CLKDV value. Whatever the CLKDV value is, the frequency of the SPI Input Clock will be divided by CLKDV + 1. For example, setting the CLKDV value to 3 will divide the SPI Input Clock by 4 and the SCLK will have frequency 25 MHz.

- Remember that this is the minimum amount we need to divide the input clock by, so the value of CLKDV should be ≥3.
- When CLKDV is odd, the duty cycle of SCLK is 50%
- When the CLKDV is even, the duty cycle of SCLK is more complicated (See Section 4.1 on pg. 25).

Step 3: Configure slave chip select polarity

The SPI module on the C5515 allows for up to four slave devices (referenced as slave 0, 1, 2, and 3). For the first part of this lab you will focus on communicating between the C5515 and just one slave DE2-70. Then later on we will experiment with communicating between multiple slave devices.

The SPI Device Configuration Registers (SPIDCR1 & SPIDCR2) store the communication configurations for each slave device. Refer to Figure 2 for the register locations for slave device 1 or consult the manual on pages 26 & 27 for information on configuration locations for all four slave devices.

Figure 15. Device Configuration Register 1 (SPIDCR1)

15 13	12 11	10	9	8	7 5	4 3	2	1	0
Reserved	DD1	CKPH1	CSP1	CKP1	Reserved	DD0	CKPH0	CSP0	CKP0
R-0	RW-0	RW-0	RW-0	RW-0	R-0	RW-0	RW-0	RW-0	RW-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

15-13	Reserved	0	Reserved.
12-11	DD1	0-3h	Data delay for chip select 1 pin (SPI_CS1).
		0	First SPI_CLK edge is delayed 0.5 clock cycles from SPI_CS1 assertion.
		1h	First SPI_CLK edge is delayed 1.5 SPI clock cycles from SPI_CS1 assertion.
		2h	First SPI_CLK edge is delayed 2.5 SPI clock cycles from SPI_CS1 assertion.
		3h	First SPI_CLK edge is delayed 3.5 SPI clock cycles from SPI_CS1 assertion.
10	CKPH1		Clock phase for chip select 1 pin (SPI_CS1). The clock phase bit, in conjunction with the clock polarity bit (CKP1), controls the clock-data relationship between master and slave.
		0	When CKP1 = 0, data shifted out on falling edge, input captured on rising edge. When CKP1 = 1, data shifted out on rising edge, input captured on falling edge.
		1	When CKP1 = 0, data shifted out on rising edge, input captured on falling edge. When CKP1 = 1, data shifted out on falling edge, input captured on rising edge.
9	CSP1		Polarity for chip select 1 pin (SPI_CS1).
		0	Active low.
		1	Active high.
8	CKP1		Clock polarity inactive state for the clock pin during accesses to chip select 1.
		0	When data is not being transferred, a steady state low value is produced at the SPI_CLK pin.
		1	When data is not being transferred, a steady state high value is produced at the SPI_CLK pin.
7-5	Reserved	0	Reserved.

Figure 2: SPIDCR1 that stores communication parameters for slave 0 and 1

Together, the clock phase and clock polarity configurations specify the SPI mode of communication (See Section 2.5 for SPI modes). For our basic initial experiment we will focus on the basics and use SPI Mode 0: active low clock polarity and data shifted out on falling edge, input captured on rising edge. **SPI modes are important to understand as they are incompatible with each other so if two devices are using different SPI modes they will not understand each other.** When you are trying to use a device that uses SPI, be sure to know what SPI mode it uses.

Notice that the C5515 stores these four configurations for each of the four slave devices **independently**, giving us the flexibility to communicate with multiple slave devices with different configurations from one master. We will attempt exactly this later on in the lab. For our first experiment, we only need to worry about setting the chip select polarity of the slave device we want to communicate with.

Data and status registers

Before discussing the last step of setting interrupts, we should look at how the C5515 stick handles receiving and propagating data.

Figure 9. Data Shift Process

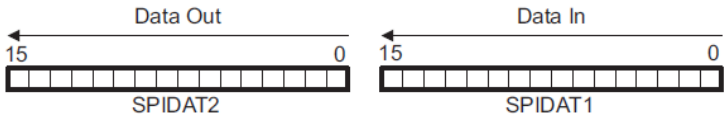

Figure 3: SPIDAT shift register

The C5515 uses SPIDAT1 and SPIDAT2 together as one 32-bit shift register. SPI incoming data from slave devices will be shifted into the SPIDAT1 register at the LSB, and SPI outgoing data will be shifted out of the SPIDAT2 from the highest significant bit. Both registers shift contents leftward. Notice that to propagate some data, our program needs to place that data in SPIDAT2 and it will be transferred out one **character** at a time until the entire **frame** is transferred. A **character** refers to some collection of 1-32 bits. A **frame** refers to the entire set of characters. It is also important to note that while Figure 3 makes it seem as if SPIDAT1 and SPIDAT2 are separate entities, the C5515 will read/write from them as two halves of a continuous 32-bit shift register. If character length exceeds 16 bits, then the MOSI data spills over to SPIDAT1's MSBs and the MISO data comes in and fills up all of SPIDAT1 and the LSBs of SPIDAT2.

There are two registers SPICMD1 and SPICMD2 that allow us to set frame and character size. We can

Figure 17. Command Register 1 (SPICMD1)

15	14	13	12	11	0
FIRQ	CIRQ	Reserved		FLEN	
RW-0	RW-0	R-0		RW-0	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 9. Command Register 1 (SPICMD1) Field Descriptions

Bit	Field	Value	Description
15	FIRQ		Frame count interrupt enable.
		0	No interrupt generated at the end of the frame count.
		1	Interrupt generated at the end of the frame count.
14	CIRQ		Character interrupt enable.
		0	No interrupt generated at the end of the character transfer.
		1	Interrupt generated at the end of the character transfer.
13-12	Reserved	0	Reserved.
11-0	FLEN	0-FFFh	Frame length bits. These bits are used to specify the length of entire transfer.
			The total number of characters transferred equals FLEN + 1.
			For example, if FLEN = 63, a frame consists of a total of 64 characters.

Figure 4: SPICMD1

access SPICMD1 during initialization to set the number of characters in a frame. As shown in Figure 4, we need to set FLEN to do this. For a frame size of 1 character for our first experiment, we will use a frame size of 1 character, so we'd leave FLEN at 0. To set the character interrupt we'll toggle CIRQ to 1.

We will not set SPICMD2 during initialization, but instead change it multiple times during program runtime. This is because whereas SPICMD1 was responding for static settings, SPICMD2 stores settings that are more dynamic in nature (for example, we need to be able to switch between reading and writing modes during runtime).

Figure 18. Command Register 2 (SPICMD2)

15 14	13 12	11 8	7 3	2	1 0
Reserved	CSNUM	Reserved	CLEN	RSV	CMD
R-0	RW-0	R-0	RW-0	R-0	RW-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 10. Command Register 2 (SPICMD2) Field Descriptions

Bit	Field	Value	Description
15-14	Reserved	0	Reserved.
13-12	CSNUM	0-3h	Device select. Sets the active chip select for the transfer.
		0	Chip select 0 is active.
		1h	Chip select 1 is active.
		2h	Chip select 2 is active.
		3h	Chip select 3 is active.
11-8	Reserved	0	Reserved.
7-3	CLEN	0-1Fh	Character length. Sets the transfer size of the individual transfer elements from 1 to 32 bits.
			The character length is set to CLEN + 1.
			For example, if CLEN = 7, the character length is set to 8 bits.
2	Reserved	0	Reserved.
1-0	CMD	0-3h	Transfer command bits. These bits specify the type of transaction being used.
		0	Reserved.
		1h	Read.
		2h	Write.
		3h	Reserved.

Figure 5: SPICMD2

Step 4: Set interrupts

Finally, once we've configured the C5515 to fire interrupts after every successful character transfer using SPICMD1, we need to capture those interrupts in functions. The way to do this is exactly what we did in previous labs. It's good to note that the interrupt fires when we succeed in sending a message over SPI.

DE2-70

For this lab we will use the DE2-70 as a slave device to the C5515 master (the C5515 stick cannot act as a slave). Later on in these experiments we will show how to use the DE2-70 as master to other DE2-70 devices.

In the lab the C5515 Breakout Board simplifies how to connect the ports on the C5515 stick to the DE2-70. The Breakout Board exposes the SPI pins on the C5515 and the GPIO pins on the DE2-70 to jumper connections between the two devices. Please consult the schematic on DE270Break.pdf to understand how the GPIO pins are mapped to the breakout board when connecting the jumper cables.

Like on the C5515, we need to go through some initializations to configure the C5515 to act as slave device. However, whereas TI gives us dedicated registers that we can understand and bit toggle, on the DE2-70 we need to implement our own modules to handle the signals going across the GPIO pins.

- Detect the SCLK
- Detect the slave select
- Collect the slave MOSI signal
- Create the MISO signal

Step 1: Detect the SCLK

Being the slave device, the DE2-70 does not set the communication frequency. Instead, our DE2-70 must check the signal on the GPIO pins for the rising and falling edges of the master SCLK.

Step 2: Detect the slave select

If we use SPI mode 0, then the master will set slave select GPIO low when it starts to communicate and set slave select GPIO high when it is done communicating. As a slave device, the DE2-70 must be aware of when communication starts, if communication is active, and when communication stops. Just like with the SCLK, it is important then to use a **shift register** to track the transition of the slave select GPIO from high to low and vice versa.

Step 3: Collect the MOSI signal

We must constantly check if slave select is active, and if it is active, depending on the communication mode, we must collect what comes across the MOSI channel into a shift register. When slave select goes from active to low, then we must pick up what we collected in the shift register into some wire to output as the MOSI message.

Step 4: Create the MISO signal

To send messages to the master device, there is some wire to store the message. When the slave select becomes 'active' then we have to 'load' that message to some shift register. Whatever goes across the MISO line should be our message from MSB to LSB, so we can just update the shift register such that this behavior is achieved. After all the bits are sent over, we can just send 0's.

2. Introduction to I2S

The second serial bus we will use is the Inter-IC Sound (IIS or I2S) interface on the C5515 and the DE2-70. Unlike SPI, the C5515 can behave as the 'slave' device, accepting an external clock set by another device. By using both the SPI and the I2S on the C5515, we can achieve asynchronous communication between the C5515 and another device. This will be useful in projects where the C5515 processes information collected by another device, such as the DE2-70. Imagine a camera is sending data to the DE2-70 at 27 MHz clock rate. If you're only using SPI, the C5515 must be the master so you need to manipulate the video data rate to conform to the SPI master's data rate. In contrast, with I2S you can set the DE2-70 as master and then send video data to the C5515 at the rate you receive them from the camera. Flexibility like this is one reason I2S is very valuable. In this lab we will demonstrate how to use the C5515 as I2S slave to the DE2-70. Then we will discuss how to use the C5515 as I2S master.

C5515 as I2S slave

Configuring and using I2S on the C5515 as a slave device is similar to setting up SPI. We need to configure register states to set up communication, and during communication we need to alter certain registers as well. Note also that the Breakout Board exposes the ports for only I2S0 and I2S2, so in practice for each USBSTICK you will have 2 I2S modules available.

Table 1-1. I2S Signal Descriptions

Name	Signal	Description
I2S*n*_CLK	INPUT /OUTPUT	I2S Clock
I2S*n*_FS	INPUT /OUTPUT	I2S Frame Sync Clock
I2S*n*_DX	OUTPUT	I2S Data Transmit
I2S*n*_RX	INPUT	I2S Data Receive

Figure 6: Pin signals for I2S on C5515

Step 1: Enable the system clock

Any I2S module requires the system clock to operate. On the C5515 the I2S module idles by default and does not get a system clock. To enable the system clock reaching the I2S module, we zero out the appropriate bit on the Peripheral Clock Gating Configuration Register 1 (PCGCR 1). This must be done for each I2S module we use.

Step 2: Update I2SnCTRL Register

Each I2S2 module in the C5515 has a 16 bit I2Sn Control Register, where the majority of configuration settings are assigned. For our exercises in this lab we will need to configure: enabling I2S, mono or stereo mode, data delay, word length, clock polarity and frame sync polarity, master or slave mode, and communication format (DSP or I2S/Left-justified).

To help you understand what some of these settings mean in terms of the signals that result from different configuration options, take a look at Figure 7 and 8 below:

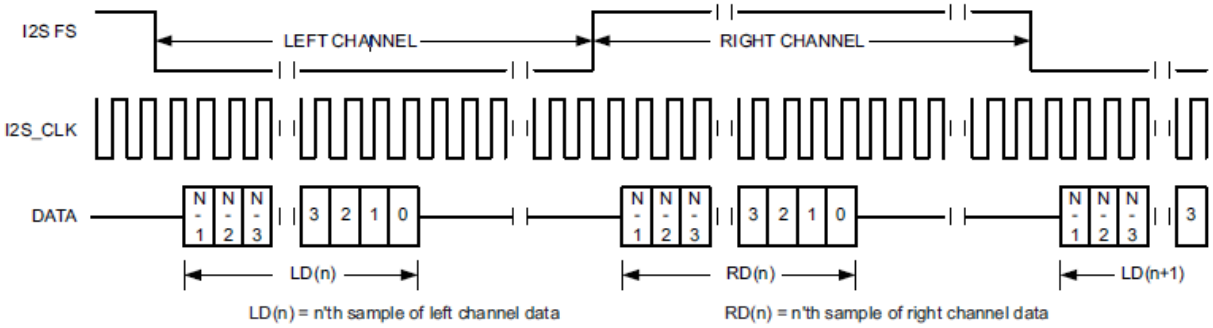

Figure 7: I2S mode, MONO = 0, CLKPOL = 0, FSPOL = 0, DATADLY = 0, FRMT = 0

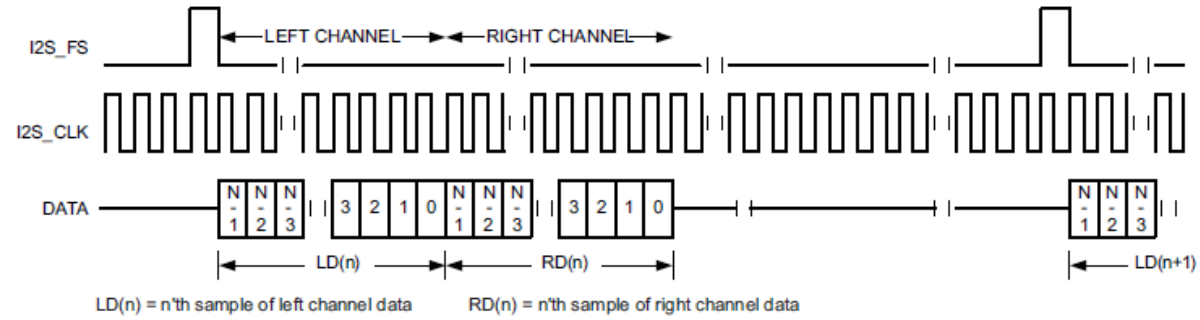

Figure 8: DSP mode, MONO = 0, CLKPOL = 0, FSPOL = 0, DATADLY = 0, FRMT = 1

The 'DATA' blocks here represent data captured on the receive line.A few things are important to notice about these two settings and the signals they correspond to. First, notice that in I2S format (FRMT = 0) must go with 'stereo' mode (MONO = 0), and the 50% duty cycle of the frame sync signal means 'mono' mode is not supported. Under the DSP format (FRMT = 1), we can do either 'mono' or

'stereo' mode. Under the I2S format, the left channel is transferred under low frame sync state, and when frame sync becomes high then right channel is transmitted. Under DSP, if we're using 'stereo', the left channel is transmitted followed immediately by right channel when frame sync is low.

In addition, understand that the CLKPOL and FSPOL can both either be set to 0 'default' or 1 'inverted'. Both of the figures are results of FSPOL = 0, and CLKPOL = 0. Under I2S format, if FSPOL = 1, then the left channel is received when FS is high, and the right channel is received when FS is low. Under DSP format, FSPOL = 1 would mean left and right channels being valid with FS is high. So we can see that FSPOL lets us know when our left and right channels are valid relative to the I2S_FS signal. We can also see that CLKPOL tells us when a **bit** is valid relative to the I2S_CLK signal. Finally, note the 1-bit data delay in the difference in 'gaps' on Figure 8. The 'gaps' indicate the MSB transmitted and the MSB received.

Beyond these basic settings, the I2SCTRL register also stores configurations for Data Pack Mode (PACK) and sign extension (SIGN_EXT) and word length (WDLNGTH). If we turn on data packing (PACK = 1), then interrupts for I2S receiving data will figure every 32 bits vs. every WDLNGTH bits. So for example, if WDLNGTH = 16, then receive events would occur every 16 bits received. However, if we turn on data packing, then receive events would occur half as much. In addition, with data packing on-chip memory is used to store the received values in 32-bit buffers. DMA events occur every 32 bits, and using the on-chip memory is more efficient.

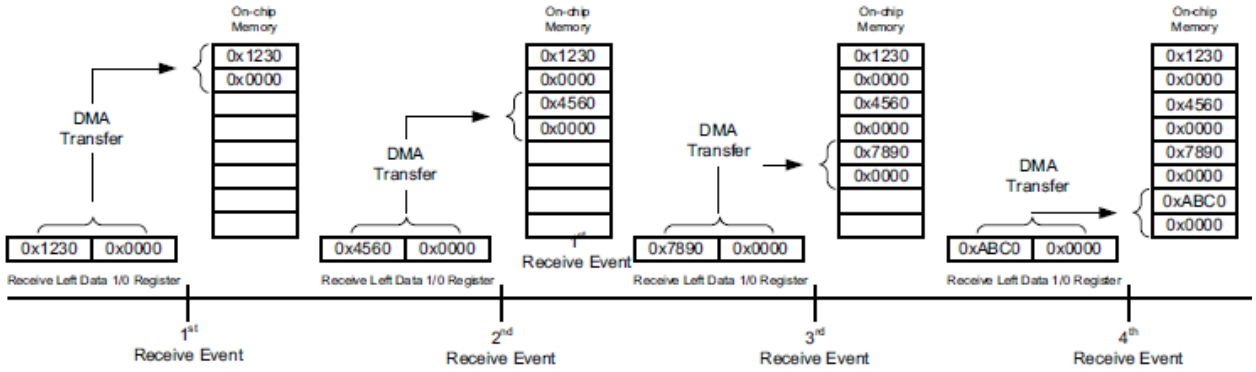

Figure 9: Without data packing (PACK = 0), WDLNGTH = 4h

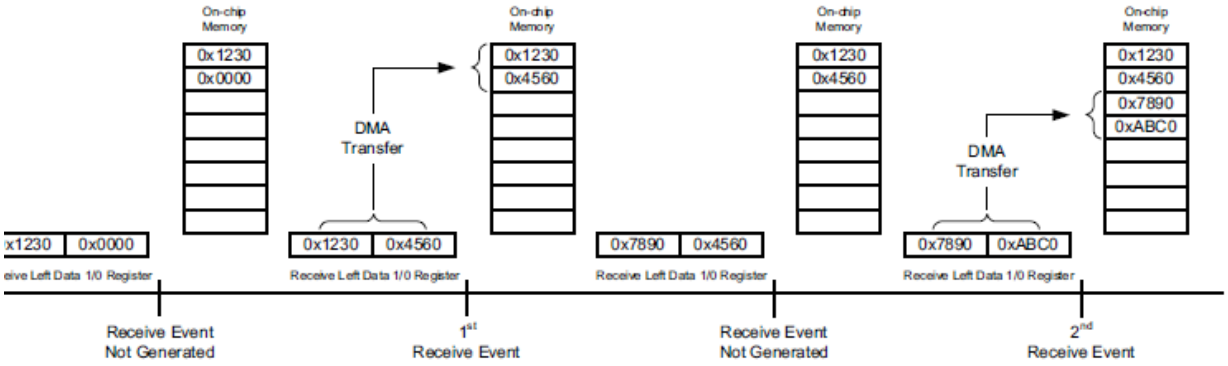

Figure 10: With data packing (PACK = 1), WDLNGTH = 4h

Finally, we can also turn on sign extension. C5515 will attempt to fill up the DMA first, and then sign extend every element to the next multiple of 16. Note the order of operations here: C5515 will fill up the on-chip memory and then sign extend after the memory has been populated.

We've covered the configure I2SCTRL and if we were using the C5515 as a slave device, the only thing left to do is enable interrupts and configure the I2SINTMASK to configure our interrupts. Note that we cannot run in both 'stereo' and 'mono' mode at the same time.

7 6	5	4	3	2	1	0
Reserved	XMITST	XMITMON	RCVST	RCVMON	FERR	OUERR
R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 1-24. I2S*n* Interrupt Mask Register (I2SINTMASK) Field Descriptions

Bit	Field	Value	Description
15-6	Reserved	0	Reserved.
5	XMITST		Enable stereo left/right transmit data interrupt. Used only when the MONO bit 12 in the I2SSCTRL register = 0 (Stereo mode). This bit is cleared on read.
		0	Disable stereo TX data interrupt.
		1	Enable stereo TX data interrupt.
4	XMITMON		Enable mono left transmit data interrupt. Used only when the MONO bit 12 in the I2SSCTRL register = 1 (Mono mode). This bit is cleared on read.
		0	Disable mono TX data interrupt.
		1	Enable mono TX data interrupt.
3	RCVST		Enable stereo left/right receive data interrupt. Used only when the MONO bit 12 in the I2SSCTRL register = 0 (Stereo mode). This bit is cleared on read.
		0	Disable stereo RX data interrupt.
		1	Enable stereo RX data interrupt.
2	RCVMON		Enable mono left receive data interrupt. Used only when the MONO bit 12 in the I2SSCTRL register = 1 (Mono mode). This bit is cleared on read.
		0	Disable mono RX data interrupt.
		1	Enable mono RX data interrupt.
1	FERR		Enable frame sync error.
		0	Disable frame-synchronization error interrupt.
		1	Enable frame-synchronization error interrupt.
0	OUERR		Enable overrun or underrun condition.
		0	Disable overrun/underrun error interrupt.
		1	Enable overrun/underrun error interrupt.

C5515 as I2S master

When we want to use the C5515 as an I2S master device, we need to go through the same steps we took to set the C5515 as an I2S slave device and then go one step further in setting the I2SnRATE register. Remember that as a master the C5515 has two more responsibilities: setting the frame sync (I2S_FS) and clock (I2S_CLK) signals. The I2SnRATE register sets up the dividers used in deriving both signals.

Table 1-18. I2S*n* Sample Rate Generator Register (I2SSRATE) Field Descriptions

Bit	Field	Value	Description
15-6	Reserved	0	Reserved.
5-3	FSDIV		Divider to generate I2S*n*_FS (frame-synchronization clock). The I2S*n*_CLK is divided down by the configured value to generate the frame-synchronization clock. (Has no effect when I2S is configured as slave device).
		0	Divide by 8
		1h	Divide by 16
		2h	Divide by 32
		3h	Divide by 64
		4h	Divide by 128
		5h	Divide by 256
		6h	Reserved
		7h	Reserved
2-0	CLKDIV		Divider to generate I2S*n*_CLK (bit-clock). The system clock (or DSP clock) to the I2S is divided down by the configured value to generate the bit clock. (Has no effect when I2S is configured as slave device).
		0	Divide by 2.
		1h	Divide by 4.
		2h	Divide by 8.
		3h	Divide by 16.
		4h	Divide by 32.
		5h	Divide by 64.
		6h	Divide by 128.
		7h	Divide by 256.

Figure 11: I2SnRATE register

Notice that the FSDIV is dividing down the I2S_CLK rate, which is in turn dividing down the CPU clock rate. So how much you decide to divide down the I2S_CLK to get I2S_FS should be based on your word length and 'stereo' or 'mono' mode.

I2S on the DE2-70

Implementing I2S on the DE2-70 is very similar to implementing SPI on the DE2-70. If we want to create an I2S slave we need to catch the signals, use shift registers to read in data on the RX port and push out data using a shift register on the TX port. We base our decisions off of what we see on the I2S_FS and I2S_CLK lines.

Implementing an I2S master is a bit more involved because we need to worry about the FS and CLK signals. We can use counters and combinational statements to divide up the clock to generate our FS signal. The same is true for generating the CLK signal.

3. Pre-lab

These exercises will prepare you for working with SPI on the C5515 and the DE2-70. All the information you need is either in the background section or the SPI Manual.

Q1. Write a C function **initSPI** that initializes the C5515 for communication with slave device 1 with:

- At a frequency of 1 MHz
- SPI communication mode 1
- Interrupts at the end of every frame
- Frames consist of 1 characters

Q2. Write the C functions **readSPI** and **writeSPI** that collect data from and send data to a slave device over SPI, respectively. Please use the following function starter code. Assume you are using SPI1 with character length 16.

```
//readSPI
Uint16 readSPI(){
//your code here
}

//writeSPI
void writeSPI(Uint16 data){
//your code here
}
```

Q3. The following is a segment of Verilog code that is involved in SPI communication. Look over the following code and describe in your own words what is happening. What mode is the DE2-70's master device communicating with?

```
reg [2:0] SSELr;
always @(posedge clk)
     SSELr <= {SSELr[1:0], SSEL};
```

```
wire SSEL_active = ~SSELr[1];

always @(posedge clk)
begin
  if(~SSEL_active)
    bitcnt <= 4'b0000;
  else
  begin
      if(SCK_risingedge)
      begin
      bitcnt <= bitcnt + 4'b0001;
      MOSIshift <= {MOSIshift[14:0], MOSI_data};
      end
     end
end
```

Q4. Write a snippet of code that does what the above code does but using SPI mode 3.

Q5. Look back at Figures 7 and 8. You're given that in both diagrams the CLKPOL register is set to 0, and know that CLKPOL tells us when to know a received bit is valid relative to the I2S_CLK signal. Using those two figures and this knowledge, answer the following question: under CLKPOL = 0, FRMT = 1, what event on the I2S_CLK tells us a bit received is valid? What about under FRMT = 0?

Q6. Let's say you are using I2S and the WDLNGTH is 4 bits and you're not using data packing. Every time you receive an interrupt, you concatenate the data on the receive register to a chunk of a 32 bit number. Explain how you could be more efficient using the I2SCTRL register.

Q7. Write a C function **initializeI2S()** that defines configurations and updates the necessary registers to set up the C5515 as I2S slave in mono mode, DSP format, with active low relative to the frame and expecting to receive on the falling edge of the I2S_CLK.

Q8. Let's say you have a word length of 16 bits and you're in 'stereo' mode. What should your FSDIV be?

4. In-lab

Part 1: C5515 with Pmods over SPI

In a previous lab we worked with the Pmods (A/D 1 and DA2) on the DE2-70. However, in certain projects students needed to use the Pmods directly with the C5515. These projects might involve collecting analog signals such as voltages or audio through the Pmod A/D 1 and / or outputting data of some form through the Pmod DA2. Understanding how to interface Pmods with the C5515 is a good starting point for using SPI in any project.

Both Pmod devices use SPI to communicate. The C5515 will supply the clock for them as the master device. First we will attempt to collect analog data from the C5515 using the Pmod AD 1. The first task is to connect the hardware. Consult the C5515BreakOut.pdf schematic found on the course website and

the Pmod AD 1 website for the pin assignments. Connect the Pmod AD 1 to SPI 1 on the C5515 Breakout Board. You can find a connector board in the lab to simplify the connection. **Note: do not to apply the blue jumpers on the C5515 breakout whenever appropriate.**

After you've connected the boards to each other, the next step is to create your project in Code Composer Studio.

1. Like in previous labs, import the **Starting_point.zip** project into your workspace and rename it to **Lab7G1**.
2. Download the Lab7_Files.zip file from CTools, extract, and copy and paste all of the files within the folder 'SPI' to your **Lab7G1** project.
3. Add your code from **initSPI** and **readSPI** you created in the pre-lab to the appropriate places in **main.c**.
4. In the file **spi_definitions.h** fill out the incomplete define lines using given definitions
5. Modify you readSPI and initSPI to use these definitions.
6. In the **main** function, implement a while loop that forever reads from SPI 1 and writes the input out to the AIC on both channels.
7. Connect the Pmod AD 1 to SPI 1 (be careful with orientation). Note that although the Pmod AD1 uses SPI to send out digital data, it has J1 interface that isn't immediately matching to the SPI1 connector on our C5515 Breakout Board. Use jumpers and refer to the C5515Breakout.pdf and AD1 links on the website. Also note that SPI exposes 1 TX and 1 RX port. You will only use the RX port for this part, and feed in one of the 'Data' pins from the AD1. You pick which 'Data' pin to use.
8. Connect the function generator outputting a waveform to the Pmod AD1 J2 side.
9. Attach the stereo out from the USBSTICK to the scope and see what you're reading in.

Note: If you are trying to use the AIC and getting linker errors, remember to examine the C5500 Linker File Search Paths and make sure they match with the projects from Lab 2.

Q1. You will not immediately see a sine wave from the AIC. The immediate reason is that the AIC is outputting values too small and close to 0. Why might this be? (Think about the input from the Pmod. How many bits does the Pmod send across?) What does this imply about how you need to change the input from the Pmod?

Modify your code (you only need to add 1 line) so that the AIC outputs a sinusoidal waveform. Remember you can adjust the DC offset of the input waveform to adjust for any truncation.

G1. Show your GSI your sine wave.

Next we're going to use the Pmod DA2 instead of the code chip to output signals. We'll start by outputting a very simple **ramp** function from the Pmod DA2. Again, use the reference documents from the class website to connect the Pmod DA2 to **SPI 2** on the C5515 Breakout Board. Do not detach the Pmod AD1 from SPI 1 because we will use it again soon.

In the same function you used for **G1**, copy over the **writeSPI** function you wrote in the Pre-lab. Modify the code so that it uses **SPI 2** instead. Use bit masking to implement a ramp waveform in the while loop of your main function. You may need to experiment with bit masking before you see an output signal. (Hint, how many bits did the Pmod AD1 send over?)

Q2. How many bits do you need to use for bit masking?

You may see a waveform not exactly a ramp with 'gaps' in between. What might be the issue? (Hint: it probably isn't your main function or your **writeSPI** function. Remember that the C5515 holds communication configurations for **each** SPI slave that are independent of each other) Modify your code so that you see the ramp waveform. You might need a bit of trial and error.

G2. Show your GSI your ramp wave.

Q3. What SPI mode does the Pmod DA2 use?

Finally, we're going to use the Pmod DA2 instead of the codec chip to output our waveform. At this point, the Pmod AD1 should still be connected to SPI 1 and the Pmod DA2 should be connected to SPI 2. In your while loop, write what you receive from the Pmod AD 1 directly out to the Pmod DA2.

You will notice that the simplest implementation will probably not work, because without interrupts nothing pre-empts the CPU to get out of 'read' or 'write' mode so it becomes stuck in which ever is called first. We will use interrupts to get around this issue. Copy and paste the code from '**G3Additions.txt**' to your **main.c**.

G3. Show your GSI the C5515 outputting a sine wave from the Pmod AD1 using the Pmod DA2.

Q4. You may have noticed the 'steps' in the outputted waveform, the time quantization. How might you reduce this quantization effect?

Q5. How much delay (approximately, microseconds) exists between the input and output signals?

Part 2: SPI Communication using the C5515 as master and DE2-70 as slave

First, create a new Starting_point project named '**Lab7G4'** and copy over the functions you wrote for **G3**. Add a while loop to the main function that constantly requests you enter a number and writes it through SPI 2 via **writeSPI** and prints out values from **readSPI** *over SPI 2* to the console. Make sure this code debugs properly.

Next, locate the Quartus folder/project **'fpgaSpiSlave'** in Lab7_Files.zip and launch it in Quartus. Within this project you'll see that we've instantiated a module called '**spiSlave**'. Take a look at the top-level code and try to understand what is going on, what hardware connections to set up, and what you should expect to see.

In the **spiSlave** module we've left a few lines of code missing for you to fill out to demonstrate your understanding. After you complete them you should be able to compile and upload to the DE2-70. Also remember to check your **readSPI** and **writeSPI** functions to make sure they are accessing the appropriate SPI channel!

G4. Show your GSI communication working in both directions. Your entered value from the C5515 console should show on a Hex display on the DE2-70, and the C5515 should print off the value of switches from the DE2-70.

Q6. Notice how the `mosiDataRegister` is updated in the `spiSlave` code, how data shifts to the left. What does this tell us about how we expect the MOSI message to come through?

Part 3: I2S with the DE2-70 as master and C5515 as slave

In this part we will use another serial protocol, I2S, to communicate between the C5515 and the DE2-70. The advantage of understanding I2S for these devices is that using I2S we can achieve *asynchronous* data transfers. Think of this way: if we just talked across one SPI channel, then the DE2-70 can only talk to the C5515 when the C5515 is speaking. However, if we had one SPI channel and another I2S channel, the DE2-70 can talk to the C5515 in one frequency and the C5515 can talk back to the DE2-70 in another frequency. This is particularly important in projects that involve one device to report analysis on data to another device in an asynchronous fashion.

Now we will attempt to implement an I2S slave on the C5515 and an I2S master on the DE2-70. Create a new CCS project from Starting_point and rename it to **'Lab7G5'**. Go to your Lab7_Files folder and locate the files in the subfolder 'I2S'. Copy all of those files to Lab7G5 project. In the **I2S_slave.c** and **main.c** files you will see that there are blanks for you to fill out as before. In the **I2S_slave.c** you will need to demonstrate a basic familiarity with the I2S specification and how it connects to communication specs. In the **main.c** file, write some code that loops forever, asking the user for a number and printing out what is received through I2S2.

Now open up the Quartus project / folder I2SMASTER from Lab7_Files. Look over the code and figure out what you expect to see during runtime and what hardware setup you need to prepare. Once you get a basic understanding of it, fill out the portion marked 'ADD CODE HERE…' to handle basic IO. Check your hardware set up (again, use the Breakout documents on the site).

G5. Demonstrate your I2S master on the DE2-70 and the I2S slave on the C5515 communicating to each other.