CE-2812, Lab Week 3, LCD API

1 Purpose

The purpose of this lab is to write a useful API for interacting with a character LCD module along with enhancement of previous work.

2 Prerequisites

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

3 Activities

3.1 Delay API

We wrote a busy wait delay subroutine for the last assignment. Of course, simply looping a bunch of times will be very prone to issues. A better way is to use our hardware, namely the SysTick timer. This will be extremely useful going forward. To make it easier to reuse we will create a delay API. This will consist of a header file (delay.h) that will declare the global functions (aka prototypes) and any needed symbols (such as register addresses), and a source file (delay.c) that will define the methods. In addition to delay_ms(), you may benefit from a shorter delays, so also implement a microsecond delay, delay_us(). Any helper methods or shared (file static) variables should be declared in the source file only.

Recall that the SysTick timer runs at either processor clock (16 MHz by default) or processor clock divided by 8. You must load your countdown value into the LOAD register, then poll the COUNTFLAG to know when it reaches 0. You should then disable the counter so it is ready to use on the next call to delayXX.

3.2 LCD API

The LCD API will provide essentially the same functionality as the version that may have been written in assembly language for CE2801. As an API, you will supply a header file with global function prototypes (declarations). The source file will contain the function definitions. Any helper methods or shared (file static) variables should be declared in the source file only.

As a minimum, supply the following global functions:

- lcd_init
  - Initializes I/O to interface to communicate with LCD module
  - Clears and homes the display
  - No arguments or return
- lcd_clear
  - clears the display
  - no arguments or return
  - includes necessary delay*
- lcd_home
  - moves cursor to the home position
  - no arguments or return
  - includes necessary delay*

- lcd_set_position
    - moves cursor to the position indicated
    - requires two arguments – a zero-based row, and a zero-based column, no return value
    - includes necessary delay*
- lcd_print_char
    - prints a single character to the display
    - accepts the character (by value), no return value
    - includes necessary delay*
- lcd_print_string
    - prints a null terminated string to the display
    - accepts the (pointer to) a null-terminated string, returns the number of characters written to display
    - includes necessary delay*
- lcd_print_num
    - prints a (decimal) number to the display
    - accepts the number to be printed, returns the number of characters written to display
    - includes necessary delay*

  * The datasheet for the LCD module lists the execution time for most operations. You must not send additional commands for data until the previous operation has completed. You should use your delay API to accomplish the necessary delays. You may consider polling the “busy flag” to know when a command completes, but this is not required.

The last two methods (lcd_print_string and lcd_print_num) require aspects of the C language that we have not yet gone through in lecture. These facilities of C will be covered early in Week 4 so that you will have time to add them to your otherwise complete project.

The last method, lcd_print_num, will benefit greatly from the standard library function sprintf(). Look it up (page 282 of The C Book)! **Do not use itoa(),** which, while available on our platform, is not part of the standard C library.

You may also benefit from a number of “helper” functions that actually interact with the LCD. These routines should be made file-scope by being declared static. Their prototype would not appear in the header file. Examples of these helper functions might be “lcd_data” to send data to the LCD, “lcd_command” to send a command.

Also be sure to use your own functions when possible. For example, lcd_print_string should repeatedly call lcd_print_char, and lcd_print_num should call lcd_print_string.

One other optimization that you may consider is to have the pointers for writing to GPIOA (the LCD’s data signals) and GPIOC (the LCD’s control signals) at file scope. This will improve performance by not having to define and initialize the pointers every time a method is called, and if they are file-scope (static) they cannot be seen by other source code hence you avoid the global variable issue. They should also be **pointer const** and **volatile** even though you will not likely be polling them.

3.3 **OPTIONAL** KEYPAD API

This is 100% optional!!! Just an idea if you have a lot of spare time over break 😊.

We may want to be able to capture keystrokes from the keypad and use them in our programs. You should already have a good understanding of how this is done from Embedded I.

3.4 DEMONSTRATION APPLICATION

In order to demonstrate the API, create a test and demonstration program of your own design. Can be simple, but must demonstrate most, if not all of the implemented LCD (and optionally keypad API functions). Ideas? A simple calculator, a programmable timer, a simple game… Maybe replicate Knight Rider lights on the LCD by having a block sweep back and forth. Your demo application can reside within main() or in a suitable subroutine located in the main source file.

3.5 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.**
    a. Pdf must be in color.
    b. Color scheme must be readable.
    c. You must include comments per the syllabus (banner comment at the top of each file, banner comments by each function definition, line comments by each control structure, etc.).
    d. Long lines of code or comments should not be wrapped in the pdf in a less than readable format.
    e. You must include the "summary of experience" paragraph in a block comment at the top of the first page / first source file.
    f. If you were unable to complete every requirement of the assignment, be sure to describe what is done and working and what is not working with your summary paragraph.
2. Ask to demo your lab to instructor.
    a. If you demo during lab in Week 3, you will earn a 10% bonus on this lab. ← really not expecting this!!
    b. If you demo during lab in Week 4, 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 4, 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 points.

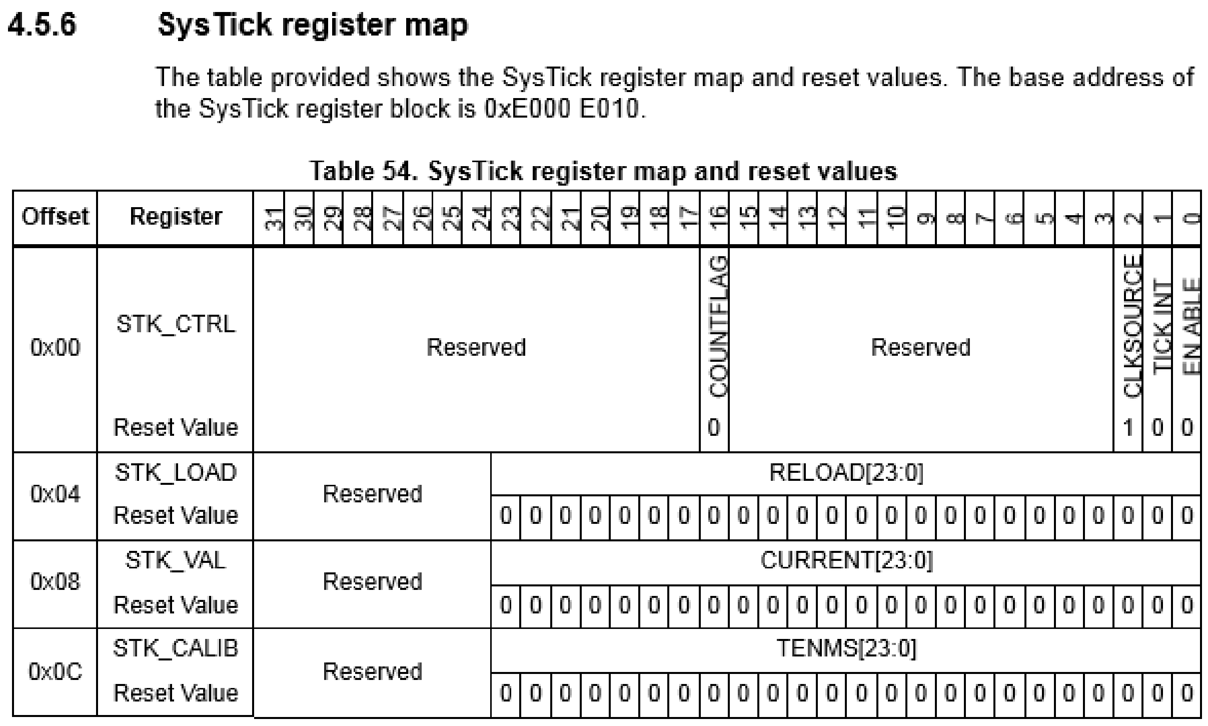
3.6 Grading Criteria

For full credit, your solution must:

- Be submitted correctly to Canvas
    - Color pdf, no wrapping of comments, comments per syllabus, summary paragraph, etc.
- Implement the delay API and the lcd API as described above, using the "API Layout" discussed in lecture. Given this, there should be at least three source files (delay.c, lcd.c, main.c) and at least two header files (delay.h, lcd.h) for this project.
    - Be sure to factor out "helper" functions in your APIs and make them file-scope (static).
- Continue to use **pointer variables** for I/O register access. **Do not directly dereference macros** for register access. Be sure pointer variables are declared **volatile** and **const** as appropriate.
- Use macros to **#define** useful symbols and minimize "magic numbers."
- "Demo" program may reside in main() if you wish or place in a separate function and/or source file.
- 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.

3.7 Hint - Using Pointers for I/O Register Interaction

Consider a typical peripheral that possess a handful of memory-mapped registers for control of that peripheral. A great example is the SysTick timer.



4.5.6 SysTick register map

The table provided shows the SysTick register map and reset values. The base address of the SysTick register block is 0xE000 E010.

Table 54. SysTick register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	STK_CTRL	Reserved															COUNTFLAG	Reserved													CLKSOURCE	TICK INT	EN ABLE
	Reset Value																0														1	0	0
0x04	STK_LOAD	Reserved								RELOAD[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	STK_VAL	Reserved								CURRENT[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	STK_CALIB	Reserved								TENMS[23:0]																							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

There are a number of ways to interact with this peripheral.

3.7.1 Pointer to Individual Location

The most straightforward would be to create a pointer that points to each desire location. Since these are 32-bit registers and have no notion of signedness, uint32_t would be the best type to use.

```
volatile uint32_t * const stk_ctrl = (uint32_t*) 0xE000E010;
*stk_ctrl = 0;  // clear control register
volatile uint32_t * const stk_load = (uint32_t*) 0xE000E014;
*stk_load = 1234;  // set reload register
*stk_ctrl = 1;  // start timer
```

This technique is fine. It works and is relatively easy to understand. It is not, however, very efficient. Each pointer variable consumes memory. One could economize and reuse one pointer variable, but that is not efficient either due to the reassignments. One improvement is to declare the pointers once at file-scope (static) and thus each method in the API can use the pointer(s) but we still have some control over what code can access registers.

3.7.2 Pointer to Base Address add Offset

One slight improvement can be made by employing "pointer arithmetic." With this technique, a single pointer can be created to the base address of the peripheral. Then, an offset is applied to access the subsequent registers. Note that when a pointer type is used in an arithmetic expression, it is manipulated in multiples of the type to which it points. In this example, when '1' is added below, it actually adds 4 to the address since it points to a 32-bit value.

```
volatile uint32_t * const stk_base = (uint32_t*) 0xE000E010;
*stk_base = 0;  // clear control register
*(stk_base + 1) = 1234;  // set reload register
```

```
    *stk_base = 1;  // start timer
```

One can improve this slightly by using symbolic names for the offsets:

```
    #define CTRL 0
    #define LOAD 1  // register #1

    volatile uint32_t * const stk_base = (uint32_t*) 0xE000E010;
    *(stk_base + CTRL) = 0;  // clear control register
    *(stk_base + LOAD) = 1234;  // set reload register
    *(stk_base + CTRL) = 1;  // start timer
```

Do note that `*stk_base+CTRL` is not the same as `*(stk_base+CTRL)`. First of all, the former expression would be a syntax error if on the left-hand-side of an assignment (LHS), but both are valid on the right-hand-side (RHS) and will give very different results. Also note the value of CTRL and what the proper address of the LOAD register.

Also note the following expressions will provide differing results – be sure to use the correct one(s). Two of the expressions below will work and properly initialize the pointer stk_load to its address of 0xE000E014.

```
    volatile uint32_t * const stk_load = (uint32_t*) 0xE000E010 + 0x04;

    volatile uint32_t * const stk_load = (uint32_t*) (0xE000E010 + 0x04);

    volatile uint32_t * const stk_load = (uint32_t*) 0xE000E010 + 0x01;

    volatile uint32_t * const stk_load = (uint32_t*) (0xE000E010 + 0x01);
```

These are all examples of "pointer arithmetic" which is valid and common. It is, however, still prone to errors. We will eventually develop yet better ways to interact with I/O registers.