REQUIREMENTS NOT MET

N/A

PROBLEMS ENCOUNTERED

The main issue I had while configuring the DMA was that I initially had the destination address fixed. However, it should have been incrementing and reloading after each burst since the DAC data register is two bytes, so it was just writing to the low byte, then overwriting it. I had a lot of issues in part 5, but the main one was that the notes initially played forever. Then when I tried implementing logic to stop the notes, my program would break when I hit the 'S' key instead of switching to the triangle LUT. I needed to reload the TRFCNT every time the 'S' key was pressed to help solve this among removing some polling logic.

FUTURE WORK/APPLICATIONS

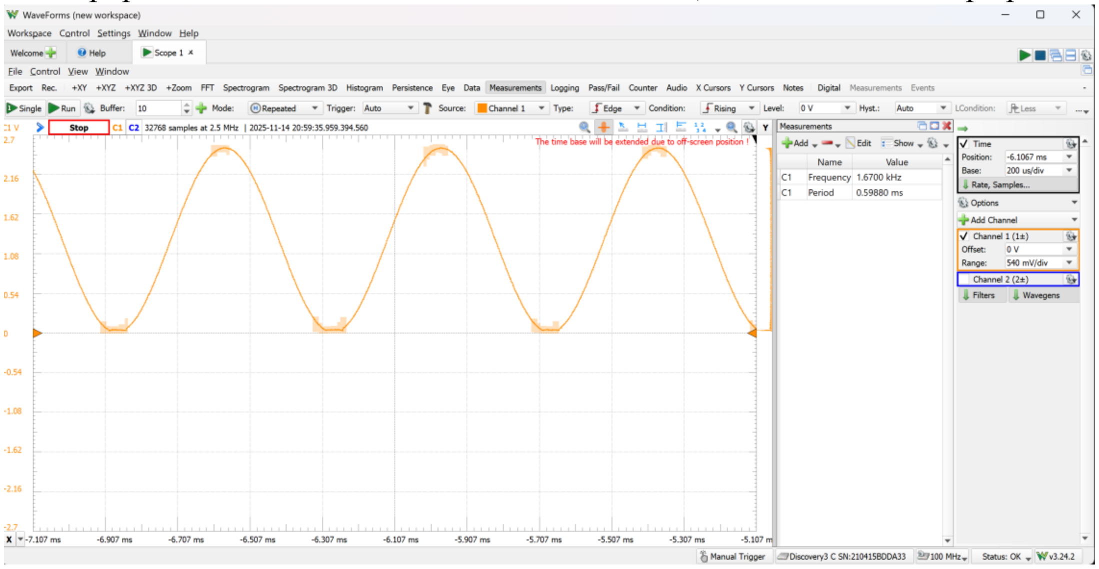
With more time and resources, this design could be expanded into a fully featured embedded synthesizer capable of generating multiple waveforms or layered voices. Additional user controls, such as an LCD, keypad, or USB-MIDI interface, could allow real-time waveform selection, octave shifting, and effects. The same DMA-driven DAC framework could also be adapted for broader applications in digital audio, signal processing, or real-time instrumentation.

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?

a. You might be unable to generate a desired frequency with this method of using an interrupt because the timer overflow is happening faster than the CPU can finish the ISR. From the disassembly, you can see that each ISR call has a lot of instructions (automatic push/pop of registers, the while loop that waits for DAC_CH0DRE, the table lookup, the write to CH0DATA, the branch back, etc.), and each instruction costs clock cycles. At higher target frequencies, the time between overflows becomes so short that there aren't enough cycles at 32 MHz to run all of those instructions 256 times per period. When this happens, the CPU spends all its time in the ISR, the effective update rate "caps out," and changing the desired frequency or PER value has little to no effect.

b. The results were slightly different because with -O1, the compiler removes some redundant instructions and shortens the ISR slightly, so the maximum achievable waveform frequency increases (for example, from ~1.06 kHz at -O0 to ~1.67 kHz at -O1). However, the behavior remains the same: the ISR still requires a minimum number of CPU cycles, and once the timer interrupt period is shorter than the ISR execution time, the CPU cannot keep up.

c. 
WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
Single Run Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto
Stop C1 C2 32768 samples at 2.5 MHz | 2025-11-14 20:59:35.959.394.560
The time base will be extended due to off-screen position !
2.7
2.16
1.62
1.08
0.54
0
-0.54
-1.08
-1.62
-2.16
-2.7
-7.107 ms
-6.907 ms
-6.707 ms
-6.507 ms
-6.307 ms
-6.107 ms
-5.907 ms
-5.707 ms
-5.507 ms
-5.307 ms
-5.107 ms
Time Position: -6.1067 ms Base: 200 us/div
Channel 1 (1±) Offset: 0 V Range: 540 mV/div
Channel 2 (2±)
Manual Trigger Discovery3 C SN:210415BDDA33 100 MHz Status: OK v3.24.2

Measurements

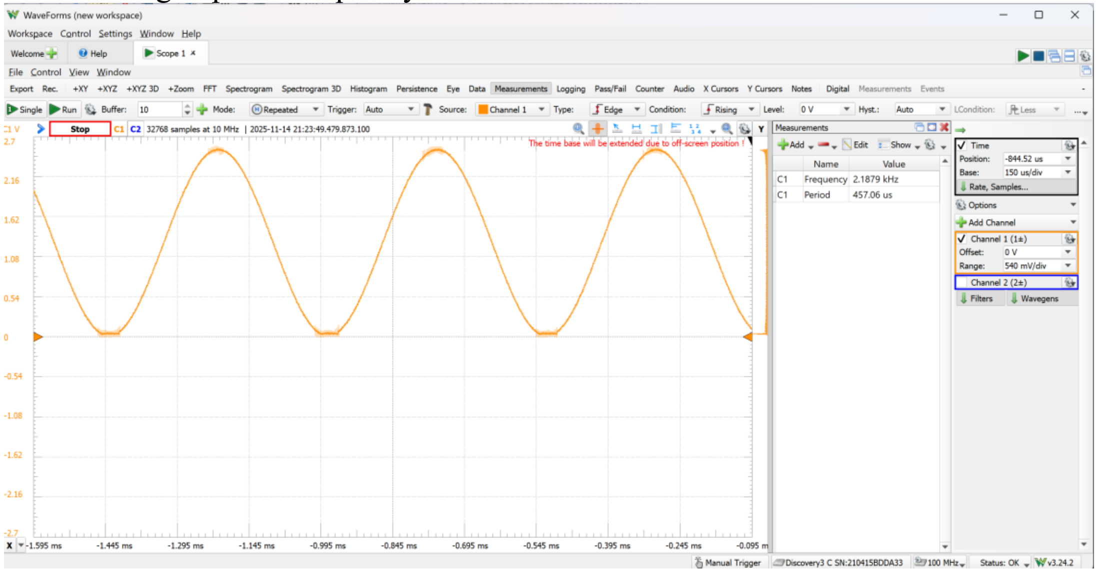
	Name	Value
C1	Frequency	1.6700 kHz
C1	Period	0.59880 ms

Figure 1: Waveform that does not meet the desired accuracy using -O1 optimization.

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.

a. A method of synchronous polling would still result in the same fundamental issue, but at a higher limiting frequency. With polling, there is no interrupt entry/exit overhead or automatic push/pop of registers, so each update of the DAC takes fewer CPU cycles, and the maximum achievable waveform frequency increases. However, the CPU still has to execute the polling loop, check the timer flag, and write the next sample every 1/(f_target * 256) seconds. Once that period becomes shorter than the time needed to run the polling code, the processor again cannot keep up and the actual output frequency no longer tracks the desired one within ±2%.

b. The desired frequency that could not be met with the interrupt-driven method would now be achieved with synchronous polling. When I switched to polling, I was able to generate a waveform at about 2.188 kHz, which allows us to go past the target frequency that previously failed using interrupts. This shows that eliminating interrupt overhead lets the CPU reach that particular frequency, even though there is still a new, higher frequency above this where the same type of limitation appears again. Additionally, with both of these methods (O1 optimization and polling), I was able to go up to a frequency of 3.383 kHz.

c. 
WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Run
Stop
C1 C2
2.7
2.16
1.62
1.08
0.54
0
-0.54
-1.08
-1.62
-2.16
-2.7
-1.595 ms
-1.445 ms
-1.295 ms
-1.145 ms
-0.995 ms
-0.845 ms
-0.695 ms
-0.545 ms
-0.395 ms
-0.245 ms
-0.095 ms
Options
Add Channel
Filters
Waveguns
Rate, Samples...

Buffer: 10 | Mode: Repeated | Trigger: Auto | Source: Channel 1 | Type: Edge | Condition: Rising | Level: 0 V | Hyst.: Auto

32768 samples at 10 MHz | 2025-11-14 21:23:49.479.873.100

The time base will be extended due to off-screen position !

Measurements

	Name	Value
C1	Frequency	2.1879 kHz
C1	Period	457.06 us

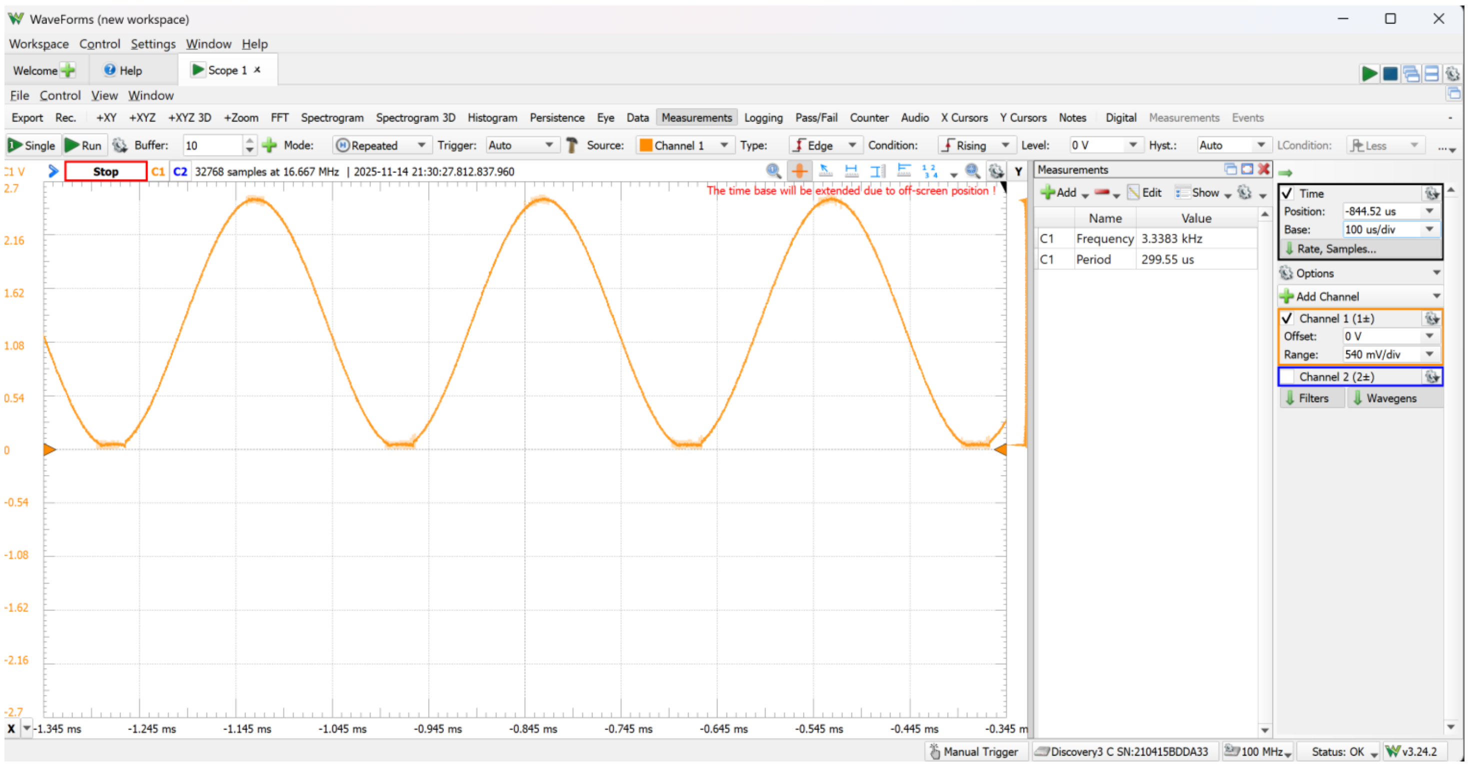
Time — Position: -844.52 us; Base: 150 us/div

Channel 1 (1±) — Offset: 0 V; Range: 540 mV/div

Channel 2 (2±)

Manual Trigger | Discovery3 C SN:210415BDDA33 | 100 MHz | Status: OK | v3.24.2

Figure 2: Screenshot of how quickly/accurately the waveform can run using polling with no optimization.


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Run
Stop C1 C2
2.7
2.16
1.62
1.08
0.54
0
-0.54
-1.08
-1.62
-2.16
-2.7
-1.345 ms -1.245 ms -1.145 ms -1.045 ms -0.945 ms -0.845 ms -0.745 ms -0.645 ms -0.545 ms -0.445 ms -0.345 ms
Add Edit Show
Options
Add Channel
Filters Wavegens
Manual Trigger

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

32768 samples at 16.667 MHz | 2025-11-14 21:30:27.812.837.960

The time base will be extended due to off-screen position !

Measurements

	Name	Value
C1	Frequency	3.3383 kHz
C1	Period	299.55 us

Time
Position: -844.52 us
Base: 100 us/div
Rate, Samples...

Channel 1 (1±)
Offset: 0 V
Range: 540 mV/div
Channel 2 (2±)

Discovery3 C SN:210415BDDA33 100 MHz Status: OK v3.24.2

Figure 3: Screenshot of how quickly/accurately the waveform can run using polling with -O1 optimization.

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

a. Using more data points in the lookup table produces a waveform that more closely resembles a true sine wave. With a higher sample count, the DAC output changes in smaller increments, which reduces distortion and makes the waveform smoother. Fewer data points force the output to jump more abruptly between values, creating a more “stepped” waveform and increasing error. Therefore, increasing the number of data points improves overall waveform quality and accuracy.

PSEUDOCODE/FLOWCHARTS

SECTION 1

lab8_1.c

```
//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
dac_init();
        while(1)
        {
                //wait for data register to be empty
                While(!(DACA.STATUS & DAC_CH0DRE_bm));
                //write digital data to CH0DATA register corresponding to 1.6 V
                //Vdacn = (CHnDATA / 0xfff) * Vref


        }


}
void dac_init(void)
{
//use only channel zero – ctrlb = chsel_single
//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb
//enable channel zero and overall ctrla = ch0en | enable
}
```

Lab8_2a.c

```
//generate a sin wave

//configure timer to 32 MHz as done in HW4 with clock.s

#define WF_FREQ 392

uint16_t sine[256] = {copy table from the calculator}

volatile uint8_t index = 0;



int main(void)

{

dac_init();

tcc0_init();

//enable pmic for low level

sei();

        while(1)

        {

        //do the next 2 lines for pre lab exercise 2 otherwise comment out the polling and use interrupts

                //wait for TCC0 OVFIF to be set

                //Clear ovifif flag

                //wait for data register to be empty

        //put in isr

                while(!(DACA.STATUS & DAC_CH0DRE_bm));

                //write digital data from look up table and increment index

                //if index == 256 reset to 0 (not sure if needed)


        }


}

void dac_init(void)

{
```

```
//use only channel zero – ctrlb = chsel_single

//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb

//enable channel zero and overall ctrla = ch0en | enable

}

void tcc0_init(void)

{

//count = 0

//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of data points

//  (1/392) / 256 = 9.96e-6 = dur

//per = fclk * dur / pre =   32MHz * 9.96e-6 = 319 with pre = 1

//enable timer overflow interrupts


//start timer with pre = 1 in ctrla

}


ISR(TCC0_OVF_vect)

{

while(!(DACA.STATUS & DAC_CH0DRE_bm));

//write digital data from look up table and increment index

//if index == 256 reset to 0 (not sure if needed)

}
```

Lab8_2b.c (same as 2a / will show changes below)

```
//generate a sin wave

//configure timer to 32 MHz as done in HW4 with clock.s

#define WF_FREQ 1567.98

uint16_t sine[256] = {copy table from the calculator}

volatile uint8_t index = 0;
```

```
int main(void)
{
dac_init();
tcc0_init();
        while(1)
        {
                //wait for data register to be empty
                while(!(DACA.STATUS & DAC_CH0DRE_bm));
                //write digital data from look up table and increment index
        }

}
void dac_init(void)
{
//use only channel zero – ctrlb = chsel_single
//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb
//make ADC conversions start when event channel 0 is triggered

//enable channel zero and overall ctrla = ch0en | enable
}
void tcc0_init(void)
{
//count = 0
//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of data points
//  (1/1567.98) / 256 = 2.491e-6 = dur
//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1
//do not enable timer overflow interrupts
//enable the timer to generate events in EVSYS.CH0MUX

//start timer with pre = 1 in ctrla
}


Lab8_3.c
```

Lab8_3.c

```
//generate a sin wave

//configure timer to 32 MHz as done in HW4 with clock.s

#define WF_FREQ 1567.98

uint16_t sine[256] = {copy table from the calculator}

volatile uint8_t index = 0;



int main(void)

{

dac_init();

dma_init();

tcc0_init();


while(1)

{

}


}

void dac_init(void)

{

//use only channel zero – ctrlb = chsel_single

//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb

//make ADC conversions start when event channel 0 is triggered


//enable channel zero and overall ctrla = ch0en | enable

}

void tcc0_init(void)

{
```

```
//count = 0
//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of data points
//  (1/1567.98) / 256 = 2.491e-6 = dur
//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1
//do not enable timer overflow interrupts
//enable the timer to generate events in EVSYS.CH0MUX

//start timer with pre = 1 in ctrla
}
void dma_init(void){
//reset dma peripheral in ctrl
//set REPCNT to zero so it repeats forever
//turn on single shot mode with burstlen 2 and repeat mode in ch0ctrla
//srcreload = block, srcdir = inc, desreload = burst, destdir = inc in addrctrl
//trigger source = event sys ch0
//trfcount = (uint16_t)(sizeof(sine))   i think 512
//src addr0 = sine, addr1 = sin>>8, addr2 = sine>>16
//dest addr0 = & DACA.CH0DATA, <<8, <<16
//enable CH0 in dmaCH0ctrla
//enable DMA in ctrl
}
```

Lab8_4.c (change all dacch0 uses to dacch1

```
//generate a sin wave
//configure timer to 32 MHz as done in HW4 with clock.s
#define WF_FREQ 1567.98
uint16_t sine[256] = {copy table from the calculator}
volatile uint8_t index = 0;
```

```
int main(void)
{
//set portc to high and output
Clock_init();
dac_init();
dma_init();
tcc0_init();

while(1)
{
}


}
void dac_init(void)
{
//use only channel zero – ctrlb = chsel_single
//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb
//make ADC conversions start when event channel 0 is triggered

//enable channel zero and overall ctrla = ch0en | enable
}
void tcc0_init(void)
{
//count = 0
//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of data points
//  (1/1567.98) / 256 = 2.491e-6 = dur
//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1
//do not enable timer overflow interrupts
//enable the timer to generate events in EVSYS.CH0MUX

//start timer with pre = 1 in ctrla
}
```

```
void dma_init(void){
//reset dma peripheral in ctrl
//set REPCNT to zero so it repeats forever
//turn on single shot mode with burstlen 2 and repeat mode in ch0ctrla
//srcreload = block, srcdir = inc, desreload = burst, destdir = inc in addrctrl
//trigger source = event sys ch0
//trfcount = (uint16_t)(sizeof(sine))   i think 512
//src addr0 = sine, addr1 = sin>>8, addr2 = sine>>16
//dest addr0 = & DACA.CH0DATA, <<8, <<16
//enable CH0 in dmaCH0ctrla
//enable DMA in ctrl
}
```

Lab8_5.c

```
//generate a sin wave
//configure timer to 32 MHz as done in HW4 with clock.s
#define WF_FREQ 1567.98
uint16_t sine[256] = {copy table from the calculator}
//Load triangle_LUT[256]
volatile uint8_t index = 0;
//current_waveform = sine
```

Corresponding pers to keyboard letters

E 120 1055 tested measurements

4 113 1114

R 106 1186

5 100 1260

T 95 1325

Y 89 1414

7 85 1498

U 80 1569

```
8 75 1672

I 71 1766

9 67 1871

O 63 1985

P 61 2120



int main(void)

{

//set portc to high and output

Clock_init();

usart_init();

dac_init();

dma_init();

tcc0_init();


while(1)

{

// if (USART RXCIF flag is set):

 //       char c = read USARTD0.DATA


//        if (c == 's'):       // switch waveform

           // if (current_waveform == SINE):

               // disable DMA channel

               // change DMA source to triangle_LUT

                //enable DMA channel

              //  current_waveform = TRIANGLE

          //  else:

              //  disable DMA channel

            //    change DMA source to sine_LUT

             //   enable DMA channel

             //   current_waveform = SINE
}


}
void dac_init(void)
{
//use only channel zero – ctrlb = chsel_single
//use AREFB (2.5V) data is right adjusted – ctrlc = refsel_arefb
//make ADC conversions start when event channel 0 is triggered


//enable channel zero and overall ctrla = ch0en | enable
}
void tcc0_init(void)
{
//count = 0
//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of data points
//  (1/1567.98) / 256 = 2.491e-6 = dur
//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1
//do not enable timer overflow interrupts
//enable the timer to generate events in EVSYS.CH0MUX


//start timer with pre = 1 in ctrla
}
void dma_init(void){
//reset dma peripheral in ctrl
//set REPCNT to zero so it repeats forever
//turn on single shot mode with burstlen 2 and repeat mode in ch0ctrla
//srcreload = block, srcdir = inc, desreload = burst, destdir = inc in addrctrl
//trigger source = event sys ch0
//trfcount = (uint16_t)(sizeof(sine))   i think 512
//src addr0 = sine, addr1 = sin>>8, addr2 = sine>>16
//dest addr0 = & DACA.CH0DATA, <<8, <<16
//enable CH0 in dmaCH0ctrla
```

//enable DMA in ctrl

}

void usartd0_init(void){

//initialize usart pins first

//outset TX pin (PD3 to idle high)

//dirset TX pin

//dirclr RX pin (PD2)

//set to odd parity, 8 bits, 1 stop bit in PORTD.CTRLC

//set baud rate to 94744 (bscale = -7, bsel = 41)

//enable tx and rx

}

PROGRAM CODE

SECTION 2

Lab8_1.c

```
/*
 * Lab8.c
 *
 * Created: 11/14/2025 5:29:16 PM
 * Author : arist
 */

/*******************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Outputs a constant 1.6 V using DAC CH0.
 *******************************************************/

#include <avr/io.h>

extern void clock_init(void);

//prototypes
void dac_init(void);


//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
       clock_init();
       dac_init();
	DACA.CH0DATA = 2621; // (1.6 / 2.5) * 4095 moved here since it does not need to be rewritten 
every loop
	while(1)
	{
		//wait for data register to be empty
		while(!(DACA.STATUS & DAC_CH0DRE_bm));
		//write digital data to CH0DATA register corresponding to 1.6 V
		//Vdacn = (CHnDATA / 0xfff) * Vref
		//DACA.CH0DATA = 2621; // (1.6 / 2.5) * 4095
	}

}

/********** dac_init ********************************************
 * Sets up DACA CH0 for single-channel output with AREFB (2.5 V)
 * reference and enables the DAC.
 *****************************************************************/
void dac_init(void)
{
	//use only channel zero - ctrlb = chsel_single
	DACA.CTRLB = DAC_CHSEL_SINGLE_gc;
	//use AREFB (2.5V) data is right adjusted - ctrlc = refsel_arefb
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;
	//enable channel zero and overall ctrla = ch0en | enable
	DACA.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;
}
```

Lab8_2a.c

```
/*
 * lab8_2a.c
 *
 * Created: 11/14/2025 6:25:33 PM
 *  Author: arist
 */
/*******************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Outputs a 256-point sine waveform on DAC CH0 using
 *   TCC0 overflow interrupts to step through the LUT.
 *******************************************************/


#include <avr/io.h>
#include <avr/interrupt.h>

//prototypes
void dac_init(void);
void tcc0_init(void);
extern void clock_init(void);

#define WF_freq 792
volatile uint8_t index = 0;
```

```
uint16_t sine[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};


//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
       clock_init();
       dac_init();
       tcc0_init();

       PMIC.CTRL = PMIC_LOLVLEN_bm;
       sei();

       DACA.CH0DATA = 2621; // (1.6 / 2.5) * 4095 moved here since it does not need to be rewritten
every loop
       while(1)
       {
              //do the next 2 lines for pre lab exercise 2 otherwise comment out the polling and use
interrupts
              //wait for TCC0 OVFIF to be set
              //while(!(TCC0.INTFLAGS & TC0_OVFIF_bm));
              //Clear ovifif flag
              //TCC0.INTFLAGS = TC0_OVFIF_bm;
              //wait for data register to be empty
              //put in isr
              while(!(DACA.STATUS & DAC_CH0DRE_bm));
              //write digital data from look up table and increment index
              DACA.CH0DATA = sine[index++];
              //if index == 256 reset to 0 (not sure if needed)
       }
}
/********************************************************
 * dac_init
 * Configures DACA for single-channel output using AREFB.
 * No parameters. No return value.
 ********************************************************/

void dac_init(void)
{
	//use only channel zero - ctrlb = chsel_single
	DACA.CTRLB = DAC_CHSEL_SINGLE_gc;
	//use AREFB (2.5V) data is right adjusted - ctrlc = refsel_arefb
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;
	//enable channel zero and overall ctrla = ch0en | enable
	DACA.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;
}

/********************************************************
 * tcc0_init
 * Sets up TCC0 to overflow at the waveform update rate
 * and enables low-level overflow interrupts.
 * No parameters. No return value.
 ********************************************************/

void tcc0_init(void)
{
	//count = 0
	TCC0.CNT = 0;
	//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of
data points
	//  (1/392) / 256 = 9.96e-6 = dur
	//per = fclk * dur / pre =   32MHz * 9.96e-6 = 319 with pre = 1
	TCC0.PER = 1;
	//enable timer overflow interrupts
	TCC0.INTCTRLA = TC_OVFINTLVL_LO_gc;
	//start timer with pre = 1 in ctrla
	TCC0.CTRLA = TC_CLKSEL_DIV1_gc;
}

/********************************************************
 * ISR: TCC0_OVF_vect
 * On each overflow, waits for CH0DRE and outputs the
 * next sine-table entry to DAC CH0.
 ********************************************************/

ISR(TCC0_OVF_vect)
{
	while(!(DACA.STATUS & DAC_CH0DRE_bm));
	//write digital data from look up table and increment index
	DACA.CH0DATA = sine[index++];
	//if index == 256 reset to 0 (not sure if needed)
	// if (index == 256)
	// index = 0;
}
```

Lab8_2b.c

```
/*
 * lab8_2b.c
 *
 * Created: 11/15/2025 2:10:41 AM
 *  Author: arist
 */

/*****************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Outputs a 256-point sine wave using DAC CH0.
 *   TCC0 overflow events trigger DAC conversions
 *   through the event system.
 *****************************************************/


#include <avr/io.h>
#include <avr/interrupt.h>

//prototypes
void dac_init(void);
void tcc0_init(void);
extern void clock_init(void);

#define WF_freq 792
volatile uint8_t index = 0;

uint16_t sine[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};


//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
	clock_init();
	dac_init();
	tcc0_init();
	
	while(1)
	{
		//wait for data register to be empty
		while(!(DACA.STATUS & DAC_CH0DRE_bm));
		//write digital data from look up table and increment index
		DACA.CH0DATA = sine[index++];
		//if (index == 256) index = 0;
	}
	
}

/*******************************************************
 * dac_init
 * Configures DAC CH0 for event-triggered output using
 * AREFB as the reference. No parameters or return value.
 *******************************************************/

void dac_init(void)
{
	//use only channel zero: ctrlb = chsel_single
	//also enable the CH0TRIG
	DACA.CTRLB = DAC_CHSEL_SINGLE_gc | DAC_CH0TRIG_bm;
	//use AREFB (2.5V) data is right adjusted - ctrlc = refsel_arefb
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;
	
	//make DAC conversions start when event channel 0 is triggered
	DACA.EVCTRL = DAC_EVSEL_0_gc;
	
	//enable channel zero and overall ctrla = ch0en | enable
	DACA.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;
	
	
}

/*******************************************************
 * tcc0_init
 * Initializes TCC0 to overflow at the waveform step rate
 * and route overflow events to EVSYS CH0.
 *******************************************************/

void tcc0_init(void)
{
	//count = 0
	TCC0.CNT = 0;
	//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of 
data points
	//  (1/1567.98) / 256 = 2.491e-6 = dur
	//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1
	TCC0.PER = 80;
	//enable the timer to generate events in EVSYS.CH0MUX
	EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc;

	//start timer with pre = 1 in ctrla
	TCC0.CTRLA = TC_CLKSEL_DIV1_gc;
}
```

Lab8_3.c

```
/*
 * lab8_3.c
 *
 * Created: 11/15/2025 3:41:02 PM
 *  Author: arist
 */

/*******************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Uses DMA channel 0 to stream a 256-point sine LUT
 *   to DAC CH0. TCC0 overflow events trigger each DMA
 *   transfer, producing a continuous waveform.
 *******************************************************/


#include <avr/io.h>
#include <avr/interrupt.h>

//prototypes
void dac_init(void);
void dma_init(void);
void tcc0_init(void);
extern void clock_init(void);

#define WF_freq 1567.98
volatile uint8_t index = 0;

uint16_t sine[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};


//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
	clock_init();
	dma_init();
	dac_init();
	tcc0_init();

	while(1)
	{
	}

}

/*******************************************************
 * dac_init
 * Configures DAC CH0 for event-triggered conversions
 * using AREFB (2.5 V). No parameters or return value.
 *******************************************************/
void dac_init(void)
{
	//use only channel zero: ctrlb = chsel_single
	//also enable the CH0TRIG
	DACA.CTRLB = DAC_CHSEL_SINGLE_gc | DAC_CH0TRIG_bm;
	//use AREFB (2.5V) data is right adjusted - ctrlc = refsel_arefb
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;

	//make DAC conversions start when event channel 0 is triggered
	DACA.EVCTRL = DAC_EVSEL_0_gc;

	//enable channel zero and overall ctrla = ch0en | enable
	DACA.CTRLA = DAC_CH0EN_bm | DAC_ENABLE_bm;


}

/*******************************************************
 * tcc0_init
 * Initializes TCC0 to overflow at the waveform rate and
 * route OVF events through EVSYS CH0 to trigger DMA.
 *******************************************************/
void tcc0_init(void)
{
	//count = 0
	TCC0.CNT = 0;
	//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of
data points
	//  (1/1567.98) / 256 = 2.491e-6 = dur
	//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1

	TCC0.PER = 80;
	//enable the timer to generate events in EVSYS.CH0MUX
	EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc;

	//start timer with pre = 1 in ctrla
	TCC0.CTRLA = TC_CLKSEL_DIV1_gc;
}

/*******************************************************
 * dma_init
 * Sets up DMA CH0 to repeatedly transfer 256 16-bit
 * samples from the sine LUT into DACA.CH0DATA. Uses
 * 2-byte burst length and block reload mode.
 *******************************************************/
void dma_init(void){
	//reset dma peripheral in ctrl
	DMA.CTRL = DMA_RESET_bm;
	//set REPCNT to zero so it repeats forever
	DMA.CH0.REPCNT = 0;
	//turn on single shot mode with burstlen 2 and repeat mode in ch0ctrla
	DMA.CH0.CTRLA = DMA_CH_REPEAT_bm | DMA_CH_SINGLE_bm | DMA_CH_BURSTLEN_2BYTE_gc;
	//srcreload = block, srcdir = inc, desreload = none, destdir = fixed in addrctrl
	DMA.CH0.ADDRCTRL = DMA_CH_SRCRELOAD_BLOCK_gc | DMA_CH_SRCDIR_INC_gc
	                          | DMA_CH_DESTRELOAD_BURST_gc | DMA_CH_DESTDIR_INC_gc;
	//trigger source = dacaCH0
	DMA.CH0.TRIGSRC = DMA_CH_TRIGSRC_DACA_CH0_gc;
	//trfcount = (uint16_t)(sizeof(sine))   i think 512
	DMA.CH0.TRFCNT = (uint16_t)(sizeof(sine));
	//DMA.CH0.TRFCNT = 512;
	//src addr0 = sine, addr1 = sin>>8, addr2 = sine>>16
	DMA.CH0.SRCADDR0 = (uint8_t)((uintptr_t)sine);
	DMA.CH0.SRCADDR1 = (uint8_t)(((uintptr_t)sine)>>8);
	DMA.CH0.SRCADDR2 = (uint8_t)(((uint32_t)((uintptr_t)sine))>>16);
	//could have just done (uint8_t)sine
	//dest addr0 = & DACA.CH0DATA, <<8, <<16 no need for & 0xff bc cast
	uint32_t dst = (uint32_t)&DACA.CH0DATA;
	DMA.CH0.DESTADDR0 = (uint8_t)(dst & 0xff);
	DMA.CH0.DESTADDR1 = (uint8_t)(dst >> 8);
	DMA.CH0.DESTADDR2 = (uint8_t)(dst >> 16);
	//enable CH0 in dmaCH0ctrla
	DMA.CH0.CTRLA |= DMA_CH_ENABLE_bm;
	//enable DMA in ctrl
	DMA.CTRL |= DMA_ENABLE_bm;
}
```

Lab8_4.c

```
/*
 * lab8_4.c
 *
 * Created: 11/15/2025 8:13:39 PM
 *  Author: arist
 */
/*****************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Uses DMA CH0 to stream a 256-point sine LUT to
 *   DAC CH1. TCC0 overflow events trigger each DMA
 *   transfer to produce a continuous waveform.
 *****************************************************/


#include <avr/io.h>
#include <avr/interrupt.h>

//prototypes
void dac_init(void);
void dma_init(void);
void tcc0_init(void);
extern void clock_init(void);

#define WF_freq 1567.98
volatile uint8_t index = 0;

uint16_t sine[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};


//generate a wavefrom with a constant voltage of 1.6V using a DAC module
//configure timer to 32 MHz as done in HW4 with clock.s
int main(void)
{
	//set portc to high and output
	PORTC.OUTSET = PIN7_bm;
	PORTC.DIRSET = PIN7_bm;
	PORTA.DIRSET = PIN3_bm; //not needed, dac already does this i think
	clock_init();
	dma_init();
	dac_init();
	tcc0_init();

	while(1)
	{
	}

/*******************************************************
 * dac_init
 * Configures DAC CH1 for event-triggered conversions
 * using AREFB (2.5 V). No parameters or return value.
 *******************************************************/
}
void dac_init(void)
{
	//use only channel zero: ctrlb = chsel_single
	//also enable the CH1TRIG
	DACA.CTRLB = DAC_CHSEL_SINGLE1_gc | DAC_CH1TRIG_bm;
	//use AREFB (2.5V) data is right adjusted - ctrlc = refsel_arefb
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;

	//make DAC conversions start when event channel 0 is triggered
	DACA.EVCTRL = DAC_EVSEL_0_gc;

	//enable channel zero and overall ctrla = ch1en | enable
	DACA.CTRLA = DAC_CH1EN_bm | DAC_ENABLE_bm;


}

/*******************************************************
 * tcc0_init
 * Initializes TCC0 to overflow at the waveform rate,
 * routing OVF events to EVSYS CH0 for DAC triggering.
 *******************************************************/
void tcc0_init(void)
{
	//count = 0
	TCC0.CNT = 0;
	//dur of each data point = total waveform period / num of data points =  (1/WF_FREQ) / num of
data points
	//  (1/1567.98) / 256 = 2.491e-6 = dur
	//per = fclk * dur / pre =   32MHz * = 2.491e-6  = 80 with pre = 1

	TCC0.PER = 80;
	//enable the timer to generate events in EVSYS.CH0MUX
	EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc;
```

```
       //start timer with pre = 1 in ctrla
       TCC0.CTRLA = TC_CLKSEL_DIV1_gc;
}

/*******************************************************
 * dma_init
 * Configures DMA CH0 to repeatedly transfer 256
 * 16-bit LUT samples to DACA.CH1DATA. Uses 2-byte
 * bursts, block reload, and event-triggered transfers.
 *******************************************************/
void dma_init(void){
       //reset dma peripheral in ctrl
       DMA.CTRL = DMA_RESET_bm;
       //set REPCNT to zero so it repeats forever
       DMA.CH0.REPCNT = 0;
       //turn on single shot mode with burstlen 2 and repeat mode in ch0ctrla
       DMA.CH0.CTRLA = DMA_CH_REPEAT_bm | DMA_CH_SINGLE_bm | DMA_CH_BURSTLEN_2BYTE_gc;
       //srcreload = block, srcdir = inc, desreload = none, destdir = fixed in addrctrl
       DMA.CH0.ADDRCTRL = DMA_CH_SRCRELOAD_BLOCK_gc | DMA_CH_SRCDIR_INC_gc
                             | DMA_CH_DESTRELOAD_BURST_gc | DMA_CH_DESTDIR_INC_gc;
       //trigger source = dacaCH1
       DMA.CH0.TRIGSRC = DMA_CH_TRIGSRC_DACA_CH1_gc;
       //trfcount = (uint16_t)(sizeof(sine))   i think 512
       DMA.CH0.TRFCNT = (uint16_t)(sizeof(sine));
       //DMA.CH0.TRFCNT = 512;
       //src addr0 = sine, addr1 = sin>>8, addr2 = sine>>16
       DMA.CH0.SRCADDR0 = (uint8_t)((uintptr_t)sine);
       DMA.CH0.SRCADDR1 = (uint8_t)(((uintptr_t)sine)>>8);
       DMA.CH0.SRCADDR2 = (uint8_t)(((uint32_t)((uintptr_t)sine))>>16);
       //could have just done (uint8_t)sine
       //dest addr0 = & DACA.CH1DATA, <<8, <<16 no need for & 0xff bc cast
       uint32_t dst = (uint32_t)&DACA.CH1DATA;
       DMA.CH0.DESTADDR0 = (uint8_t)(dst & 0xff);
       DMA.CH0.DESTADDR1 = (uint8_t)(dst >> 8);
       DMA.CH0.DESTADDR2 = (uint8_t)(dst >> 16);
       //enable CH0 in dmaCH0ctrla
       DMA.CH0.CTRLA |= DMA_CH_ENABLE_bm;
       //enable DMA in ctrl
       DMA.CTRL |= DMA_ENABLE_bm;
}
```

Lab8_5.c

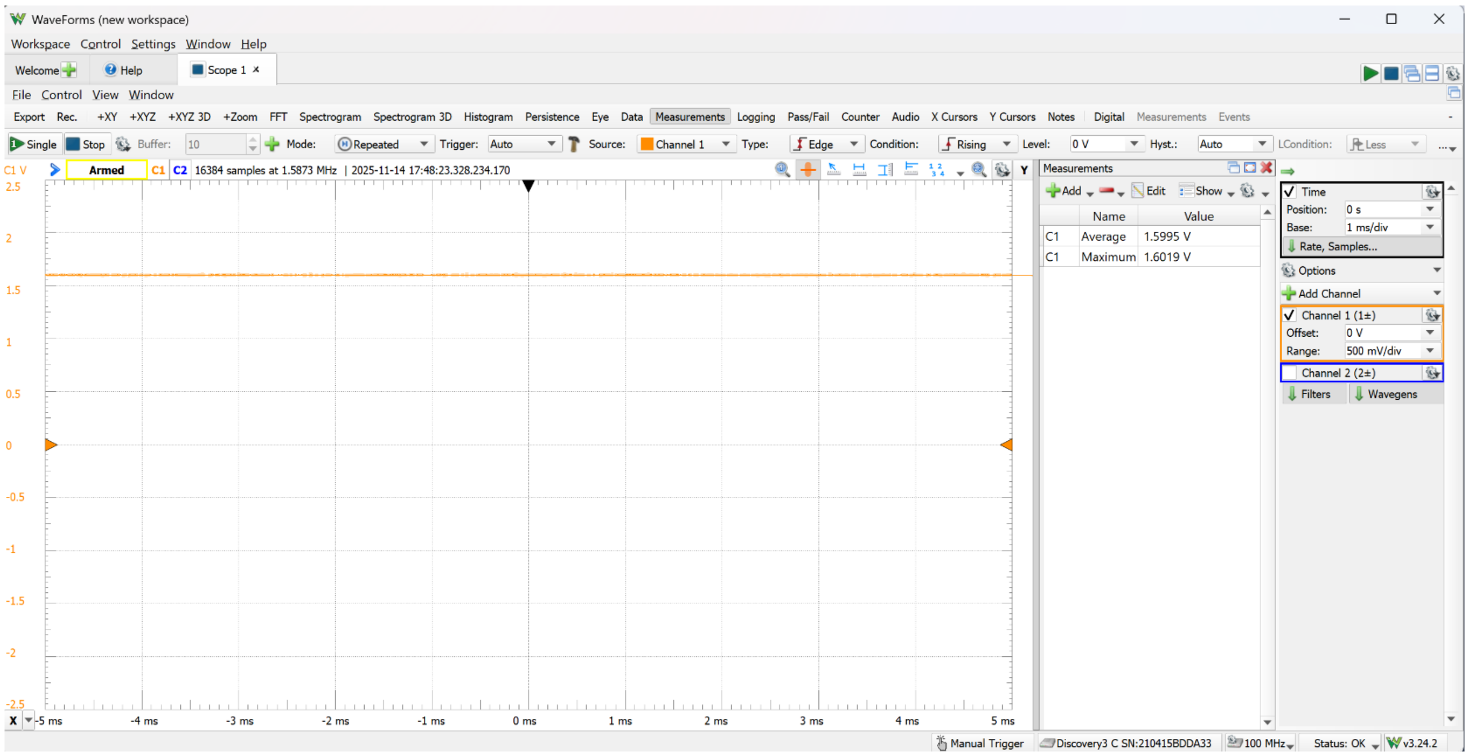
```
/*
 * lab8_5.c
 *
 * Created: 11/15/2025 9:17:25 PM
 *  Author: arist
 */
/******************************************************
 * Lab 8, Section 22
 * Name: Arion Stern
 * Class #: 11303
 * PI Name: Ian Santamauro
 * Description:
 *   Keyboard-controlled waveform generator using
 *   DMA + DAC CH1. 'S' toggles sine/triangle LUTs,
 *   and note keys change TCC0.PER to play pitches.
 ******************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>     // for uintptr_t

// Prototypes
void usartd0_init(void);
void dac_init(void);
void dma_init(void);
void tcc0_init(void);
void tcc1_init(void);
extern void clock_init(void);
void dma_set_source(uint16_t *lut);
uint16_t get_per(char c);

#define WAVE_SINE      0
#define WAVE_TRIANGLE  1

#define BSEL     (3693)
#define BSCALE   (-7)

volatile uint8_t  current_wf = WAVE_SINE;
volatile char     rec        = 0;
volatile uint16_t note_timer = 0;

// === LUTs ===
uint16_t sine[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
};

uint16_t triangle[256] = {
        0x20, 0x40, 0x60, 0x80, 0xa0, 0xc0, 0xe0, 0x100, 0x120, 0x140, 0x160, 0x180, 0x1a0, 0x1c0, 0x1e0,
0x200,
        0x220, 0x240, 0x260, 0x280, 0x2a0, 0x2c0, 0x2e0, 0x300, 0x320, 0x340, 0x360, 0x380, 0x3a0, 0x3c0,
0x3e0, 0x400,
        0x420, 0x440, 0x460, 0x480, 0x4a0, 0x4c0, 0x4e0, 0x500, 0x520, 0x540, 0x560, 0x580, 0x5a0, 0x5c0,
0x5e0, 0x600,
        0x620, 0x640, 0x660, 0x680, 0x6a0, 0x6c0, 0x6e0, 0x700, 0x720, 0x740, 0x760, 0x780, 0x7a0, 0x7c0,
0x7e0, 0x800,
        0x81f, 0x83f, 0x85f, 0x87f, 0x89f, 0x8bf, 0x8df, 0x8ff, 0x91f, 0x93f, 0x95f, 0x97f, 0x99f, 0x9bf,
0x9df, 0x9ff,
        0xa1f, 0xa3f, 0xa5f, 0xa7f, 0xa9f, 0xabf, 0xadf, 0xaff, 0xb1f, 0xb3f, 0xb5f, 0xb7f, 0xb9f, 0xbbf,
0xbdf, 0xbff,
        0xc1f, 0xc3f, 0xc5f, 0xc7f, 0xc9f, 0xcbf, 0xcdf, 0xcff, 0xd1f, 0xd3f, 0xd5f, 0xd7f, 0xd9f, 0xdbf,
0xddf, 0xdff,
        0xe1f, 0xe3f, 0xe5f, 0xe7f, 0xe9f, 0xebf, 0xedf, 0xeff, 0xf1f, 0xf3f, 0xf5f, 0xf7f, 0xf9f, 0xfbf,
0xfdf, 0xfff,
        0xfdf, 0xfbf, 0xf9f, 0xf7f, 0xf5f, 0xf3f, 0xf1f, 0xeff, 0xedf, 0xebf, 0xe9f, 0xe7f, 0xe5f, 0xe3f,
0xe1f, 0xdff,
        0xddf, 0xdbf, 0xd9f, 0xd7f, 0xd5f, 0xd3f, 0xd1f, 0xcff, 0xcdf, 0xcbf, 0xc9f, 0xc7f, 0xc5f, 0xc3f,
0xc1f, 0xbff,
        0xbdf, 0xbbf, 0xb9f, 0xb7f, 0xb5f, 0xb3f, 0xb1f, 0xaff, 0xadf, 0xabf, 0xa9f, 0xa7f, 0xa5f, 0xa3f,
0xa1f, 0x9ff,
        0x9df, 0x9bf, 0x99f, 0x97f, 0x95f, 0x93f, 0x91f, 0x8ff, 0x8df, 0x8bf, 0x89f, 0x87f, 0x85f, 0x83f,
0x81f, 0x800,
        0x7e0, 0x7c0, 0x7a0, 0x780, 0x760, 0x740, 0x720, 0x700, 0x6e0, 0x6c0, 0x6a0, 0x680, 0x660, 0x640,
0x620, 0x600,
        0x5e0, 0x5c0, 0x5a0, 0x580, 0x560, 0x540, 0x520, 0x500, 0x4e0, 0x4c0, 0x4a0, 0x480, 0x460, 0x440,
0x420, 0x400,
        0x3e0, 0x3c0, 0x3a0, 0x380, 0x360, 0x340, 0x320, 0x300, 0x2e0, 0x2c0, 0x2a0, 0x280, 0x260, 0x240,
0x220, 0x200,
        0x1e0, 0x1c0, 0x1a0, 0x180, 0x160, 0x140, 0x120, 0x100, 0xe0, 0xc0, 0xa0, 0x80, 0x60, 0x40, 0x20,
0x0
};

/* PER = 32 MHz / (256 * f_note) */
/*******************************************************
 * get_per
 * Returns the TCC0.PER value for the given note key.
 * Input: ASCII key character.
 * Output: 16-bit period value (0 if invalid key).
 *******************************************************/
uint16_t get_per(char c)
{
        switch(c)
	{
		case 'E': return 120; // C6
		case '4': return 114; // C#6
		case 'R': return 107; // D6
		case '5': return 101; // D#6
		case 'T': return 95;  // E6
		case 'Y': return 90;  // F6
		case '7': return 84;  // F#6
		case 'U': return 80;  // G6
		case '8': return 75;  // G#6
		case 'I': return 71;  // A6
		case '9': return 67;  // A#6
		case 'O': return 63;  // B6
		case 'P': return 60;  // C7
		default:  return 0;   // not a musical key
	}
}

/*******************************************************
 * main
 * Initializes all peripherals and polls USART input.
 * 'S' switches waveform via DMA source swap.
 * Note keys change pitch; timer auto-stops notes.
 *******************************************************/
int main(void)
{
	// Status LED
	PORTC.DIRSET = PIN7_bm;
	PORTC.OUTSET = PIN7_bm;

	// Debug LED for 'S'
	PORTD.DIRSET = PIN6_bm;

	// DAC CH0 output pin
	PORTA.DIRSET = PIN3_bm;

	clock_init();
	dac_init();
	usartd0_init();
	dma_init();
	tcc0_init();
	tcc1_init();   // note-length timer

	PMIC.CTRL |= PMIC_LOLVLEN_bm;
	sei();

	while(1)
	{
		// ---------- READ USART ----------
		if (USARTD0.STATUS & USART_RXCIF_bm)
		{
			rec = USARTD0.DATA;

			// Normalize to uppercase so PuTTY case doesn't matter
			if (rec >= 'a' && rec <= 'z')
				rec -= 32;
		}

		// ---------- PROCESS NEW KEY ----------
		if (rec != 0)
		{
```

```
                        switch(rec)
                        {
                                case 'S':
                                        // Toggle wave (0 <-> 1)
                                        current_wf ^= 1;
                                        PORTD.OUTTGL = PIN6_bm;  // visual feedback

                                        if (current_wf == WAVE_SINE)
                                        {
                                                dma_set_source(sine);
                                        }
                                        else
                                        {
                                                dma_set_source(triangle);
                                        }


                                        TCC0.CNT = 0;
                                        break;

                                default:
                                {
                                        uint16_t per = get_per(rec);
                                        if (per > 0)
                                        {
                                                // Update pitch & start timer
                                                TCC0.PER  = per;
                                                TCC0.CNT  = 0;
                                                TCC0.CTRLA = TC_CLKSEL_DIV1_gc;   // ensure running

                                                // hold note for N ms (N = note_timer value)
                                                note_timer = 200; // ~500 ms at 1 kHz TCC1
                                        }
                                }
                                break;
                        }

                        // CLEAR KEY TO AVOID STALE INPUT
                        rec = 0;
                }

                // AUTO NOTE STOP
                if (note_timer == 0)
                {
                        // stop sound
                        TCC0.CTRLA = TC_CLKSEL_OFF_gc;
                }
        }
}

/*******************************************************
 * tcc1_init
 * Sets up TCC1 to overflow at ~1 kHz, used to
 * decrement note_timer for automatic note stopping.
 *******************************************************/
/*** TCC1: ~1 kHz tick for note hold timer ***/
void tcc1_init(void)
{
        // 32 MHz / 1 = 32 MHz / 32000 = 1 kHz overflow
        TCC1.PER = 32000;
        TCC1.CNT = 0;
	TCC1.INTCTRLA = TC_OVFINTLVL_LO_gc;
	TCC1.CTRLA = TC_CLKSEL_DIV1_gc;
}


/*******************************************************
 * ISR(TCC1_OVF_vect)
 * Decrements note_timer every 1 ms until it reaches 0.
 *******************************************************/
ISR(TCC1_OVF_vect)
{
	if (note_timer > 0)
		note_timer--;

	// Clear flag
	TCC1.INTFLAGS = TC1_OVFIF_bm;
}

/*******************************************************
 * dac_init
 * Configures DAC CH1 for event-triggered conversions
 * using AREFB=2.5 V and EVSYS CH0 as the trigger.
 *******************************************************/
/*** DAC setup: CH0, AREFB, triggered by EVSYS CH0 ***/
void dac_init(void)
{
	DACA.CTRLB = DAC_CHSEL_SINGLE1_gc | DAC_CH1TRIG_bm;
	DACA.CTRLC = DAC_REFSEL_AREFB_gc;
	DACA.EVCTRL = DAC_EVSEL_0_gc;
	DACA.CTRLA = DAC_CH1EN_bm | DAC_ENABLE_bm;
}

/*******************************************************
 * tcc0_init
 * Initializes TCC0 to generate OVF events routed to
 * EVSYS CH0, which triggers each DAC conversion.
 *******************************************************/
/*** TCC0: drives EVSYS CH0 for DAC conversions ***/
void tcc0_init(void)
{
	TCC0.CNT = 0;
	TCC0.PER = 80;   // default G6-ish
	EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc;
	TCC0.CTRLA = TC_CLKSEL_DIV1_gc;   // running at 32 MHz (will be stopped by logic)
}

/*******************************************************
 * dma_init
 * Sets up DMA CH0 to stream 256×16-bit samples from the
 * current LUT to DACA.CH1DATA using 2-byte bursts and
 * block reload. Trigger source = DAC CH1.
 *******************************************************/
/*** DMA: CH0 -> DACA.CH0DATA from LUT ***/
void dma_init(void)
{
	DMA.CTRL = DMA_RESET_bm;

	DMA.CH0.REPCNT = 0; // repeat forever

	DMA.CH0.CTRLA = DMA_CH_REPEAT_bm | DMA_CH_SINGLE_bm | DMA_CH_BURSTLEN_2BYTE_gc;
	DMA.CH0.ADDRCTRL =
		DMA_CH_SRCRELOAD_BLOCK_gc |
		DMA_CH_SRCDIR_INC_gc      |
		DMA_CH_DESTRELOAD_BURST_gc |
		DMA_CH_DESTDIR_INC_gc;

	DMA.CH0.TRIGSRC = DMA_CH_TRIGSRC_DACA_CH1_gc;
	DMA.CH0.TRFCNT = (uint16_t)sizeof(sine);   // 256 samples * 2 bytes

	// Source = sine LUT by default
	uintptr_t src = (uintptr_t)&sine[0];
	DMA.CH0.SRCADDR0 = (uint8_t)(src & 0xFF);
	DMA.CH0.SRCADDR1 = (uint8_t)((src >> 8) & 0xFF);
	DMA.CH0.SRCADDR2 = (uint8_t)((src >> 16) & 0xFF);

	// Dest = DAC CH0DATA
	uintptr_t dst = (uintptr_t)&DACA.CH1DATA;
	DMA.CH0.DESTADDR0 = (uint8_t)(dst & 0xFF);
	DMA.CH0.DESTADDR1 = (uint8_t)((dst >> 8) & 0xFF);
	DMA.CH0.DESTADDR2 = (uint8_t)((dst >> 16) & 0xFF);

	DMA.CH0.CTRLA |= DMA_CH_ENABLE_bm;
	DMA.CTRL |= DMA_ENABLE_bm;
}

/*******************************************************
 * dma_set_source
 * Updates DMA CH0 source pointer to a new LUT.
 * Input: pointer to a 256-sample uint16_t array.
 *******************************************************/
/*** Safely change DMA source LUT (no busy-wait on BUSY) ***/
void dma_set_source(uint16_t *lut)
{
	// Disable channel while we update reload address
	DMA.CH0.CTRLA &= ~DMA_CH_ENABLE_bm;

	uintptr_t src = (uintptr_t)&lut[0];
	DMA.CH0.SRCADDR0 = (uint8_t)(src & 0xFF);
	DMA.CH0.SRCADDR1 = (uint8_t)((src >> 8) & 0xFF);
	DMA.CH0.SRCADDR2 = (uint8_t)((src >> 16) & 0xFF);

	// TRFCNT stays the same (256 samples * 2 bytes)
	 DMA.CH0.TRFCNT = 256 * 2;

	// Re-enable DMA channel
	DMA.CH0.CTRLA |= DMA_CH_ENABLE_bm;
}

/*******************************************************
 * usartd0_init
 * Configures USARTD0 for ~67 kbaud RX/TX
 * (8-bit, odd parity). Used to receive note commands.
 *******************************************************/
/*** USART D0 init @ ~67,000 baud, odd parity, 8N1-like ***/
void usartd0_init(void)
{
	PORTD.OUTSET = PIN3_bm; // TXD idle high
	PORTD.DIRSET = PIN3_bm; // TXD
	PORTD.DIRCLR = PIN2_bm; // RXD

	USARTD0.BAUDCTRLA = (uint8_t)BSEL;
	USARTD0.BAUDCTRLB = (uint8_t)((BSCALE << 4) | (BSEL >> 8));

	USARTD0.CTRLC = (USART_CMODE_ASYNCHRONOUS_gc |
	                 USART_PMODE_ODD_gc |
	                 USART_CHSIZE_8BIT_gc) &
	                ~USART_SBMODE_bm;

	USARTD0.CTRLB = USART_RXEN_bm | USART_TXEN_bm;
	// RX interrupt not used; we just poll
}
```

APPENDIX

Supporting ASM/C Code and Additional Information

- Included/Referenced Files and Headers:
    - USART files from lab6
    - HW4 files
- Additional Screenshots/Images:


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
Armed C1 C2
C1 V
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-5 ms -4 ms -3 ms -2 ms -1 ms 0 ms 1 ms 2 ms 3 ms 4 ms 5 ms
Add Edit Show
Options
Add Channel
Rate, Samples...
Filters Wavegens

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

16384 samples at 1.5873 MHz | 2025-11-14 17:48:23.328.234.170

Measurements

	Name	Value
C1	Average	1.5995 V
C1	Maximum	1.6019 V

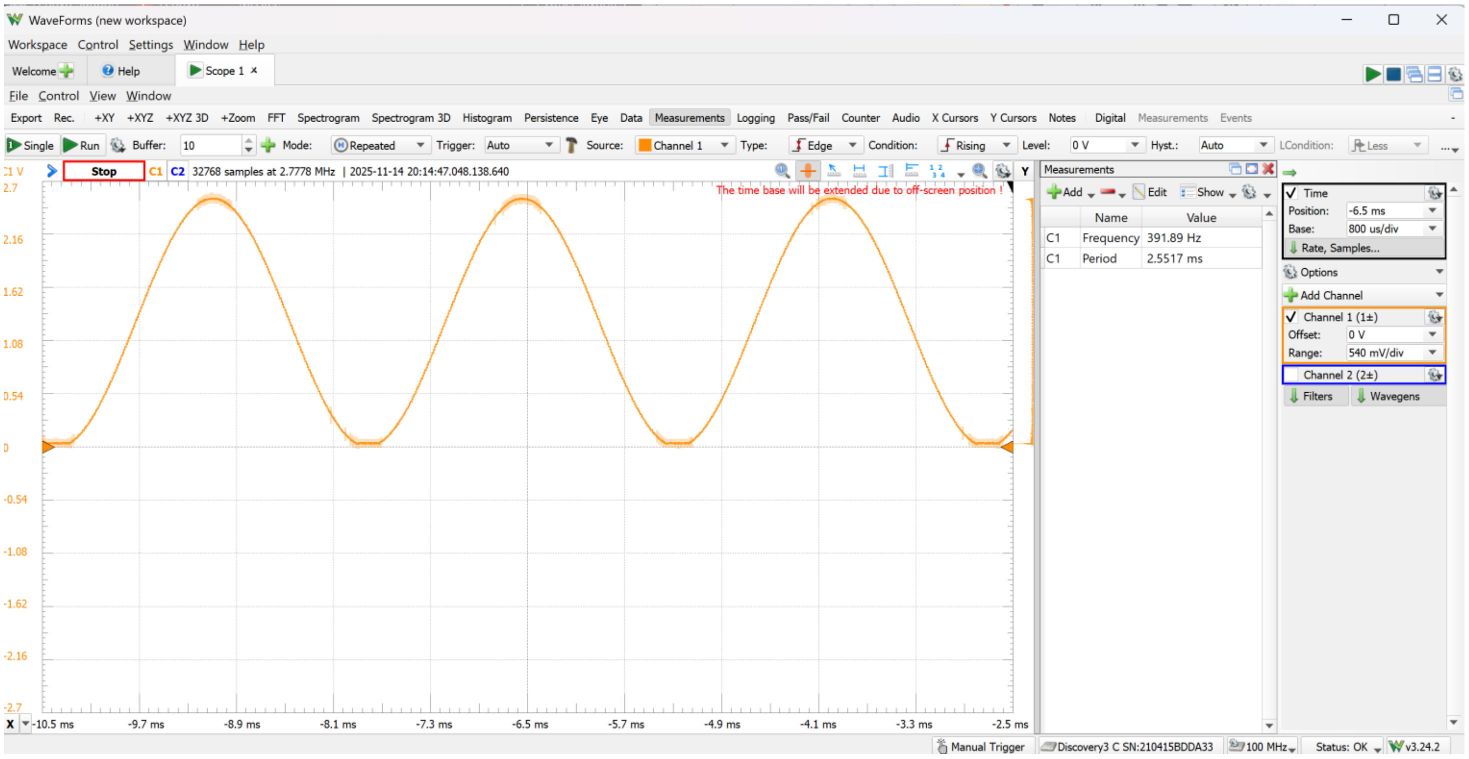
Time
Position: 0 s
Base: 1 ms/div

Channel 1 (1±)
Offset: 0 V
Range: 500 mV/div

Channel 2 (2±)

Manual Trigger | Discovery3 C SN:210415BDDA33 | 100 MHz | Status: OK | v3.24.2

Figure 4: DAC CH0 outputs a steady 1.6 V DC level, confirming proper configuration of the DAC module using AREFB.


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Run
Stop C1 C2
2.7 2.16 1.62 1.08 0.54 0 -0.54 -1.08 -1.62 -2.16 -2.7
-10.5 ms -9.7 ms -8.9 ms -8.1 ms -7.3 ms -6.5 ms -5.7 ms -4.9 ms -4.1 ms -3.3 ms -2.5 ms
Add Edit Show
Options
Add Channel
Filters Wavegens
Manual Trigger

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

32768 samples at 2.7778 MHz | 2025-11-14 20:14:47.048.138.640

The time base will be extended due to off-screen position !

Measurements

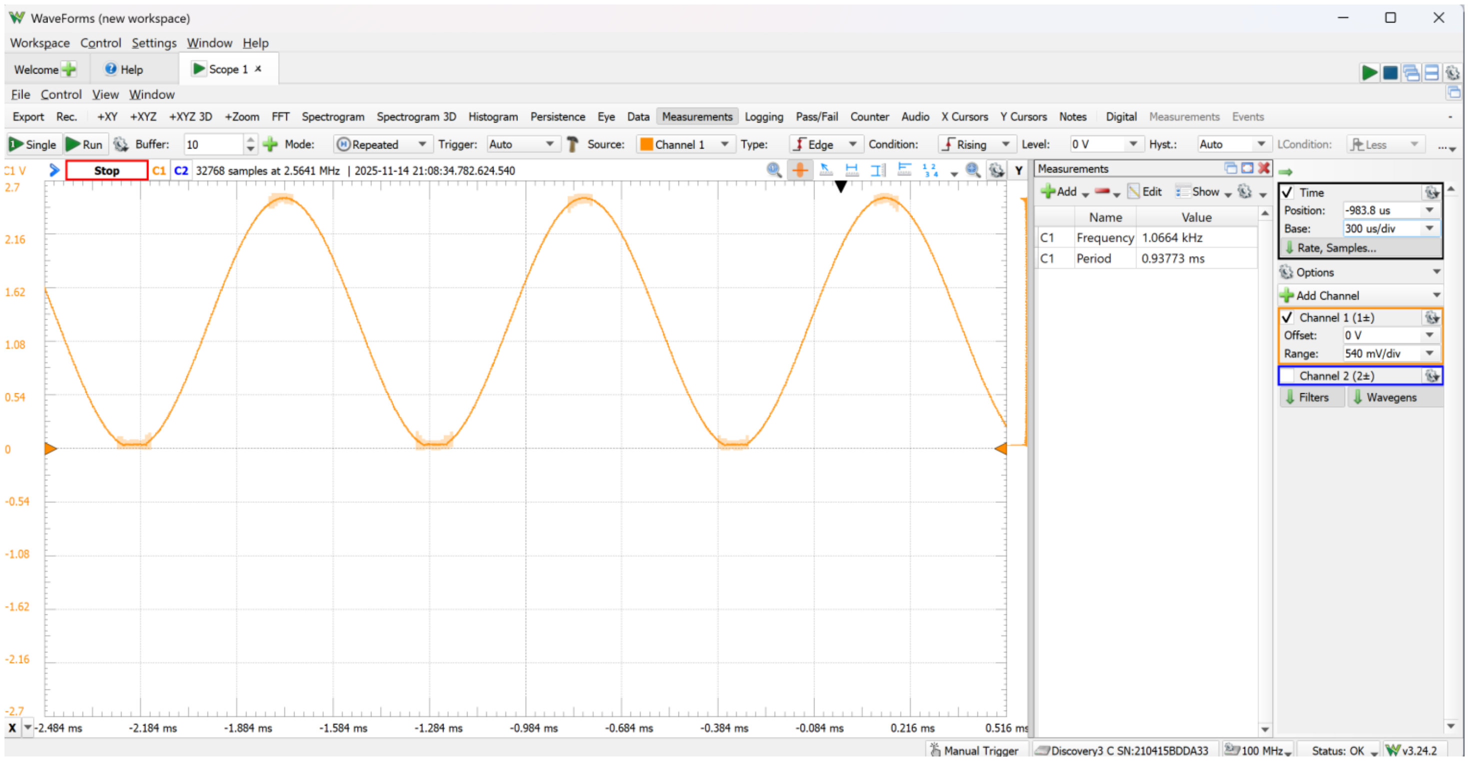
	Name	Value
C1	Frequency	391.89 Hz
C1	Period	2.5517 ms

Time
Position: -6.5 ms
Base: 800 us/div
Rate, Samples...

Channel 1 (1±)
Offset: 0 V
Range: 540 mV/div
Channel 2 (2±)

Discovery3 C SN:210415BDDA33 100 MHz Status: OK v3.24.2

Figure 5: 792 Hz sine wave generated using TCC0 and manual polling of DAC_CH0DRE_bm. The waveform follows the 256-point lookup table.


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Run
Stop
C1 C2
2.7
2.16
1.62
1.08
0.54
0
-0.54
-1.08
-1.62
-2.16
-2.7
-2.484 ms
-2.184 ms
-1.884 ms
-1.584 ms
-1.284 ms
-0.984 ms
-0.684 ms
-0.384 ms
-0.084 ms
0.216 ms
0.516 ms
Options
Add Channel
Filters Wavegens
Manual Trigger
Discovery3 C SN:210415BDDA33
100 MHz
Status: OK
v3.24.2

Buffer: 10 | Mode: Repeated | Trigger: Auto | Source: Channel 1 | Type: Edge | Condition: Rising | Level: 0 V | Hyst.: Auto | LCondition: Less

32768 samples at 2.5641 MHz | 2025-11-14 21:08:34.782.624.540

Measurements

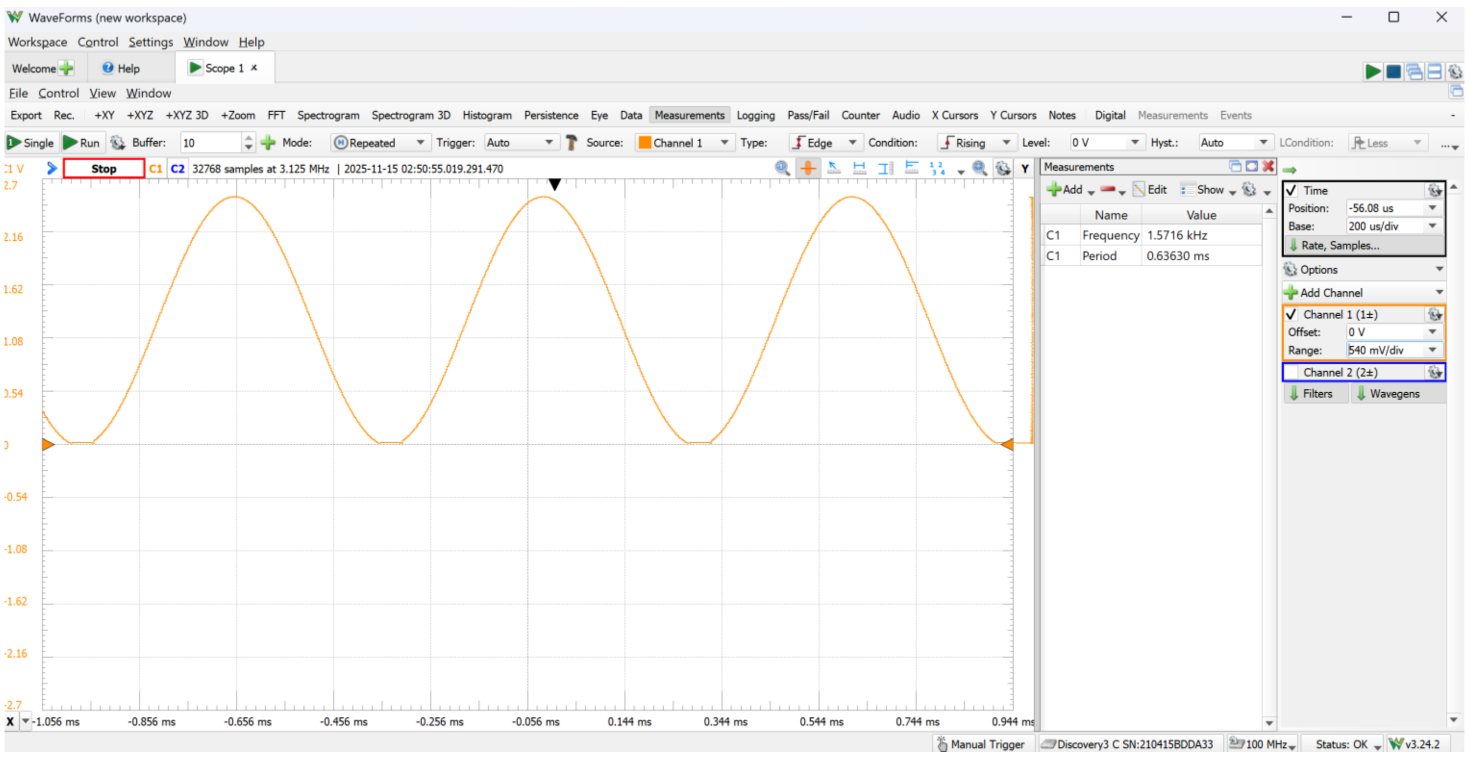
	Name	Value
C1	Frequency	1.0664 kHz
C1	Period	0.93773 ms

Time — Position: -983.8 us; Base: 300 us/div

Channel 1 (1±) — Offset: 0 V; Range: 540 mV/div

Channel 2 (2±)

Figure 6: First waveform that failed to meet the desired accuracy in lab8_2a.c. It should have been 1092.


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Scope 1
File Control View Window
Export Rec. +XY +XYZ +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Run
Stop
C1 C2
2.7
2.16
1.62
1.08
0.54
-0.54
-1.08
-1.62
-2.16
-2.7
-1.056 ms
-0.856 ms
-0.656 ms
-0.456 ms
-0.256 ms
-0.056 ms
0.144 ms
0.344 ms
0.544 ms
0.744 ms
0.944 ms
Options
Add Channel
Filters Wavegens
Manual Trigger
Discovery3 C SN:210415BDDA33
100 MHz
Status: OK
v3.24.2

Buffer: 10 | Mode: Repeated | Trigger: Auto | Source: Channel 1 | Type: Edge | Condition: Rising | Level: 0 V | Hyst.: Auto | LCondition: Less

32768 samples at 3.125 MHz | 2025-11-15 02:50:55.019.291.470

Measurements

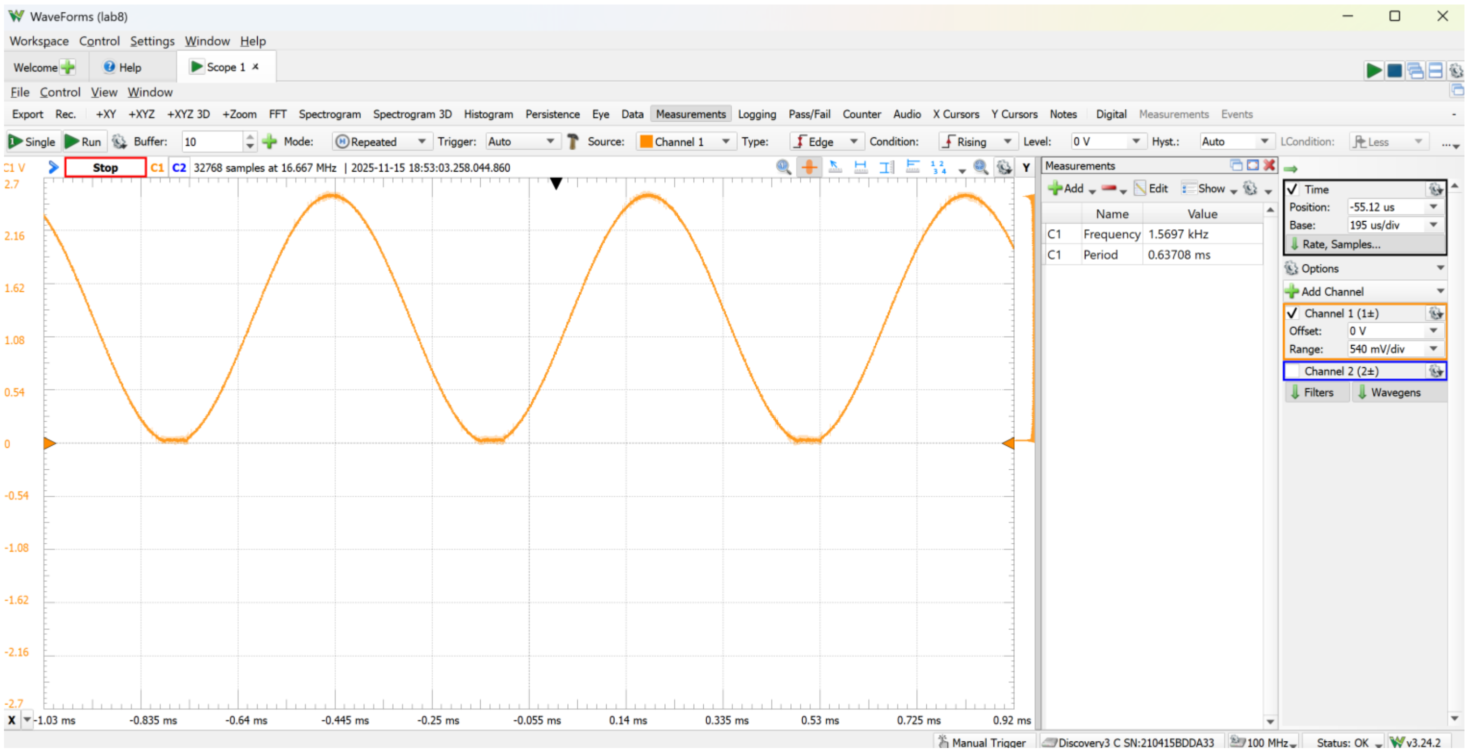
	Name	Value
C1	Frequency	1.5716 kHz
C1	Period	0.63630 ms

Time — Position: -56.08 us; Base: 200 us/div

Channel 1 (1±) — Offset: 0 V; Range: 540 mV/div

Channel 2 (2±)

Figure 7: Lab8_2b waveform within the desired accuracy using the event system (should be a sine wave of 1567.98 Hz).



Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

C1 C2 32768 samples at 16.667 MHz | 2025-11-15 18:53:03.258.044.860

Measurements

Name	Value
C1 Frequency	1.5697 kHz
C1 Period	0.63708 ms

Time
Position: -55.12 us
Base: 195 us/div

Channel 1 (1±)
Offset: 0 V
Range: 540 mV/div

Channel 2 (2±)

Discovery3 C SN:210415BDDA33 | 100 MHz | Status: OK | v3.24.2

Figure 8: Lab8_3 waveform within the desired accuracy using the DMA (should be a sine wave of 1567.98 Hz).