Lab 6 – DAC and C Programming

Objectives

The objectives for the second part of this lab are to:

1. use a DAC to output a desired voltage
2. define function prototypes for internal and external procedures
3. access C8051 SFRs using C statements
4. place and access data in specific memory spaces
5. use C constructs which you may not have used before: pointers, shifting, bit manipulation, etc.
6. implement an interrupt handler in C

Background

The lab this last week and this week both deal with some basic routines for the digital-to-analog converters (DAC0 and DAC1) that we will use in the project. The DACs will be directly driving the output on our MP3 player. You will need the C8051F120 datasheet for the following information:

- Sections 8 and 9 on the DAC and voltage reference
- Section 23 on Timer 2

In order for the sound reproduction to be as accurate as possible, the DACs needs to be updated promptly. We can handle this issue by using Timer 2 to keep the DACs happy. Both DACs can be configured so that the data can be buffered and output only when a timer overflows. You will get an interrupt when the timer overflows, and the handler will then prepare the next byte(s) of data for each channel.

The DACs use two SFRs (**DACxH** and **DACxL**) to hold the data. Each DAC has 12-bit of precision or resolution. This raises some issues you need to consider:

1. Which 12 bits of the 16 bits in **DACxH** and **DACxL** are used?
2. The sound data is only 8 bits. What (if anything) should be done with the other 4 bits?

In addition to loading the DACs, you are also responsible for providing the user some control over the volume and stereo balance. There are a number of ways to handle this, but the approach you will take is to use software to adjust the amplitude of the values that the DAC sees. How to do this is left as an exercise for you; however, you may need one fact. Our sound data is stored as 8-bit unsigned values. Consequently, the mean waveform value is 127 or 128 (0x7F or 0x80). In other words, a silent passage in a song would just be a series of 127/128 bytes.

The primary goal for this lab is to write C code to control the DACs. The pre-lab questions below focus almost exclusively on preparations for the DACs (each DAC is identical except for the output pin and SFR page setting) and accordingly will be worth more points than previous pre-labs.

Pre-lab

Prior to the lab, you and your lab partner must turn in a "pre-lab" which answers the following questions. Your group will need the answers to these questions in order to get this lab working correctly.

1. How are the 12 bits of data stored in the 16 bits of DAC0/DAC1 data SFRs? Where should your 8 bit data sample be written?
2. What are the addresses of these SFRs: DAC0L, DAC1L, TMR2L, RCAP2L, DAC0H, DAC1H, TMR2H and RCAP2H?
3. How can you adjust the volume and balance by manipulating only the
   a) sound data?
   b) DAC settings?
4. Write the logic (pseudo-code, flowchart, etc) for the **dacout** procedure.

Lab work

At the completion of this lab, you need to turn in an assembly module named **dacC.c** with all the procedures listed below. Your C module will need to include a header file which defines the C8051 SFRs and SFR bits. The include file is **c8051F120.h** and the statement you need to use is "*#include <c8051F120.h>*". Speaking of header files, the data types used in the procedure are primarily unsigned values. Some are 8-bit and some are 16-bit. The size of C data types and those used in the the Keil C compiler are not always as clear. To make this more obvious, we will use the following type declarations (you should put these into a file named **types.h** and include using a "*#include*" statement in your **dacC.c** file:

```
typedef unsigned char uint8;

typedef char int8;

typedef unsigned short uint16;

typedef short int16;

typedef unsigned long uint32;

typedef long int32;
```

Your C module will access the three global variables created last week in **daca.asm**. Since you are accessing them from C, you need to declare them with the proper data types:

- *uint16 bytesleft* : this is a 16-bit unsigned integer. It contains the number of bytes which remaming to be sent to the DACs
- *uint8 xdata *bufptr* : this is a 16-bit pointer to 8-bit data in external RAM (XRAM). It contains

the address of the next byte of data to be sent to the DACs. (Note: the pointer is 16 bits because external memory addresses are 16 bits; the *data it is pointing at* is 8 bits).

- *bit dacactive* : this is a bit variable. It signals whether the code is sending data to the DACs

You will need to specify that these variables are external to **dacC.c** using the **extern** (not the assembly language **extrn**) keyword. In addition to these, your module will need to declare some global variables (described below) which allow the interrupt handler **dacout** to communicate with the other procedures.

The procedures you will write for your **dacC.c** module are:

1. **void dac2init(void):** This procedure initializes any global variables used by your C code. It is called after **dacinit** in your assembly module is called. You must do this instead of using an initializer when the variable is declared:

   ```
   uint8 fred = 2;
   ```

   because this type of initialization this will cause problem later in the project.

2. **void dacrate(uint16 rate)**: This procedure sets the Timer 2 reload value based on the frequency parameter. *rate* will be one of 8000, 11025, or 22050.

3. **void dacstereo(uint8 channel)**: This procedure sets the number of channels (1 or 2) for the song in the global variable *isStereo*. If *channel* is zero then the sound is set for mono, otherwise it is set for stereo.

4. **void dacvolume(int8 ud)**: This procedure increases or decreases the volume level and sets global variables *volumeL* and *volumeR* (i.e., the amplitude of the DAC outputs). A positive value increases the volume, a negative value decreases it. If the volume of either channel is at the maximum or minimum, it has no effect. Your code should support at least three volume levels (you may want to include more, which is OK, just don't go nuts).

5. **void dacbalance(int8 lr)**: This procedure changes the balance between the left and right channels. A positive value increases the difference of the left over the right, a negative value increases the difference of the right over the left. It examines the left and right volumes and if the change would set a channel past the minimum or maximum, it has no effect.

6. **void dacout (void) interrupt *n***: As in Lab 4, this interrupt procedure is the most important of all and does most of the work. It uses the external values *bytesleft*, *bufptr,* and *dacactive*. The code only operates if *dacactive* is set. If *bytesleft* is zero, a constant 80H is written to both DACs and the handler returns. Otherwise, it reads data from *bufptr* and writes it to DAC0 and DAC1. Each DAC always receives one byte of data. If there is only channel, both DACs are given the same byte of data from the buffer. If there are two channels, DAC0 gets the first byte and DAC1 the second byte. *bytesleft* is decremented by the number of bytes sent to the DACs. Obviously, for two channels the data in the buffer is consumed twice a quickly. If *bytesleft* then becomes 0, *dacactive* is cleared. Your code must work as quickly and efficiently as possible. Your goal is to execute the minimum number of instructions to do you work.

   Note: the "***n***" in the **dacout** procedure declaration above refers to the interrupt priority number for Timer 2. You need to fill this value yourself using the priority number from Table 11.4.

A hint: the Keil compiler has a **sfr16** datatype specifically for situations like timers and DACs, where a pair of SFRs are combined to hold 16-bit number. It works *only* if when (1) the pair of SFR are in consecutive memory addresses and (2) the SFR which is the LSB has the lower address (i.e., is little endian). You can then use a declaration like this:

```
sfr16 T0 = 0x8a;    // access TL0 and TH0 as a 16-bit value
```

These two conditions are true for Timer 0 and Timer 2; you should verify if it is true for **RCAP2**, **DAC0** and **DAC1**.

In addition to the **dacC.c** file, you need to create a header file named **dac.h**. All this file contains are the prototypes for each of the procedures in **daca.asm** and **dacC.c** except **dacout**, and any constants which would be useful by the main program (I'm not saying there are any, just if you think of some). This file will be included in **dacC.c** as well as the main module using a "#include" statement.

Testing for this project will be done by removing **lab5dac.obj** from the list of external OBJ files and instead including your **dacC.c** in the project build. In addition, you need to copy **lab5main.c** to **lab6main.c** and modify to respond to the additional keys as follows:

- "<" or "," key: change the balance to the left
- ">" or "." key: change the balance to the right
- "u" or "U" key: increase the volume
- "d" or "D" key: decrease the volume
- "r" or "R" key: change the rate between 8K, 11.025K and 22.05K
- "c" or "C" key: toggle the channels between mono and stereo. Note that playing the stereo sine waves as mono will not sound very nice. That's OK for now.

You will need to add calls to **dac2init()** (*after* the call to **dacinit()**), and possibly **dacstereo()** and **dacrate()**, to initialize your code before the main *while* loop begins.

Your should observe the Timer 2 toggle output and the DAC waveforms as you did in Lab 5; however, be sure to observe the result when you change the frequency, volume and balance.

At the completion of lab, submit your **dacC.c, dac.h** and **lab6main.c** files to Sakai.