Programming Assignment
Seven-Segment Display and C Libraries

Hazem Selmi, Ahmad Khayyat

Version 162, 8 March 2017

Table of Contents

1. Objectives

- Using seven-segment displays
- Organizing your code into libraries

2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- Seven-segment display
- 330-Ohm Resistors
- Jumper wires

3. Background

3.1. Seven-Segment Displays

Seven-segment displays can be useful in many types of projects as an output device.

The part number of the seven-segment display included in your kit is LTD-4608JR.

You should be able to use the data sheet to figure out how to display a number on the seven-segment display.

This seven-segment display part can show two digits; it is actually two seven-segment displays in one part, with common seven pins connected to them both. Simply setting the seven pins would show the same digit on both displays.

To show a different digit on each display simultaneously, we need to employ time multiplexing. Each display has its own *common* pin, which enables and disables that display. Setting the seven pins affects enabled displays only. Alternatively enabling each display while disabling the other allows you to set each display to show a different digit. However, leaving a display disabled will clear any previously shown digit. Therefore, time multiplexing is required, whereby the two displays are continuously enabled alternatively.

3.2. Building a C Library

A C library consists of one or more `c`-files and `h`-files that can be used as part of a project to make your code modular and more efficient.

3.2.1. The `c` file

The `c` file can have one or more, usually related, C functions. To make your code readable and easily reusable, it is crucial to carefully comment your code and wisely choose the function and parameter names.

In LPCXpresso, you can add new C source files to your project by right-clicking the `src` folder and adding a new *source file*.

3.2.2. The `h` file

To call functions defined in a library, your source file needs to tell the compiler what the function signatures, or prototypes, are. Instead of having to write the function prototypes every time you use a library, libraries

provide a file containing those prototypes, without the function implementations. Such a file has the `.h` extension, and is called a *header file*.

The `c` files contain the full functions that the library defines. In contrast, the `h` files contain only the function prototypes, without their implementations. This allows you to use libraries without necessarily having access to their full sources, just their function prototypes and binary implementation. Moreover, header files may contain other macros, or preprocessor directives, such as `define` and `include` statements.

To use a library, you only need to include its header file(s) in your program. Including a file is equivalent to inserting the contents of the included file in your program file.

As with any source file, it is good practice to comment your library implementation (`c` files) and headers (`h` files).

To avoid including multiple copies of the same header file, which would result in function name collision, it is recommended to use the following template for your header files.

```
#ifndef LIBRARY_NAME_H
#define LIBRARY_NAME_H

/* Header file statements */

#endif /* LIBRARY_NAME_H */
```

Exercise

Study the above template and explain how it ensures that a header file that uses this template will always be included at most once.

The above template is automatically generated for you by LPCXpresso when you choose the *default C-header template* when adding a new header file.

3.3. A Seven-Segment Display Library

We want to write a C library that would help us use the seven-segment display, by abstracting out the details of decoding each digit into the actual bit pattern that will set the display to show the required digit. The ultimate objective is to set the pins that are connected to the display to show the digit. However, for the library to be reusable, it would be wise not to tie it to specific pins, and leave the mapping of the bits to the actual I/O pins outside the library.

Therefore, the library would include one public function that acts as a BCD-to-seven-segment decoder. Think of the function as a BCD-to-seven-segment decoder chip. If you use such a chip, you still need to map each of its outputs to the seven-segment display pins.

The library function will generate the seven bits. Then, in your main program, or perhaps in a separate function in your main program file, you assign each of these bits to one of the seven-segment display pins. This way, you can use the library regardless of where the seven-segment display is connected.

Modular design, which is achieved by appropriate abstraction, improves the reusability of your code, which makes you more productive.

In this case, the library function abstracts the decoding of a BCD digit to seven-segment bits, and does nothing else.

It is also useful to write another library for reading and writing to GPIO pins. The prototypes for the functions in such a library may look something like the following.

```
void set_gpio_pin(int port, int pin, int value); // write to an output pin
int  get_gpio_pin(int port, int pin);            // read an input pin
```

In this case you need to call the write function seven times (once per bit).

Consider using the newer 8-bit `uint8_t` type instead of the typically 32-bit `int` type when you can guarantee that 8 bits are enough to represent the values. `uint8_t` is a part of the C99 standard, among other similarly named types, which are guaranteed to have the same exact width across all platforms, unlike the plain `int` type.

`uint8_t` is defined in `inttypes.h` and `stdint.h`.

4. Tasks

4.1. Display a Hard-wired Number on the Seven-Segment Display

1. Use the seven-segment display along with VDD and GND pins on the microcontroller to display a number. You do not need to program the microcontroller. Display the same number on both displays.

   This task will help you confirm your understanding of how the seven-segment display works.

4.2. Display a Number on the Seven-Segment Display by Software

2. Write a program that displays different numbers on the seven-segment display. For example, a program that counts from 0 to 9. Display the same number on both displays.

4.3. Write a Seven-Segment Display Library

3. Write a reusable library consisting of a `c` file and a corresponding header (`h`) file for decoding a BCD digit to seven-segment display bits. The library should not be tied to any specific I/O pins; it just implements a seven-segment display decoder.
4. Write a library for easily accessing GPIO pins (reading and writing).
5. Write a program to test your seven-segment display decoder library and your GPIO library.

4.4. Display Two Different Digits Simultaneously

6. Write a program that implements time multiplexing to show a different digit on each display.