EDA223/DIT161 Real-Time Systems January 12, 2018
Laboratory Assignment — Part 0

Objective: The purpose of Part 0 of the lab is to get you familiarized with the I/O capabilities of the MD407 card and learn the background for Part 1 and Part 2 of the laboratory assignment. First, you will learn how to compile a given test application using the cross compiler for the Cortex-M4 microcontroller and upload the executable on the target hardware (Step 1). Second, you will learn how to take input from the keyboard of your workstation to the application hosted on the target microcontroller via the USB serial port of the MD407 card and how to generate output to the console program on your workstation (Step 2). Third, you will learn how to compute the periods of the task that generates the waveform for the tones of the Brother John tune, and decide on the data structures that you will require in Part 2 of the laboratory assignment (Step 3). Finally, you will familiarize with some general guidelines and hints that will help you to get through Part 1 and Part 2 of the laboratory assignment without unnecessary problems.

Approval: When you see the text "*Assistant's approval: ....*" below a problem description you should show your solutions to the laboratory assistant. If the solutions are found to be satisfactory the laboratory assistant will sign your lab-PM, and you can continue with the next problem. When all problems in Part 0 have been approved the assistant will mark the corresponding examination objectives as 'Passed' in PingPong.

Step 1: Hello, hello...

Open the `CodeLite-TinyTimber quickstart tutorial` which is available under the **Resources** link on the course homepage. Follow the guidelines given in the tutorial to compile and upload the test application on the target hardware. If you run the test application successfully, you will see an output similar to the following on the console:

```
TinyTimber v2.05 (2017-03-01)
Hello, hello...
```

Step 2: I/O Using the serial port

During this step, you will practice input and output using the serial port to read data from the workstation keyboard and generate output on the console. You will work with the code in the `application.c` file. Create a backup of the file before changing it as this file will be required later.

Locate the `reader` method in the `application.c` file. Whenever you press a key on the keyboard, the `sci_interrupt` method (an interrupt handler) located in the `sciTinyTimber.c` file is invoked and the entered character is read from the serial port. This interrupt handler then invokes the `reader` method and provides to it the character read as a parameter. In summary, the `reader` method is executed whenever you press a key on the keyboard. It also prints the character to the console. For example, if you type character 'R', you will see `Rcv: 'R'` printed on the console.

Only *one* character can be read *at a time* from the serial port. Now you will learn how to take *integer* input. In order to read an integer, we have to type each character that constitutes the integer. In addition, you also need to type a special *delimiter* character, for example, 'e' to specify the end of the integer input. For example, reading integer -53 from the keyboard requires you to type '-' (the sign), '5' (the first digit), '3' (the second digit), and finally, 'e' (the delimiter).

Since the `reader` method is called each time a character is typed, we have to *store* all the previous characters of the integer number being typed in the keyboard before reading the next character. Therefore, in order to store every character that is entered until the delimiter is hit, you need to declare a string, that is, a character array of a **fixed** size. Every time a character is hit, it is compared with the delimiter ('e'). If the character typed is the delimiter, then you need to store the null character `'\0'` in the string to specify the end of the input integer[1]. Now the string can be converted to its integer representation using the `atoi` function. An example declaration of a fixed-size string and use of the `atoi` function is the following:

```
int num;
char buf[20];
buf[0]='-';
buf[1]='5';
buf[2]='3';
buf[3]='\0';
num = atoi(buf);   //variable num is equal to integer -53
```

Now you will learn how to output to the console. You can only print a character or a string in the console using the TinyTimber's build-in macros `SCI_WRITECHAR` or `SCI_WRITE`, respectively. Locate at least one use of `SCI_WRITECHAR` and one use of `SCI_WRITE` macro in the `application.c` file. In order to write the value of an integer variable in console, you first have to convert the integer into a string, and then print that string using `SCI_WRITE`. For the conversion, the `sprintf` function can be used.

Problem 2.a: Read integer −106 from keyboard and store it in an integer variable called `myNum`. Add 13 to `myNum` and print the result to the console.

Assistant's approval: ....

[1]Make sure that the character array is large enough to store any number you could possible use, including an initial '-' and the trailing `'\0'`. Unintended writes of data outside the limits of a character array can cause many hard-to-solve problems in your software!

Problem 2.b: Implement a function that continuously reads a series of integer numbers. Print the running sum each time a new number is entered. Typing 'F' should reset the running sum. An example output produced when entering numbers 13, -7, and 20 looks like this:

```
TinyTimber v2.05 (2017-03-01)
Hello, hello...
Rcv: '1'
Rcv: '3'
Rcv: 'e'
The entered number is 13
The running sum is 13
Rcv: '-'
Rcv: '7'
Rcv: 'e'
The entered number is -7
The running sum is 6
Rcv: '2'
Rcv: '0'
Rcv: 'e'
The entered number is 20
The running sum is 26
Rcv: 'F'
The running sum is 0
```

Assistant's approval: ....

Step 3: Preliminaries for Part 1 and Part 2

You have already listened to the Brother John tune in your first exercise class. A **tune** is a sequence of several **tones**. Brother John has 32 tones. The Brother John tune is generated by playing its tones one by one: the first tone is played, then the second tone is played, ... , and finally, the 32^{nd} tone is played. And, after the 32^{nd} tone finishes, again the sequence of tones is repeated starting from the first tone. In others words, the Brother John tune is a cyclic sequence of 32 tones.

You need to implement program code to generate one particular tone in **Part 1** of the laboratory assignment. Before that, in this part, you will learn the background regarding how to generate a tone, particularly, how to compute the period of the task that produces the waveform of the tone. A particular tone is generated by *periodically* writing alternating '1's and '0's to the DAC (digital-to-analog converter) of the MD407 card. In other words, a *tone-generating task* periodically writes the digits of the following sequence to the DAC:

$$1, \quad 0, \quad 1, \quad 0, \quad 1, \quad 0, \ldots$$

The time between two consecutive writes is the ***period*** of the tone-generating task. The period of the task is computed from the ***frequency*** of the tone. If the required frequency of the tone is f Hz, then the corresponding period of the task must be $\frac{1}{2f}$ sec.

Problem 3.a: Consider that alternating '1's and '0's are written to the DAC so that a tone of frequency 200 Hz is produced. The period of the tone-generating task is equal to one of the following. Tick the correct one. [Hint: 1 sec = 10^3 ms =$10^6\mu$s]

Answer: (a) 5000 μs (b) 2500 μs (c) 2000 μs

Problem 3.b: Consider that the DAC is written with alternating '1's and '0's so that a tone of frequency 1 kHz is produced. Find the period of the tone-generating task.

Answer: .... μs

Assistant's approval: ....

At this point, you know how to compute the period of the tone-generating task for a *given* frequency. Therefore, if the frequency for each of the 32 tones of the Brother John tune is known, you can compute the corresponding task period for each of the frequencies. For practical reasons, Part 2 of the laboratory assignment refers to a ***frequency index***, rather than the frequency itself, for each of the 32 tones. A frequency index i is an integer, e.g., $i = -3$. For a given frequency index i, the corresponding frequency is denoted by $p(i)$. Now you will learn how to compute the frequency corresponding to any given frequency index.

The frequency index $i = 0$ is called the *base frequency index*. The frequency for the base index is called the **base frequency**, which for our purposes is $p(0) = 440$ Hz[2].

Problem 3.c: Consider that the DAC is written with alternating '1's and '0's so that a tone with base frequency $p(0)$ is produced. Find the period of the tone-generating task.

Answer: .... μs

Assistant's approval: ....

Given the base frequency $p(0)$, frequencies for other indices can be computed based on one of the various *temperament standards* used for musical instruments. In Part 2 of the laboratory assignment, one such standard called equal-tempered 12-tone scale is used. According to this standard, the ratio of $\frac{p(i+1)}{p(i)}$ always equals the twelfth root of 2. In

[2]This is the frequency of the musical note A, which is a commonly-used audio frequency reference for calibration of acoustic equipment and tuning of musical instruments.

other words, given any two consecutive frequency indices i and $(i+1)$, the frequencies $p(i)$ and $p(i+1)$ are related as follows:

$$\frac{p(i+1)}{p(i)} = 2^{\frac{1}{12}}$$

Simple arithmetic shows that the frequency $p(k)$ and $p(i)$ for frequency indices k and i are related as follows:

$$\ldots \quad = \frac{p(i+1)}{p(i)} = \frac{p(i+2)}{p(i+1)} = \quad \ldots \quad = \frac{p(k-1)}{p(k-2)} = \frac{p(k)}{p(k-1)} = \quad \ldots \quad = 2^{\frac{1}{12}}$$

which yields

$$\frac{p(k)}{p(i)} = 2^{\frac{k-i}{12}} \tag{1}$$

Problem 3.d: Given that $p(0) = 440$ Hz, the frequency $p(k)$ for index k can be computed using Equation (1). Complete the following statement:

$$p(k) \;=\; 440 \;\times\; \ldots\ldots\ldots\ldots \tag{2}$$

Problem 3.e: As mentioned before the period of the tone-generating task must be $\frac{1}{2f}$ sec in order to produce a tone of frequency f Hz. What should the period of the tone-generating task be when (a) the frequency index is -7, and (b) the frequency index is 9?

Answer: (a) .... μs (b) .... μs

Assistant's approval: ....

Now you know how to compute the period of the tone-generating task that corresponds to a given frequency index k, assuming base frequency $p(0)$. Since the base frequency $p(0) = 440$ Hz is known, we can compute the task periods corresponding to the tones in any given tune if all frequency indices for the tones are available. The Brother John tune consists of the following 32 tones expressed in the form of frequency indices:

0	2	4	0	0	2	4	0	4	5	7	4	5	7	7	9	7	5	4	0	7	9	7	5	4	0	0	-5	0	0	-5	0

Table 1: The 32 frequency indices for the Brother John tune

Problem 3.f: Add an integer array of length 32 to your application code and copy into it the 32 frequency indices for the Brother John tune, in the order given by Table 1.

Problem 3.g: Inspect the 32 frequency indices given in Table 1 and find the maximum and minimum frequency indices.

Answer: Maximum is .... Minimum is ....

Assistant's approval: ....

The magnitude (value) of all the frequency indices in Table 1 can be increased or decreased using a *transpose* parameter called `key`. The value of `key` can be in range $[-5 \ldots 5]$. The frequency indices in Table 1 are for `key` = 0 (that is, the Brother John tune as played in the key of A). The value of `key` is added to each of the 32 frequency indices in Table 1, thereby producing new frequency indices for the given `key`. For example, if `key` = -2 (transposing the tune to the key of G), then an offset of -2 is added to each of the 32 frequency indices given in Table 1 and a new set of frequencies will be used when playing the tune.

Problem 3.h: What are the first 10 frequency indices of the Brother John tune for `key` = -5?

Answer: ....

Problem 3.i: Inspect the 32 frequency indices given for `key` = 0. If the `key` can be within the range of $[-5 \ldots 5]$, what is the maximum and minimum frequency index of the Brother John tune for any given `key`?

Answer: Maximum is .... Minimum is ....

Assistant's approval: ....

Let `max_index` and `min_index` be the maximum and minimum indices, respectively, obtained in the solution of Problem 3.i.

Problem 3.j: Pre-compute the periods of the tone-generation task corresponding to all frequency indices in the range [`min_index` ...`max_index`]. If a computed period is not an integer number, use only the integer part of the result. There is no need to involve floating-point arithmetic at run-time as the effects of using only integer values will not be perceived by the human ear. Add another integer array, named `period`, of appropriate length to your application code and copy the pre-computed periods into the array.

[Hint: It is recommended that you use a tool to do the manual computation of periods, for example, you can use a spreadsheet program like Excel, a programming language like Python, or even MATLAB.]

Note that the indices for the new array differ from the frequency indices. For example, array element `period[0]` refers to frequency index .... Therefore, when the frequency index is `f` (where `min_index` ≤ `f` ≤ `max_index`), the array element `period[......]` needs to be accessed (find the answer in terms of `f`).

Assistant's approval: ....

You stored frequency indices of Brother John tune in an array based on Table 1, and now also have an array of all required periods for the tone-generating task.

Problem 3.k: Add a function to your application code that prints the periods corresponding to all 32 frequency indices of the Brother John tune for `key=0`.

Assistant's approval: ....

Problem 3.l: Add a function to your application code that takes the `key` as input from the keyboard and prints the periods corresponding to the 32 frequency indices of the Brother John tune for the input `key`.

Assistant's approval: ....

That's it for Part 0.
Good luck with Part 1 and Part 2!

General Guidelines and Hints for Part 1 and Part 2

- Follow the software design guidelines advocated by the course material and the lab assistants. The laboratory assignment is challenging enough even when you do things "by the book". By deviating from recommended programming guidelines you run a high risk (based on past experiences from the course) of not being able to succeed with the programming assignment.

- Before starting to write your program code, draw access graphs and timing diagrams to make sure that your software design works "on paper".

- Variables and data structures that are used by multiple concurrently executing tasks in your program should be placed inside an object, and be accessed only via synchronized methods. Exception: if you have big data structures, such as lists and tables, with non-changing contents you may instead define them as global variables and refer to them directly.

- While you are testing your software it is convenient to print debug messages to the console. For example, if you change a variable by pressing a key on the workstation keyboard, it could be helpful to display the variable's new value. Or if you receive a CAN message, it may be beneficial to display the contents of the message.

- Use the `atoi` function for converting string representations of integers to their numeric counterparts.

- Use the `snprintf` function for composing strings of arbitrary format for output to the console. Make sure that the character array that you use with `snprintf` is large enough to store the entire composed string, including any expanded formatting characters (e.g., `'%d'`) and the trailing `'\0'`. Caution: We discourage the use of `sprintf` since it does not prevent unintended writes of data outside the limits of a character array (which can cause many hard-to-solve problems in your software!)

- Printing floating point numbers with `snprintf` is disabled in order to keep code size down. Instead, type cast your numbers to integers and print as such.

- If the console output gets messed up, i.e, not all intended text gets printed at the right time, it is an indicator that your program is overloading the system with too frequent text updates. Do not print debug messages in code sections that are executed hundred or thousand times per second.

- If you get the message "PANIC!!! Empty queue" on the console output, it means that the system is temporarily overloaded by data input. Probable causes: a key on the workstation keyboard has been pressed for too long (activating auto repeat), or some computer board has transmitted CAN messages too frequently.

- We do not recommend the use of dynamic memory allocation in TinyTimber for applications with hard real-time constraints as it may cause unpredictable delays in the execution of the program.