# Lab 6. Digital Piano using a Digital to Analog Converter



## Preparation

Read Sections 9.1, 9.2, 9.4, 9.6, 10.1, 10.2 and 10.3,
Update the SVN, move edXLab13.dll into your Keil/ARM/bin folder
Starter files **PeriodicSysTickInts_123** in EE319Kware, and **Lab6** (see **dac.xls** inside this project) in SVN
https://www.youtube.com/playlist?list=PLyg2vmIzGxXGxGIgbOrqYWALtXnDvxCCd
http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C13_Interactives.htm

## Purpose

There are three objectives for this lab:

1. to learn about Digital to Analog Converters (DACs)
2. to understand how digital data stored in a computer could be used to represent sounds and music;
3. to study how the DAC can be used to create sounds.

## Programming Requirements

All software for this lab must be written in C. You can debug your code in the simulator but your final code must run on the board with a DAC circuit. Keil uVision4 projects in C are a little different than the assembly projects used for Labs 1 to 4. The project configuration and the startup.s file are different. The Lab 6 starter file has the appropriate connections to the Lab 6 simulator/grader. However, you should also look at **SysTickInts.c**

## System Requirements

In this lab you will create a very simple sound generation system that illustrates an application of the DAC. Your goal is to create an embedded system that plays three notes, which will be a digital piano with three keys.

- Design a minimum of a 4-bit, binary weighted DAC
- Design a device driver for your DAC
- Interface three switches to act as synthesizer keys
- Implement a synthesizer with three notes with the switches and device driver
- Connect the DAC output to a speaker or headphones

## Procedure

All code written for this lab must be written in C.

## **Part a - Port Assignment**

Decide which port pins you will use for the inputs and outputs. Avoid the pins with hardware already connected. Table 4.4 in the book lists the pins to avoid. There is a Lab 6 simulator, but will we not use the grader in EE319K this semester. To use the simulator/grader the choices are listed in Tables 6.1 and 6.2. In particular, Table 6.1 shows you three possibilities for how you can connect the DAC output. Table 6.2 shows you three possibilities for how you can connect the four positive logic switches that constitute the piano keys. Obviously, you will not connect both inputs and outputs to the same pin. Also, note that these tables show 6-bits of DAC and 4 piano keys; whereas the minimum requirements are a 4-bit DAC and 3 piano keys.

| | | | |
|---|---|---|---|
| DAC bit 5 | PA7 | PB5 | PE5 |
| DAC bit 4 | PA6 | PB4 | PE4 |
| DAC bit 3 | PA5 | PB3 | PE3 |
| DAC bit 2 | PA4 | PB2 | PE2 |
| DAC bit 1 | PA3 | PB1 | PE1 |
| DAC bit 0 | PA2 | PB0 | PE0 |

**Table 6.1. Possible ports to interface the DAC outputs (DAC bits 4 and 5 are optional).**

| | | | |
|---|---|---|---|
| Piano key 3 | PA5 | PB3 | PE3 |
| Piano key 2 | PA4 | PB2 | PE2 |
| Piano key 1 | PA3 | PB1 | PE1 |
| Piano key 0 | PA2 | PB0 | PE0 |

**Table 6.2. Possible ports to interface the piano key inputs (piano key 3 is optional).**

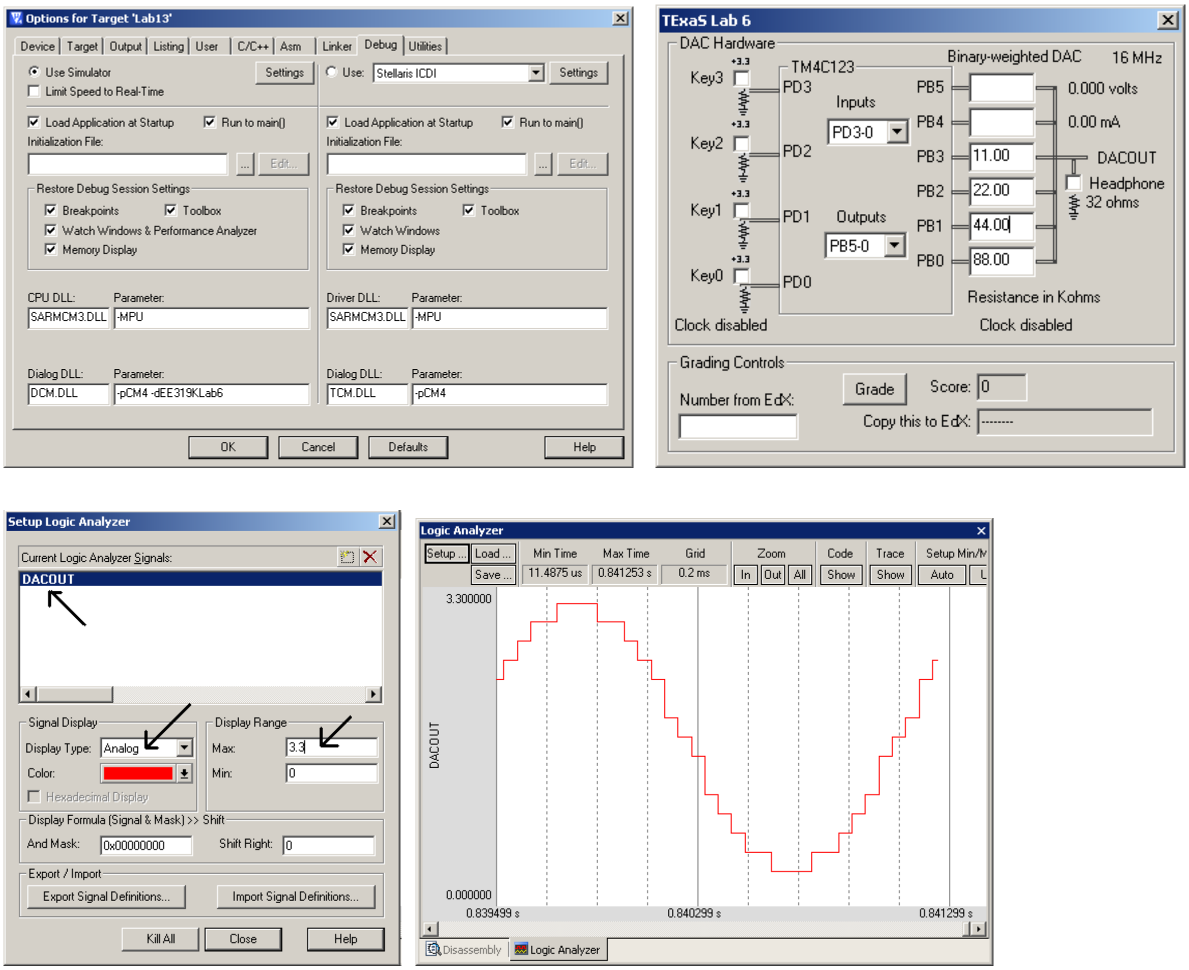


*Figure 6.1b. The screens needed to run Lab 6 in simulation.*

# Part b - Design DAC

Draw the circuit required to interface the binary-weighted DAC to the TM4C123. This DAC should operate using a simple resistor network. A 4-bit binary-weighted DAC uses resistors in a 1/2/4/8 resistance ratio. Select values in the 1.5 kΩ to 240 kΩ range. For example, you could use 1.5 kΩ, 3 kΩ, 6 kΩ, and 12 kΩ. Notice that you could create double or half resistance values by placing identical resistors in series or parallel, respectively. It is a good idea to email a pdf file of your design to your TA and have him/her verify your design before you build it. You can solder 24 gauge solid wires to the audio jack to simplify connecting your circuit to the headphones. Plug your headphones into your audio jack and use your ohmmeter to determine which two wires to connect. You have the option of connecting just the left, just the right, or both channels of the headphones. Draw interface circuits for three switches, which will be used for the piano keyboard.

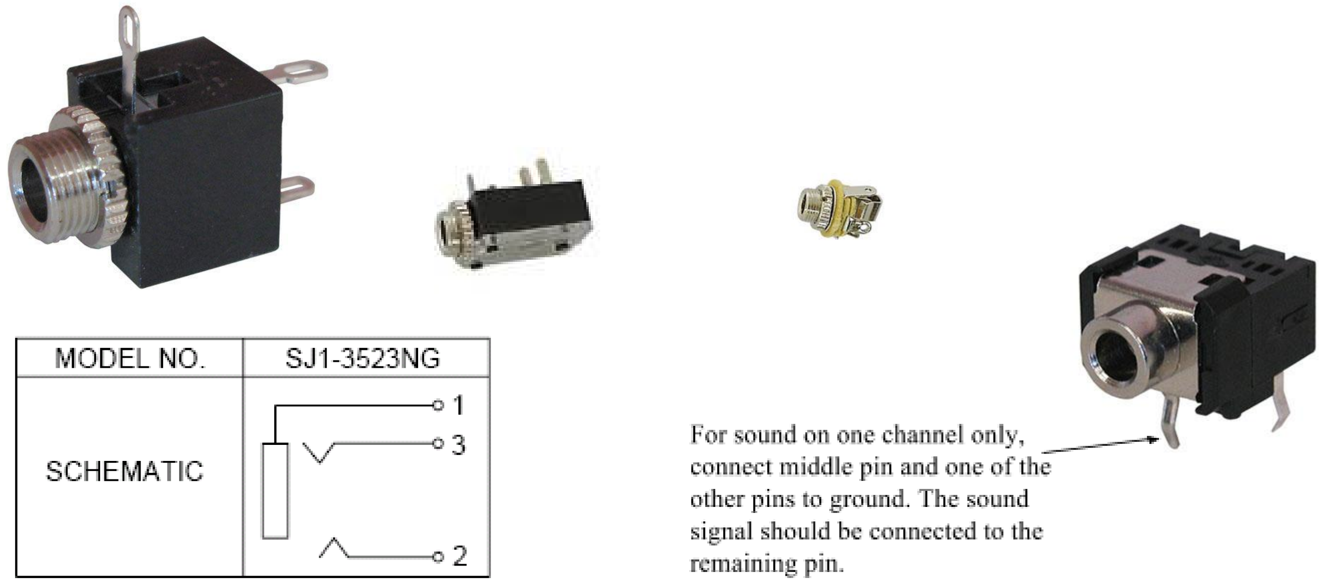


*Figure 6.2. A mono jack has three connections (for one channel connect pin 1 to ground and pin 3 to the DAC output).*

## Part c - Write DAC Driver

Implement a device driver for your binary weighted DAC. Include at least two functions that implement the DAC interface. For example, you could implement the function **`DAC_Init`** to initialize the DAC, and the function **`DAC_Out`** to send new data value to the DAC. Place all code that accesses the DAC in a separate **DAC.c** file. Add a **DAC.h** header file with the prototypes for public functions. Describe how to use the functions defined by the module in the comments of the header file.

## Part d - Test DAC

Measure the range, precision, resolution, and accuracy of your DAC's analog output by connecting it to your voltmeter or oscilloscope. This requires you to write a simple main program to test the DAC, similar in style as Program 6.1. You are free to debug this system however you wish, but you must debug The DAC module separately. You should initially debug your software in the simulator (Figure 6.3). You can single step this program, comparing digital **`Data`** to the analog voltage at the $V_{out}$ without the speaker attached (i.e., left side of Figure 6.1).

```
#include "DAC.h"

int main(void){

  unsigned long Data; // 0 to 15 DAC output

  PLL_Init();         // like Program 4.6 in the book, 80 MHz

  DAC_Init();

  for(;;) {

    DAC_Out(Data);

    Data = 0x0F&(Data+1); // 0,1,2...,14,15,0,1,2,...
```

```
    }
}
```

*Program 6.1. A simple program that outputs all DAC values in sequence.*

Using Ohm's Law and fact that the digital output voltages will be approximately 0 and 3.3 V, make a table of the theoretical DAC voltage and as a function of digital value (without the speaker attached). Calculate resolution, range, precision and accuracy. Complete Table 6.3 and attach it as a deliverable (alternatively you could enter this data into **DACdata.xls**).

| Bit3 bit2 bit1 bit0 | Theoretical DAC voltage | Measured DAC voltage |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |

| | | |
|---|---|---|
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | | |

*Table 6.3. Static performance evaluation of the DAC (if you implement a 6-bit DAC, then make a table with 16 measurements: 0,1,7,8,15,16,17,18,31,32,33,47,48,49,62,63).*

Figure 6.3 shows the static testing using program 6.1 running in the simulator.

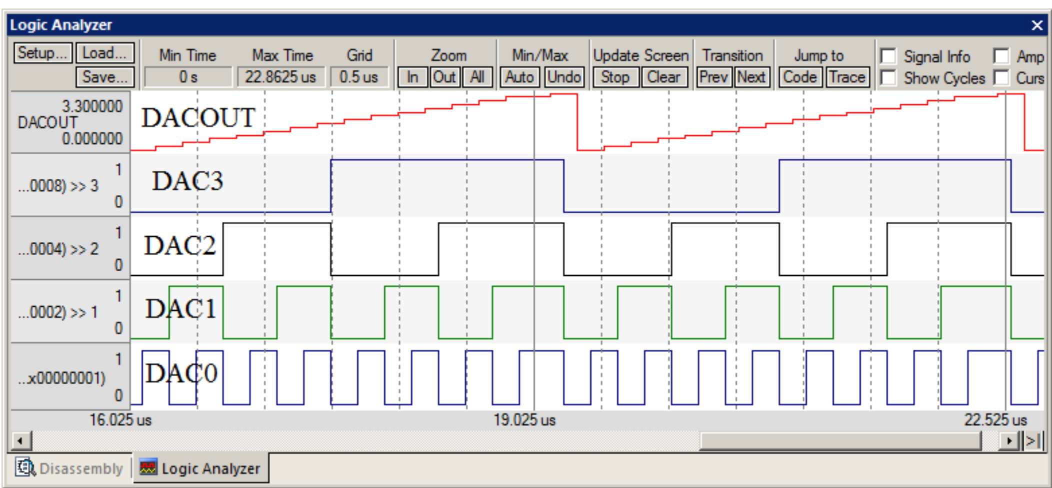


*Figure 6.3. A screenshot in simulation mode showing the static testing of the DAC. (the current version has a bug requiring you place a break point at the line that you output to the DAC so the scope window is updated)*

## Part e - Write Piano Driver

Design and write the piano keyboard device driver software. These routines facilitate the use of the at least three piano keys. Include at least two functions that implement the piano keyboard interface. For example, you could implement the function **`Piano_Init`** to initialize the switch inputs, and the function **`Piano_In`** that returns a logical key code for the pattern of switches that are pressed. Place all code that directly accesses the switches in a separate **Piano.c** code file. Add a **Piano.h** header file with the prototypes for public functions. Add comments that describe what it does in the **Piano.h** file and how it works in the **Piano.c** file.

## Part f - Write Synthesizer

Design and write the sound device driver software. The input to the sound driver is the pitch of the note to play. SysTick interrupts will be used to set the time in between outputs to the DAC. Add minimally intrusive debugging instruments to allow you to visualize when interrupts are being processed. Include at least two functions that implement the sound output. For example, you could implement the function **`Sound_Init`** to initialize the data structures, calls `DAC_Init`, and initializes the SysTick interrupt. You could implement a function **`Sound_Play(note)`** that starts sound output at the specified pitch. Place all code that implements the waveform generation in a separate **Sound.c** code file. Add a **Sound.h** header file with the prototypes for public functions. Add comments that describe what it does in the **Sound.h** file and how it works in the **Sound.c** file.

When you wish to play a new note you could write to **`NVIC_ST_RELOAD_R`**, changing the interrupt period, without initializing SysTick.

One approach to debugging is to attempt to create a sine wave with a constant frequency as shown in Figures 6.4 and 6.5. Just run the SysTick periodic interrupts and output one DAC value each interrupt. The toggling digital line shows you the interrupts are happening, and the top sine wave shows your table and DAC output are working.

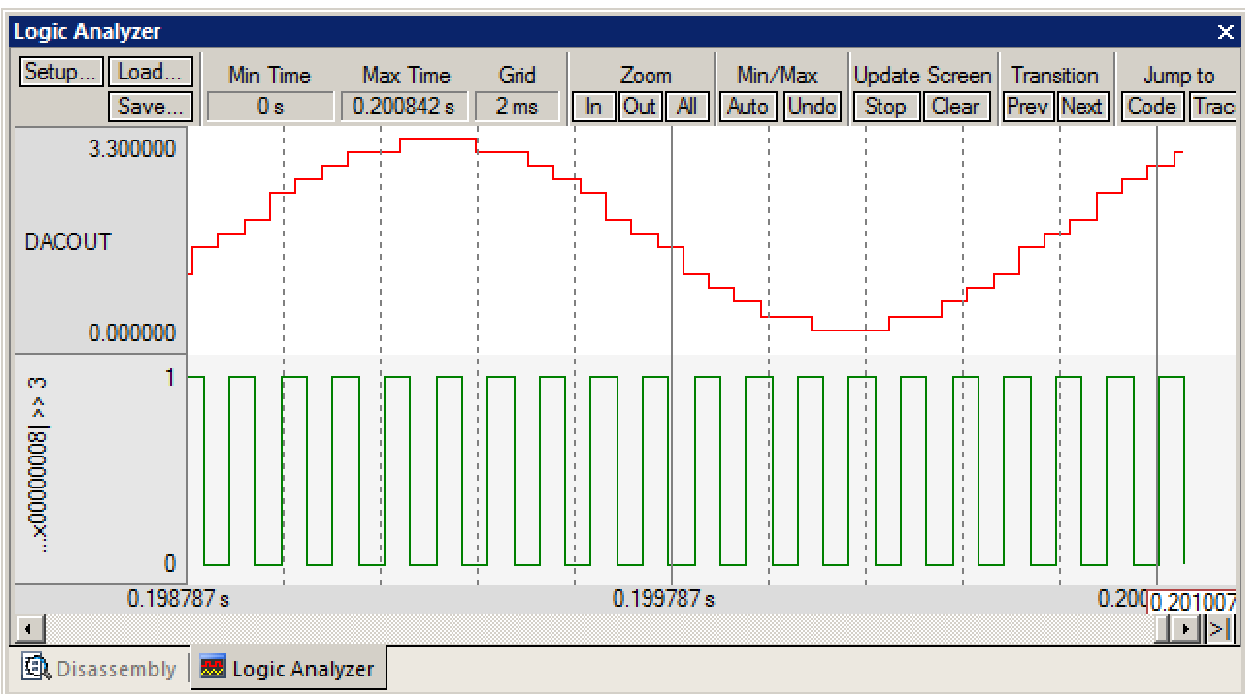


*Figure 6.4. The 4-bit DAC with a 32-element table is used to create a sine wave in simulation. The top trace is the DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt. (the current version has a bug requiring you place a break point at the line that you output to the DAC so the scope window is updated)*

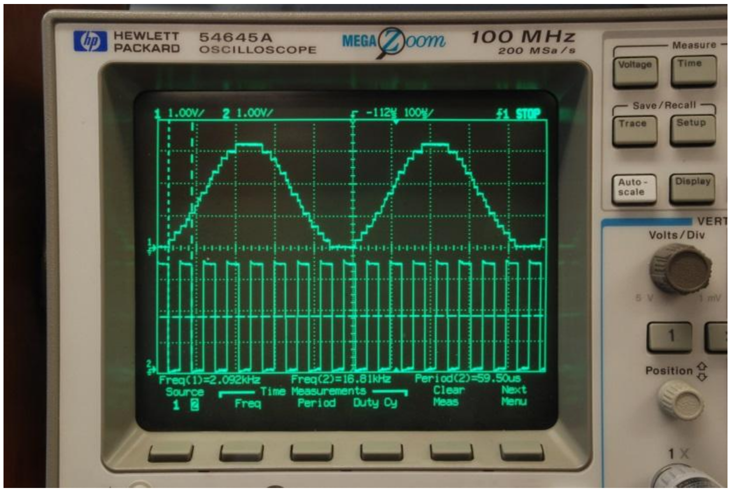


*Figure 6.5. The 4-bit DAC with a 32-element table is used to create a sine wave. In this system, the software was attempting to create 2093 Hz, and the measured frequency was 2092 Hz. The top trace is the DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.*

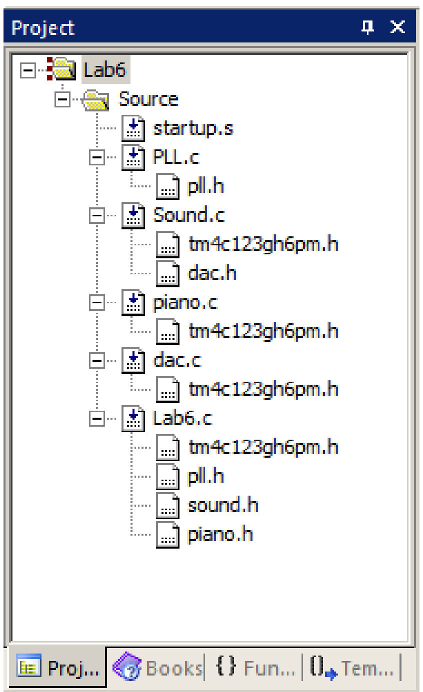


*Figure 6.1 An example project source tree with all header files listed where they are included*

## Part g - Write Main Program

Write a main program to implement the three-key piano. Make a heartbeat connected to an LED so you can see that your program is running. Document clearly the operation of the routines. Figure 6.1 shows what a lab 6 project might look like at this step in the process. Figure 6.6 shows a possible data flow graph of the music player. Debug the system first in the simulator then on the real TM4C123 with an oscilloscope. Take a photograph of the scope traces (like Figure 6.10) to capture the waveform generated by your digital piano. When no buttons are pressed, the output will be quiet. When switch 1 is pressed, output a sine wave at one frequency. When switch 2 is pressed, output a sine wave at a second frequency. When switch 3 is pressed, output a sine wave at a third frequency. Only one button will be pressed at a time. The sound lasts until the button is released.

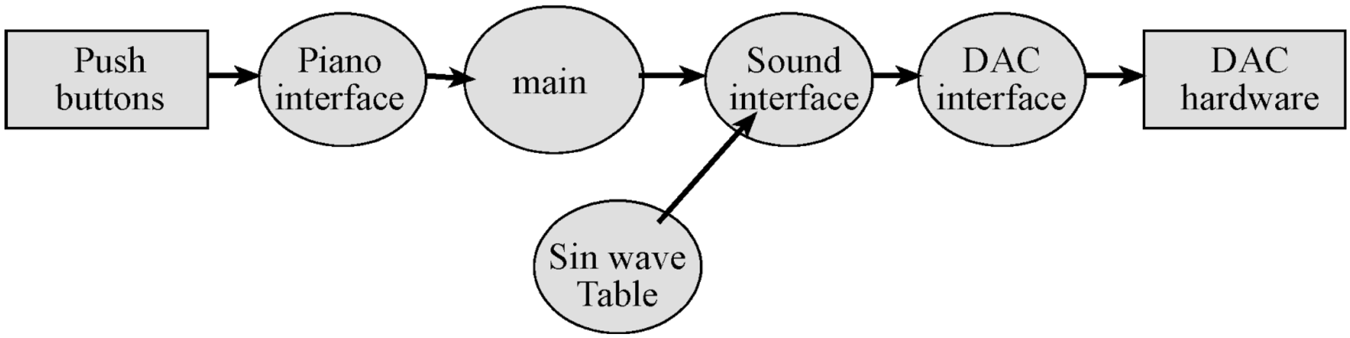


*Figure 6.6. Data flows from the memory and the switches to the speaker.*

Figure 6.7 shows a possible call graph of the Lab 6 system. Figure 6.8 is one possible way to implement the extra credit. Dividing the system into modules allows for concurrent development and eases the reuse of code.

r

*Figure 6.7. A call graph showing the three modules used by the regular part of Lab 6. Notice the SysTick hardware calls the SysTick ISR.*

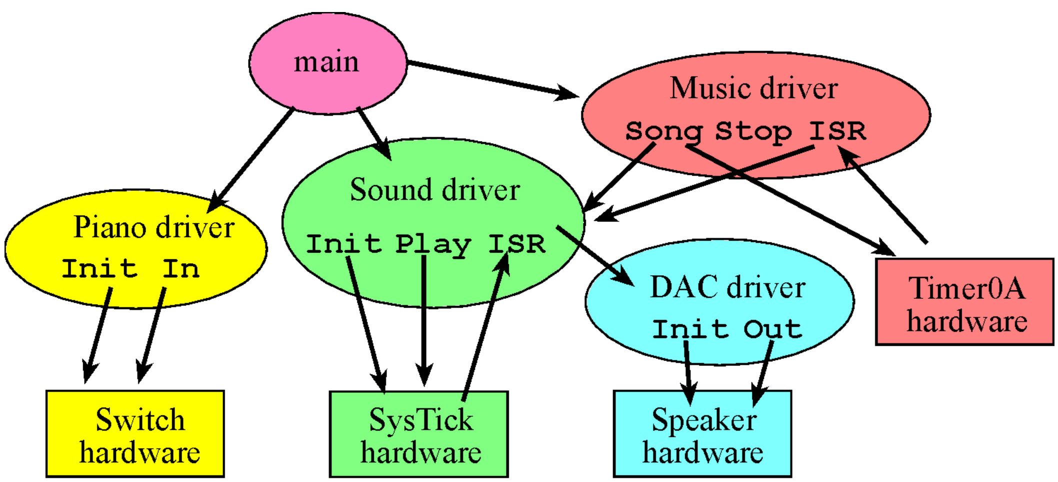


*Figure 6.8. A call graph showing the three modules used by the extra credit part of Lab 6. Notice there are two interrupts.*

## Part h - Add Debug Monitor

Add a debugging monitor to your interrupt service routine. I.e., set a digital output pin high at the start of the ISR and clear it low at the end of the ISR. The scopes in the lab should have probes attached. Connect one channel of an oscilloscope to a digital output showing when the interrupts are occurring and a second channel to the DAC output (without the headphones). Set the oscilloscope to trigger on the digital output occurring in the ISR. Observe the relationship between software and hardware.

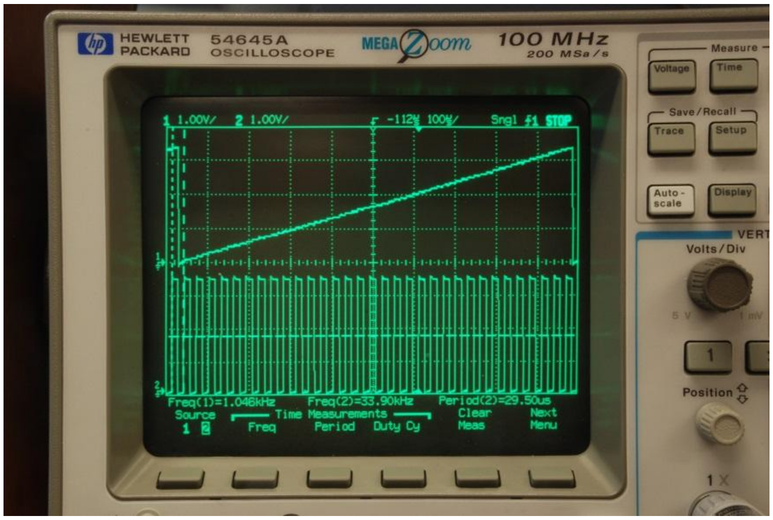


*Figure 6.9. Photograph of the DAC output and debugging monitor. This system uses a 6-bit DAC and a 64-element table. Notice the DAC is monotonic (each increment of the digital input causes the DAC output to go up. (You only need 4 bits). The top trace is the 6-bit DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.*

## Demonstration

**(both partners must be present, and demonstration grades for partners may be different)**

You should be able to demonstrate the three notes. Be prepared to explain how your software works. You should be prepared to discuss alternative approaches and be able to justify your solution. The TA may look at your data and expect you to understand how the data was collected and how DAC works. In particular, you should be able to design a DAC with 5 to 10 bits. What is the range, resolution and precision? You will tell the TA what frequency you are trying to generate, and they may check the accuracy with a frequency meter or scope. TAs may ask you what frequency it is supposed to be, and then ask you to prove it using calculations. Just having three different sounding waves is not enough, you must demonstrate the frequency is proper and it is a sinewave (at least as good as you can get with a 4-bit DAC). You will be asked to attach your DAC output to the scope (part g). Many students come for their checkout with systems that did not operate properly. You may be asked SysTick interrupt and DAC questions. If the desired frequency is **f**, and there are **n** samples in the sine wave table, what SysTick interrupt period would you use?

This lab mentions 32 samples per cycle. Increasing the DAC output rate and the number of points in the table is one way of smoothing out the “steps” that in the DAC output waveform. If we double the number of samples from 32 to 64 to 128 and so on, keeping the DAC precision at 4-bit, will we keep getting a corresponding increase in quality of the DAC output waveform?

As you increase the number of bits in the DAC you expect an increase in the quality of the output waveform. If we increase the number of bits in the DAC from 4 to 6 to 8 and so on, keeping the number of points in the table fixed at 32, will we keep getting a corresponding increase in quality of the DAC output waveform?

## Deliverables

**(Items 2, 3, 4, 5, and 6 are one pdf file uploaded to SVN, have this file open during demo.)**

1. Lab 6 grading sheet. You can print it yourself or pick up a copy in lab. You fill out the information at the top.

***Create one pdf file with the following for electronic submission (**UTEID1_UTEID2.pdf***)***

2. Circuit diagram showing the DAC and any other hardware used in this lab, PCB Artist
3. Software Design
    a. Draw pictures of the data structures used to store the sound data
    b. If you organized the system different than Figure 6.6 and 6.7, then draw its data flow and call graphs
4. A picture of the dual scope like Figures 6.5 or 6.11. You may also show screenshots of TexasDisplay or Audacity.
5. Measurement Data
    a. Show the theoretical response of DAC voltage versus digital value (part c, Table 6.3)
    b. Show the experimental response of DAC voltage versus digital value (part c, Table 6.3)
    c. Calculate resolution, range, precision and accuracy
6. Brief, one sentence answers to the following questions
    a. When does the interrupt trigger occur?
    b. In which file is the interrupt vector?
    c. List the steps that occur after the trigger occurs and before the processor executes the handler.
    d. It looks like **`BX LR`** instruction simply moves LR into PC, how does this return from interrupt?
7. Optional Feedback : [http://goo.gl/forms/rBsP9NTxSy](http://goo.gl/forms/rBsP9NTxSy)

All source files that you have changed or added (like main.c DAC.c DAC.h) should be committed to SVN. Please do not commit other file types.

## Extra Credit

Extend the system so that is plays your favorite song (a sequence of notes, set at a specific tempo and includes an envelope). The song should contain at least 5 different pitches and at least 20 notes. But, what really matters is the organization is well-done using appropriate data structures that is easy to understand and easy to adapt for other songs.

This extra credit provides for up to 20 additional points, allowing for a score of 120 out of 100 for this lab. Extra credit is for playing a song. To earn the credit you must use more than one interrupt. A fast SysTick ISR outputs the sinewave to the DAC (Figures 6.4, 6.11). The rate of this interrupt is set to specify the frequency (pitch) of the sound. A second slow Timer ISR occurs at the tempo of the music. For example, if the song has just quarter notes at 120, then this interrupt occurs every 500 ms. If the song has eight notes, quarter notes and half notes, then this interrupt occurs at 250, 500, 1000 ms respectively. During this second ISR, the frequency of the first ISR is modified according to the note that is to be played next. To earn your credit, there must be at least 3 distinct notes in the song.

Compressed data occupies less storage, but requires runtime calculations to decompress. On the other hand, a complete list of points will be simpler to process, but requires more storage than is available on the TM4C123.

Although you will be playing only one song, the song data itself will be stored in the main program, and the device driver will perform all the I/O and interrupts to make it happen. You will need public functions **`Song`** and **`Stop`**. The **`Song`** function has an input parameter that defines the song to play.

If you complete the extra credit (with input switches that can be used to start and stop), then the three-key piano functionality still must be completed. In other words, the extra credit part is in addition to the regular part. You could interface more switches (which you can get from checkout) or you could use the on-board switches to activate the **`Song`** and **`Stop`** operations

## Some web links about music

Simple Music Theory http://library.thinkquest.org/15413/theory/theory.htm

Free sheet music http://www.8notes.com/piano/

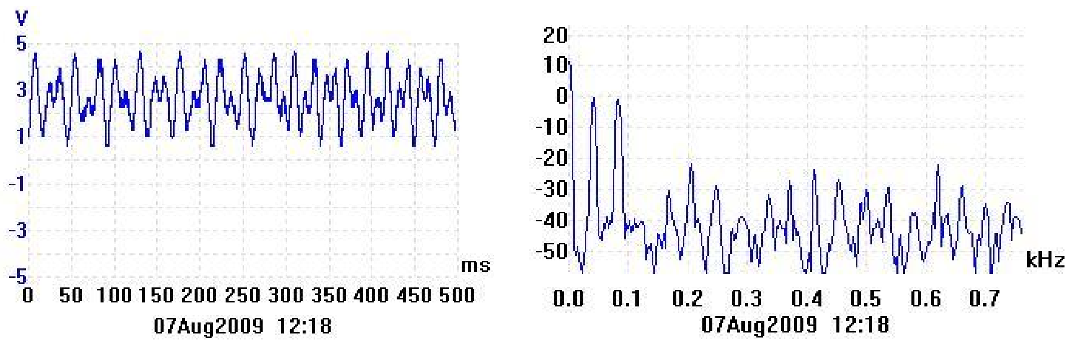


*Figure 6.10. A song being played with a harmony and melody (see two peaks in the spectrum).*

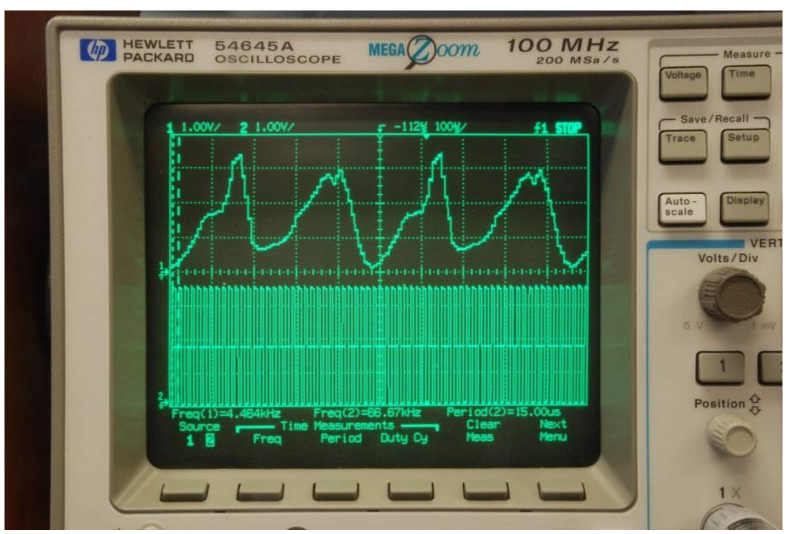


*Figure 6.11. This waveform sounds like a horn (6-bit DAC, 64-element table). The top trace is the 6-bit DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.*

## Notes

There should be no calls to **`DAC_Out`** in the main loop. There should be zero or one calls to **`DAC_Out`** each execution of the ISR. Some students will discover they can make sounds without interrupts, by simply using a software delay like Labs 4, and 5. This approach is not allowed. Other students will discover they can create sounds by calling **`DAC_Out`** in the main loop. This approach is also not allowed. 20 points will be deducted if zero or one calls to **`DAC_Out`** do not occur in the ISR. These other approaches are not allowed because we will be employing sounds in Lab 10, and we will need the sound to completely run within the ISR. The Lab 10 main program will start and stop sounds, but the ISR will do all the **`DAC_Out`** calls.

## FAQ: Frequently Asked Questions

1. **The deliverables say to have pictures of the data structures. Does it mean data flow graphs/flow charts?**

   No, it's talking about diagrams of the structs or arrays you define in C. You should include all the members of your struct in the diagram.

2. **What is the purpose of the SysTick_Handler? Is it called automatically every time the CURRENT_R reaches 0?**

   Yes, as long as you initialized SysTick and enabled the SysTick interrupt.

3. **What is the purpose of the Piano_Init function and the Sound_Init functions ? What does the implementation for these functions do ?**

   Those functions will essentially setup your inputs and outputs. For example, Piano_Init will configure the three pins your external switches are connected to as inputs and enable them. If you have any data structures or arrays, you can also set their initial values in your init functions if they are not constant or too large to write out by hand (int x[2] = {1,2} is easy, int x[1000] = {1,2,3....} would take forever and can be put into a loop during initialization).

4. **How do I export variables in C? (C file calling another C file) More specifically, I am using bit specific addressing and want my dac.c file to be able to use the variable I defined in lab6.c**

   I'm guessing the variable you want to share is a global variable? In that case, you'll want to use the "extern" declaration. You said you already have a definition for your variable in lab6.c. Keep that there as it is, but in dac.c, declare the same variable again, except this time with the "extern" keyword in front of it. This tells the compiler that when it's compiling dac.c, it should not allocate space for the variable, because the space has already been allocated in another C file. To share a #define constant between two files, you can declare it in a header file, and then #include that header file in both C files. Remember that #define is a preprocessor directive, so the constants get substituted before even compiling starts. This means that no space in the memory is needed for these constants. Global variables are different, because they are stored in memory, and can change at runtime.

5. **What are the difference between the Sound_Play() and DAC_Out() functions?**

   DAC_Out() interacts with the 4 DAC pins to simply output the 4-bit "int." Sound_Play(), on the other hand, will actually play a note. Playing a note involves moving the DAC_Out() values according to a sinusoidal curve. Your DAC_Out() and Sound_Play() subroutines don't necessarily return anything to your main code, but rather perform tasks on their own based in inputs you give them. For DAC_Out() think about what you need to pass to the portB pins (high or low) and how it changes with time. For Sound_Play() think about what you need to change about systick to create the note you want and how it changes with the keys pressed.

6. **We are currently in the software debugging process and for some reason on the peripheral it is not displaying the PB5-PB0 changing as we traverse the sin wave and write new**

**values to PortB. We have initialized portb and set the 6 bits to input. What would you suggest we do to figure out the problem?**

Keep in mind that you are outputting voltage from portB to your DAC, your headphones are receiving the signal put out by your controller. The peripheral will not show port B bits, but in the blank spaces it requires you to input your resistance values (e.g. write "1.5" in PB5 if that is the resistor you are outputting to (1.5k) ). The peripheral will now show you the voltage change that is received at the audio jack.

7. **Why does Sound_Play(uint32_t period) have a parameter? What exactly is Sound_Play() supposed to do?**

   As the lab manual says, "Sound_Play(note) starts sound output at the specified pitch." The parameter is to specify the period of the note you want to play, allowing you to play different types of notes. If the period in order to play the note A was *x*, then Sound_Play(*x*) should play that note.

8. *** FAIL: At least 4 DR8R bits in PORTB should be high* What's the DR8R register?**

   That is the 8-mA drive control register. It is to make sure that you are driving only 8mA out of the pins you set high. Guess it is a good thing to do to ensure you don't fry your board.

9. **Is it necessary that we use all of the different subroutines that have been defined in the starter file? I know it's for the sake of abstraction, so that anyone should be able to look at the main program and understand what the inputs should mean, and what the output should be as a result. However, it's rather difficult to keep track of where variables have been defined (e.g., globally, locally within the function, or locally elsewhere). It would be significantly easier to have only a few subroutines rather than 6 or so different files that use much of the same variables.**

   The lab manual specifies separating each component into a separate file with at least two functions for piano and DAC, so I would stick close to that; however, you can change around the function names/implementations a bit as long as it makes sense to you. It can definitely get confusing, especially the first time, but that's why it's good to start practicing modular software development now.

10. **Will the SysTick timer begin counting down and rolling-over from the moment it is initialized?**

    When you set the enable bits within NVIC_ST_CTRL_R, the counter will begin to decrement and roll over as needed. However if you have disabled interrupts (EnableInterrupts(), DisableInterrupts()), then the SysTick_Handler will not be called until you re-enable the interrupts.

11. **My partner and I are having some issues with getting the simulator (particularly the DAC simulation to update when we change any of the relevant data for each of the ports. When I look at the status of the data registers in the system viewer, the values are accurate, but the simulator doesn't reflect those values. The simulator for Port F, works, however.**

    Don't forget to add the resistor values for your DAC in the simulator window (blank spots for PB0-5). The voltage should change after that. When you plug in headphones (the check box), you should some voltage drop and current flow.

12. **It seems like the .h files for the Sound, Piano, and DAC modules are already provided to us in the Lab6 folder. Do we need to add anything to the .h files besides comments as to what each function does and its hardware connections?**

No. The .h files contains only the declaration of your functions, and the .c files contains the definition (the actual code for the function). All you need to do is complete the function definition in your .c files. Although, if you'd like to add your own functions and variables to your program, you will need to include declarations for these in the header files so that they can be accessed from other files.

13. **In the header files, I keep getting an error when using uint32_t instead of "unsigned long." Why is this happening/is it okay to declare a variable as a uint32_t in the .c file and unsigned long in the .h file?**

    So the type uint32_t is not one of the built-in C variable types. It is one that is defined in "stdint.h", so if you wanted to use uint32_t, you need to add "#include <stdint.h>" at the beginning of your code. However, using unsigned long also suffices. A long is a built-in C type which is why it works fine.

14. **What is meant by "the function Sound_Init [initializes] the data structures?" Is this just referring to the Sine table?**

    The sound_init function initializes the systic counter by enabling interrupt bit. By data structure, we are referring to the sine array and you should initialize the index of that array to zero. You should figure out in which function you will increment that index.

15. **For the heartbeat in the main program:Can we just use one of the LED's on the microcontroller, or do we have to interface an external LED?**

    You can use the onboard LED.

16. **What exactly is the implementation for the DAC_Out function supposed to do? Doesn’t the DAC circuit convert the signals from digital to analog anyway ?**

    DAC_Out should take the input data passed as argument and write it to PORTB. Then your circuit converts the 4 (or 6) digital outputs at the pins of PORTB to an analog voltage.

17. **Is this Systick periodic interrupts or edge triggered interrupts?**

    The Systick interrupt is an example of a periodic interrupt. An example of an edge triggered would be handling a button press on a GPIO pin.

18. **I'm trying to understand what resolution means, I understand that precision is the number of levels or entries, and the more we have the smoother the wave is. Range is the max/min of the voltages, so if we divide precision by the range, what exactly are we getting?**

    Range=precision * resolution. The **resolution** is the smallest change in value that is significant. You can get more from the e-book.

19. **So my hardware doesn't seem to work at all but I gone through it multiple times and it seems to be wired correctly with the correct resistor values. I'm assuming the code works since it's working in the simulator. Any ideas to why my lab isn't working?**

    That's what most people have trouble with. Use multimeter and oscilloscope to see if the hardware is receiving/outputting the correct things. There's no really short cut for this.

20.