

Lab 6g Music Player and Audio Amp

This laboratory assignment accompanies the book, *Embedded Microcomputer Systems: Real Time Interfacing*, Second edition, by Jonathan W. Valvano, published by Thomson, copyright © 2006.

- Goals**
- Design a data structure to represent music,
 - Develop a system to play sounds.
- Review**
- Valvano Section 4.15.3 on periodic interrupts
 - Valvano Section 11.4.1 on DAC parameters
 - Valvano Section 11.4.6 on waveform generation.
- Starter files**
- none

Background

Continuing the design started in the previous lab, the next step is to convert the DAC analog output to speaker current using a current-amplifying audio amplifier. It doesn't matter what range the DAC is, as long as there is an approximately linear relationship between the digital data and the speaker current. To do this you will have to run the amplifier in its linear range. The performance score of this lab is not based on loudness, but sound quality. The quality of the music will depend on both hardware and software factors. The precision of the DAC, the linearity of the audio amp, the frequency response of the audio amp and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the complexity of the stored music data. Consider using the MC34119 when designing the audio amp.

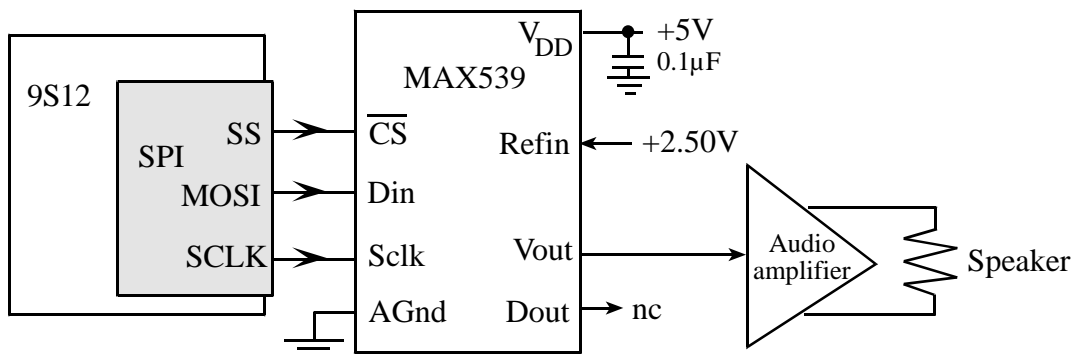


Figure 6.1. DAC allows the software to create music.

You will design a data structure to store the sound waveform. You are free to design your own format, as long as it uses a formal data structure (i.e., **struct**). Compressed data occupies less storage, but requires runtime calculation. On the other hand, a complete list of points will be simpler to process, but requires more storage than is available on the 9S12. Next, you will organize the music software into a device driver. 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 **Rewind**, **Play** and **Stop**, which perform operations like a cassette tape player. The **Play** function has an input parameter that defines the song to play. A background thread implemented with output compare will fetch data out of your music structure and send them to the DAC. The last step is to write a main program that inputs from three binary switches and performs the three public functions.

If you output a sequence of numbers to the DAC that form a sine-wave, then you will hear a continuous tone on the speaker, as shown in Figures 5.2 and 6.2. The **loudness** of the tone is determined by the amplitude of the wave. The **pitch** is defined as the frequency of the wave. Table 6.1 contains frequency values for the notes in one octave.

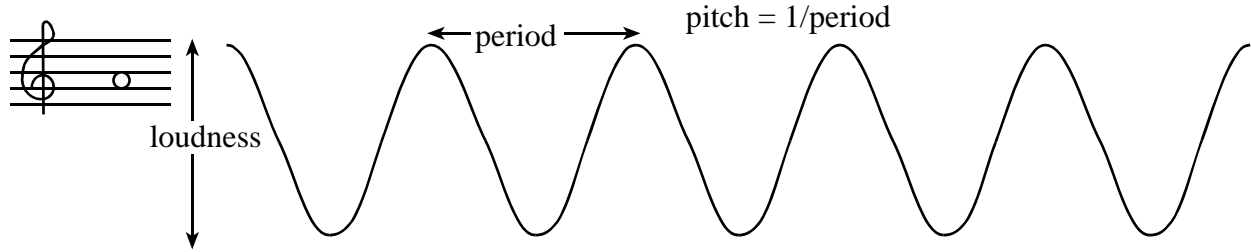


Figure 6.2. A sine-wave generates a pure tone.

Note	frequency
C	523 Hz
B	494 Hz
B ^b	466 Hz
A	440 Hz
A ^b	415 Hz
G	392 Hz
G ^b	370 Hz
F	349 Hz
E	330 Hz
E ^b	311 Hz
D	294 Hz
D ^b	277 Hz
C	262 Hz

Table 6.1. Fundamental frequencies of standard musical notes. The frequency for 'A' is exact.

The frequency of each note can be calculated by multiplying the previous frequency by $\sqrt[12]{2}$. You can use this method to determine the frequencies of additional notes above and below the ones in Table 6.1. There are twelve notes in an octave, therefore moving up one octave doubles the frequency. Figure 6.3 illustrates the concept of **instrument**. You can define the type of sound by the shape of the voltage versus time waveform. Brass instruments have a very large first harmonic frequency.



Figure 6.3. A waveform shape that generates a trumpet sound.

The **tempo** of the music defines the speed of the song. In 2/4 3/4 or 4/4 music, a **beat** is defined as a quarter note. A moderate tempo is 120 beats/min, which means a quarter-note has a duration of 1/2 second. A sequence of notes should be separated by pauses (silences) so that each note is heard separately. The **envelope** of the note defines the amplitude versus time. A very simple envelope is illustrated in Figure 6.4. The 9S12 has plenty of processing power to create these types of waves.

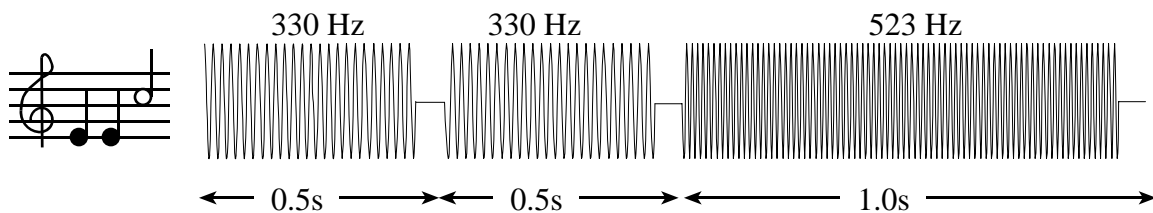


Figure 6.4. You can control the amplitude, frequency and duration of each note (not drawn to scale).

The smooth-shaped envelope, as illustrated in Figure 6.5, causes a less staccato and more melodic sound. This type of sound generation may be difficult to produce in real-time on the 9S12.

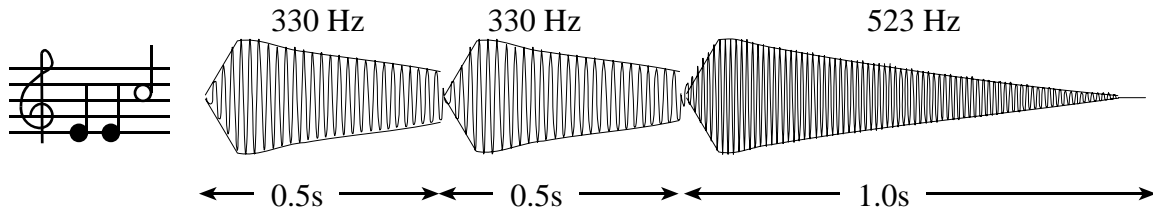


Figure 6.5. The amplitude of a plucked string drops exponentially in time.

A chord is created by playing multiple notes simultaneously. When two piano keys are struck simultaneously both notes are created, and the sounds are mixed arithmetically. You can create the same effect by adding two waves together in software, before sending the wave to the DAC. Figure 6.6 plots the mathematical addition of a 262 Hz (low C) and a 392 Hz sine wave (G), creating a simple chord.

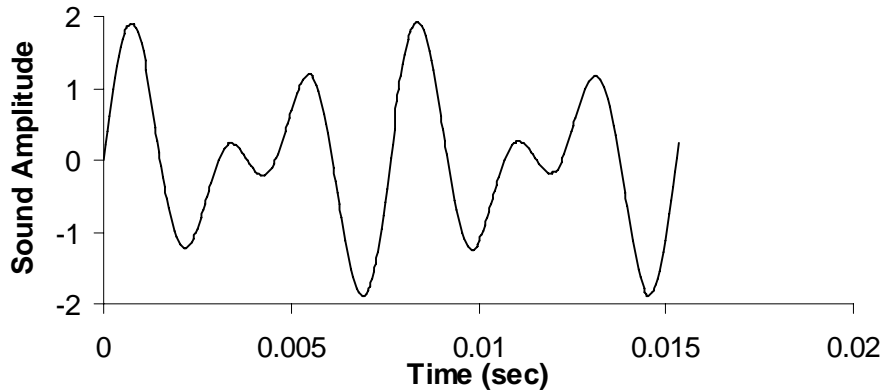


Figure 6.6. A simple chord mixing the notes C and G.

Preparation (do this before your lab period)

1. Design the audio amplifier that runs on the +5V power from the board. Using Ohm's law and the properties of the audio amplifier, make a table of the speaker voltage and current as a function of digital value.
2. Design and write the music device driver software. Create separate **Music.h** and **Music.c** files. Place the data structure format definition in the header file. Add minimally intrusive debugging instruments to allow you to visualize when interrupts are being processed.
3. Write a main program to run the entire system.

A "syntax-error-free" hardcopy listing for the software is required as preparation. The TA will check off your listing at the beginning of the lab period. You are required to do your editing before lab. The debugging will be done during lab. Document clearly the operation of the routines. Figure 6.7 shows the data flow graph of the music player.

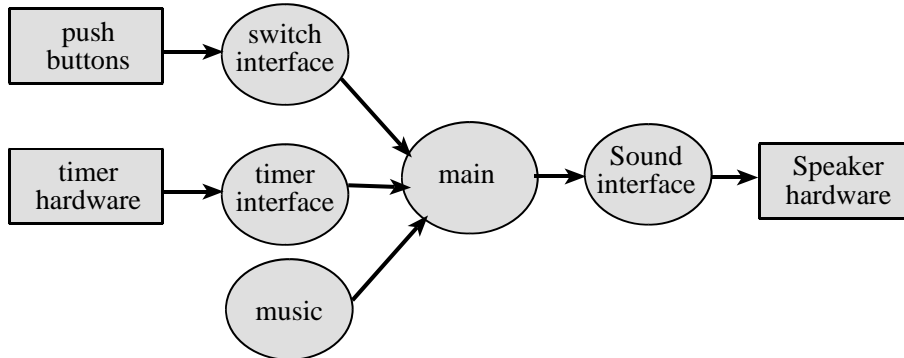


Figure 6.7. Data flows from the memory and the switches to the speaker.

Figure 6.8 shows a possible call graph of the system. Dividing the system into modules allows for concurrent development and eases the reuse of code.

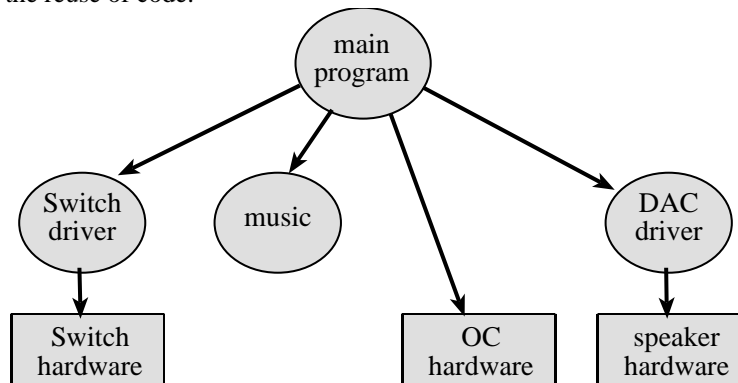


Figure 6.8. A call graph showing the three modules used by the music player.

Procedure (do this during your lab period)

1. Use the simple main programs to debug the SPI/DAC/speaker interface. Experimentally measure the speaker voltage/current versus digital value. Compare the measured data from the predicted data calculated as part of the preparation. Adjust resistance and capacitor values to get an approximately linear relationship between the digital output and the speaker current.
2. Using debugging instruments, measure the maximum time required to execute the periodic interrupt service routine. Adjust the interrupt rate to guarantee no data are lost.
3. Debug the music system.

Deliverables (exact components of the lab report)

- A) Objectives (1/2 page maximum)
- B) Hardware Design
 - Detailed circuit diagram of all hardware attached to the 9S12 (preparation 1)
- C) Software Design (a hardcopy software printout is due at the time of demonstration)
 - Draw pictures of the data structures used to store the sound data
 - If you organized the system different than Figure 6.7 and 6.8, then draw its data flow and call graphs
- D) Measurement Data
 - Show the theoretical response of speaker current versus digital value (preparation 1)
 - Show the experimental response of speaker current versus digital value (procedure 1)
- E) Analysis and Discussion (1/2 page maximum)

Checkout (show this to the TA)

You should be able to demonstrate the three functions **Rewind**, **Play** and **Stop**. You should be prepared to discuss alternative approaches and be able to justify your solution.

A hardcopy printout of your software will be given to your TA, and graded for style at a later time.

Do you want your machine to sound better than everyone else's?

It is possible (but not required) to create multiple sine-waves at the same time. This way, you can play music containing melody and harmony. For example, if you have a 12-bit DAC, then you can implement two 11-bit sine-waves, so that when the two waves are added together overflow is avoided. You will need three interrupts: one for outputting the sine-wave for the melody, one for outputting the sine-wave for the harmony, and a third to interpret the music (updating the frequencies and envelopes for the other two.)