Michael Tangy

ECE 381 – Spring 2015

Lab 10 & 11

SPI SRAM &

Data Acquisition System

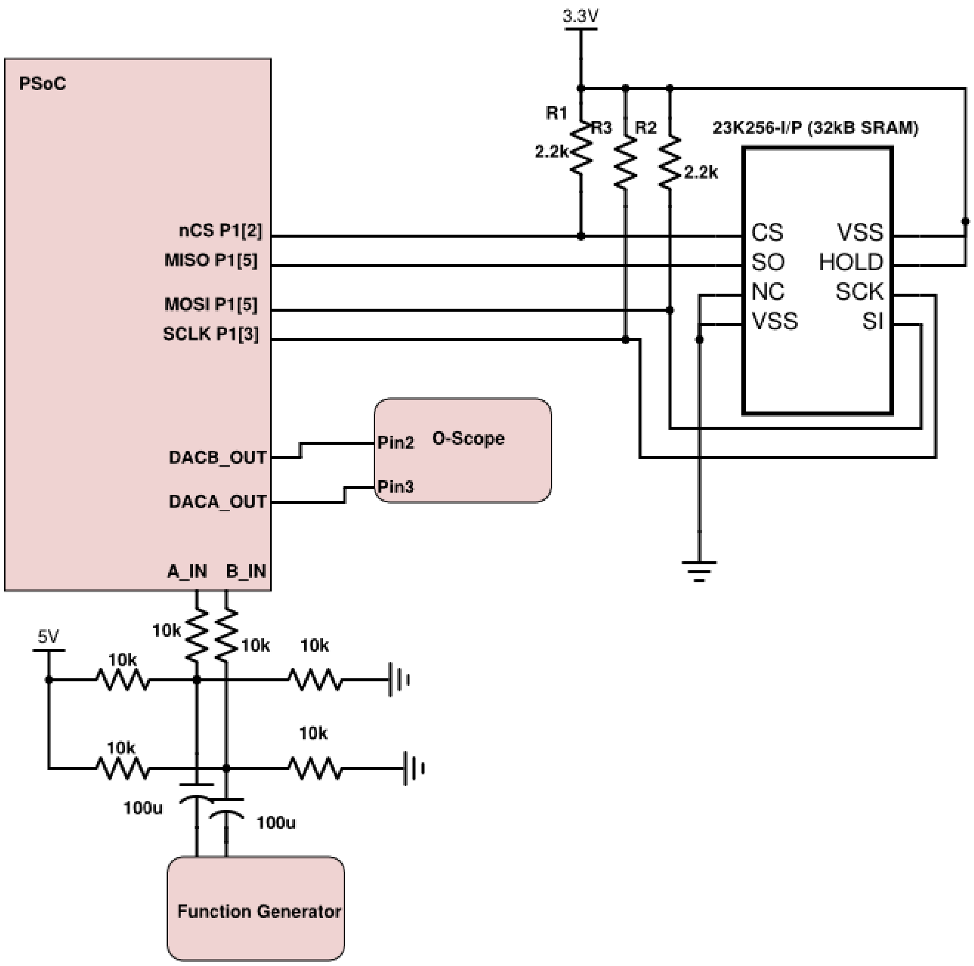
The purpose of lab 10 was for us to gain an understanding of how to implement SPI communication with our PSoC microcontroller to interface with peripheral components such as our SRAM IC. We also learned how to use the PSoC's Digital to analog converter (DAC) to output signals generated by our program. In the Lab we gave the user the option to write one of four different signals to a corresponding 8KB memory block in our 32KB SRAM. For the second lab we developed a Data Acquisition System that allowed the user to save signals from two of the PSoC's analog GPIO pins. The user is greeted with an interface that allows them to choose which pin the system will read from, which block to save to and which sampling rate they would like to be used. The system also gives the user the option to output saved signals to two different GPIO pins.

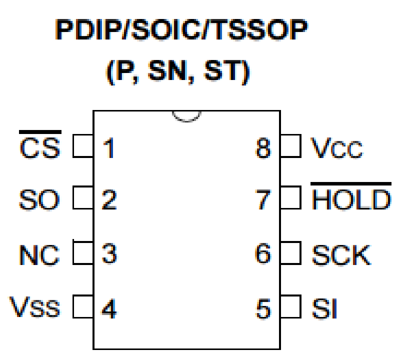
Parts Needed:

- PSoC 1 Microcontroller
- 23K256 32KB SPI SRAM
- RS232 DB9 cable
- 3- 2.2 kΩ Resistors & 6-10k kΩ Resistors
- Decoupling capacitor
- 2 channel DC power supply
- 2-100μF capacitors
- 2 channel Function generator and Oscilloscope (for testing)

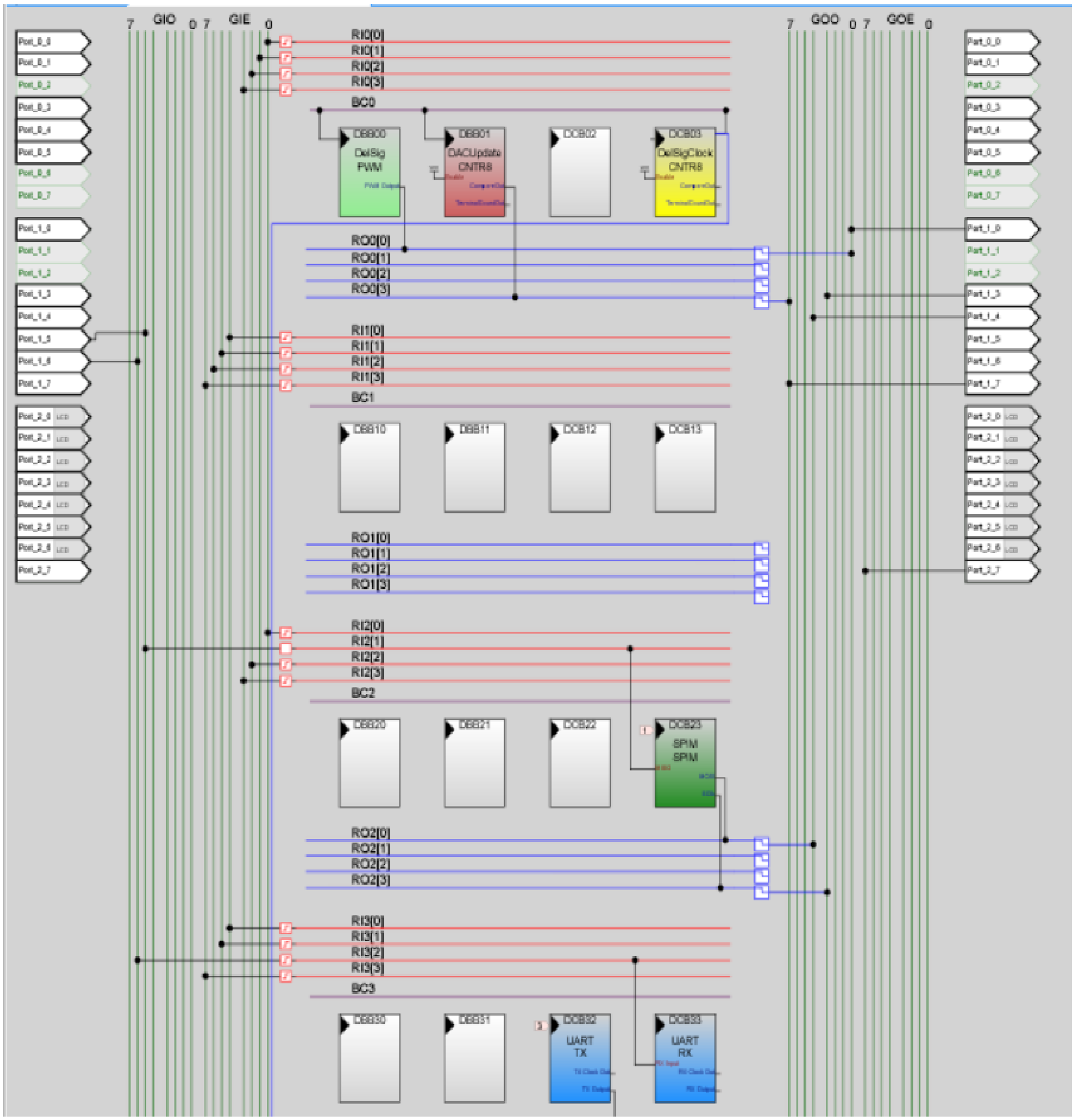
Hardware/Pin-Out Configuration:

After hooking the PSoC up with a PC the SPI SRAM and the protection circuit needs to be wired and conned to the PSoC's SPI pins, follow the wiring diagram and SRAM chip layout below. Make sure to tie "HOLD" high and "NC" low as well as "SO" to P1[5], "SI" to P1[4], SCK to P1[3] and CS to P1[2] of the PSoC. Next make sure that the function generator is hooked up to the inputs of each protection circuit and that their outputs are hooked up to channel A and B P0[1] and P0[7] respectively.

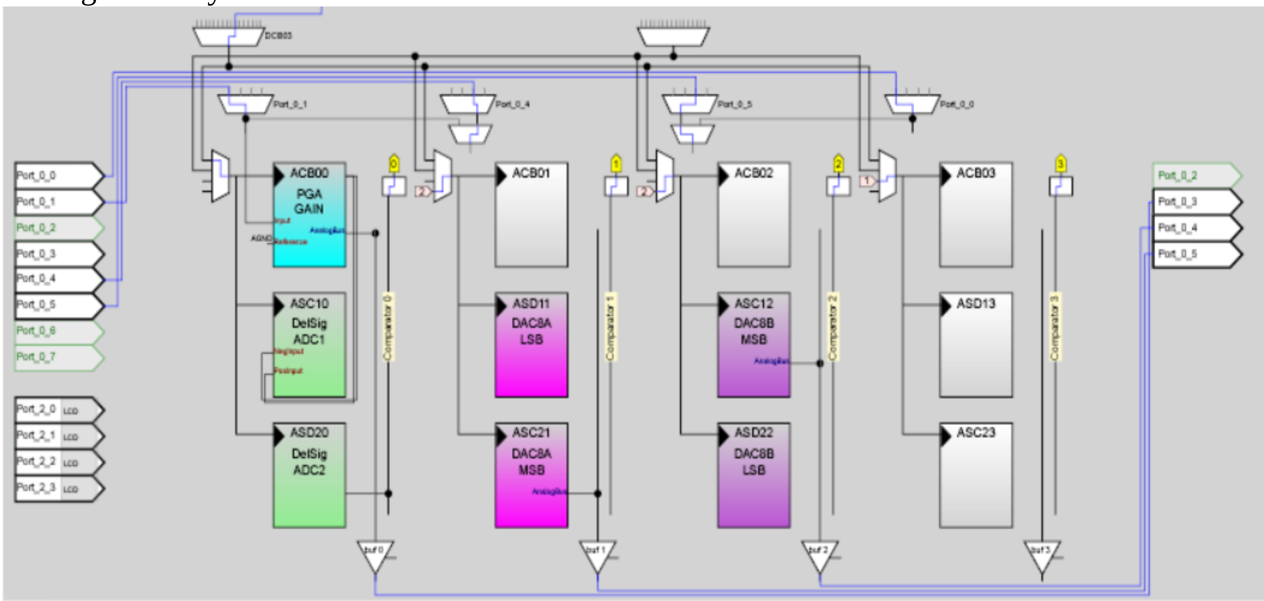




Digital Block layout:



Analog block layout:



Pinout:

Pinout - DataAcqSystem

⊞ P0[0]	Port_0_0, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[1]	CHANNELA_IN, AnalogInput, High Z Analog, DisableInt, 0
⊞ P0[2]	Port_0_2, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[3]	PGA_OUT, AnalogOutBuf_0, High Z Analog, DisableInt, 0
⊞ P0[4]	DAC8B_OUT, AnalogOutBuf_2, High Z Analog, DisableInt, 0
⊞ P0[5]	DAC8A_OUT, AnalogOutBuf_1, High Z Analog, DisableInt, 0
⊞ P0[6]	Port_0_6, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[7]	CHANNELB_IN, AnalogInput, High Z Analog, DisableInt, 0
⊞ P1[0]	ADCPWM_OUT, GlobalOutOdd_0, Strong, DisableInt, 0
⊞ P1[1]	TRIGGER, StdCPU, Strong, DisableInt, 0
⊞ P1[2]	nCS, StdCPU, Open Drain Low, DisableInt, 1
⊞ P1[3]	SCLK, GlobalOutOdd_3, Open Drain Low, DisableInt, 1
⊞ P1[4]	MOSI, GlobalOutOdd_4, Open Drain Low, DisableInt, 1
⊞ P1[5]	MISO, GlobalInOdd_5, High Z, DisableInt, 0
⊞ P1[6]	RX_IN, GlobalInOdd_6, High Z, DisableInt, 0
⊞ P1[7]	DACUPDATE_OUT, GlobalOutOdd_7, Strong, DisableInt, 0
⊞ P2[0]	LCDD4, StdCPU, Strong, DisableInt, 0
⊞ P2[1]	LCDD5, StdCPU, Strong, DisableInt, 0
⊞ P2[2]	LCDD6, StdCPU, Strong, DisableInt, 0
⊞ P2[3]	LCDD7, StdCPU, Strong, DisableInt, 0
⊞ P2[4]	LCDE, StdCPU, Strong, DisableInt, 0
⊞ P2[5]	LCDRS, StdCPU, Strong, DisableInt, 0
⊞ P2[6]	LCDRW, StdCPU, Strong, DisableInt, 0
⊞ P2[7]	TX_OUT, GlobalOutEven_7, Strong, DisableInt, 0

The protection circuit limits the current entering the PSoC’s analog pins as well as provide a 2.5V DC offset to compensate for the negative half cycle produced by the function generator.

Once the wiring is complete download the template project file for the lab that contains the chip level configurations of the PSoC (refer to the picture below). After that run the SRAM test file from lab 10 to verify the SRAM is wired properly.

Software Description:

The first code that should be written is the read and write byte and array functions in the SPI SRAM.c file. These functions should first parse the two byte address into two separate bytes. Next the program should set the chip select bit low and make sure the TX buffer is empty. After that the two functions need to send the command bits to either read or write depending on the operation. Once the command bits are sent, the parsed memory address is sent (MSB's first). For the write function send the data to be written (byte or array passed to the function) along with two dummy reads to move the next two bytes through the SPI shift registers. Before all the write commands, check that the TX buffer is empty and before all the read commands, check that the SPI transmissions is complete. Before the function ends, set the CS bit high and re-enable global interrupts. For the read byte function place a dummy write after the address (to make room in the serial transmission for the data being read) and assign the last byte read to the variable passed to the function. The write array function is exactly the same as the write byte function except while writing, the function enters a for-loop that iterates through every element of the array and writes that byte along with a dummy read. The read array function is the same as the read byte function, except at the end it has a for-loop that populates the array passed to it with consecutive data read from the SRAM.

For lab 10 we generated signals to output through the PSoC's digital to analog converter by using the iterator of a for loop as the independent variable of a mathematical function that was set to a variable that was passed to the write byte function. These values were later written back from and passed to the DAC to represent voltage that can be measured and viewed by the oscilloscope.

In lab 11 we implemented a lot of the skills developed in the previous lab to read and save signals fed to the PSoC. We first created a user interface that asked the user questions in order to set the systems settings. The user was first asked ask if they want to capture or display a signal, this was to properly redirect them to another set of questions. When capturing, they were asked which channel A or B. This would change which input the chip level multiplexer would select. The user would then be asked which memory block they would like to write to. This would assign a constant factor used to multiply the bounds of the for-loop used to read and write from the SRAM. Next the user would choose the sampling rate, which would change the period and compare values of the counter that was used to clock the analog to digital converter. Next the wave form was saved to SRAM by continuously taking samples from the analog to digital converter in a for-loop and saving those samples to SRAM with the write byte function. When the user chooses to display the signals they are asked which memory block they would like each DAC to output. This changes which iterator is passed into a while-loop that continuously outputs the DAC's until the user pushes a button. This while-loop is nested in another while-loop that iterates through both blocks and continues to write those values to their corresponding DAC in order to simultaneously display both signals. Once the user presses a button, the program breaks out of two loops and starts from the beginning.

Testing:

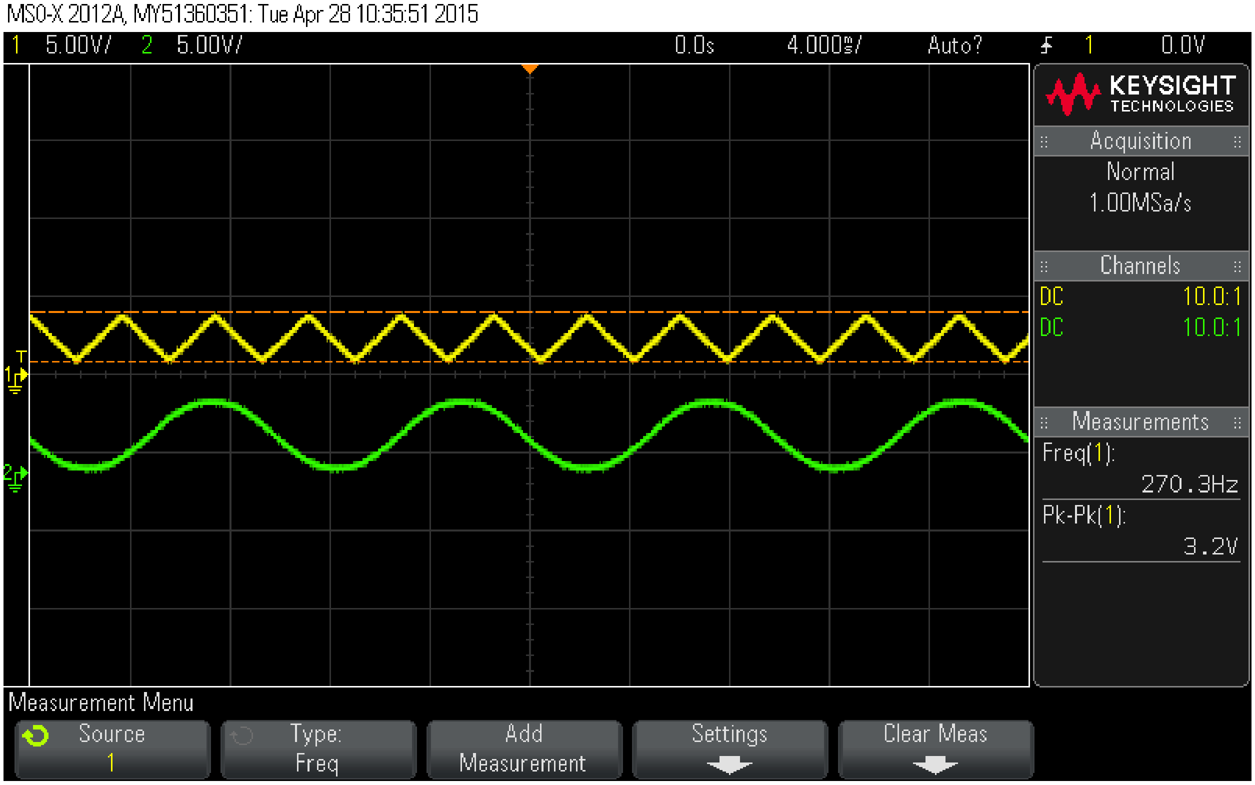
Compile the program and upload the corresponding HEX file to the PSoC. Turn on both function generators and set them both to two different signals at around a 100Hz with a 2V peak to peak voltage. Place your two Oscilloscope probes to pins P0[4] and P0[5] (use a jumper to the bread board if the probes won't stay next to each other) to view the outputs of the DAC. Next open realterm and capture both signals from the two different channels and save them to two different blocks. Next display them and check the Oscilloscope for the signals. Refer to the pictures below.

Real term prompt:

```
Lab 11 Data Acquisition System

Would you like to (d)isplay waveform or (c)apture waveform?
>>c

Which Input would you like to save 'a' or 'b' ?
>>a

Which block to save to? (Choose 1, 2, 3, or 4)
>>1

Choose sample rate
1 for 1.25ksps, 2 for 1.5ksps, 3 for 1.87ksps, 4 for 2.5ksps
5 for 3.125ksps, 6 for 3.75ksps, 7 for 6.25ksps, 8 for 7.5ksps, 9 for 9.375ksps
>>9

Saving waveform

Would you like to (d)isplay waveform or (c)apture waveform?
>>c

Which Input would you like to save 'a' or 'b' ?
>>b

Which block to save to? (Choose 1, 2, 3, or 4)
>>2

Choose sample rate
1 for 1.25ksps, 2 for 1.5ksps, 3 for 1.87ksps, 4 for 2.5ksps
5 for 3.125ksps, 6 for 3.75ksps, 7 for 6.25ksps, 8 for 7.5ksps, 9 for 9.375ksps
>>9

Saving waveform

Would you like to (d)isplay waveform or (c)apture waveform?
>>d

Which block to read from DAC A? (Choose 1, 2, 3, or 4)
>>1

Which block to read from DAC B? (Choose 1, 2, 3, or 4)
>>2
```

```
Displaying waveform (press any key to exit)
```

Oscilloscope picture:



CODE:

spi_sram.c:

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "spi_sram.h"

// Write byte "value" to SRAM Status Register
BYTE SPIRAM_WriteStatusRegister(BYTE value)
{
      BYTE b;

      // If bits 5 through 1 aren't zero, return an error as per
      // Section 2.5 in the 23K256 datasheet
      if (value & 0b00111110)
            return(1);
      // Mode SPIRAM_SEQUENTIAL_MODE|SPIRAM_PAGE_MODE are invalid.
      if ((value & 0b11000000) == 0b11000000)
            return(1);
      // Make sure the TX buffer is empty (it should be but let's be proper)
      while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
      // SPI transfers begin by bringing CS LOW
```

```
	nCS_LOW;
	// Send the Status Register Write command
	SPIM_SendTxData(SPIRAM_WRITE_STATUS_REG);
	// It will be almost immediately loaded into the TX shift register, freeing
	// up the TX buffer, and the SPIM module will start transmission.
	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	// Prime the TX buffer for the next byte by loading it with the new status
	// register byte while the first byte is still transmitting.
	SPIM_SendTxData(value);
	// Wait for the first TX/RX cycle to finish. We don't care what we read.
	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	// Reading the data clears the RX_BUFFER_FULL flag, even if we don't want it.
	SPIM_bReadRxData(); // ignore byte from SPIRAM_WRITE_STATUS_REG TX
	// Wait for the second TX/RX cycle to finish so that we know that the entire
	// two byte transaction is finished.
	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData(); // We don't care about this read either
	// SPI transfers end by bringing CS LOW
	nCS_HIGH;
	return(0);
}

// Read SRAM Status Register and return the result.
BYTE SPIRAM_ReadStatusRegister(void)
{
	BYTE statReg;
	BYTE b;

	// Make sure the TX buffer is empty before starting
	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	nCS_LOW;
	// Send the Read Status Register command
	SPIM_SendTxData(SPIRAM_READ_STATUS_REG);
	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	// Send a dummy byte in order to initiate a TX/RX transfer
	SPIM_SendTxData(SPIRAM_DUMMY_BYTE);
	// Wait for the first RX byte to arrive and ignore it; it is meaningless.
	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();
	// Wait for the second RX byte to arrive; it contains the status reg value.
	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	statReg = SPIM_bReadRxData();
	nCS_HIGH;
	return(statReg);

}

// Write byte "out" to SRAM address "addr"
// NOTE: This function assumes the SRAM has already been put in Byte Mode.
void SPIRAM_WriteByte(WORD addr, BYTE out)
```

```
{
	BYTE hiAddr;
	BYTE loAddr;

	// Break the SRAM word address into two bytes
	hiAddr = (BYTE)((addr >> 8) & 0x00ff);
	loAddr = (BYTE)(addr & 0x00ff);
	M8C_DisableGInt;
	// Place your SPI code here

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));

	nCS_LOW;
	SPIM_SendTxData(SPIRAM_WRITE);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(hiAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(loAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(out);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();

	nCS_HIGH;

	M8C_EnableGInt;
}

// Read and return byte at SRAM address "addr"
// NOTE: This function assumes the SRAM has already been put in Byte Mode.
BYTE SPIRAM_ReadByte(WORD addr)
{
	BYTE hiAddr;
	BYTE loAddr;
	BYTE in;

	// Break the SRAM word address into two bytes
	hiAddr = (BYTE)((addr >> 8) & 0x00ff);
	loAddr = (BYTE)(addr & 0x00ff);
	M8C_DisableGInt;
	// Place your SPI code here
	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));

	nCS_LOW;
	SPIM_SendTxData(SPIRAM_READ);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(hiAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
```

```
	SPIM_SendTxData(loAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(SPIRAM_DUMMY_BYTE);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	in = SPIM_bReadRxData();

	nCS_HIGH;

	M8C_EnableGInt;
	return(in);
}

// Write "count" bytes starting at address "addr" from array "out".
// The M8C is limited to 256 byte pages. This limits the maximum
// array size to 256 bytes, which means that "count" is only useful
// as a BYTE.
// NOTE: This function assumes the SRAM has already been put in Sequential
Mode
void SPIRAM_WriteArray(WORD addr, BYTE *out, BYTE count)
{
	BYTE hiAddr;
	BYTE loAddr;
	BYTE b;
	int i;

	// If some clown tries to write 0 bytes, just return.
	// XXX - Always beware of clowns!
	if (!count)
		return;
	// Break the SRAM word address into two bytes
	hiAddr = (BYTE)((addr >> 8) & 0x00ff);
	loAddr = (BYTE)(addr & 0x00ff);
	M8C_DisableGInt;
	// Place your SPI code here

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	nCS_LOW;
	SPIM_SendTxData(SPIRAM_WRITE);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(hiAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();


	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(loAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();
```

```
	for (i = 0; i < count; i++ ){

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(out[i]);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();
	}

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();

	nCS_HIGH;

	M8C_EnableGInt;
}

// Read "count" bytes starting at address "addr" into array "in"
// The M8C is limited to 256 byte pages. This limits the maximum
// array size to 256 bytes, which means that "count" is only useful
// as a BYTE.
// NOTE: This function assumes the SRAM has already been put in Sequential
Mode
void SPIRAM_ReadArray(WORD addr, BYTE *in, BYTE count)
{
	BYTE hiAddr;
	BYTE loAddr;
	BYTE b;
	int i;
	// If some clown tries to write 0 bytes, just return.
	// XXX - Always beware of clowns!
	if (!count)
		return;
	// Break the SRAM word address into two bytes
	hiAddr = (BYTE)((addr >> 8) & 0x00ff);
	loAddr = (BYTE)(addr & 0x00ff);
	M8C_DisableGInt;

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	nCS_LOW;
	SPIM_SendTxData(SPIRAM_READ);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(hiAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
	SPIM_SendTxData(loAddr);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY));
      SPIM_SendTxData(SPIRAM_DUMMY_BYTE);

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));
	SPIM_bReadRxData();

	while(!(SPIM_bReadStatus() & SPIM_SPIM_SPI_COMPLETE));

	for (i = 0; i < count; i++ ){
```

```
      in[i] = SPIM_bReadRxData();
      }

      nCS_HIGH;

      M8C_EnableGInt;
}
```

Lab 10:

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "stdlib.h"
#include "spi_sram.h"
#include "string.h"
#include "math.h"
// Define our I/O array size. Powers of 2 are nice but not necessary
#define ARRAY_SIZE 64

BOOL exit = FALSE;
int q;
int p;
BYTE Pointer;
BOOL finishFlag;
char rawString[2];
char charIn;
float x;
int j;
float sValue;
BYTE sByte;
char input;
BYTE Sinusoid2[256];

const char Sinusoid[] = {
127, 139, 152, 164, 176, 187, 198, 208, 217, 225, 233, 239, 244, 249, 252,
253,
254, 253, 252, 249, 244, 239, 233, 225, 217, 208, 198, 187, 176, 164, 152,
139,
127, 115, 102, 90, 78, 67, 56, 46, 37, 29, 21, 15, 10, 5, 2, 1,
0, 1, 2, 5, 10, 15, 21, 29, 37, 46, 56, 67, 78, 90, 102, 115
};
// Globals
BYTE DataIn[ARRAY_SIZE];
BYTE DataOut[ARRAY_SIZE];
// Test reading and writing the 23K256 status register:
BYTE SPIRAM_StatusRegisterTest(void)
{
      // NOTE: SPIRAM_SEQUENTIAL_MODE|SPIRAM_PAGE_MODE is "Reserved", don't
use it
      BYTE mode[6] = { SPIRAM_BYTE_MODE,
                              SPIRAM_BYTE_MODE|SPIRAM_DISABLE_HOLD,
                              SPIRAM_SEQUENTIAL_MODE,
                              SPIRAM_SEQUENTIAL_MODE|SPIRAM_DISABLE_HOLD,
                              SPIRAM_PAGE_MODE,
                              SPIRAM_PAGE_MODE|SPIRAM_DISABLE_HOLD };
      BYTE status;
      BYTE b;
```

```
	UART_CPutString("Status Register W/R Test: 0x  ");
	for (b=0; b<6 ; b++) {
		UART_PutChar(0x08);
		UART_PutChar(0x08);
		UART_PutSHexByte(mode[b]);
		if (SPIRAM_WriteStatusRegister(mode[b])) {
			UART_CPutString("\r\nWrite of invalid Status Register
value. System halted.\r\n");
			M8C_Stop;
		}
		status = SPIRAM_ReadStatusRegister();
		if (status != mode[b]) {
			UART_CPutString(" FAIL\r\n");
			return(1);
		}
	}
	UART_CPutString("\b\b\b\b\b PASS\r\n");
	// Place the SRAM back in Byte Mode
	SPIRAM_WriteStatusRegister(SPIRAM_BYTE_MODE|SPIRAM_DISABLE_HOLD);
	return(0);
}
// Test reading and writing the 23K256 in Byte Mode:
WORD SPIRAM_ByteModeTest(void)
{
	BYTE status;
	BYTE b;
	BYTE in;
	WORD addr;

	SPIRAM_WriteStatusRegister(SPIRAM_BYTE_MODE|SPIRAM_DISABLE_HOLD);
	UART_CPutString("      Byte Mode W/R Test: Addr 0x    ");

	for (addr=0; addr<0x8000 ; addr++) {
		if (((addr-1) & 0x000f) == 0x000f) {
			UART_CPutString("\b\b\b\b");
			UART_PutSHexInt(addr);
		}
		b = 0;
		do {
			SPIRAM_WriteByte(addr, b);
			in = SPIRAM_ReadByte(addr);
			if (in != b) {
				UART_CPutString("\b\b\b\b");
				UART_PutSHexInt(addr);
				UART_CPutString(" FAIL\r\n");
				return(1);
			}
			if (!b)
				b = 0x01;
			else
				b = b << 1;
		} while(b);
		if (UART_cReadChar()) {
			UART_CPutString("\b\b\b\b\b\b\b\b\b\b\b\b\b ABORTED
\r\n");
			return(0);
```

```
		}
	}
	UART_CPutString("\b\b\b\b\b\b\b\b\b\b\b\b\b PASS       \r\n");
	return(0);
}

// Test reading and writing the 23K256 in Sequential Mode:
WORD SPIRAM_SequentialModeTest(void)
{
	BYTE status;
	BYTE a;
	BYTE b;
	BYTE in;
	WORD addr;

	SPIRAM_WriteStatusRegister(SPIRAM_SEQUENTIAL_MODE|SPIRAM_DISABLE_HOLD);
	UART_CPutString("Sequential Mode W/R Test: Addr 0x    ");

	for (addr=0; addr<0x8000 ; addr+=ARRAY_SIZE) {
		UART_CPutString("\b\b\b\b");
		UART_PutSHexInt(addr);
		b = 0;
		do {
			for (a=0 ; a<ARRAY_SIZE ; a++) {
				DataOut[a] = b;
			}
			SPIRAM_WriteArray(addr, DataOut, ARRAY_SIZE);
			SPIRAM_ReadArray(addr, DataIn, ARRAY_SIZE);
			for (a=0 ; a<ARRAY_SIZE ; a++) {
				if (DataIn[a] != b) {
					UART_CPutString("\b\b\b\b");
					UART_PutSHexInt(addr+a);
					UART_CPutString(" FAIL\r\n");
					return(1);
				}
			}
			if (!b)
				b = 0x01;
			else
				b = b << 1;
		} while(b);
		if (UART_cReadChar()) {
			UART_CPutString("\b\b\b\b\b\b\b\b\b\b\b\b\b ABORTED
\r\n");
			return(0);
		}
	}
	UART_CPutString("\b\b\b\b\b\b\b\b\b\b\b\b\b PASS       \r\n");
	return(0);
}

void main(void)
{
	DAC8_Start(DAC8_FULLPOWER);
	// Make sure nCS is high before doing anything
	nCS_HIGH;
```

```
	// Enable user module interrupts
	SleepTimer_EnableInt();

	// Enable global interrutps
	M8C_EnableGInt;

	// Start the user modules
	UART_Start(UART_PARITY_NONE);
	UART_PutCRLF();
	SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);
	SleepTimer_Start();

	while(1){

      instructions ();

	readFunction ();

	stringParser();

	}

	UART_CPutString("23K256 SPI SRAM\r\n");
	while(1) {
		// Test the status register, looping every 1/2s until it succeeds
		while(SPIRAM_StatusRegisterTest()) {
			SleepTimer_SyncWait(4, SleepTimer_WAIT_RELOAD);
		}
//		// Test Byte Mode, looping every 1/2s until it succeeds
//
//		while(SPIRAM_ByteModeTest()) {
//			SleepTimer_SyncWait(4, SleepTimer_WAIT_RELOAD);
//		}
//
		// Test Sequential Mode, looping every 1/2s until it succeeds
		while(SPIRAM_SequentialModeTest()) {
			SleepTimer_SyncWait(4, SleepTimer_WAIT_RELOAD);
		}
		SleepTimer_SyncWait(8, SleepTimer_WAIT_RELOAD);
	}
}


void generate_signal(int signalNum){

	switch(signalNum){

	case 1:
			UART_PutCRLF();
			UART_CPutString("Creating and saving Sinusoid");
			UART_PutCRLF();
			//create and save sine
			for (j=0; j<8192; j++) {

			sValue = 127.0 * sinf(3.14159 *(j)/32.0);

			sByte = (BYTE) sValue;
```

```
                  SPIRAM_WriteByte(j, sByte);
            }


                  UART_CPutString("Producing Sinusoid (Press 'r' to reset)");
                  UART_PutCRLF();
                  while (1){
                        for (j=0; j<8192; j++){
                        if (UART_cReadChar()==0x72){M8C_Reset;}
                      DAC8_WriteBlind(SPIRAM_ReadByte(j));

                    }
                        if (UART_cReadChar()==0x72){M8C_Reset;}
                  }

      break;
      case 2:
                  //create and save rectified sine
                  UART_PutCRLF();
                  UART_CPutString("Creating and saving rectified sinusoid");
                  UART_PutCRLF();
                  for (j=8192; j<16384; j++) {

                        x = j - 8192;

                        sValue = abs(127.0 * sinf(3.14159 *(x)/32.0));

                  sByte = (BYTE) sValue;
                  SPIRAM_WriteByte(j, sByte);
            }
            UART_CPutString("Producing rectified sinusoid (press 'r' to
reset)");
                  UART_PutCRLF();
                  while (1){
                        for (j=8192; j<16384; j++) {
                      if (UART_cReadChar()==0x72){M8C_Reset;}
                              DAC8_WriteBlind(SPIRAM_ReadByte(j));

                    }
                        if (UART_cReadChar()==0x72){M8C_Reset;}
                  }

      break;
      case 3:

                  //create and save sinc
                  UART_PutCRLF();
                  UART_CPutString("Creating and saving sinc");
                  UART_PutCRLF();
                  for (j=16384; j<24576; j++) {

                        x = j -16384;

                        sValue = 127.0 * (sinf(3.14159*(x-
4096)/512.0)/(3.1416*(x-4096)/512.0));

                  sByte = (BYTE) sValue;
```

```
                SPIRAM_WriteByte(j, sByte);
        }

                UART_CPutString("Producing sinc (press 'r' to reset)");
                UART_PutCRLF();
                while (1){

                      for (j=16384; j<24576; j++) {
                      //4096

                      DAC8_WriteBlind(SPIRAM_ReadByte(j));
                      if (UART_cReadChar()==0x72){M8C_Reset;}

                  }
                if (UART_cReadChar()==0x72){M8C_Reset;}
                }

      break;
      case 4:
            //create and save cosine

                UART_PutCRLF();
                UART_CPutString("Creating and saving cosine");
                UART_PutCRLF();
                for (j=24576; j<32769; j++) {

                x = j -     24576;


                //sValue = 127.0 * 2 (x - floor(x)) - 1;
                sValue = 127.0 *((sqrtf(3.14159 *(x)/(256.0))) );

                sByte = (BYTE) sValue;
                SPIRAM_WriteByte(j, sByte);
        }

                UART_CPutString("Producing cosine (press 'r' to reset)");
                UART_PutCRLF();
                while (1){

                      for (j=24576; j<32769; j++){
                    DAC8_WriteBlind(SPIRAM_ReadByte(j));
                      if (UART_cReadChar()==0x72){M8C_Reset;}
                  }
                      if (UART_cReadChar()==0x72){M8C_Reset;}
                }
      break;

      default:

      break;

      }///end switch

}//end function
```

```
void readFunction (void)
{
        int placeCounter = 0;
        finishFlag= FALSE;

            UART_CPutString(">");

        while (!finishFlag) {

                charIn = UART_cReadChar();
                        while (charIn == 0x00){charIn = UART_cReadChar();}
                                if (placeCounter<2){

                            if (charIn == 0x0d){ //if carriage return
                                                UART_CPutString("\r\n");
                                                UART_CPutString("\r\n");
                                            finishFlag = TRUE;
                                    } //end if CR
                                                      //backspace
                                else if ((charIn == 0x08 || charIn ==0x7f) &&
placeCounter > 0 ){ //if backspace
                                        placeCounter--;
                                        UART_CPutString("\x8\x20\x8");
                                } //end if backspace
                                else {
                                                rawString[placeCounter] =
charIn;

UART_PutChar(rawString[placeCounter]);
                                                placeCounter++;
                                        }// end else write into string

                                } // end placeCounter if
                        else
                                {

                                       // UART_CPutString("String Full");

//UART_PutChar(0x07);
                                         finishFlag = TRUE;
                                                         //addbell
                                }

        }//end while (~finsihFlag)

        finishFlag= FALSE ;

}//end readFunction
```

Lab 11:

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include "stdlib.h"
#include "spi_sram.h"
#include "math.h"
```

```
#define ARRAY_SIZE 64


/// testing
///
// GPIO Defines
#define TRIGGER_HIGH    {TRIGGER_Data_ADDR |=  TRIGGER_MASK;}
#define TRIGGER_LOW           {TRIGGER_Data_ADDR &= ~TRIGGER_MASK;}

// Define Sampling Rates
#define SAMPLING_RATE_1250 149 // (150-1)
#define SAMPLING_RATE_1500 124 // (125-1)
#define SAMPLING_RATE_1875  99 // (100-1)
#define SAMPLING_RATE_2500  74 //  (75-1)
#define SAMPLING_RATE_3125  59 //  (60-1)
#define SAMPLING_RATE_3750  49 //  (50-1)
#define SAMPLING_RATE_6250  29 //  (30-1)
#define SAMPLING_RATE_7500  24 //  (25-1)
#define SAMPLING_RATE_9375  19 //  (20-1)

int i;

// DACUpdate Period = 4*DelSig_DecimationRate = 128 for DS232
#define DACUPDATE_PERIOD   127 // (128 - 1)

// Globals
BYTE DACUpdateDone = 0;
// add your globals here

float fScaleFactor;

int trigAdr;

char *pResult;
float voltage;
char sampleRead;

char Svalue;
int j;
int k;
int blockNum2;
int iStatus;
char rawString[64];
char charIn;
BOOL finishFlag;
int parseNum;
int blockNum;
BOOL displayOrcapture;
BOOL done = FALSE;
BOOL inA = TRUE;
BOOL error = FALSE;
BOOL validMode = TRUE;
void readFunction (void);
void stringParser(int parseNum);
void instruction1(void);
void instruction2(void);
```

```
void instruction3(void);
void instruction4(void);
void instruction5(void);
void instruction6(void);
void instruction7(void);
char toLower(char k);
unsigned int hexToDec(char c1, char c2, char c3, char c4);
unsigned char ascii_to_hex(unsigned char*  addressString);
WORD SPIRAM_ByteModeTest(void);
WORD SPIRAM_SequentialModeTest(void);

int sampleRateS[9]= {
SAMPLING_RATE_1250,
SAMPLING_RATE_1500,
SAMPLING_RATE_1875,
SAMPLING_RATE_2500,
SAMPLING_RATE_3125,
SAMPLING_RATE_3750,
SAMPLING_RATE_6250,
SAMPLING_RATE_7500,
SAMPLING_RATE_9375
};

int sampleRate;



void main(void)
{
	int count=0;
	fScaleFactor = (float)5/(float)64;
	// Make sure nCS is high before doing anything
	nCS_HIGH;
	// Make the oscilloscope external trigger signal low. Trigger must be
quickly
	// brough high-then-low when you want the oscilloscope to draw the
signals
	// on DACA and DACB. Trigger (P1[1]) must be connected to the EXT TRIG
input
	// on the back of the oscilloscope and the Trigger Source must be set
to
	// External. The oscilloscope should also be set for Normal Mode
Triggering.
	TRIGGER_LOW;
	// Enable global interrutps
	M8C_EnableGInt;

//	SleepTimer_Start();
//
//	SleepTimer_EnableInt();
//	SleepTimer_Start();
	LCD_Start();

	// Start the UART
	UART_Start(UART_PARITY_NONE);
	UART_PutCRLF();
```

```c
	// Start the SPIM Module
	SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);

	// Start the DelSig custom clock source at the default sampling rate
	//DelSigClock_WritePeriod(SAMPLING_RATE_1250); //SAMPLING_RATE_1250
	DelSigClock_WritePeriod(SAMPLING_RATE_3125);
	DelSigClock_WriteCompareValue(SAMPLING_RATE_3125>>1);
	//DelSigClock_WriteCompareValue(SAMPLING_RATE_1250>>1);
//SAMPLING_RATE_1250>>1
	DelSigClock_Start();


	// Start the analog mux and select P0[1] (Channel A) as default
	AMUX4_Start();
	AMUX4_InputSelect(AMUX4_PORT0_1);

	// Start the PGA
	PGA_Start(PGA_HIGHPOWER);

	// Start the DelSig but do not start taking samples yet.
	// Note: The DelSig PWM block output can be monitored on P1[0]. This
	// can be used to verify the sampling rate.
	DelSig_Start(DelSig_HIGHPOWER);



	// Enable interrupts on the counter that sets the DAC output rate.
	// Start the module only when actually outputting samples and
	// stop it when done. Don't forget to write the period after stoping
	// to reset the count register.
	// NOTE: You can watch this counter on P1[7] to compare desired
	// output rate with your actual output rate.
	DACUpdate_WritePeriod(DACUPDATE_PERIOD);
	DACUpdate_EnableInt();



	// Start the DACs
	DAC8A_Start(DAC8A_HIGHPOWER);
	DAC8B_Start(DAC8B_HIGHPOWER);
	UART_PutCRLF();
	UART_CPutString("Lab 11 Data Acquisition System\r\n");
	UART_PutCRLF();

	// Enter the main loop

	while(1) {

		validMode = TRUE;

		UART_PutCRLF();
		instruction1();
		readFunction();
		stringParser(1);
		UART_PutCRLF();
		if (validMode){
```

```
            if(displayOrcapture){

                    error = FALSE;
                      instruction3();
                    readFunction();
                    stringParser(2);

                  if(!error){

                  instruction7();    //block num
                readFunction();
                stringParser(5);
                  }


            if(!error){

                          UART_PutCRLF();
                          UART_CPutString("Displaying waveform (press any key
to exit)");
                        UART_PutCRLF();
                          UART_PutCRLF();

                  done = FALSE;
                  while (!done){
                        j=(8192*(blockNum-1));
                        k=(8192*(blockNum2-1));
                        while (j<(blockNum*8192) &&  k<(blockNum2*8192)){

                        k++;
                        j++;

                        if (UART_cReadChar()){done = TRUE;}

//                      if (j = trigAdr){
//                            TRIGGER_HIGH;
//                            for(i=0 ; i<100 ; i++);
//                            TRIGGER_LOW;
//                      }

                            DAC8A_WriteStall(SPIRAM_ReadByte(j));
                              DAC8B_WriteStall(SPIRAM_ReadByte(k));


                        if (UART_cReadChar()){done = TRUE;}

                  }//end DAC1 while

                        if (UART_cReadChar()){done = TRUE;}

                  }//end DAC while
            }//end error-if

            }else {

                  error = FALSE;
```

```
                  if(!error){
                  instruction6();   //input
                readFunction();
                stringParser(3);
                  }
                  if(!error){
                  instruction2();   //block num
                readFunction();
                stringParser(2);
                  }

                  if(!error){
                  instruction4();   //sample rate
                readFunction();
                stringParser(4);
                  }

                  if(!error){
                  DelSigClock_WritePeriod(sampleRate);
                  DelSigClock_WriteCompareValue(sampleRate>>1);
                  }

//                if(!error){
//                instruction5();
//              readFunction();
//                trigAdr =
hexToDec(rawString[0],rawString[1],rawString[2],rawString[3]);
//              //stringParser(4);
//                }

                  if (!error){
                   UART_CPutString("Saving waveform");
                   UART_PutCRLF();
                   UART_PutCRLF();

                  DelSig_StartAD();
                  for (j=(8192*(blockNum-1)); j<(blockNum*8192); j++){

                  while (!DelSig_fIsDataAvailable()){}

                  Svalue = DelSig_bGetData();
                  SPIRAM_WriteByte(j,Svalue);
                  DelSig_ClearFlag();

                  }//end save for
                  DelSig_StopAD();
                  }//end final if not error
            }//endSaveElse

            ///////////////////////////////
            }//end ifValidMode
      }//end_While1

} //endmain

/*************************************************************************
*/
```

```
/********************* Interrupt Service Routines Below
*********************/
/****************************************************************************
*/
#pragma interrupt_handler DACUpdate_ISR

// DACUpdate_ISR is called at the terminal count of the DACUpdate user
module.
// Since it's clock source is the same as DelSig, setting its period to
// match the DelSig PWM (4*DecimationRate) will cause it to interrupt at the
// same rate as the DelSig's sampling rate. If the samples are only sent to
// the DACs when the variable DACUpdateDone is one, the output sampling rate
// can be controlled.
void DACUpdate_ISR(void)
{

      // Updating the DACs inside the ISR takes more clock cycles
      // than simply setting a flag and exiting. This is because
      // the C-compiler does a full preserve and restore of the
      // CPU context which takes 190+185 CPU cycles.

      DACUpdateDone = 1;
}


void readFunction (void)
{
        int placeCounter = 0;
        finishFlag= FALSE;

            UART_CPutString(">");

        while (!finishFlag) {

                        charIn = UART_cReadChar();
                        while (charIn == 0x00){charIn = UART_cReadChar();}
                                if (placeCounter<7){

                            if (charIn == 0x0d){ //if carriage return
                                            UART_CPutString("\r\n");
                                            UART_CPutString("\r\n");
                                        finishFlag = TRUE;
                                } //end if CR
                                                    //backspace
                                else if ((charIn == 0x08 || charIn ==0x7f) &&
placeCounter > 0 ){ //if backspace
                                        placeCounter--;
                                        UART_CPutString("\x8\x20\x8");
                                } //end if backspace
                                else {
                                                rawString[placeCounter] =
charIn;

UART_PutChar(rawString[placeCounter]);
                                                placeCounter++;
                                        }// end else write into string
```

```
                                    } // end placeCounter if
                        else
                                    {

                                                UART_CPutString("Too many
characters.");

UART_PutChar(0x07);
                                              finishFlag = TRUE;
                                                            //addbell
                                    }

        }//end while (~finsihFlag)

        finishFlag= FALSE ;

}//end readFunction
void stringParser(int parseNum){

      switch (parseNum) {

      case 1:
                        switch (rawString[0]){
                              case 'd':  displayOrcapture = TRUE;
                              break;
                              case 'c':  displayOrcapture = FALSE;
                              break;
                              default:
                                    UART_CPutString("Invalid Mode");
                                    UART_PutChar(0x07);
                                    validMode = FALSE;
                                    UART_PutCRLF();
                                    UART_PutCRLF();
                              break;
                              }
      break;

      case 2:
            switch (rawString[0]){
                        case '1': blockNum = 1;
                        break;
                        case '2': blockNum = 2;
                        break;
                        case '3': blockNum = 3;
                        break;
                        case '4': blockNum = 4;
                        break;
                        default:
                              UART_CPutString("Invalid block #");
                              UART_PutChar(0x07);
                              error = TRUE;
                          UART_PutCRLF();
                              UART_PutCRLF();
                              //M8C_Reset;
                        break;
                        }
      break;
```

```
case 3:
                  switch (rawString[0]){
                        case 'a':
                                    AMUX4_InputSelect(AMUX4_PORT0_1);
                                         inA = TRUE;
                        break;
                        case 'b':  AMUX4_InputSelect(AMUX4_PORT0_7);
                                         inA = FALSE;
                        break;
                        default:
                              UART_CPutString("Invalid input");
                              UART_PutChar(0x07);
                              error = TRUE;
                              UART_PutCRLF();
                              UART_PutCRLF();
                        break;
                  }
break;

case 4:

            switch (rawString[0]){
                      case '1': sampleRate = sampleRateS[0];
                        break;
                        case '2':sampleRate = sampleRateS[1];
                        break;
                        case '3': sampleRate = sampleRateS[2];
                        break;
                        case '4': sampleRate = sampleRateS[3];
                        break;
                        case '5': sampleRate = sampleRateS[4];
                        break;
                        case '6': sampleRate = sampleRateS[5];
                        break;
                        case '7': sampleRate = sampleRateS[6];
                        break;
                        case '8': sampleRate = sampleRateS[7];
                        break;
                        case '9': sampleRate = sampleRateS[8];
                        break;
                        default:
                              UART_CPutString("Invalid sample rate #");
                              UART_PutChar(0x07);
                              error = TRUE;
                              UART_PutCRLF();
                              UART_PutCRLF();
                              //M8C_Reset;
                        break;
                  }
break;

case 5:
      switch (rawString[0]){
                        case '1': blockNum2 = 1;
                        break;
                        case '2': blockNum2 = 2;
```

```
                              break;
                              case '3': blockNum2 = 3;
                              break;
                              case '4': blockNum2 = 4;
                              break;
                              default:
                                    UART_CPutString("Invalid block #");
                                    UART_PutChar(0x07);
                                    error = TRUE;
                                UART_PutCRLF();
                                    UART_PutCRLF();
                              break;
                        }
      break;

      default:
                  M8C_Reset;
                  break;
      }

}
void instruction1(void)
{
            UART_CPutString("Would you like to (d)isplay waveform or
(c)apture waveform?  ");
            UART_PutCRLF();
            UART_CPutString(">");

}

void instruction2(void)
{
            UART_CPutString("Which block to save to? (Choose 1, 2, 3, or 4)
");
            UART_PutCRLF();
            UART_CPutString(">");

}

void instruction3(void)
{
            UART_CPutString("Which block to read from DAC A? (Choose 1, 2, 3,
or 4) ");
            UART_PutCRLF();
            UART_CPutString(">");
}

void instruction4(void)
{
            UART_CPutString("Choose sample rate");
            UART_PutCRLF();
            UART_CPutString("1 for 1.25ksps, 2 for 1.5ksps, 3 for 1.87ksps, 4
for 2.5ksps");
            UART_PutCRLF();
            UART_CPutString("5 for 3.125ksps, 6 for 3.75ksps, 7 for 6.25ksps,
8 for 7.5ksps, 9 for 9.375ksps");
            UART_PutCRLF();
```

```
            UART_CPutString(">");
}

void instruction5(void)
{
            UART_CPutString("Enter address to set trigger on within block in
Hex");
            UART_PutCRLF();
            UART_CPutString(">");

}
void instruction6(void)
{
            UART_CPutString("Which Input would you like to save 'a' or 'b'
?");
            UART_PutCRLF();
            UART_CPutString(">");

}
void instruction7(void)
{
            UART_CPutString("Which block to read from DAC B? (Choose 1, 2, 3,
or 4) ");
            UART_PutCRLF();
            UART_CPutString(">");
}
//char asciiToHex(char addressByte)
//{
//        char output;
//        BOOL lowerCase = FALSE;

//          if(addressByte>= 0x41 && addressByte<= 0x5a)
//          {
//                addressByte -= 0x37;
//
//          }
//
//        if (addressByte >= 0x30  && addressByte <= 0x39){
//
//                                  UART_CPutString("integer");
//                                  UART_PutCRLF();
//                      output = addressByte - 0x30;
//          }
//           if (addressByte >= 0x61  && addressByte <= 0x66){
//
//                                  /// Add conditional if
//                                  UART_CPutString("Upper case");
//                                  UART_PutCRLF();
//                        output = addressByte - 0x51;
//          }
//
//          if (!(addressByte >= 0x30  && addressByte <= 0x39)&&
!(addressByte >= 0x61  && addressByte <= 0x7a))
////        {
////
//////                        invaildFlag=TRUE;
//////                  UART_CPutString("Not valid Hex address");
```

```
//////                         UART_PutChar(0x07);
////        }
//        return output;
//
//}//end asciiToHex

unsigned char ascii_to_hex(unsigned char*  addressString)
{
   unsigned char hundred, ten, unit, value;

//   hundred = (*addressString-0x30)*100;
   ten = (*(addressString + 1)-0x30)*10;
   unit = *(addressString+2)-0x30;

   value = (hundred + ten + unit);
   //printf("\nValue: %#04x \n", value);

   return value;
}

char toLower(char k) {
      if (k >= 'A' && k <= 'Z') {
            return k + 0x20;
      } else {
            return k;
      }
}

unsigned int hexToDec(char c1, char c2, char c3, char c4) {
      int total = 0;

      c1=toLower(c1);
      c2=toLower(c2);
      c3=toLower(c3);
      c4=toLower(c4);

      if (c1 >= 'a' && c1 <= 'z') c1 -= 0x51;
      else c1 -= 0x30;
      if (c2 >= 'a' && c2 <= 'z') c2 -= 0x51;
      else c2 -= 0x30;
      if (c3 >= 'a' && c3 <= 'z') c3 -= 0x51;
      else c3 -= 0x30;
      if (c4 >= 'a' && c4 <= 'z') c4 -= 0x51;
      else c4 -= 0x30;

      total += c1*1000;
      total += c2*100;
      total += c3*10;
      total += c4;

      return total;
 }
```

SPI module functions used:

SPIM_SendTxData

Description:

Initiates the SPI transmission to a slave SPI device. Just before this call, the specified SPI slave device’s signal must be asserted low. This should be done in a user-supplied routine.

C Prototype:

```
void  SPIM_SendTxData(BYTE bSPIMData)
```

Assembly:

```
mov   A, bSPIMData
lcall  SPIM_SendTxData
```

Parameters:

BYTE bSPIMData: Data to be sent to the SPI slave device. It is passed in Accumulator.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadRxData

Description:

Returns a received data byte from a slave device. The Rx Buffer Full flag should be checked before calling this routine, to verify that a data byte has been received.

C Prototype:

```
BYTE  SPIM_bReadRxData(void)
```

Assembly:

```
lcall  SPIM_bReadRxData
mov   bRxData, A
```

Parameters:

None

Return Value:

Data byte received from the slave SPI and returned in the Accumulator.

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadStatus

Description:

Reads and returns the current SPIM Control/Status register.

C Prototype:

```
BYTE  SPIM_bReadStatus(void)
```

Assembly:

```
lcall  SPIM_bReadStatus
and   A, SPIM_SPIM_SPI_COMPLETE | SPIM_SPIM_RX_BUFFER_FULL
jnz   SpimCompleteGetRxData
```

Parameters:

None

Return Value:

Returns status byte read and is returned in the Accumulator. Utilize defined masks to test for specific status conditions. Note that masks can be OR’ed together to test for multiple conditions. Also note that the instance name of the user module is prepended to the symbolic name listed below. For example, if you named the user module SPIM1 when you placed it, the symbolic name of the first mask is SPIM1_SPIM_SPI_COMPLETE.

SPIM Status Masks	Value
SPIM_SPIM_SPI_COMPLETE	0x20
SPIM_SPIM_RX_OVERRUN_ERROR	0x40
SPIM_SPIM_TX_BUFFER_EMPTY	0x10
SPIM_SPIM_RX_BUFFER_FULL	0x08

Side Effects:

The status bits are cleared after this function is called. The A and X registers may be altered by this function.