EEL4511C

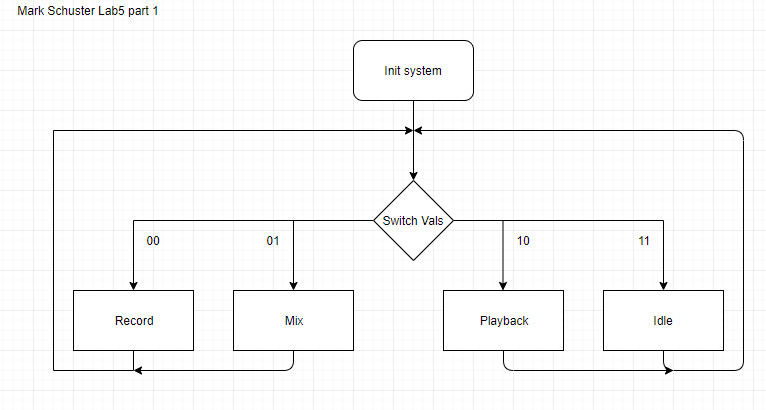
Schuster, Mark

Lab 5

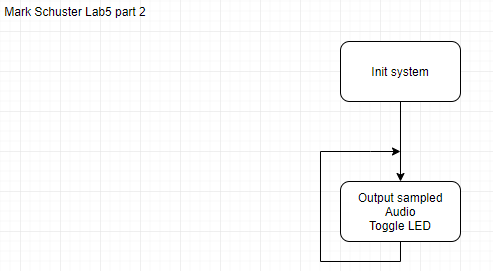
03/19/2018

**1.1 Flowcharts:**

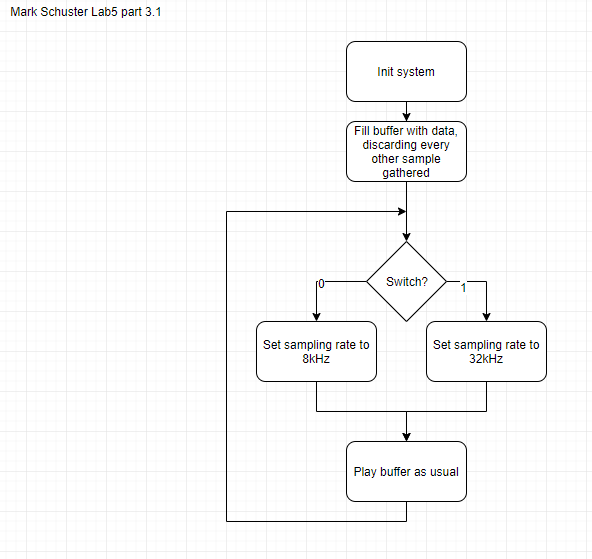
Part 1 flowchart:

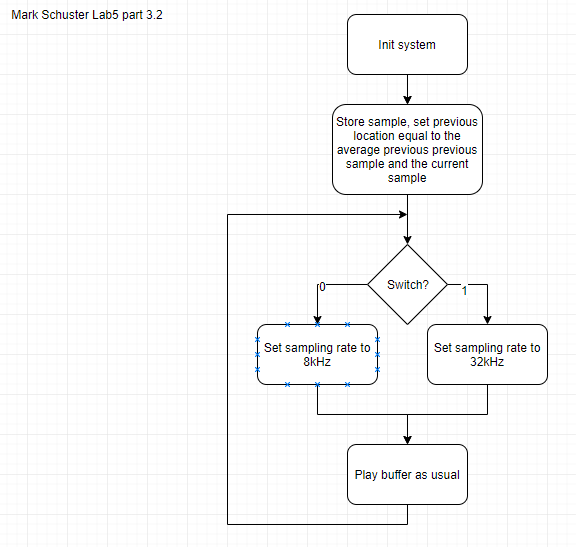


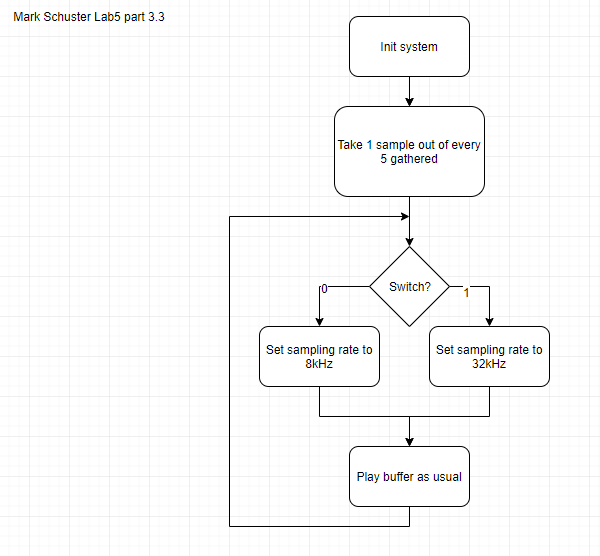
Part 2 flowchart:



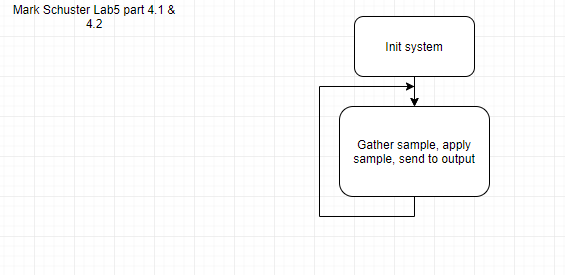
Part 3 flowchart:







Part 4 flowchart:



**1.2 Schematics/Decoding Logic:**

None for this lab.

**1.3 Problems Encountered:**

In this lab I had no major problems aside from a few minor casting errors.

**1.4 Program Code:**

Headers:

**msLib.h**

// File: msLib.h

// Date: 03/08/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

**#ifndef** MSLIB\_H\_

**#define** MSLIB\_H\_

**#include** <DSP2833x\_Device.h>

**#include** "DSP28x\_Project.h"

**#include** "ADC\_Utils.h"

**#include** "Codec\_Utils.h"

**#include** "extSram.h"

**#include** "I2C\_LCD\_Utils.h"

**#include** "Interrupt\_ISRs.h"

**#include** "Interrupt\_Utils.h"

**#include** "LEDs\_switches\_Utils.h"

**#include** "OneToOneI2CDriver.h"

**#include** "Timer1\_Utils.h"

**#endif**

**ADC\_Utils.h**

// File: I2C\_LCD\_Utils.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#ifndef ADCUTILS\_H\_

#define ADCUTILS\_H\_

void initADC(void);

#endif

**Codec\_Utils.h:**

// File: Codec\_Utils.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#ifndef CODECUTILS\_H\_

#define CODECUTILS\_H\_

extern interrupt void codecIsr();

void initCodec(void);

#endif

**I2C\_LCD\_Utils.h**

// File: I2C\_LCD\_Utils.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#ifndef I2CLCDUTILS\_H\_

#define I2CLCDUTILS\_H\_

// Initializes the LCD and clears the display.

void initLCD(void);

// Sends a command to the LCD over I2C.

void sendCmdListLCD(Uint16 \*, Uint16);

// Writes a character to the LCD after it has been

// initialized with "initLCD".

void sendCharLCD(char);

// Writes a string to the LCD using "sendCharLCD".

void sendStringLCD(char\*);

// Send the clear screen to the LCD,

// wiping the display.

void clearLCD(void);

// Set the LCD's cursor the first

// position in the top right corner.

void cursorHomeLCD(void);

#endif

**Interrupt\_ISRs.h**

// File: Interrupt\_ISRs.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

**#ifndef** INTERRUPTISRS\_H\_

**#define** INTERRUPTISRS\_H\_

**enum**{

*REC*,

*MIX*,

*PLAY*,

*IDLE*,

};

**enum**{

*REC\_48*,

*PLAY\_8*,

*PLAY\_32*,

*PLAY\_48*,

};

**interrupt** **void** **timer1Isr**(**void**);

**interrupt** **void** **audioIsr**(**void**);

**#endif**

**Interrupt\_Utils.h**

// File: Interrupt\_Utils.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#ifndef INTERRUPTUTILS\_H\_

#define INTERRUPTUTILS\_H\_

void initInterrupts(void);

#endif

**OneToOneI2CDriver.h**

/\*

\* OneToOneI2CDriver.h

\*

\* Created on: Sep 24, 2016

\* Author: Raz Aloni

\*/

#ifndef ONETOONEI2CDRIVER\_H\_

#define ONETOONEI2CDRIVER\_H\_

/\*

\* <summary>

\* Initializes the I2C to run in Master Mode for a One-To-One connection

\* </summary>

\* <param="slaveAddress">Address of the slave device to write to</param>

\* <param="sysClkMhz">System Clock Frequency in Mhz</param>

\* <param="I2CClkKHz">Desired I2C Clock Frequency in KHz</param>

\*/

void I2C\_O2O\_Master\_Init(Uint16 slaveAddress, float32 sysClkMhz, float32 I2CClkKHz);

/\*

\* <summary>

\* Sends bytes via I2C

\* </summary>

\* <param="values">Pointer to array of bytes to send</param>

\* <param-"length">Length of array</param>

\*/

void I2C\_O2O\_SendBytes(Uint16 \* const values, Uint16 length);

#endif /\* ONETOONEI2CDRIVER\_H\_ \*/

**Timer1\_Utils.h**

// File: Timer1\_Utils.h

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#ifndef TIMER1UTILS\_H\_

#define TIMER1UTILS\_H\_

extern interrupt void timer1Isr();

void initTimer1(void);

#endif

**ADC\_Utils.c**

// File: ADC\_Utils.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#include "msLib.h"

void initADC(void){

InitAdc();

EALLOW;

PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

AdcRegs.ADCTRL1.all = 0x0170;

AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0;

AdcRegs.ADCREFSEL.bit.REF\_SEL = 0x0;

AdcRegs.ADCMAXCONV.all = 0;

AdcRegs.ADCTRL1.bit.CPS = 1;

AdcRegs.ADCTRL2.bit.SOC\_SEQ1 = 1;

return;

}

**Codec\_Utils.c**

// File: Codec\_Utils.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#include "msLib.h"

void initCodec(void){

EALLOW;

PieVectTable.MRINTB = audioIsr; //link it to my interrupt

PieCtrlRegs.PIEIER6.bit.INTx3 = 1;

IER |= M\_INT6;

EDIS;

}

**I2C\_LCD\_Utils.c**

// File: I2C\_LCD\_Utils.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#include "msLib.h"

// Initializes the LCD and clears the display.

void initLCD(void)

{

Uint16 LCDCommandList[] = {0x33,0x32,0x28,0x0F,0x01};

I2C\_O2O\_Master\_Init(0x3F, 150, 400);

sendCmdListLCD(LCDCommandList, sizeof(LCDCommandList)/sizeof(Uint16));

cursorHomeLCD();

return;

}

// Sends a command to the LCD over I2C.

void sendCmdListLCD(Uint16 \*cmdList, Uint16 cmdListLen)

{

for(Uint16 itter=0; itter<cmdListLen; itter++)

{

Uint16 upperNibble, lowerNibble;

lowerNibble = ((cmdList[itter] & 0x000F) << 4) | 0x8;

upperNibble = (cmdList[itter] & 0x00F0) | 0x8;

Uint16 nibbleEnableCmds[] = { (upperNibble | 0x4), upperNibble,

(lowerNibble | 0x4), lowerNibble};

I2C\_O2O\_SendBytes(nibbleEnableCmds, sizeof(nibbleEnableCmds)/sizeof(Uint16));

}

DELAY\_US(1000);

return;

}

// Writes a character to the LCD after it has been

// initialized with "initLCD".

void sendCharLCD(char c){

Uint16 upperNibble, lowerNibble;

lowerNibble = (((Uint16)c & 0x000F) << 4) | 0x9;

upperNibble = ((Uint16)c & 0x00F0) | 0x9;

Uint16 nibbleEnableCmds[] = { (upperNibble | 0x4), upperNibble,

(lowerNibble | 0x4), lowerNibble};

I2C\_O2O\_SendBytes(nibbleEnableCmds, sizeof(nibbleEnableCmds)/sizeof(Uint16));

return;

}

// Writes a string to the LCD using "sendCharLCD".

void sendStringLCD(char\* str){

for(Uint16 itter=0; str[itter]!='\0'; itter++)

sendCharLCD(str[itter]);

return;

}

// Send the clear screen to the LCD,

// wiping the display.

void clearLCD(void)

{

Uint16 clearVal[] = {0x01};

sendCmdListLCD(clearVal, sizeof(clearVal)/sizeof(Uint16));

return;

}

// Set the LCD's cursor the first

// position in the top right corner.

void cursorHomeLCD(void){

Uint16 homeVal[] = {0x02};

sendCmdListLCD(homeVal, sizeof(homeVal)/sizeof(Uint16));

return;

}

**Interrupt\_ISRs.c**

// File: Interrupt\_ISRs.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

**#include** "msLib.h"

Uint16 audioState;

**interrupt** **void** **timer1Isr**()

{

CpuTimer1Regs.TCR.bit.TIF = 1;

**return**;

}

/\* ---PART 1's ISR ---

interrupt void audioIsr(void)

{

enum{REC\_MIX\_NOT\_DONE,REC\_DONE, MIX\_DONE,};

static Uint16 recState = REC;

static Uint32 currentSramIndex = 0;

if( audioState == REC )

{

if(currentSramIndex <= SRAM\_END && recState != REC\_DONE)

sram[currentSramIndex++] = McbspbRegs.DRR2.all;

else

{

setLEDS(0x01);

recState = REC\_DONE;

currentSramIndex = 0;

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

}

int16 temp = McbspbRegs.DRR1.all;

McbspbRegs.DXR1.all = 0x00;

McbspbRegs.DXR2.all = 0x00;

}

else if( audioState == MIX )

{

if(currentSramIndex <= SRAM\_END && recState != MIX\_DONE )

sram[currentSramIndex++] = ((int16)sram[currentSramIndex])/2 + ((int16)McbspbRegs.DRR2.all)/2;

else

{

setLEDS(0x01);

recState = MIX\_DONE;

currentSramIndex = 0;

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

}

int16 temp = McbspbRegs.DRR1.all;

McbspbRegs.DXR1.all = 0x00;

McbspbRegs.DXR2.all = 0x00;

}

else if( audioState == PLAY )

{

if(currentSramIndex <= SRAM\_END)

{

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

McbspbRegs.DXR2.all = sram[currentSramIndex++];

}

else

{

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

McbspbRegs.DXR2.all = 0x00;

currentSramIndex = 0x00;

}

McbspbRegs.DXR1.all = 0x00;

recState = REC\_MIX\_NOT\_DONE;

setLEDS(0x00);

}

else if( audioState == IDLE )

{

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

McbspbRegs.DXR1.all = 0x00;

McbspbRegs.DXR2.all = 0x00;

recState = REC\_MIX\_NOT\_DONE;

currentSramIndex = 0x00;

setLEDS(0x00);

}

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

\*/

/\* ---PART 2's ISR ---

interrupt void audioIsr(void)

{

GpioDataRegs.GPATOGGLE.bit.GPIO14 = 1;

McbspbRegs.DXR1.allGpioDataRegs.GPATOGGLE.bit.GPIO14 = 1; = McbspbRegs.DRR1.all;

McbspbRegs.DXR2.all = McbspbRegs.DRR2.all;

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

\*/

/\* ---PART 3's ISR ---

interrupt void audioIsr(void)

{

static Uint32 currentSramIndex = 0;

if( audioState == REC\_48 )

{

if(currentSramIndex <= SRAM\_END)

{ --- PART3.2 ---

if(currentSramIndex == 0x00)

sram[currentSramIndex] = McbspbRegs.DRR2.all;

else

{

sram[currentSramIndex] = McbspbRegs.DRR2.all;

sram[currentSramIndex-1] = (int16)sram[currentSramIndex-2]/2 + (int16)sram[currentSramIndex]/2;

}

currentSramIndex += 2;

--- PART3.3 ---

static Uint16 state = 0;

if(state < 4)

{ Uint16 temp = McbspbRegs.DRR2.all; }

else

{

sram[currentSramIndex++] = McbspbRegs.DRR2.all;

state = 0;

}

state++;

\*//\*

}

else

{

setLEDS(0x01);

audioState = PLAY\_32;

currentSramIndex = 0;

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

}

int16 temp = McbspbRegs.DRR1.all;

McbspbRegs.DXR1.all = 0x00;

McbspbRegs.DXR2.all = 0x00;

}

else if( audioState == PLAY\_32 || audioState == PLAY\_8 || audioState == PLAY\_48)

{

if(currentSramIndex <= SRAM\_END)

{

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

McbspbRegs.DXR2.all = sram[currentSramIndex++];

}

else

{

Uint16 temp = McbspbRegs.DRR1.all;

temp = McbspbRegs.DRR2.all;

McbspbRegs.DXR2.all = 0x00;

currentSramIndex = 0x00;

}

McbspbRegs.DXR1.all = 0x00;

}

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

\*/

**#define** ALPHA 0.3

/\* ---PART 4.1's ISR ---\*/

**interrupt** **void** **audioIsr**(**void**)

{

**static** Uint32 currentSramIndex = 0;

**if**(currentSramIndex <= SRAM\_END)

{

sram[currentSramIndex++] = McbspbRegs.DRR2.all;

int32 echoIndex = currentSramIndex + (getSwitches()\*480);

**if**(echoIndex < 0)

echoIndex = SRAM\_END + echoIndex;

**float** f = (1-ALPHA)\*(int16)sram[echoIndex] + ALPHA\*(int16)sram[currentSramIndex];

McbspbRegs.DXR2.all = (int16)f;

}

**else**

{

currentSramIndex = 0;

McbspbRegs.DXR2.all = sram[currentSramIndex];

}

int16 temp = McbspbRegs.DRR1.all;

McbspbRegs.DXR1.all = 0x00;

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

/\* ---PART 4.2's ISR ---

interrupt void audioIsr(void)

{

static Uint32 currentSramIndex = 0;

if(currentSramIndex < (SRAM\_END+1)/2)

{

int32 echoIndex = currentSramIndex - (getSwitches()\*12000);

if(echoIndex < 0)

echoIndex = (SRAM\_END+1)/2 - echoIndex;

sram[currentSramIndex] = McbspbRegs.DRR2.all;

float f = (1-ALPHA)\*(int16)sram[currentSramIndex] + ALPHA\*(int16)sram[echoIndex + (SRAM\_END+1)/2];

sram[currentSramIndex + (SRAM\_END+1)/2] = (int16)f;

McbspbRegs.DXR2.all = sram[currentSramIndex + (SRAM\_END+1)/2];

currentSramIndex++;

}

else

{

currentSramIndex = 0;

McbspbRegs.DXR2.all = sram[currentSramIndex];

}

int16 temp = McbspbRegs.DRR1.all;

McbspbRegs.DXR1.all = 0x00;

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

\*/

**Interrupt\_Utils.c**

// File: Interrupt\_Utils.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#include "msLib.h"

void initInterrupts(void){

InitPieCtrl();

InitPieVectTable();

EnableInterrupts();

}

**OneToOneI2CDriver.c**

/\*

\* OneToOneI2CDriver.c

\*

\* Created on: Sep 24, 2016

\* Author: Raz Aloni

\*/

#include <DSP2833x\_Device.h>

#include "OneToOneI2CDriver.h"

/\* Ideal module clock frequency for I2C \*/

static const Uint16 IdealModClockFrqMHz = 12;

/\*

\* <summary>

\* Initializes the GPIO for the I2C

\* </summary>

\*/

static void InitI2CGpio();

/\*

\* <summary>

\* Calculates and sets the ClockDivides for the I2C Module

\* </summary>

\* <param="sysClkMhz">System Clock Frequency in Mhz</param>

\* <param="I2CClkKHz">Desired I2C Clock Frequency in KHz</param>

\*/

static inline void SetClockDivides(float32 sysClkMHz, float32 I2CClkKHz);

/\*

\* <summary>

\* Initializes the I2C to run in Master Mode for a One-To-One connection

\* </summary>

\* <param="slaveAddress">Address of the slave device to write to</param>

\* <param="sysClkMhz">System Clock Frequency in Mhz</param>

\* <param="I2CClkKHz">Desired I2C Clock Frequency in KHz</param>

\*/

void I2C\_O2O\_Master\_Init(Uint16 slaveAddress, float32 sysClkMhz, float32 I2CClkKHz)

{

// Init GPIO

InitI2CGpio();

EALLOW;

// Enable Clock for I2C

SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 1;

// Put I2C into Reset Mode

I2caRegs.I2CMDR.bit.IRS = 0;

// Set Slave Address

I2caRegs.I2CSAR = slaveAddress;

// Set Clocks

SetClockDivides(sysClkMhz, I2CClkKHz);

// Release from Reset Mode

I2caRegs.I2CMDR.bit.IRS = 1;

EDIS;

}

/\*

\* <summary>

\* Sends bytes via I2C

\* </summary>

\* <param="values">Pointer to array of bytes to send</param>

\* <param-"length">Length of array</param>

\*/

void I2C\_O2O\_SendBytes(Uint16 \* const values, Uint16 length)

{

// Set to Master, Repeat Mode, TRX, FREE, Start

I2caRegs.I2CMDR.all = 0x66A0;

// Write values to I2C

for (Uint16 i = 0; i < length; i++)

{

// Wait if Transmit is not ready

while(!I2caRegs.I2CSTR.bit.ARDY);

I2caRegs.I2CDXR = values[i];

}

// Stop Bit

I2caRegs.I2CMDR.bit.STP = 1;

}

/\*

\* <summary>

\* Calculates and sets the ClockDivides for the I2C Module

\* </summary>

\* <param="sysClkMhz">System Clock Frequency in Mhz</param>

\* <param="I2CClkKHz">Desired I2C Clock Frequency in KHz</param>

\*/

static inline void SetClockDivides(float32 sysClkMHz, float32 I2CClkKHz)

{

/\* Calculate Module Clock Frequency - Must be between 7-12 MHz

\* Module Clock Frequency = sysClkMhz/(IPSC + 1)

\*/

Uint16 IPSC = (Uint16)(sysClkMHz/IdealModClockFrqMHz);

/\* Calculate Divide Downs for SCL

\* FreqMClk = sysClkMHz/((IPSC + 1)[(ICCL + d) + (ICCH + d)])

\*

\* Assume an even clock size -> ICCH == ICCL

\* ICCL = ICCH = sysclkMHz/(2000 \* I2CClkKHz \* (IPSC + 1)) - d

\*/

// Find value for d

Uint16 d = 5;

if (IPSC < 2)

{

d++;

if (IPSC < 1)

{

d++;

}

}

Uint16 ICCLH = (Uint16)(1000 \* sysClkMHz/(2 \* I2CClkKHz \* (IPSC + 1)) - d);

// Set values

I2caRegs.I2CPSC.all = IPSC;

I2caRegs.I2CCLKL = ICCLH;

I2caRegs.I2CCLKH = ICCLH;

}

/\*

\* <summary>

\* Initializes the GPIO for the I2C

\* </summary>

\*/

static void InitI2CGpio()

{

EALLOW;

/\* Enable internal pull-up for the selected pins \*/

// Pull-ups can be enabled or disabled disabled by the user.

// This will enable the pullups for the specified pins.

// Comment out other unwanted lines.

GpioCtrlRegs.GPBPUD.bit.GPIO32 = 0; // Enable pull-up for GPIO32 (SDAA)

GpioCtrlRegs.GPBPUD.bit.GPIO33 = 0; // Enable pull-up for GPIO33 (SCLA)

/\* Set qualification for selected pins to asynch only \*/

// This will select asynch (no qualification) for the selected pins.

// Comment out other unwanted lines.

GpioCtrlRegs.GPBQSEL1.bit.GPIO32 = 3; // Asynch input GPIO32 (SDAA)

GpioCtrlRegs.GPBQSEL1.bit.GPIO33 = 3; // Asynch input GPIO33 (SCLA)

/\* Configure SCI pins using GPIO regs\*/

// This specifies which of the possible GPIO pins will be I2C functional pins.

// Comment out other unwanted lines.

GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 1; // Configure GPIO32 for SDAA operation

GpioCtrlRegs.GPBMUX1.bit.GPIO33 = 1; // Configure GPIO33 for SCLA operation

EDIS;

}

**Timer1\_Utils.c**

// File: Timer1\_Utils.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

#include "msLib.h"

void initTimer1(void){

InitCpuTimers();

ConfigCpuTimer(&CpuTimer1, 150, 1E5);

EALLOW;

PieVectTable.XINT13 = timer1Isr;

IER |= M\_INT13;

EDIS;

CpuTimer1.RegsAddr->TCR.bit.TSS = 0;

}

**main.c**

// File: main.c

// Date: 02/24/2018

// Name: Mark Schuster

// Class: EEE4511C (DSP)

**#include** <DSP2833x\_Device.h>

**#include** "DSP28x\_Project.h"

**#include** "AIC23.h"

**#include** "InitAIC23.h"

**#include** "msLib.h"

**extern** Uint16 audioState;

// Simple enum for bool values.

**enum**{

*FALSE*,

*TRUE*,

};

// Function prototypes:

Uint16 **main**(**void**)

{

// Disable the WDT, and init the phase lock loop.

**InitSysCtrl**();

initLEDSAndSwitches();

InitMcBSPb();

InitSPIA();

InitAIC23();

initCodec();

initExtSRAM();

setLEDS(0x00);

/\* --- PART 2 ---

SpiTransmit(CLKsampleratecontrol(SR8));

SpiTransmit(CLKsampleratecontrol(SR32));

\*/

/\* --- PART 3.1 & PART 3.4 --- \*/

SpiTransmit(CLKsampleratecontrol(SR48));

/\* --- PART 3.2 --- \*/

// SpiTransmit(CLKsampleratecontrol(SR32));

// audioState = REC\_48;

**while**(*TRUE*)

{

/\* --- PART 1 ---

if( !(getSwitches() & 2) && !(getSwitches() & 1) )

audioState = REC;

else if( !(getSwitches() & 2) && (getSwitches() & 1) )

audioState = MIX;

else if( (getSwitches() & 2) && !(getSwitches() & 1) )

audioState = PLAY;

else if( (getSwitches() & 2) && (getSwitches() & 1) )

audioState = IDLE;

\*/

/\* --- PART 3.1 ---

if( !(getSwitches() & 1) && audioState != REC\_48)

{

SpiTransmit(CLKsampleratecontrol(SR8));

}

else if( (getSwitches() & 1) && audioState != REC\_48)

{

SpiTransmit(CLKsampleratecontrol(SR32));

}

\*/

/\* --- PART 3.2 & PART 3.3 ---

static Uint16 set = 32;

if( !(getSwitches() & 1) && audioState != REC\_48)

{

if(set == 32)

{

SpiTransmit(CLKsampleratecontrol(SR8));

set = 8;

}

}

else if( (getSwitches() & 1) && audioState != REC\_48)

{

if(set == 8)

{

SpiTransmit(CLKsampleratecontrol(SR32));

set = 32;

}

}

\*/

/\* --- PART 3.4 --- \*/

}

**return** 0;

}

**1.5 Program Description:**

Use the switches to determine whether to sample audio input, mix the recorded audio input, stand idle, or the stored samples.

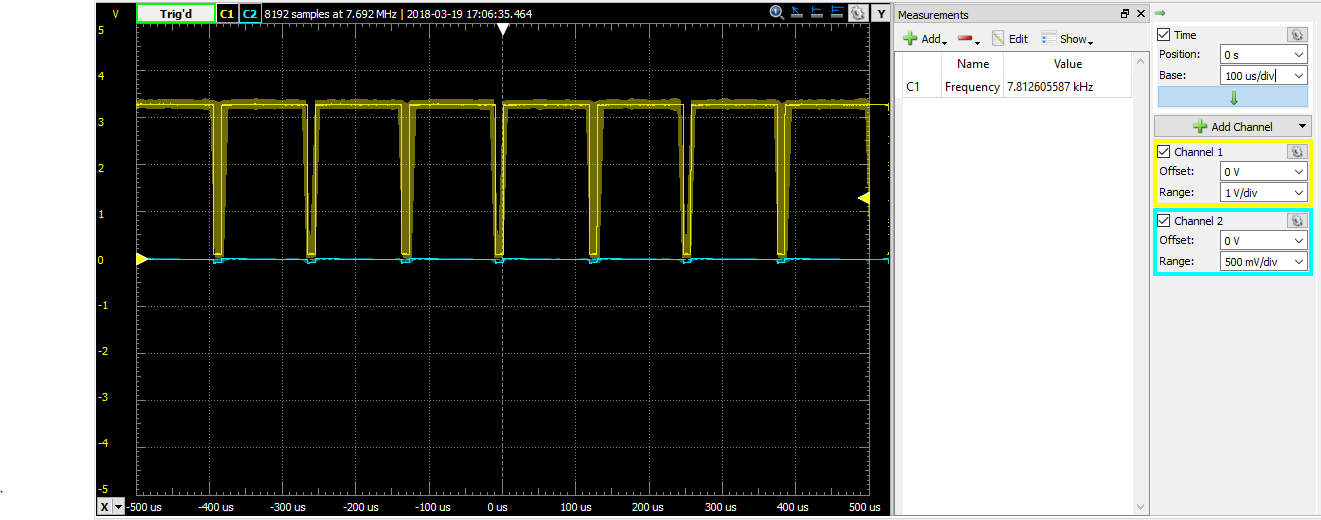
Simply passed the samples to output and toggle an LED.

Take in samples and interpolate them by skipping every other location and setting the skipped location to the average of the surrounding samples.

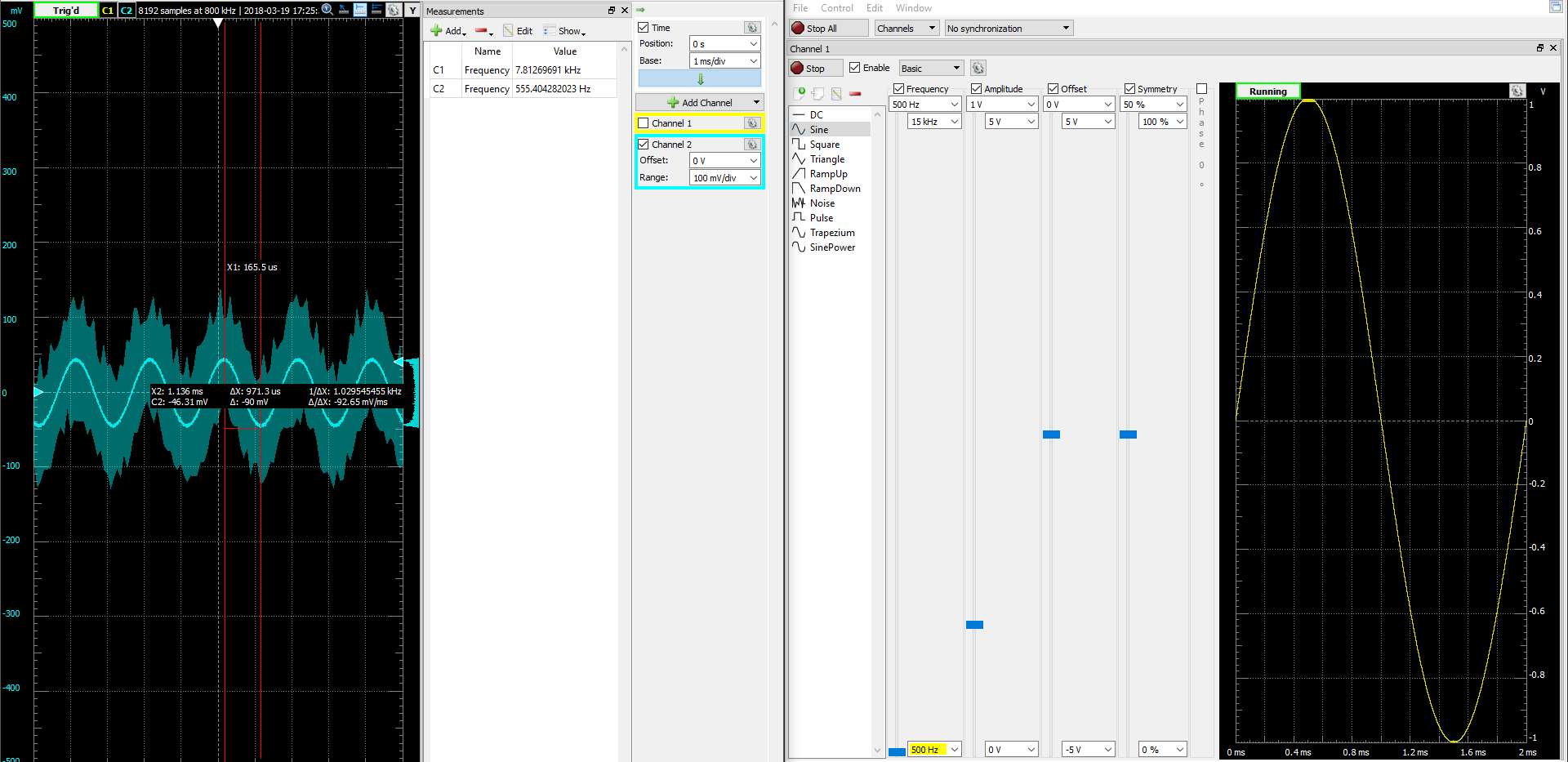
Take in samples and apply a special filter to the set and output them in real-time.

**1.6 Question Answers:**

2.1 Screenshot of frequency:

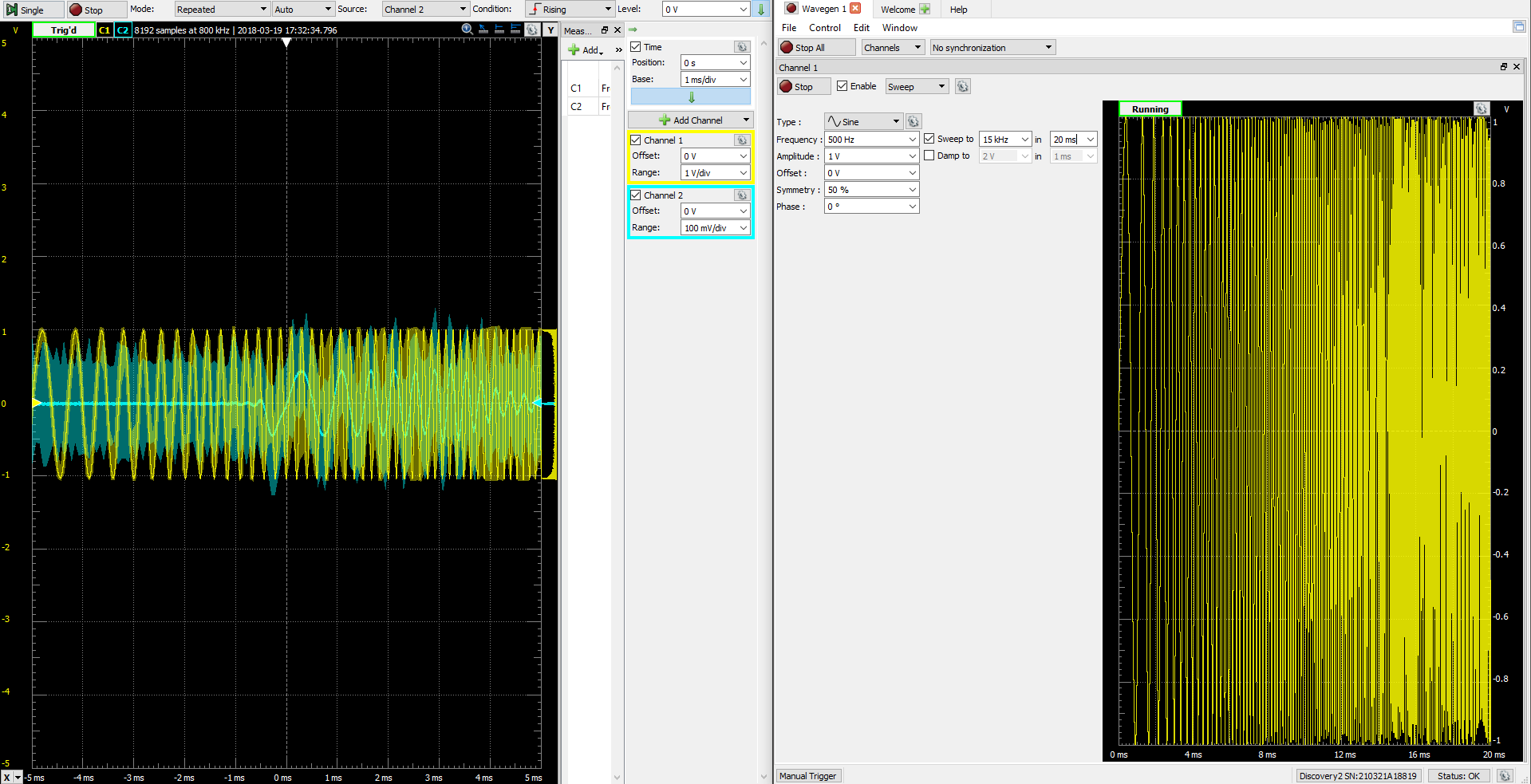


2.2



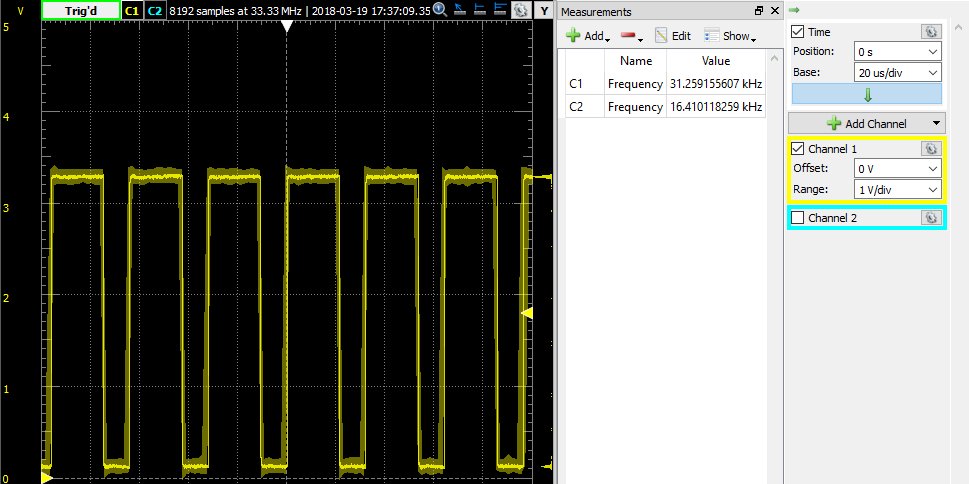
Gain of 0.05, and a phase shift of approximately 270 degrees.

2.3

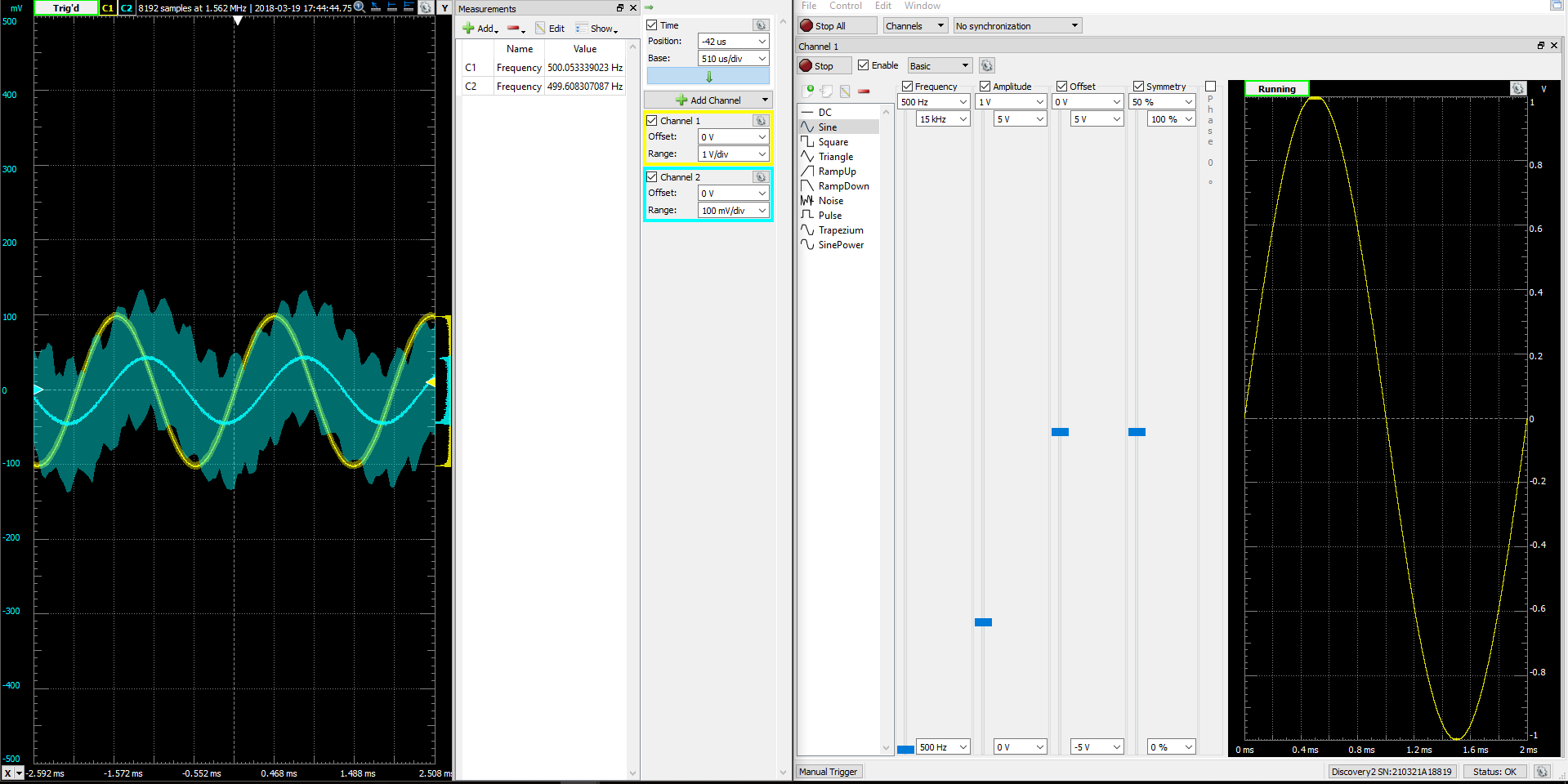


Note the amplitude attenuate to zero as the frequency approaches 4kHz. After 4kHz (the Nyquist rate) the sampling aliases to 1Hz.

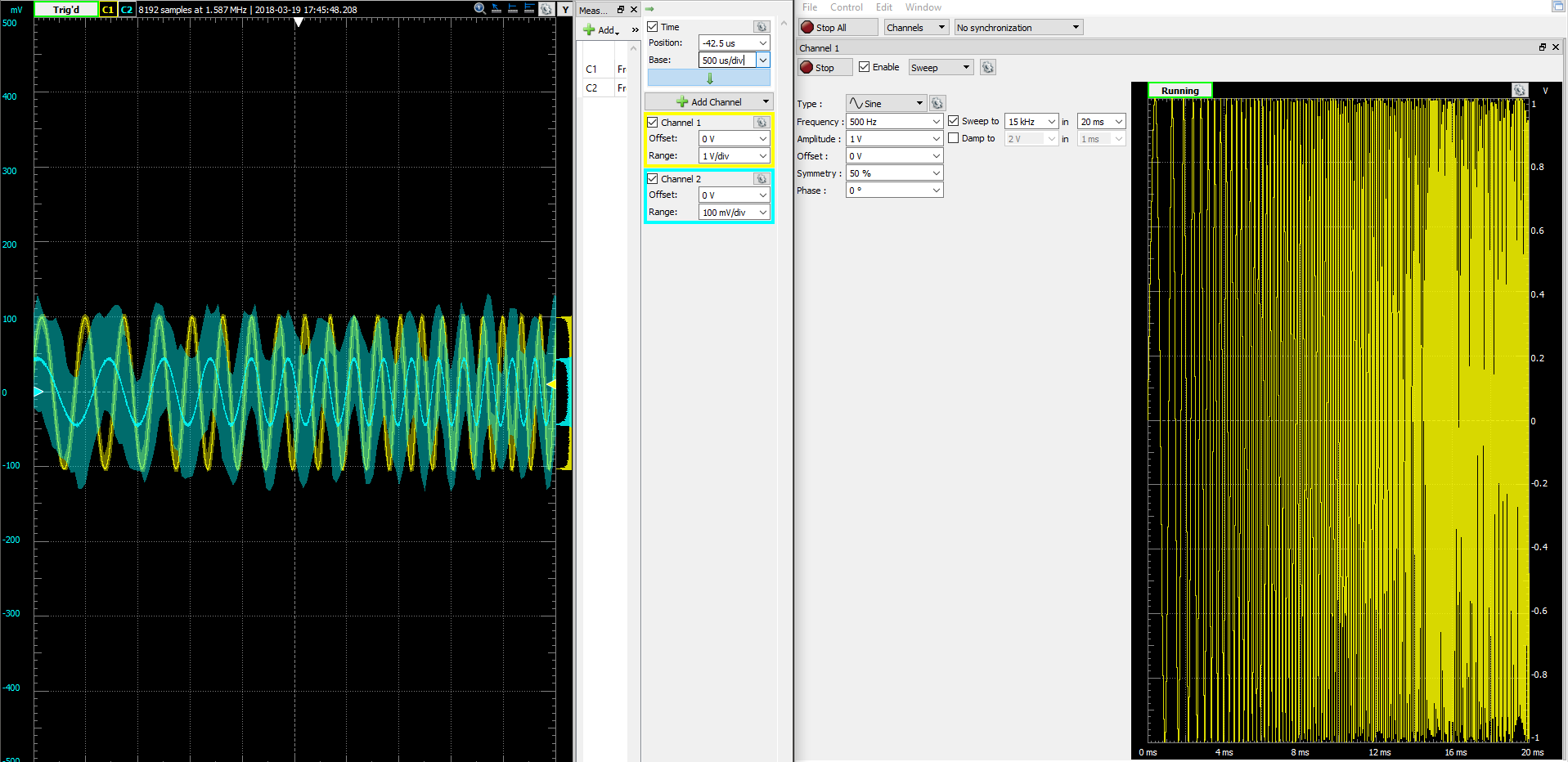
3.4



Shows the sampling rate of 32kHz.



Gain of 0.05, and a phase shift of approximately 270 degrees still for 32 kHz Sf.



Note that since the new Nyquist rate is 16kHz the amplitude of the output never drops off.

4.1