Kadeem Samuel
10/4/18
2559-2971

Function Generator Serial D2A Micro-Controller Applications

Introduction

The processor used for this module is the Atmel ATMega324P. This is an 8-bit processor that is programmed through the usage of the Atmel Ice, using the in-system programming method (functionally an SPI system). This microcontroller has a 1 MHz clock, 1k bytes of EEPROM, and 2K bytes of internal SRAM, in addition to 4 I/O ports, multiple timer/counters, and more. The goal of the design with this microcontroller is to output the digital data that represents a waveform to an external DAC, which generates the analog waveform. The microcontroller can output multiple different waveform types, and can also modulate the frequency and amplitude, all based upon user inputs.

Design

For the hardware, the Atmel Ice is used as the programmer for the ATMega device. This is connected to the ATMega as shown in Figure 1 in the appendix. A 4.7k pull-up resistor is used to connect the reset in place of a wire. In Atmel Studio 7, the interface mode is set to ISP for serial communication. A 5V supply is used to supply power to the circuit, which also represents the voltage level for logical high values. A layout of the design is provided in Figure 2, with the actual board shown in Figure 3. A potentiometer is used to adjust the frequency of the output waveform between 10 Hz and 100 Hz, and another is used to adjust the output voltage peak between 1V max and 5V max. Two switches are used to alternate between the four waveforms: sawtooth, sine, triangle, and square wave.

For the software, a flowchart of the code is shown in Figures 4 through 5. It works as follows: the four lookup tables for the waveforms are stored in memory. Easily accessible global variables are initialized for various parameters, such as the amplitude and period offsets. Port D is set as an input for the switches, as is port A. The SPI and the ADC are initialized. A frequency shift value is set based upon one ADC, and the amplitude shift value is set based upon the other. Math is done to calculate the necessary delay time and amplitude scaling factor, and the waveform to use is determined based upon the switch inputs. The data to write to the DAC is calculated based upon the amplitude scaling factor and the next value in the lookup table, then the data is written to the DAC, and a delay occurs to account for the frequency. The full code is provided in Figures 6 – 14. The bill of materials is shown in Table 1, with the total cost coming to $107.48.

Conclusion

The amplitude range was measured to be about 1.004V to 5.12V, both being within the 5% error range. Similarly, the frequency ranged from 9.96 Hz to 102.01 Hz, also within the error range, as shown in Figures 15-20. These values held up for all four waveform options. The difficulty creating the function generator lay in generating the proper frequency and amplitude for the output waveform: the CKDIV8 programmable fuse was turned off, in addition to varying the values of F_CPU, the amplitude scaling factor, and the frequency scaling factor. Once these values were determined through trial and error, the rest was simple to execute. The SPI with the external DAC is straightforward as is receiving inputs from the switches and the potentiometers.

Appendix

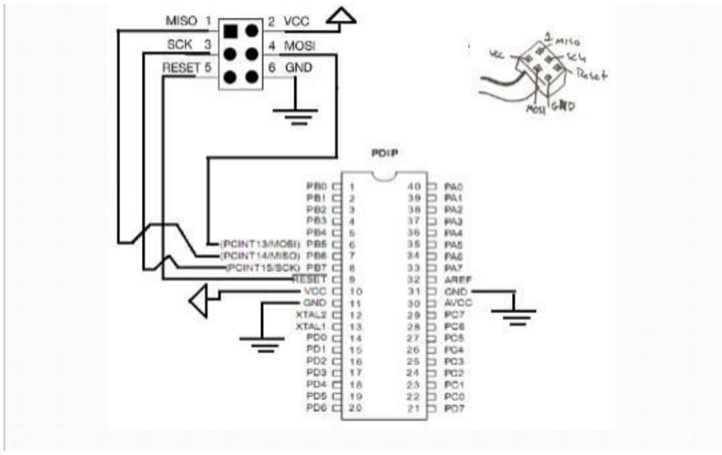

Figure 1: Pin Layout for Connecting Atmel Ice to ATMega324P, Red Wire is SCK

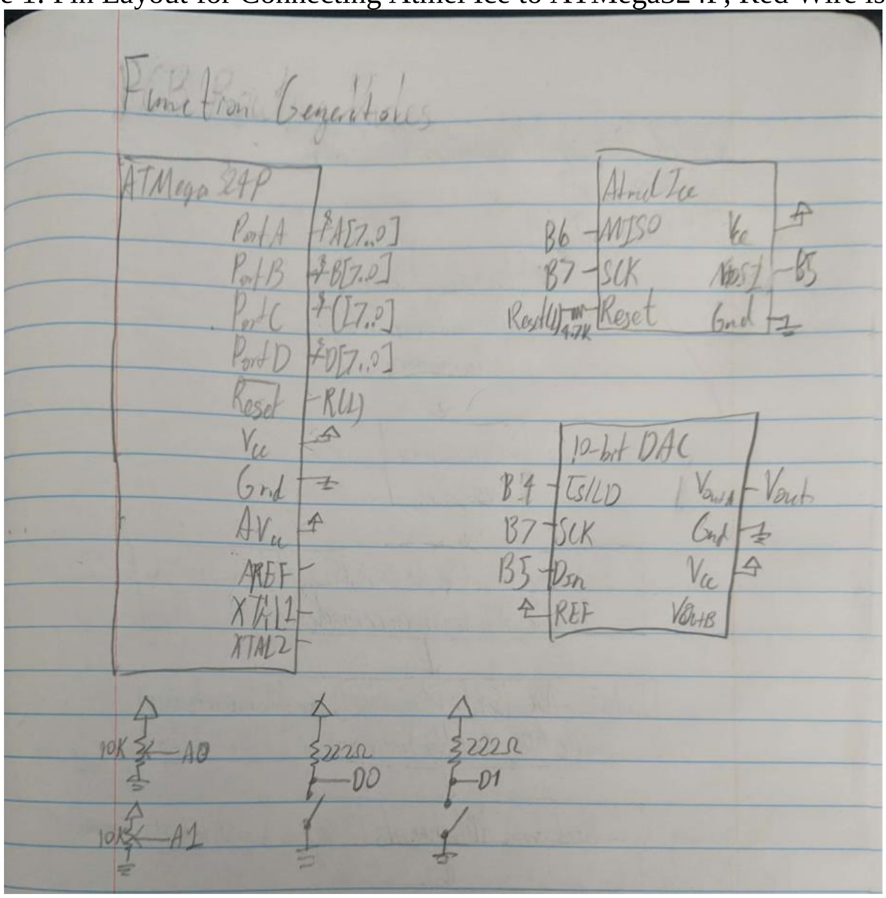

Figure 2: Hardware Connection Block Diagram of the Complete Circuit

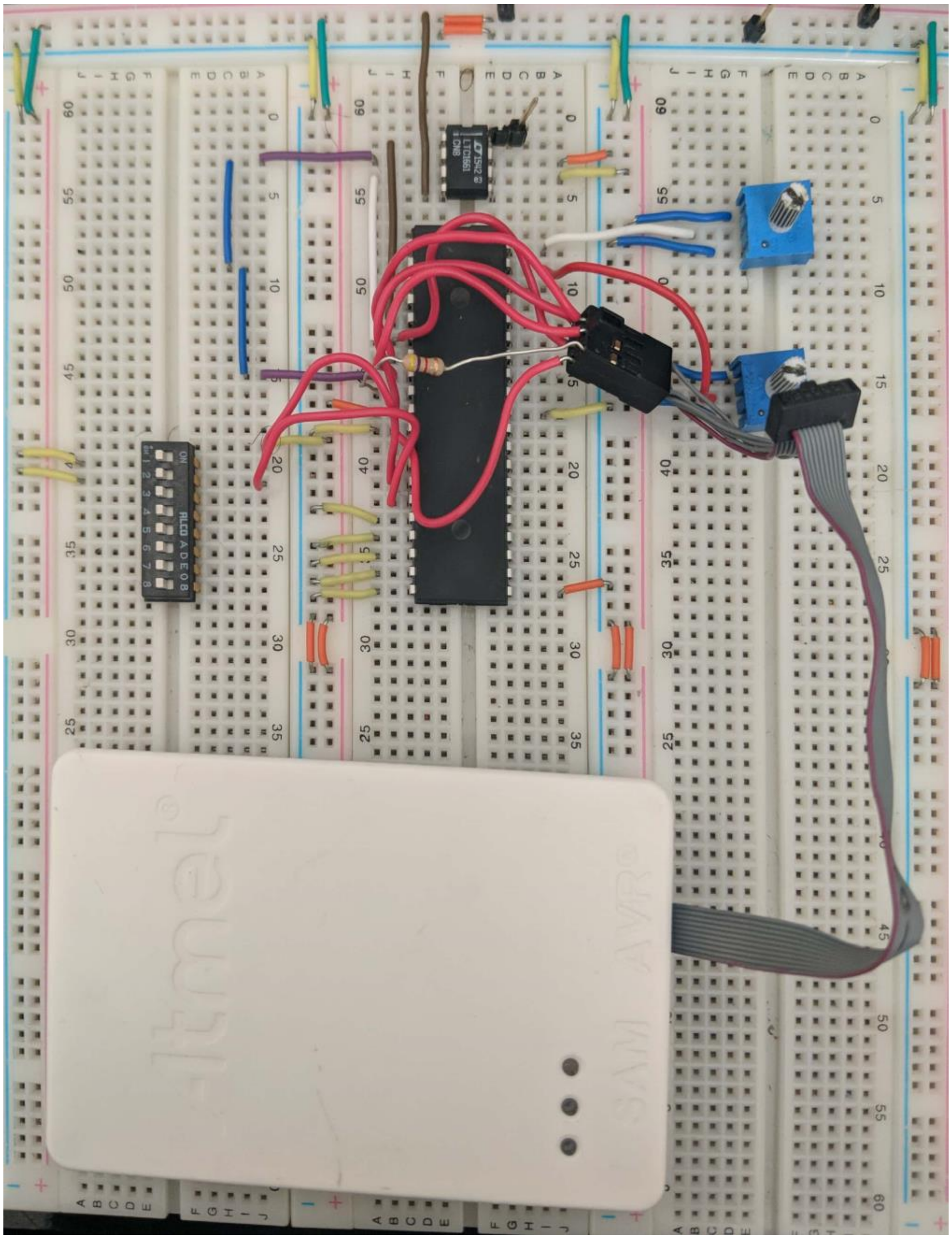

Figure 3: Actual Circuit Showing Atmel Ice (Bottom), ATMega324P (Center), and DAC (Top)

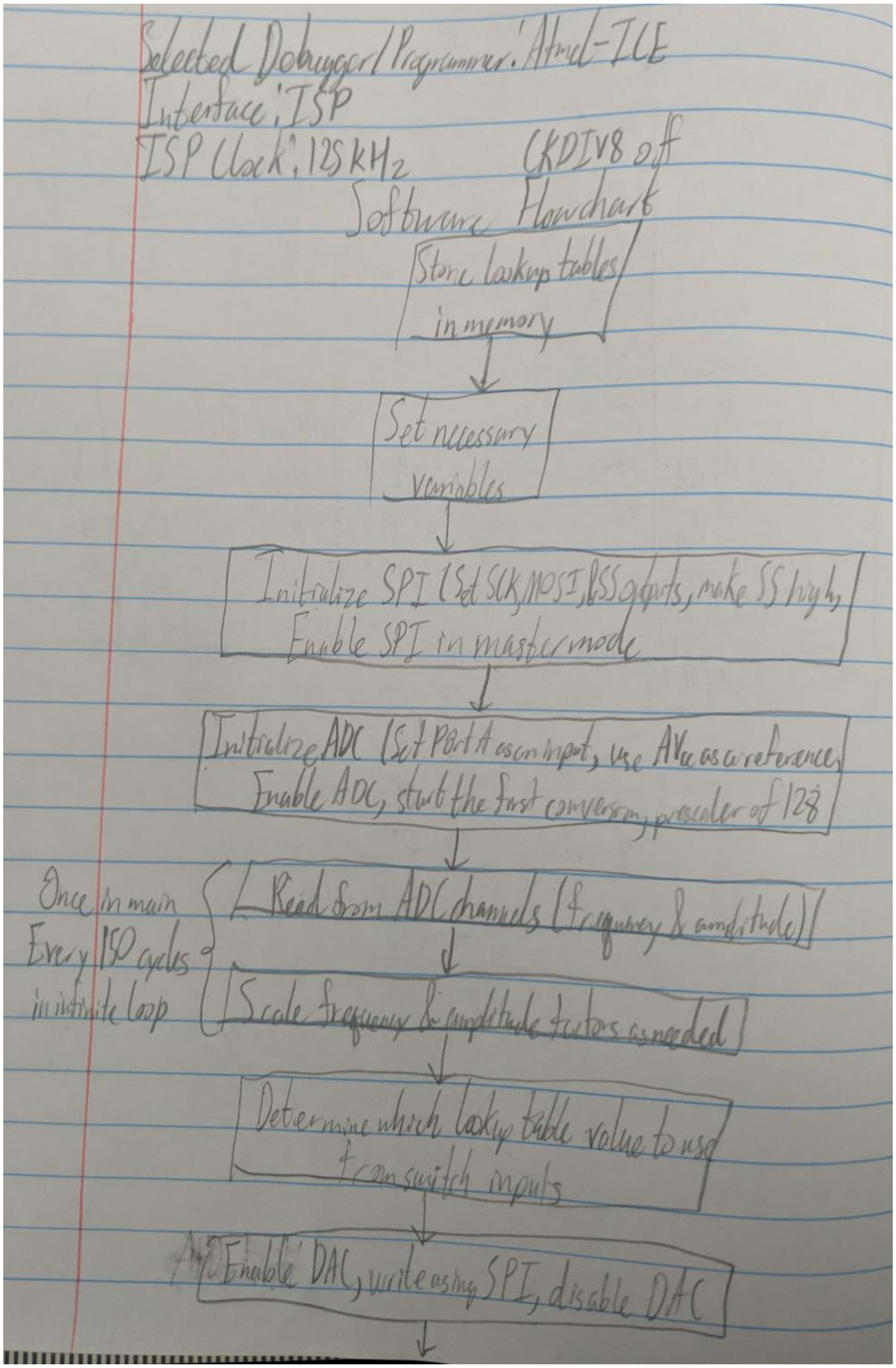

Figure 4: Software Flowchart, Page 1

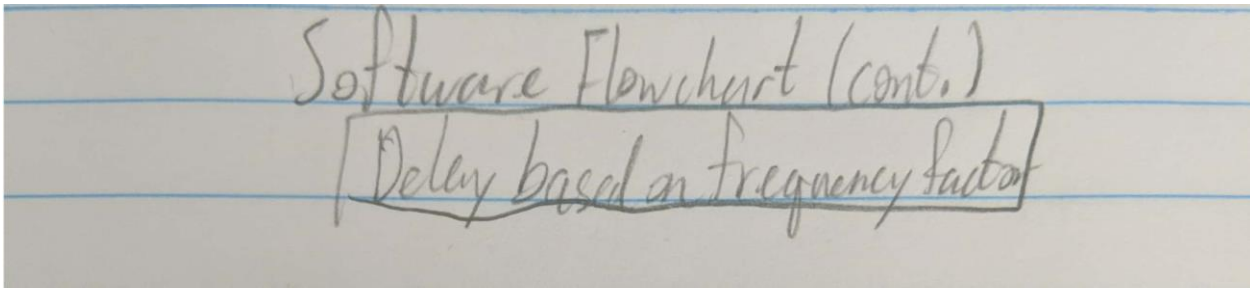

Figure 5: Software Flowchart, Page 2

```c
/*
 * FunctionGenerator.c
 *
 * Created: 10/3/2018 8:25:26 AM
 * Author : Kadeem
 */

#define F_CPU 98000 //1 MHz system clock

#include <avr/io.h>
#include <util/delay.h>

//Table for sine values
const uint16_t sineTable[] = {
    0x200,0x232,0x264,0x295,0x2c4,0x2f1,0x31c,0x345,
    0x36a,0x38c,0x3aa,0x3c4,0x3d9,0x3ea,0x3f6,0x3fe,
    0x3ff,0x3fe,0x3f6,0x3ea,0x3d9,0x3c4,0x3aa,0x38c,
    0x36a,0x345,0x31c,0x2f1,0x2c4,0x295,0x264,0x232,
    0x200,0x1ce,0x19c,0x16b,0x13c,0x10f,0xe4,0xbb,
    0x96,0x74,0x56,0x3c,0x27,0x16,0xa,0x2,
    0x0,0x2,0xa,0x16,0x27,0x3c,0x56,0x74,
    0x96,0xbb,0xe4,0x10f,0x13c,0x16b,0x19c,0x1ce
    };

```

Figure 6: Program Code, Page 1

```
const uint16_t squareTable[] = {
    0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
    0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
    0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
    0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
    0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
    0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
    0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
    0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
    0x3ff,0x3ff,0x3ff,0x3ff
    };
const uint16_t triangleTable[] = {
    0x20,0x40,0x60,0x80,0xa0,0xc0,0xe0,0x100,
    0x120,0x140,0x160,0x180,0x1a0,0x1c0,0x1e0,0x200,
    0x220,0x240,0x260,0x280,0x2a0,0x2c0,0x2e0,0x300,
    0x320,0x340,0x360,0x380,0x3a0,0x3c0,0x3e0,0x3ff,
    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
    };
const uint16_t sawtoothTable[] = {
    0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,
    0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0,0x100,
```

Figure 7: Program Code, Page 2

```
    0x110,0x120,0x130,0x140,0x150,0x160,0x170,0x180,
    0x190,0x1a0,0x1b0,0x1c0,0x1d0,0x1e0,0x1f0,0x200,
    0x210,0x220,0x230,0x240,0x250,0x260,0x270,0x280,
    0x290,0x2a0,0x2b0,0x2c0,0x2d0,0x2e0,0x2f0,0x300,
    0x310,0x320,0x330,0x340,0x350,0x360,0x370,0x380,
    0x390,0x3a0,0x3b0,0x3c0,0x3d0,0x3e0,0x3f0
    };

//Using ADC0 for voltage reference, and ADC1 for frequency
void ADC_init(void);
uint16_t ADC0_read(void);
uint16_t ADC1_read(void);
void spi_init(void);
void spi_write(uint16_t finalData);
void delays(int delayTime);

//Global variables
/*Use two switches to determine the waveform
    0b00: Sine wave
    0b01: Square wave
    0b10: Triangle wave
    0b11: Sawtooth wave
*/
//Amplitude is between 1 and 5
```

Figure 8: Program Code, Page 3

```
//Vary value of ampShift between 0 and 4 based upon ADC1
//Add 1, and divide the lookup value by this
double volatile amplitude = 2;
double volatile ampShift = 0;
//Frequency varies from 10Hz to 100Hz
//Vary value of freqShift between 0 and 90 based upon ADC0
//Add 10, and divide the lookup value by this
double volatile period = 58;
double volatile freqShift = 0;
int volatile peak = 64;
int main(void)
{

    //Variable to know when interrupt occurred
    uint8_t volatile counter = 0;
    uint8_t volatile counter2 = 0;
    double volatile fShift = 0;
    double volatile aShift = 0;
    DDRD = 0; //Use PortD as an input for the switches, on 0 and 1
    spi_init(); //Initialize SPI
    ADC_init(); //Initialize ADC
    double volatile data = 0;
    int volatile delayTime = 0;
    fShift = ADC0_read();
```

Figure 9: Program Code, Page 4

```
    freqShift = (period + period*(fShift/1024));
    delayTime = (int)(freqShift);
    aShift = ADC1_read();
    ampShift = (uint16_t)(amplitude + 4*(aShift/1024));
    uint16_t volatile finalData = (uint16_t)(data / ampShift);
    uint8_t volatile switches = 0;
    while (1)
    {
        if(counter2 == 150)
        {
            counter2 = 0;
            fShift = ADC0_read();
            freqShift = (period*(1 + 15*((fShift-25)/1024)));
            delayTime = (int)(freqShift);
            aShift = ADC1_read();
            ampShift = (1 + 5*(aShift/1024));
        }
        switches = PIND;
        switch (switches)
        {
        case 0x00:
            data = sineTable[counter];
            break;
        case 0x01:
```

Figure 10: Program Code, Page 5

```c
            data = squareTable[counter];
            break;
        case 0x02:
            data = triangleTable[counter];
            break;
        case 0x03:
            data = sawtoothTable[counter];
            break;
        default:
            break;
        }
        finalData = (uint16_t)(data / ampShift);
        ++counter;
        ++counter2;
        spi_write(finalData);
        if (counter == peak)
        {
            counter = 0;
        }
        if(switches == 0x01)
        {
            delays(delayTime+14);
        } else {
            delays(delayTime);
```

Figure 11: Program Code, Page 6

```c
        }
    }
}

void delays(int delayTime)
{
    while (0 < delayTime)
    {
        _delay_us(82);
        --delayTime;
    }
}

void spi_init(void)
{
    DDRB = 0xB0; ////SCK, MOSI, and SS as outputs
    PORTB = 0x10; //Make SS high
    SPCR0 = (1<<SPE0)|(1<<MSTR0); //Enable the SPI, master mode, no interrupts
}

void spi_write(uint16_t finalData)
{
    uint8_t volatile data = 0;
    PORTB = 0x00; //Drive SS low
```

Figure 12: Program Code, Page 7

```
    //Load control code and 4 bits of input data
    SPDR0 = (0x90)|(uint8_t)(finalData>>6);
    while((SPSR0 & 0x80) != 0x80);
    //Load latter 6 bits of data
    data = SPDR0;
    SPDR0 = (uint8_t)(finalData<<2);
    while((SPSR0 & 0x80) != 0x80);
    data = SPDR0;
    //Drive SS pin high again
    PORTB = 0x10;
}

void ADC_init(void)
{
    DDRA = 0x80; //PA is all inputs
    DIDR0 = (1<<ADC0D); //Disable digital input buffer on PA0 to reduce power consumption
    ADMUX = (1<<REFS0)|(0x0); //AVcc pin as reference, right adjusted format, gain of 1, using ADC0
    ADCSRA |= (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(ADPS0); //Enable the ADC, and start the first conversion, prescaler of 128
}

//For frequency
uint16_t ADC0_read(void)
{
    ADMUX = (1<<REFS0)|(0x0); //Using ADC0
```

Figure 13: Program Code, Page 8

```
    ADCSRA |= (1<<ADSC);
    while(ADCSRA & (1<<ADSC));
    //Continuously poll the flag
    return (ADC);
}

//For amplitude
uint16_t ADC1_read(void)
{
    ADMUX = (1<<REFS0)|(0x1); //Using ADC1
    ADCSRA |= (1<<ADSC);
    while(ADCSRA & (1<<ADSC));
    //Continuously poll the flag
    return (ADC);
}
```

Figure 14: Program Code, Page 9

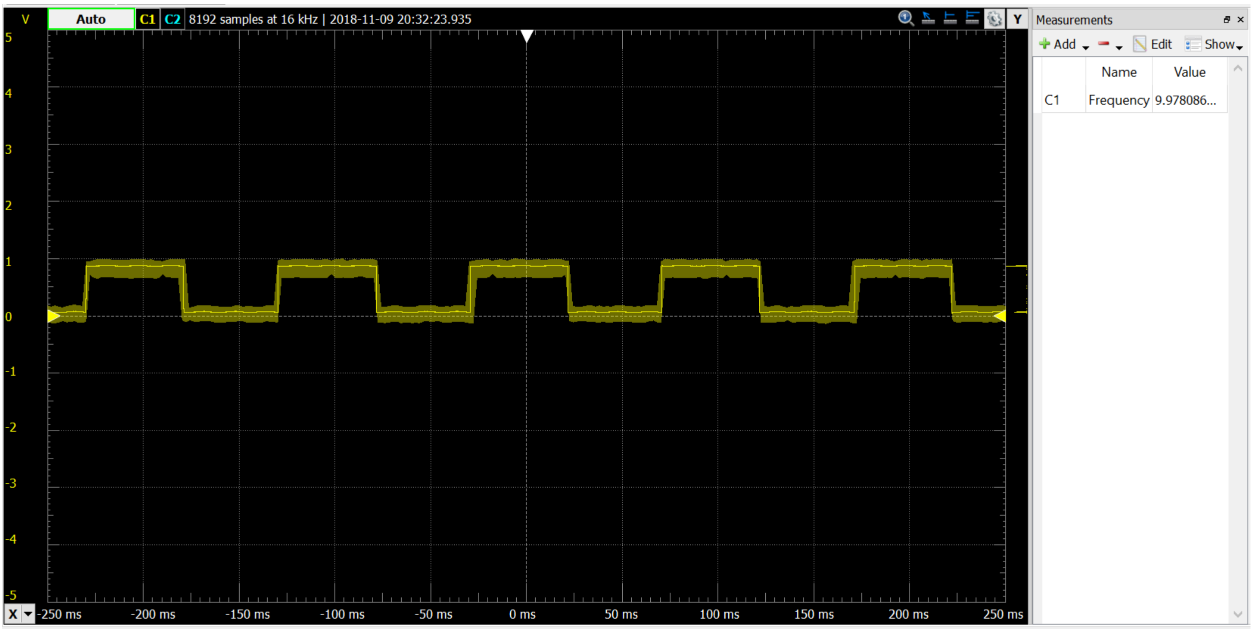

Figure 15: 10 Hz Square Wave Output, 1Vpp

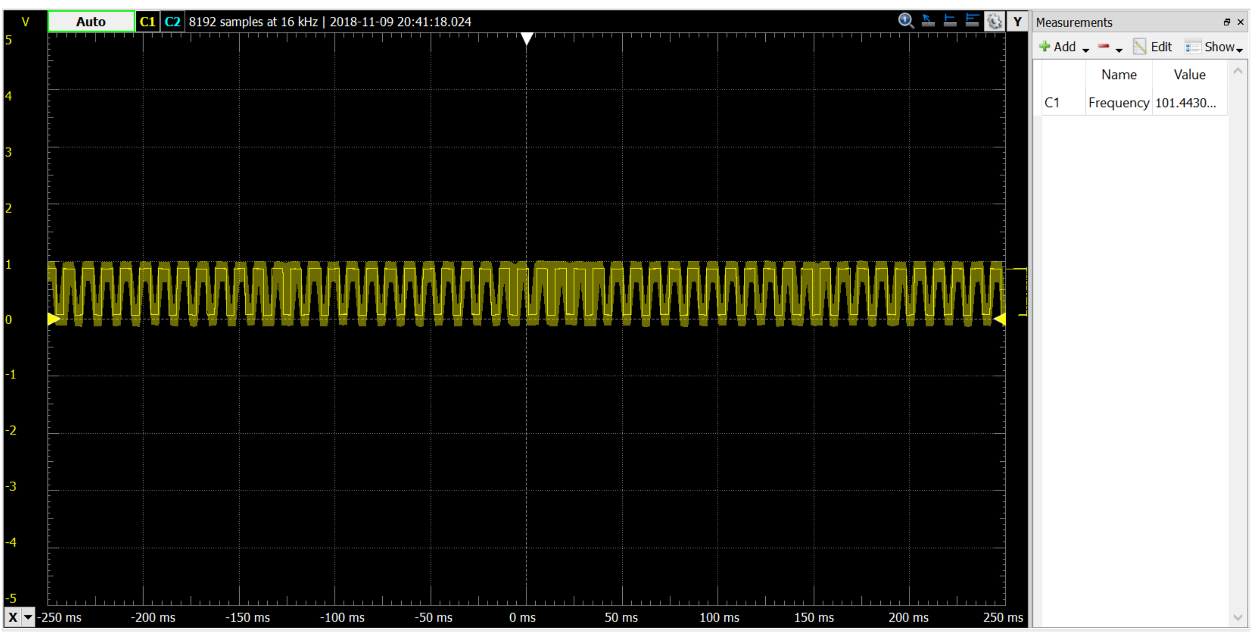

Figure 16: 100Hz Square Wave Output, 1Vpp

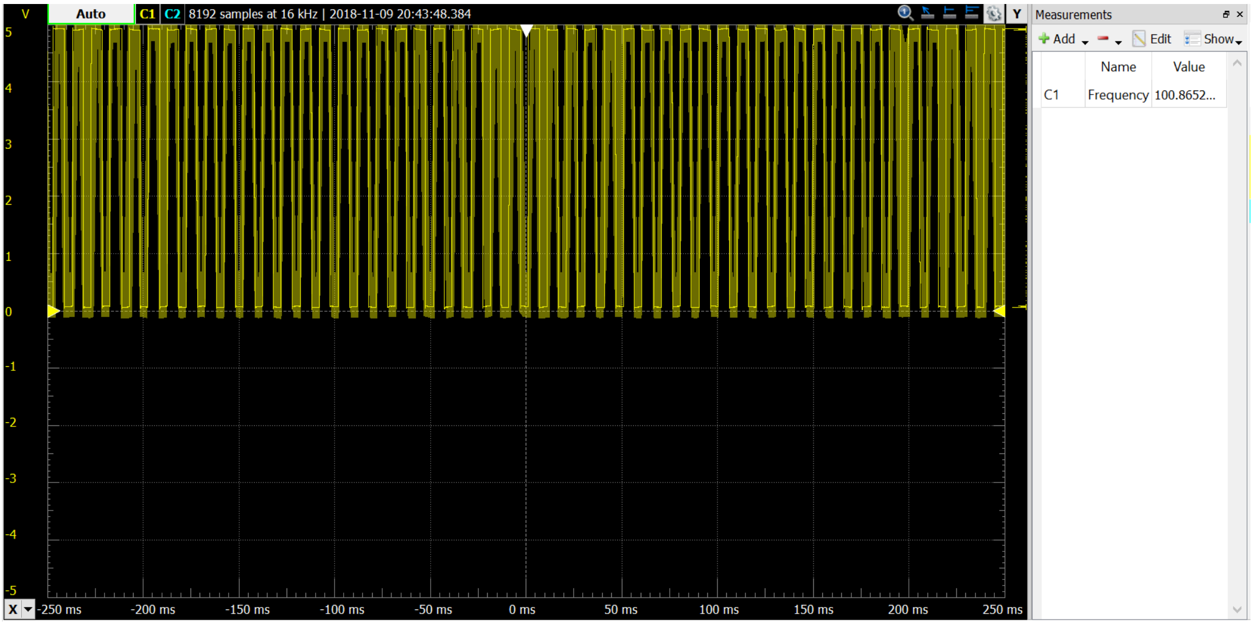

Figure 17: 100Hz Square Wave Output, 5Vpp

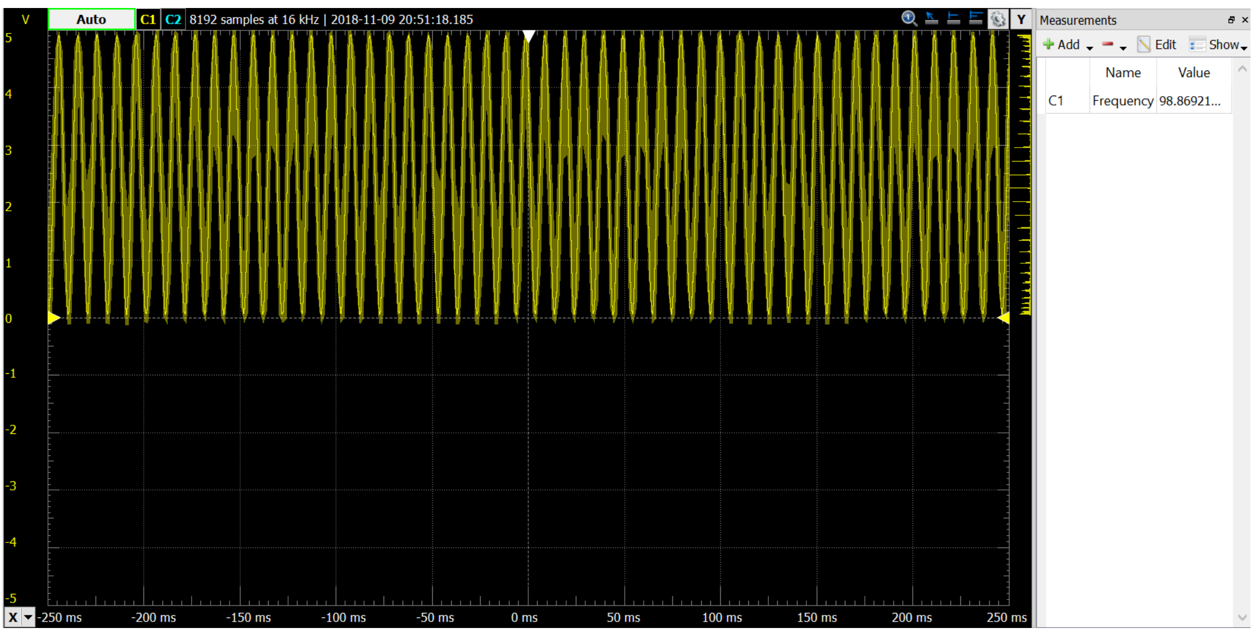

Figure 18: 100Hz Sine Wave Output, 5Vpp

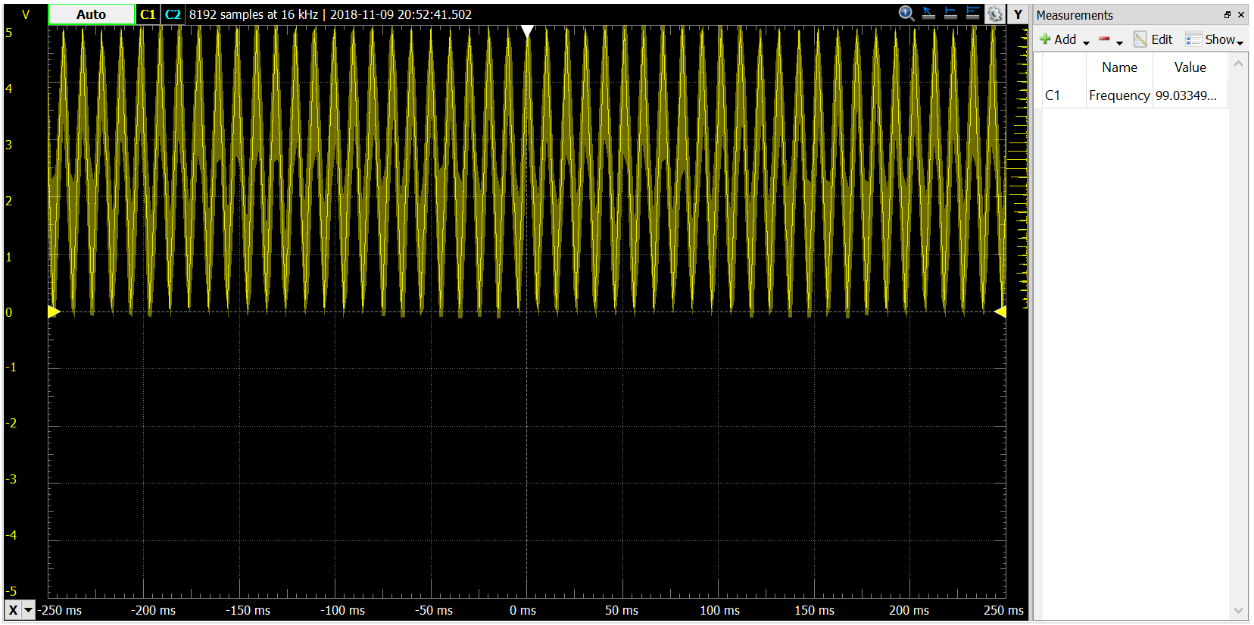

Figure 19: 100Hz Triangle Wave Output, 5Vpp

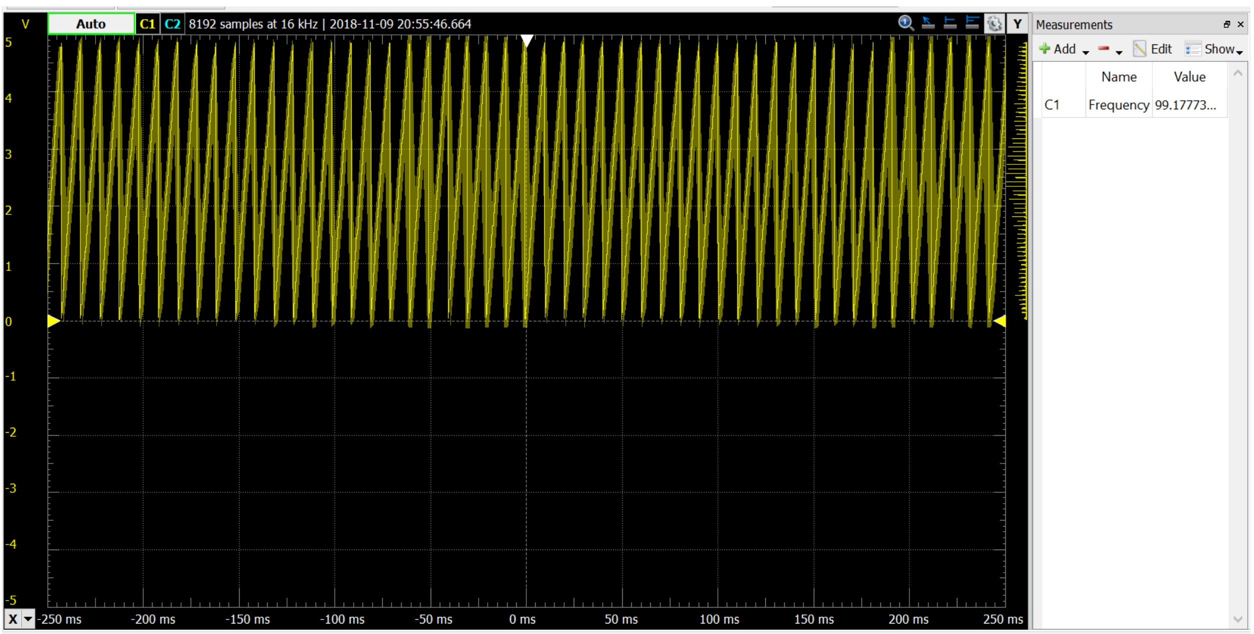

Figure 20: 100Hz Sawtooth Wave Output, 5Vpp

Part Number	Part Name	Cost per Part	Volume Discount (Price per unit for 100 units)	Source
1	ATMega324P	$4.87	$4.04	Mouser Electronics
2	Atmel Ice with Connector	$93.61	N/A	Digi-Key Electronics
3	4.7K Resistor	$0.10	$0.04	Jameco Electronics
4	Breadboard with Wires	$3.59	N/A	Amazon
5	LTC1661 DAC	$3.46	$1.79	DigiKey
6	10K Potentiometer (x2)	$0.57	$0.45	Mouser Electronics
7	10K Resistor	$0.25	$0.22	Jameco Electronics
8	222 Ohm Resistor	$0.12	N/A	Galco Industrial Electronics
9	CTS Switch Array	$0.91	$0.76	DigiKey
Total		$107.48		

Table 1: Bill of Materials