**REQUIREMENTS NOT MET**

**N/A**

**PROBLEMS ENCOUNTERED**

**N/A**

**FUTURE WORK/APPLICATIONS**

Digital to analog converters are frequently used in:  
synthesizers  
measuring devices  
game controllers  
music players  
DAC’s can also be used for transferring analog voltages across telephone lines for dial up internet, or cable lines for cable internet.

**PRE-LAB EXERCISES**

**i. Why might you be unable to generate a desired frequency with this method of using an interrupt? Refer to the disassembly of the interrupt service routine. Additionally, temporarily change the optimization level of your compiler to -O1. Are the results any different? Why or why not?   
 Because the compiled code is more inefficient than hand writing raw assembly code, the counter runs for a few extra clock cycles. Hence, lowering the frequency.  
 When changing the optimization level, the frequency increases due to the elimination of extra instructions**

**ii. Would a method of synchronous polling (i.e., a method with no interrupts) result in the same issue identified in the previous exercise? In other words, would the desired frequency not initially met now be achieved? Alter your program to check your answer, and then take a screenshot of the waveform generated, again denoting a precise frequency measurement of this waveform within the screenshot.  
 No, it would not be fixed. The assembly code is still bloated.  
A screen shot of a graph

Description automatically generated  
 Figure 0: Waveform after using synchronous polling**

**iii. What is the correlation between the amount of data points used to recreate the waveform and the overall quality of the waveform?**

**As more data points are used, the smoother the waveform is.**

**PSEUDOCODE/FLOWCHARTS**

**SECTION 1**

**A diagram of a computer system

Description automatically generated  
Figure 1: Flowchart for “lab8\_1.C”**

**SECTION 2**

**A diagram of a process

Description automatically generated  
  
Figure 2: Flowchart for “lab8\_2.C”**

**SECTION 3**

**A diagram of a process flow

Description automatically generated  
Figure 2: Flowchart for “lab8\_2b.C”**

**PROGRAM CODE**

**SECTION 1**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Lab 8, Section 1

//Name: Steven Miller

//Class #: 11318

//PI Name: Anthony Stross

//Description: outputs constant 1.7 volts using DAC

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <avr/io.h>

#define onepointsevenvolts 0xAE1

#define maxvoltage 4095 //2.5v converted to decimal

#define slope (2.5/4095)

extern void clock\_init(void);

int main(void)

{

//set system clock to 32 mhz

clock\_init();

//initialize DAC

dac\_init();

while (1)

{

//wait until channel 0 is empty (conversion is finished)

while(!(DACA.STATUS & DAC\_CH0DRE\_bm))

{

//do nothing

}

DACA.CH0DATA = onepointsevenvolts;

}

}

void dac\_init(void)

{

//use only channel 0

DACA.CTRLB = DAC\_CHSEL\_SINGLE\_gc;

//use arefb

DACA.CTRLC = DAC\_REFSEL\_AREFB\_gc;

//enable channel 0

DACA.CTRLA = DAC\_CH0EN\_bm | DAC\_ENABLE\_bm;

}

**SECTION 2a**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Lab 8, Section 2a

//Name: Steven Miller

//Class #: 11318

//PI Name: Anthony Stross

//Description: outputs sinewave at 988 hz

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <avr/io.h>

#include <avr/interrupt.h>

#define maxvoltage 4095 //2.5v converted to decimal

#define slope (2.5/4095)

extern void clock\_init(void);

extern *uint16\_t* sine\_wave[256];

volatile *uint8\_t* index = 255;

int main(void)

{

//set system clock to 32 mhz

clock\_init();

//initialize DAC

dac\_init();

//initialize timer counter

tcc0\_init();

//start tcc0 timer

TCC0.CTRLA = TC\_CLKSEL\_DIV2\_gc;

//enable global interrupts

PMIC.CTRL = PMIC\_MEDLVLEN\_bm;

sei();

//dummy loop

while (1)

{

}

}

void dac\_init(void)

{

//use only channel 0

DACA.CTRLB = DAC\_CHSEL\_SINGLE\_gc;

//use arefb

DACA.CTRLC = DAC\_REFSEL\_AREFB\_gc;

//enable channel 0

DACA.CTRLA = DAC\_CH0EN\_bm | DAC\_ENABLE\_bm;

}

void tcc0\_init(void)

{

*uint16\_t* period = 63;

*int8\_t* offset = -21;

//INITIALIZE COUNT REGISTER

TCC0.CNT = 0;

//initialize tcc0 period register

TCC0.PER = period + offset;

TCC0.CNT = 0;

//enable tcc0 overflow interrupts as medium priority

TCC0.INTCTRLA = TC\_OVFINTLVL\_MED\_gc;

};

ISR(TCC0\_OVF\_vect)

{

//reset counter

TCC0.CNT = 0;

//begin conversion

DACA.CH0DATA = sine\_wave[index];

if(index == 0)

{

index = 255;

}

else

{

index = index-1;

}

}

**SECTION 2b**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Lab 8, Section 2b

//Name: Steven Miller

//Class #: 11318

//PI Name: Anthony Stross

//Description: outputs sinewave at 1567.98 Hz

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <avr/io.h>

#include <avr/interrupt.h>

#define maxvoltage 4095 //2.5v converted to decimal

#define slope (2.5/4095)

extern void clock\_init(void);

extern *uint16\_t* sine\_wave[256];

volatile *uint8\_t* index = 255;

int main(void)

{

//set system clock to 32 mhz

clock\_init();

//initialize DAC

dac\_init();

//initialize timer counter

tcc0\_init();

//start tcc0 timer

TCC0.CTRLA = TC\_CLKSEL\_DIV2\_gc;

while (1)

{

while(!(DACA.STATUS & DAC\_CH0DRE\_bm))

{

//do nothing

}

DACA.CH0DATA = sine\_wave[index];

if(index == 0)

{

index = 255;

}

else

{

index = index-1;

}

}

}

void dac\_init(void)

{

//use only channel 0

DACA.CTRLB = DAC\_CHSEL\_SINGLE\_gc| DAC\_CH0TRIG\_bm;

//use arefb

DACA.CTRLC = DAC\_REFSEL\_AREFB\_gc;

//enable channel 0

DACA.CTRLA = DAC\_CH0EN\_bm | DAC\_ENABLE\_bm;

//enable conversion trigger on channel event 0

DACA.EVCTRL = DAC\_EVSEL\_0\_gc;

}

void tcc0\_init(void)

{

*uint16\_t* period = 40;

*int8\_t* offset = 0;

//INITIALIZE COUNT REGISTER

TCC0.CNT = 0;

//initialize tcc0 period register

TCC0.PER = period + offset;

//set tcc0 as event channel 0 source

EVSYS.CH0MUX = EVSYS\_CHMUX\_TCC0\_OVF\_gc;

};

**APPENDIX**

A screen shot of a computer

Description automatically generated **Figure x:Screenshot of section 1**

**A screen shot of a graph

Description automatically generated  
Figure x: Screenshot of section 2a**

**A screen shot of a graph

Description automatically generated  
Figure x: Screenshot of section 2a. The maximum frequency that can be obtained it 1.06khz**

**A screen shot of a graph

Description automatically generated**

**Figure X: Screenshot of section 2b.**

**Clock.s**

/\* clock.s

\*

\* Last Updated: 21 June 2023

\* Authors: Wes, Dr. Schwartz

\*/

#include <avr/io.h>

.section .text

.global clock\_init

clock\_init:

push r24

ldi r24, OSC\_RC32MEN\_bm

sts OSC\_CTRL, R24 ;Enables the 32MHz internal oscillator

check32MHzStatus:

lds r24, OSC\_STATUS

;Ensure that the 32MHz clock is ready before proceeding

sbrs r24, OSC\_RC32MRDY\_bp

rjmp check32MHzStatus

;Writing to CCP disables interrupts for a certain number of cycles

;to give the clock time to switch sources. It also enables writes to certain registers.

ldi r24, 0xD8

sts CPU\_CCP, r24

;Finally, select the now-ready 32MHz oscillator as the new clock source.

ldi r24, 0x01

sts CLK\_CTRL, r24

skip32MHZ\_enable:

;CPU CLK prescaler settings

;Use values that are powers of 2 from 1 to 512 (1, 2, 4, 8, 16, ..., 512) for A. See Table 7-2 in the manual.

;You can also change B/C. See Table 7-3 in the manual.

ldi r24, 0xD8

sts CPU\_CCP, r24

ldi r24, ((0x00 << 2) | (0x00 << 0)) ;32MHz

;ldi r24, ((0x05<<2) | (0x00<<0)) ;4MHz

sts CLK\_PSCTRL, r24

pop r24

ret

**sinewave\_table.C**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//sinewave\_table.c

//Name: Steven Miller

//Class #: 11318

//PI Name: Anthony Stross

//Description: array that contains data points to make a sinewave from 0 to 4095

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <avr/io.h>

volatile *uint16\_t* sine\_wave[256] =

{

0x800,0x832,0x864,0x896,0x8c8,0x8fa,0x92c,0x95e,

0x98f,0x9c0,0x9f1,0xa22,0xa52,0xa82,0xab1,0xae0,

0xb0f,0xb3d,0xb6b,0xb98,0xbc5,0xbf1,0xc1c,0xc47,

0xc71,0xc9a,0xcc3,0xceb,0xd12,0xd39,0xd5f,0xd83,

0xda7,0xdca,0xded,0xe0e,0xe2e,0xe4e,0xe6c,0xe8a,

0xea6,0xec1,0xedc,0xef5,0xf0d,0xf24,0xf3a,0xf4f,

0xf63,0xf76,0xf87,0xf98,0xfa7,0xfb5,0xfc2,0xfcd,

0xfd8,0xfe1,0xfe9,0xff0,0xff5,0xff9,0xffd,0xffe,

0xfff,0xffe,0xffd,0xff9,0xff5,0xff0,0xfe9,0xfe1,

0xfd8,0xfcd,0xfc2,0xfb5,0xfa7,0xf98,0xf87,0xf76,

0xf63,0xf4f,0xf3a,0xf24,0xf0d,0xef5,0xedc,0xec1,

0xea6,0xe8a,0xe6c,0xe4e,0xe2e,0xe0e,0xded,0xdca,

0xda7,0xd83,0xd5f,0xd39,0xd12,0xceb,0xcc3,0xc9a,

0xc71,0xc47,0xc1c,0xbf1,0xbc5,0xb98,0xb6b,0xb3d,

0xb0f,0xae0,0xab1,0xa82,0xa52,0xa22,0x9f1,0x9c0,

0x98f,0x95e,0x92c,0x8fa,0x8c8,0x896,0x864,0x832,

0x800,0x7cd,0x79b,0x769,0x737,0x705,0x6d3,0x6a1,

0x670,0x63f,0x60e,0x5dd,0x5ad,0x57d,0x54e,0x51f,

0x4f0,0x4c2,0x494,0x467,0x43a,0x40e,0x3e3,0x3b8,

0x38e,0x365,0x33c,0x314,0x2ed,0x2c6,0x2a0,0x27c,

0x258,0x235,0x212,0x1f1,0x1d1,0x1b1,0x193,0x175,

0x159,0x13e,0x123,0x10a,0xf2,0xdb,0xc5,0xb0,

0x9c,0x89,0x78,0x67,0x58,0x4a,0x3d,0x32,

0x27,0x1e,0x16,0xf,0xa,0x6,0x2,0x1,

0x0,0x1,0x2,0x6,0xa,0xf,0x16,0x1e,

0x27,0x32,0x3d,0x4a,0x58,0x67,0x78,0x89,

0x9c,0xb0,0xc5,0xdb,0xf2,0x10a,0x123,0x13e,

0x159,0x175,0x193,0x1b1,0x1d1,0x1f1,0x212,0x235,

0x258,0x27c,0x2a0,0x2c6,0x2ed,0x314,0x33c,0x365,

0x38e,0x3b8,0x3e3,0x40e,0x43a,0x467,0x494,0x4c2,

0x4f0,0x51f,0x54e,0x57d,0x5ad,0x5dd,0x60e,0x63f,

0x670,0x6a1,0x6d3,0x705,0x737,0x769,0x79b,0x7cd

};