OBJECTIVES

- Understand how to utilize a *digital-to-analog converter* (*DAC*), also known as a *D/A system*.
- Learn how to effectively recreate a waveform with a *lookup table* (*LUT*) and TC/DAC systems.
- Understand why and how to free up CPU utilization with a *Direct Memory Access* (*DMA*) system.
- Create a functional piano synthesizer keyboard using your computer keyboard and the USART, DMA, and DAC systems available within the *ATxmega128A1U* microcontroller.

INTRODUCTION

As seen in a previous lab, a microprocessor can use an *analog-to-digital converter* (*ADC*) to interpret analog signals. Conversely, a *digital-to-analog converter* (*DAC*) allows a microprocessor to convert a digital value, whose limits are also determined by the hardware and software configurations of the system, into an analog signal, i.e., a voltage waveform. This ability to generate analog voltage waveforms allows a microprocessor yet another method of communication with external devices.

Separately, as mentioned before, timing is generally imperative in digital systems. Throughout this course, the majority of all operation and computation performed within a microcontroller has been performed directly by a *central processing unit* (*CPU*). Even though the peripherals within the microcontroller execute independently from the CPU, with exception to when using the event system, all *input and output* (*I/O*) between systems up to this point has been handled by the central processing unit. In other words, the CPU has been almost completely responsible for reading/writing data, waiting for asynchronous events, executing interrupt service routines, etc. In applications that require a lot of computation, or those that are known to be *CPU-bound*, this tight coupling between systems can cause CPU execution time to be wasted on I/O.

Many applications rely on moving a large amount of data between application subsystems. This desire for flexible data transfer motivated the creation of the *Direct Memory Access* (*DMA*) system. Simply put, a DMA system allows data to move from one system to another via a data bus, *without processor intervention*.

LAB STRUCTURE

In this lab, you will begin to utilize the DAC and DMA systems within the *ATxmega128A1U*. In order to do so, you will first learn the basics of the DAC system by generating a waveform with a constant voltage. Then, you will generate (or more specifically, emulate) a sine waveform at various frequencies, with a table of predefined data points. This method of using a table of predefined data, otherwise known as a *lookup table* (*LUT*), to generate a waveform is quite common and efficient, although it should not be surprising that with only a finite amount of data points, the generated waveform will amount some degree of (possibly negligible) error. Around this point within the lab, the need for a DMA system will be apparent, and you will begin to use such a system available within the *ATxmega128A1U*.

Following this, you will use your emulated sine waveform to output a musical note from the speaker on your *OOTB Analog Backpack*, after learning some fundamental music terminology. Finally, you will learn some basic music theory regarding pianos, and then finish the lab by creating a simple keyboard synthesizer program. This program will allow you to use twelve of your computer keyboard keys to generate specific musical notes within two selected musical octaves, namely the 6th and 7th octaves.

REQUIRED MATERIALS

- Atmel *ATxmega128A1U* AU Manual (doc8331)
- Atmel *ATxmega128A1U* Manual (doc8385)
- *OOTB µPAD v2.0* with USB A/B cable
- Digital Analog Discovery (DAD) kit, with *WaveForms*
- *OOTB Analog Backpack*, with accompanying schematic
- *IS31AP4991* Audio Power Amplifier Datasheet

SUPPLEMENTAL MATERIALS

- *Atmel 8046D - AVR1304 (DMA Controller)*
- *Create, Simulate, and Emulate a Project* tutorial
- *Changing the System Clock* assignment

PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

As required, you must re-read the *Lab Rules & Policies* before submitting any pre-lab assignment and before attending any lab.

NOTE: All software written in this lab must first configure your microcontroller system clock frequency to 32 MHz! See *Homeowrk 4* for information on how to change the clock frequency.

1. INTRODUCTION TO DAC

As you will soon discover, using a DAC system is very straightforward after having utilized an ADC system.

1.1. Read § 29 (*DAC – Digital to Analog Converter*) of the 8331 manual to learn how to configure and use the DAC system within the *ATxmega128A1U*.

Now, to demonstrate your understanding of the DAC system, you will generate a waveform with a constant voltage, and then measure this waveform with the *Scope* feature of your *Waveforms* software.

1.2. Create a program, **`lab8_1.c`**, to generate a waveform with a constant voltage of 1.5 V, using a DAC module for which you have access to probe via the *µPAD*. Measure the generated waveform with your DAD and the *Scope* feature of *Waveforms*, and then take a screenshot of the relevant waveform.

2. GENERATING A WAVEFORM WITH A LOOKUP TABLE

Sometimes, a DAC system may be used to output analog voltages based on sampled ADC values. In other cases, instead of sampling a waveform first, it may be desired to simply recreate a known waveform. In either case, timing requirements are often imposed, and the capabilities of the microcontroller completely dictate whether or not these requirements can be met.

When it is desired to recreate a known waveform, the data of the waveform is often stored within a data structure such as an array or table, to avoid approximating the waveform with (potentially) computationally expensive operations. When a data structure such as an array or table is used for this purpose, the data structure is often referred to as a *lookup table* (*LUT*).

NOTE: A *data structure* refers to a format of organizing some unit(s) of data e.g., an array in the "C" programming language is a data structure that represents a contiguous block of memory (either program, data, BSS, stack, or heap memory), where a reference to an array is a pointer to the first element within the relevant block of memory.

Although lookup tables can allow a microprocessor to avoid approximating waveform data, they can also require a considerable amount of memory, depending on the desired accuracy of the waveform.

In this part of the lab, you will use some "C" array to create a lookup table for a specified sine wave. Then, you will utilize this lookup table along with a DAC and TC system to *continually* recreate the sine waveform at a given rate.

To generate the necessary data points for your lookup table, you may use *MATLAB*, *Excel*, or an online lookup table calculator such as the following: Sine Waveform Data Calculator.

Make sure that the lookup table data in the same format expected by your DAC, i.e., left-adjusted or right-adjusted.

2.1. Write a "C" program, **`lab8_2a.c`**, to generate a sine wave of at least 256 data points, with voltages ranging between 0V and AREFB, and with a frequency of 1000 Hz, where an error of up to 2% is permitted for the generated frequency. (Do not worry if there are any slight voltage errors.) Your program must utilize some built-in DAC and TC module, and interrupts must be used to handle the relevant timing. Remember that, for the purposes of this course, the optimization tool within the relevant "C" compiler must be turned off. (Refer to the *Create, Simulate, and Emulate a Project* tutorial for more information on altering optimization levels.)

2.2. Use the *Scope* feature of your *Waveforms* software to measure the created sine waveform. Take a screenshot of the waveform, identifying its frequency.

2.3. In increments of 100 Hz, increase the target frequency until you cannot meet the desired accuracy of ±2%. Take a screenshot of the first waveform that fails to meet this criterion and include a *precise* frequency measurement of the waveform within this screenshot.

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?

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.

In the final section of this part, you will use events to generate a sine waveform of a specified frequency. An event should be configured to trigger upon an appropriate timer/counter overflow, which should then start a digital-to-analog conversion within the appropriate DAC module.

2.4. Write a "C" program, **`lab8_2b.c`**, to generate a sine wave of at least 256 data points, with voltages ranging between 0V and AREFB, and with a frequency of 1760 Hz,

where an error of up to 2% is permitted for the generated frequency. (Do not worry if there are any slight voltage errors.) Your program must utilize events. Again, remember that optimization within the "C" compiler tool must be turned off.

2.5. Use the *Scope* feature of your *Waveforms* software to measure the created sine waveform. Take a screenshot of the waveform and include a *precise* frequency measurement of the waveform within this screenshot.

PRE-LAB EXERCISES

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

3. INTRODUCTION TO DMA

By now in this course, you hopefully are starting to understand that writing software for a microprocessor requires a good understanding of computer hardware and, if programming with any language more abstract than assembly, how some compiler may generate machine code for a given processor.

In the previous parts of this lab, you generated various waveforms with a DAC and TC module. In general, using a timer/counter system is the best way to generate a precise waveform, but as you probably have noticed, *the output waveform has still been somewhat inaccurate*. This is often because some amount of latency will likely occur whenever our processor needs to execute some particular code, ultimately delaying our time-contingent program. A more effective way to generate the same waveform would be to utilize a *Direct Memory Access* (*DMA*) system along with a timer/counter. A DMA system allows data to move from one of our systems to another via the data bus, without processor intervention.

In the remaining parts of this lab, you will use a timer and a set of events to accurately "trigger" the DMA system within your microcontroller to move data from your lookup table (located within data memory) to the DAC system, as to avoid CPU utilization.

When configuring the DMA system, there are a few things that need to be considered:

- How many bytes will be transferred in a burst (one piece of data)?
- How many bytes will be in a complete transfer (i.e., is there more than one burst)?
- How many times will the specified set of data be transferred?
- From where should the DMA get the necessary data? If more than one byte is to be transferred, after a byte is retrieved, where should the DMA retrieve the next one?
- To where should the DMA transfer the data? Again, if there is more than one byte to be sent to the destination, where should the DMA transfer each additional byte after the first?
- Regarding the source and destination of the DMA transfer, does the DMA ever need to point back to the original memory location(s) after some transfer?
- What will trigger the DMA transfer? A completed ADC conversion? Some other interrupt? Some event?

NOTE: Be careful to note that when configuring the DMA, *there are several registers that can only be configured when the DMA is disabled*. Also, be aware that for many configurations, the DMA channel will be automatically disabled after a certain number of bursts or blocks.

3.1. Read § 5 (*DMAC – Direct Memory Access Controller*) of the 8331 manual, and the *Atmel 8046D - AVR1304 (DMA Controller Application Note)* document.

3.2. Write a "C" program, **`lab8_3.c`**, to utilize the DMA and event systems within the *ATxmega128A1U* to emulate the same sine waveform described in § 2, with a frequency of 1760 Hz. Note that an event other than that configured in the previous part will need to be used. Again, the frequency generated should have no more than ± 2% error.

3.3. Use your DAD and *Waveforms* software to measure the newly created sine waveform. Take a screenshot, including an *appropriate* measurement of the waveform's frequency.

4. INTRODUCTION TO MUSIC

In the previous two parts of this lab, you created a sine wave with a frequency of 1760 Hz. In musical contexts, this waveform represents the note $\mathbf{A_6}$, one of the 12 notes in the 6th octave. To further understand what this means, some basic music terminology is defined below.

In music, a *note* is a sound defined by a pitch and duration, where a *pitch* is a frequency of vibration (Hz), and where *sound* is created from vibrations that travel through the air or some other medium.

A special collection of notes, ordered by pitch, defines a musical *scale*, where a single scale can be manifested at many different pitch levels. The most common musical scales are typically written using eight notes, where the first and last notes are an octave apart. An *octave* is defined to be an interval of notes where any given base note within the interval has a frequency of vibration twice that of the same base note in the previous octave (or half of the same base note in a following octave). For example, the base note **A** in the 6th octave ($\mathbf{A_6}$), has a frequency of 1760 Hz, whereas the base note **A** in the 5th octave ($\mathbf{A_5}$) has a frequency 880 Hz. Similarly, the base note **A** in the 7th octave ($\mathbf{A_7}$) has a frequency of 3520 Hz.

Now, you will begin to utilize the speaker located on the *OOTB Analog Backpack*, to create your processor's first musical tone! (They grow up so fast.) However, to be able to use your on-board speaker, a few points regarding audio electronics will first be mentioned.

In the field of *audio electronics*, the main goal is to interpret and produce either sound or pressure wave signals. For a device to interpret such signals, an analog signal must first be encoded into a digital signal, with an analog-to-digital converter. (Often, a

preamplifier is used to amplify any weak digital signals into more manageable signals for any processing components.) Conversely, to produce sound or pressure wave signals, a digital signal must be sent from a digital-to-analog converter to a speaker system. To be able to drive a speaker system with enough power, an amplifier circuit is almost always needed.

NOTE: Normally, audio applications, as well as other digital signal processing applications, are handled by a specialized microprocessor known as a *digital signal processor* (*DSP*). Such a processor is generally built with specialized ADC and DAC systems, along with other specialized components.

On your *OOTB Analog Backpack*, there exists an IS31AP4991 audio power amplifier used to drive your on-board speaker (labeled J4 on your *Analog Backpack* schematic) with enough power, by amplifying any signals sent from an internal DAC system within your processor. Since amplifiers are not completely efficient, i.e., the level of power output is less than that of what is input, your amplifier component will need to be supplied with additional power than that of what is input. To provide the IS31AP4991 component located on your *OOTB Analog Backpack* with additional power, you will need to connect your DC Barrel Jack power adapter to the VIN supply, and also connect the other end of the adapter to an AC wall outlet.

NOTE: Do *not* use any power supply other than the one provided in your *µPAD* kit, or you will likely supply too much current to your speaker and fry the component.

Furthermore, the amplifier component on your *OOTB Analog Backpack* has a pin used to enable shutdown. *To utilize the IS31AP4991, shutdown will have to be prevented.*

4.1. Locate the IS31AP4991 audio power amplifier within your *OOTB Analog Backpack* schematic. Identify the DAC system that the amplifier input originates from. Identify how you will prevent the IS31AP4991 from entering its shutdown mode.

4.2. Create a program, **`lab8_4.c`**, to continually output an **A_6** note (1760 Hz) to your on-board speaker.

5. MAKING A MUSICAL INSTRUMENT

In this part of the lab, you will create a small interactive synthesizer keyboard, using your computer's keyboard, and the appropriate USART, DAC, TC, and DMA modules within your microcontroller.

A *synthesizer* (often abbreviated as *synth*) is an electronic musical instrument that generates electric signals which are then converted to sound through amplifiers and speakers. Synthesizers typically imitate traditional musical instruments like pianos, organs, flutes, or even vocals; they may also imitate any other wide range of sounds such as ocean waves, animal noises, simple electronic timbres, etc. The interface for a synthesizer is commonly in the form of a piano keyboard.

Within a synthesizer keyboard (as well as a piano keyboard), sound is generated electronically, with either simple switches or dedicated circuits associated with each key, which is unlike a normal acoustic piano, where upon a key being pressed, sound is generated by hammers hitting finely tuned strings.

Most modern pianos contain 88 keys. This means that the piano can generate around 88 different pitches (with each pitch corresponding to a musical note). Each key's frequency can be calculated through the following equation, where n represents the n^{th} key (for n = 1, …, 88):

$$f(n) = 440 \times 2^{\frac{n-49}{12}} \text{ Hz}$$

NOTE: **A_4** is a standard pitch that is often used to tune various musical instruments. **A_4** is generally the 49^{th} key on a keyboard, i.e., n = 49 in the above equation.

Every twelve keys on the piano make up an octave (see the previous part of this lab for the definition of an octave). Each octave on a piano is organized into a group of two black keys, a group of three black keys, and a group of seven white keys. The "C" note within each octave generally designates the start of the octave (when using a "C" scale) and is always placed before the group of two black keys.

When you create your synthesizer keyboard later in this part, you will emulate notes from the 6^{th} octave. The 6^{th} octave is chosen because it is the lowest octave in which all twelve notes have frequencies within the allowable range of our speaker's rated frequencies. To play each of these twelve notes, you will use

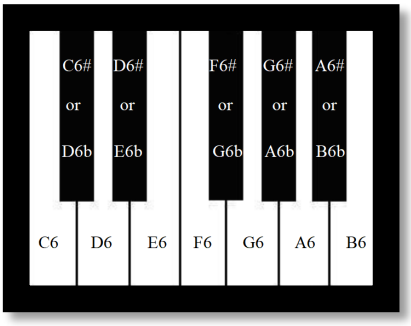

Figure 3: Piano key layout for 6^{th} octave

specific keys on a connected computer keyboard. For a list of the the specified key/note pairs and their corresponding frequencies,

see Table 1. Additionally, see Figure 3 for a graphic depicting a layout of the 6th octave with piano keys.

Beyond being able to play each of these note frequencies, the synthesizer must also support two different modes: *Sine*, where all notes must be output with a sine waveform, and *Triangle,* where all notes must be output with a *triangle* waveform. (Review the concept of a *triangle* waveform, if necessary.) To accomplish this, your program should utilize two separate lookup tables. In addition, to toggle between each of the two modes, the 's' key on a connected keyboard must be utilized.

Finally, *each note on your synth should only play for roughly as long as you hold the corresponding key*, and there must be no noticeable delay from the time a key is pressed to the time the corresponding note is output to your speaker. There will be a minimum note length required to achieve a continuous tone while the key is held, which will limit the responsiveness of your program to new key presses during that interval. This phenomenon is dependent on the computer that you are using to send commands to the *ATxmega128A1U.* (See the appendix for more details). You are not expected to play more than one key at the same time.

NOTE: Before you attempt to complete the entire program, it is recommended that you try to output a single frequency of both a sine and triangle waveform, switching between the two when the 's' key on a connected computer keyboard is pressed.

5.1. Write a "C" program, **`lab8_5.c`**, to create the specified synthesizer keyboard program. Each of the outputted note frequencies must have no more than ±2% error from the given frequency/period.

Computer Key	Musical Note	Note Frequency (Hz)
'W'	C_6	1046.50
'3'	$C^{\#}_6/D^{b}_6$	1108.73
'E'	D_6	1174.66
'4'	$D^{\#}_6/E^{b}_6$	1244.51
'R'	E_6	1318.51
'T'	F_6	1396.91
'6'	$F^{\#}_6/G^{b}_6$	1479.98
'Y'	G_6	1567.98
'7'	$G^{\#}_6/A^{b}_6$	1661.22
'U'	A_6	1760.00
'8'	$A^{\#}_6/B^{b}_6$	1864.66
'I'	B_6	1975.53

Table 1: Computer synthesizer keyboard details

PRE-LAB PROCEDURE SUMMARY

1) Answer any pre-lab exercises, when appropriate.
2) In § 1, configure a DAC system for the first time, and then generate a constant voltage waveform. Measure this waveform with your DAD, and then take a screenshot of the measured waveform.
3) In § 2, emulate sine waveform with a lookup table, DAC system, and various timing techniques. Take screenshots when appropriate.
4) In § 3, learn how to utilize the DMA system within the *ATxmega128A1U*, and then output the required sine waveform. Take a screenshot when appropriate.
5) In § 4, first learn music terminology, before outputting a $\mathbf{A_6}$ note to the speaker on your *OOTB Analog Backpack*.
6) In § 5, create a synthesizer keyboard program using the appropriate USART, DAC, TC, and DMA modules.

APPENDIX

A. COMPUTER KEYBOARD REPEAT DELAY

Your computer can most likely detect a keypress made to a connected keyboard in the order of picoseconds. To prevent a single keypress from being repeated unintentionally, your computer's operating system likely has a built-in repeat delay for any connected keyboard. Sometimes, like within this lab, we want our computer to recognize a single keypress multiple times, as quickly as possible.

To change the "Repeat delay" setting within the Windows operating system, perform the following steps:

1. Navigate to the *Keyboard Properties* dialog box from the Start menu (or alternately, through *Devices and Printers*). (In Windows 10, type *Run* at a Cortona prompt, then type *main.cpl @1*. The third letter in *cpl* is a lowercase L and the number 1 follows the @.)
2. Click on the *Speed* tab (see Figure 5).
3. Use the sliders beneath *Repeat delay* to increase/lower the delay used before a single keypress is detected multiple times.
4. Click *Apply* to confirm any changes made.

NOTE: The "Repeat rate" setting also available in the *Speed* tab is the frequency at which a single keypress is repeated, after the initial repeat delay. The configuration of this setting is *not* vital to this lab.

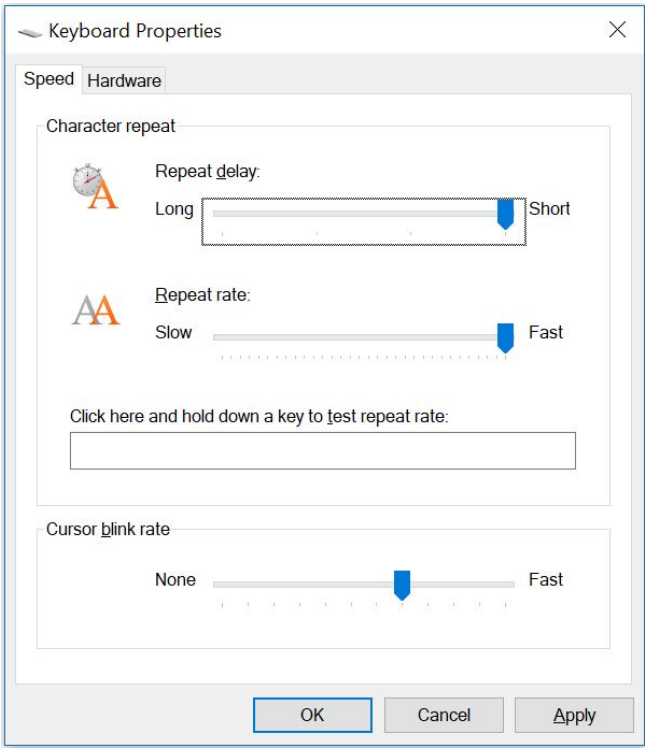

Figure 3: Repeat delay setting in *Keyboard Properties*