CE-2812, Lab Week 5, Play a Tune

1 PURPOSE

The purpose of this lab is to practice structure and create another API.

2 PREREQUISITES

- The Nucleo-F446RE board had been mounted onto the Computer Engineering Development Board.

3 ACTIVITIES

Remember the other timer counters? The piezo speaker? Write an API that will play a song. This API will need to:

- Initialize the appropriate timer/counter to connect to the piezo speaker and configure in a mode to play tones.
- Create a structure that has at least two elements – one element to represent the frequency of a note (could be frequency, could be a character 'a','b','c' with another element for octave, etc, could be timer counts for that note) and one element to represent duration. Be sure to use our "best practice" of creating a typedef for the struct.
- Create an instance of an array of your structure with values included for your favorite song and one other (**at least two songs**). I recommend Imperial March by John Williams for one of them. You can find some help here in terms of representing notes in your program: http://www.instructables.com/id/How-to-easily-play-music-with-buzzer-on-arduino-Th/ although my suggested format is a bit different.
- Create a subroutine (e.g. play_song()) that can be called from your console application that will sequence through the array of structure elements and play the song. It should accept as a parameter a pointer to the array of structures that contains the note information any other information needed to play the song.
    - While it is conceivable that we could write an ISR for the timer/counter that could handle playing the song automatically, we do not need to go quite that far. Rather, setup the timer/counter to toggle the output pin at the correct rate for the particular note, and while it is playing, use your delay_ms routine to allow the note to play the correct duration, then move to the next note, etc.
- Add a menu selection to your console program to play a selected song.
- Your code should be in "API" format as discussed extensively.
- (**Optional**) You should access timer/counter registers via a struct created to match register layout. Be sure to use our "best practice" of creating a typedef for this struct.
- You must use TIM3's output compare functionality to drive the piezo (output compare mode (toggle-on match) or PWM mode).

4 DELIVERABLES

When completed:

1. Submit to Canvas a **single pdf** printout of your completed source code to Canvas. **Include in a comment block at the top of your code a summary of your experience with this project.**
2. Ask to demo your lab to instructor. You can do this via writing your name on the whiteboard.
    a. If you demo during lab in Week 5, you will earn a 10% bonus on this lab.
    b. If you demo during lab in Week 6, you will be eligible for full credit.

- Demos are ONLY accepted during lab periods. If you are unable to demo by the end of lab in Week 6, you lose the 10% of the assignment attributed to the demo (per syllabus).

- Demos must be ready a reasonable amount of time before the end of the lab period. If you write your name on the board at 9:45 and lab ends at 9:50, and there are five names in front of yours, you will be unlikely to complete your demo by the end of lab and hence lose the bonus or demo points.

4.1 GRADING CRITERIA

For full credit, your solution must:

- Use a timer/counter in an appropriate mode.
- Use the best practice for declaring the note structure.
- Implement play_song() to access an array of note structures and iterate the array properly.
- Add features to existing menu system from previous lab.
- Minor errors usually result in a deduction of ~ 3 points (three such errors results in ~ a letter grade reduction)
- Major errors, such as not achieving a requirement, usually result in a deduction of 5 to 10 points.

4.2 Musical Notes

Numerous resources are available that document the frequency of musical notes. Wikipedia has a great article here: https://en.wikipedia.org/wiki/Piano_key_frequencies.

From that article, the lowest frequency note on a standard 88 key piano is A_0 at 27.5 Hz and the highest is C_8 at 4186.009 Hz. That being said, most musical melodies will likely be in octaves 3, 4, and 5, so you could choose to support a smaller range of frequencies if you wish.

Assuming we wish to support the entire range of a standard piano, what is the range of period? This is a little more relevant to us than frequency.

$$P_{27.5}\ \text{Hz} \rightarrow 1/27.5 = 36.36\ \text{ms}$$

$$P_{4186.009} \rightarrow 1/4186.009 = 0.23889\ \text{ms}$$

4.3 Using a Prescaler

Do you need to use the prescaler? It depends. For this platform, the main clock we are running with is 16 MHz, and with no configuration to the RCC or TIM3's prescaler, TIM3 will count at that rate too. TIM3 is a 16-bit counter, so the question is can we accommodate the needed periods with a 16-bit counter running at 16 MHz. If we use PWM mode, we will need the entire period to be within the range of the counter, but if we use output compare mode (toggle-on match), we only need half of the period to be in range.

First off, max period for TIM3 running at 16 MHz:

$$65{,}536\ \text{ticks} * 1\ \text{s}/16{,}000{,}000\ \text{ticks} = 4.096\ \text{ms} \rightarrow \sim 244\ \text{Hz}$$

We could also try to figure out the minimum periods too, but, knowing that each tick is 62.5 ns, the minimum period is well below the 0.239 ms we need.

So back to the max period. We have a potential problem. We cannot create a square wave signal of 27.5 Hz with a 16-bit counter in PWM mode. The lowest frequency we can support is about 244 Hz. This is a little lower than middle C (C_4) so may limit some musical freedom.

What about output compare mode (toggle-on match)? In that case, the 4.096 ms represents half of the period, so we could generate a square wave with a period of 8.192 ms which is about 122 Hz.

Is that sufficient? Maybe. Back to the table of notes. 122 Hz is just below C_3, so you can support octave 3 and above with no prescaler if using output compare mode (toggle-on match).

Incidentally, looking at the Imperial March music (from link in section 3), the lowest note appears to be $A\flat_3$, or 207.65 Hz. Achievable with output compare mode (toggle-on match) but not PWM mode.

If you wish to support the entire range of a piano, you will need to employ the prescaler and reduce the count rate of the counter.

4.4 Calculating Counts

This section will assume output compare mode (toggle-on match) with the counter running at 16 MHz. If you employ PWM mode or a prescaler, adjust the math accordingly.

Essentially, we need to convert the frequency for a note to the number of counts to place in the counter's registers. In output compare mode (toggle-on match), we need the number of counts in a half period, and place that count into both the CCR1 and ARR registers.

The math is pretty simple.

$$P = 1/\text{Note Freq}$$

$$\text{Counts} = P * \text{Clock Rate} = \text{Clock Rate} / \text{Note Freq}$$

We are working with half-period for output compare mode (toggle-on match), so

$$\text{Half-period} = ½\ P, \text{ or } 1/(2*\text{Freq})$$

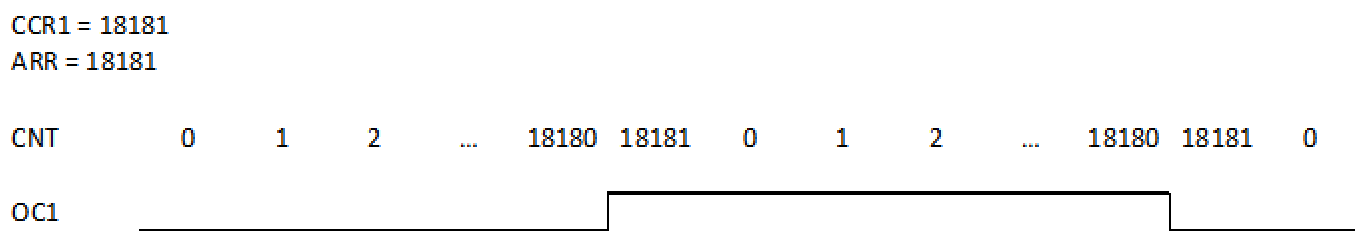
$$\text{Counts}_{\text{Half-period}} = \text{Clock Rate} / (\ 2 * \text{Note Freq}\ )$$

Let’s verify. Try out calculations for middle A, which is 440 Hz.

$$\text{Counts}_{\text{Half-period – Middle A}} = 16{,}000{,}000 / (\ 2 * 440\ ) = 18{,}181.81$$

Of course, we cannot use fractional counts, so this will be truncated to 18,181. Also note that we are fine with integer math. There is no reason to carry intermediate floating point numbers, such as 1/Clock Frequency.

How does this work with the timer? Properly configured, the timer’s counter will start counting (at 0) and count up. When the count reaches the value in CCR1, the associated output pin will toggle. Since we set ARR to the same value, the counter will also reset to 0 and start counting again, and when it reached the value in CCR1, it will toggle again and reset to 0 again. Hence, we get a square wave with a period twice the value in CCR1.



This does reveal a bit of a problem. The toggle occurs when the count reaches 18181, but the counter actually resets on the next tick. So, astute observers would see the half-period is actually 18182 ticks and not the desired 18181.

Let’s check out math. If the half-period was 18181, that would equate to a frequency of:

Period = 18,181 * 2 = 36,362 → multiply by 62.5 ns / tick → 2.272625 ms period → Freq = 1/Period → 440.0198 Hz

Accounting for the discrepancy noted above,

Period = 18,182 * 2 = 36,364 → multiply by 62.5 ns / tick → 2.27275 ms period → Freq = 1/Period → 439.9956 Hz

Due to the nature of the counter, we will not be able to hit every note exactly no matter what. This particular discrepancy can be fixed, however, simply by subtracting 1 from the value you write to CCR1 and ARR.

4.5 Floating Point Math

Do we need to employ floating point math? It depends.

Let’s first consider using the math above, but truncate or round the frequency so that we always have integer division. To see the impact, let’s take a look at count calculations for F_4, which is 349.2282 Hz.

16,000,000 / (2*349) (rounded down) = 22,922 (truncated)

If we stick with floating point division, we get

16,000,000 / (2*349.2282) = 22,907 (truncated)

Different answers, for sure. What do we get from these counts in practice?

$$1 / (22{,}922 * 2 * 62.5 \text{ ns}) = 349.010 \text{ Hz}$$

and

$$1 / (22{,}907 * 2 * 62.5 \text{ ns}) = 349.238 \text{ Hz}$$

So, error - the floating-point result is more accurate, of course. Tolerable error? Tolerable for a piezo speaker? Tolerable for a concert-grade synthesizer?

4.6 Avoiding Floating Point Math

There are a couple of strategies. You can round the frequencies as noted in the previous section which is just fine for this application. However, if you really want to be more accurate, the easiest, but perhaps not the best strategy is to pre-calculate the number of ticks for each note and use that information to represent the note in your program's setup. For example, instead of:

```
#define F4 349.2282
```

use

```
#define F4 22907
```

Of course, this is now very specific to a counter running at 16 MHz and not terribly portable.

But, check this out…

```
#define F_CPU 16000000

#define F4_FREQ 349.2282

#define F4_COUNTS F_CPU/(2*F4_FREQ)
```

What will F4 be? Well, the expression above is replicated everywhere you use F4 in your program which would lead you to believe there will be floating point math. But, something else will happen. The expression expanded 'F_CPU/(2*349.2282)' is a constant expression and will actually be pre-calculated by the compiler and not calculated by your hardware. Need proof? Check this out:

```
#define F_CPU 16000000
#define F4_FREQ 349.2282
#define F4_COUNTS F_CPU/(2*F4_FREQ)

int main(void)
{
    unsigned int counts = F4_COUNTS;
```

compiles to:

```
08000204 <main>:
#define F_CPU 16000000
#define F4_FREQ 349.2282
#define F4_COUNTS F_CPU/(2*F4_FREQ)

int main(void)
{
 8000204:   b480         push    {r7}
 8000206:   b083         sub sp, #12
 8000208:   af00         add r7, sp, #0
    unsigned int counts = F4_COUNTS;
 800020a:   f645 137b    movw    r3, #22907  ; 0x597b
 800020e:   607b         str r3, [r7, #4]

```

You can clearly see that the compiler has done the math for us and our variable initialization is happening with a hardcoded constant. Pretty cool.

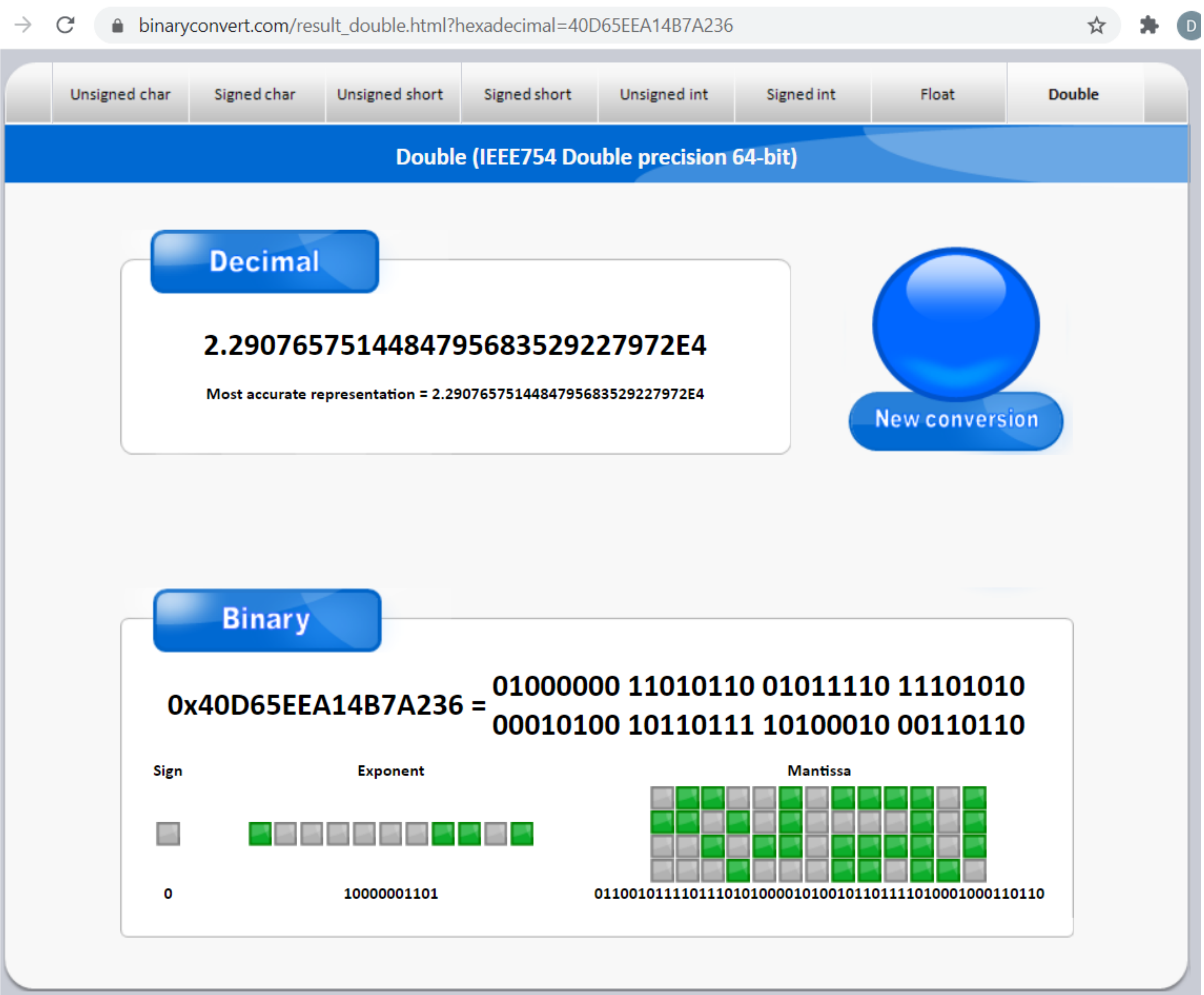
Is the compiler limited to integer expressions? Clearly this expression was evaluated with a floating point divide or based on previous discussion, otherwise it would be 22922 and not 22907. Also, what if we assign to a floating point type?

```
08000208 <main>:
#define F_CPU 16000000
#define F4_FREQ 349.2282
#define F4_COUNTS F_CPU/(2*F4_FREQ)

int main(void)
{
 8000208:   b490         push    {r4, r7}
 800020a:   b084         sub sp, #16
 800020c:   af00         add r7, sp, #0
    unsigned int counts = F4_COUNTS;
 800020e:   f645 137b    movw    r3, #22907  ; 0x597b
 8000212:   60fb         str r3, [r7, #12]
    double dcounts = F4_COUNTS;
 8000214:   a404         add r4, pc, #16 ; (adr r4, 8000228 <main+0x20>)
 8000216:   e9d4 3400    ldrd    r3, r4, [r4]
 800021a:   e9c7 3400    strd    r3, r4, [r7]
```

Where:

```
 8000228:   14b7a236    .word   0x14b7a236
 800022c:   40d65eea    .word   0x40d65eea

```

A double is 8 bytes long, so both words starting at 0x08000228 are part of the number. The entire number is 0x40d65eea14b7a236 taking into account the little endian memory format. What value is this? Well, doubles use IEEE754 encoding. We could study up on that or find a website that decodes it for us. Check this out:



Hex 0x40D65EEA14B7A236 == 2.2907657…E4, aka 2.2907657 x 10^4, aka 22,907.657. This is the exact value of the floating point solution, again, performed by the compiler, not our hardware.

When do we get stuck doing math in hardware? If any part of the expression we are evaluating is **not a constant**, the compiler cannot do the math. Recall that in C, having a **const** variable does not necessarily mean we will get a constant expression.

```
080009e8 <main>:
const unsigned int F_CPU = 16000000;
#define F4_FREQ 349.2282
#define F4_COUNTS F_CPU/(2*F4_FREQ)

int main(void)
{
 80009e8:   b590        push    {r4, r7, lr}
 80009ea:   b083        sub sp, #12
 80009ec:   af00        add r7, sp, #0
    unsigned int counts = F4_COUNTS;
 80009ee:   4b0c        ldr r3, [pc, #48]   ; (8000a20 <main+0x38>)
 80009f0:   4618        mov r0, r3
 80009f2:   f7ff fd4b   bl  800048c <__aeabi_ui2d>
 80009f6:   a308        add r3, pc, #32 ; (adr r3, 8000a18 <main+0x30>)
 80009f8:   e9d3 2300   ldrd    r2, r3, [r3]
 80009fc:   f7ff feea   bl  80007d4 <__aeabi_ddiv>
 8000a00:   4603        mov r3, r0
 8000a02:   460c        mov r4, r1
 8000a04:   4618        mov r0, r3
 8000a06:   4621        mov r1, r4
 8000a08:   f7ff ffcc   bl  80009a4 <__aeabi_d2uiz>
 8000a0c:   4603        mov r3, r0
 8000a0e:   607b        str r3, [r7, #4]
```

In this example, F_CPU has been made a **const unsigned int**. An appropriate type, but you can see the resulting code now involves a lot of extra work by our processor including calls to __aeabi_ddiv (double divide by double, double result) and __aeabi_d2uiz which is truncating the double to an unsigned int (note, hardware floating point is not enabled in this example).

Of course, the lesson here is to not use **const**, but to go back to #define. Incidentally, C++ fixes this issue, but we are not using the C++ compiler right now.

The other easily overlooked situation that will trigger fp math is if you use the notes' frequency #define as a function argument directly or even in a struct. When that argument is received by the function, it cannot be a constant expression.

```
080009e8 <play>:
#define F_CPU 16000000
#define F4_FREQ 349.2282
//#define F4_COUNTS F_CPU/(2*F4_FREQ)

void play(double note)
{
 80009e8:	b590      	push	{r4, r7, lr}
 80009ea:	b085      	sub	sp, #20
 80009ec:	af00      	add	r7, sp, #0
 80009ee:	ed87 0b00 	vstr	d0, [r7]
    unsigned int counts = F_CPU/(2*note);
 80009f2:	e9d7 0100 	ldrd	r0, r1, [r7]
 80009f6:	4602      	mov	r2, r0
 80009f8:	460b      	mov	r3, r1
 80009fa:	f7ff fc0b 	bl	8000214 <__adddf3>
 80009fe:	4603      	mov	r3, r0
 8000a00:	460c      	mov	r4, r1
 8000a02:	461a      	mov	r2, r3
 8000a04:	4623      	mov	r3, r4
 8000a06:	a108      	add	r1, pc, #32	; (adr r1, 8000a28 <play+0x40>)
 8000a08:	e9d1 0100 	ldrd	r0, r1, [r1]
 8000a0c:	f7ff fee2 	bl	80007d4 <__aeabi_ddiv>
 8000a10:	4603      	mov	r3, r0
 8000a12:	460c      	mov	r4, r1
```

Here we can see that since a double was passed to the function play(), all of the fp math code must take place.