CHAPTER 4

Using Libraries

You have used libraries all your life—well, at least as long as you have programmed in C. Want to display text on the screen? `printf`. What about determining the length of a string? `strlen`. Need to sort an array? `qsort`. You can find these functions, along with numerous others, in the C standard library. A *library* consists of a collection object files (`.o`), that have been combined into an archive file (`.a`): for example, the C standard library `libc.a`. Using a library requires you to include the associated header files (`.h`) and link with the archive file. The header file (e.g., `stdio.h`) declares the functions, constants, and data types used by the library while the archive file contains function implementations. Libraries make it easy to share code between multiple projects without needing to repeatedly compile the code.

In addition to the C standard library, Microchip provides some other libraries specific to programming PIC32s. In Chapter 3, we learned about the header file `xc.h` which includes the processor-specific header `pic32mx795f512h.h`, providing us with definitions for the SFRs. The "archive" file for this library is `processor.o`.[1] Microchip also provides a higher-level framework called Harmony, which contains libraries and other source code to help you create code that works with multiple PIC32 models; we use Harmony later in this book.

Libraries can also be distributed as source code: for example, the NU32 library consists of `<PIC32>/skeleton/NU32.h` and `<PIC32>/skeleton/NU32.c`. To use libraries distributed as source code you must include the library header files, compile your source code and the library code, and link the resulting object files. You can link as many object files as you want, as long as they do not declare the same symbols (e.g., two C files in one project cannot both have a `main` function).

The NU32 library provides initialization and communication functions for the NU32 board. The `talkingPIC.c` code in Chapter 1 uses the NU32 library, as will most of the examples throughout the book. Let us revisit `talkingPIC.c`, and examine how it includes libraries during the build process.

[1] The library consists of only one object file so Microchip did not create an archive, which holds multiple object files.

4.1 Talking PIC

In the previous chapter, to keep things as simple as possible, we built the executable from `simplePIC.c` by directly issuing the `xc32-gcc` and `xc32-bin2hex` commands at the command line. In this chapter, and all future chapters, we use the `Makefile` with `make` to build the executable, as with `talkingPIC.c` in Chapter 1.

Recall from Chapter 1 that the `Makefile` compiles and links all `.c` files in the directory. Since the project directory `<PIC32>/talkingPIC` contains `NU32.c`, this file was compiled along with `talkingPIC.c`. To see how this process works, we examine the commands that `make` issues to build your project.

Navigate to where you created talkingPIC in Chapter 1 (`<PIC32>/talkingPIC`). Issue the following command:

```
> make clean
```

This command removes the files created when you originally built the project, so we can start fresh. Next, issue the `make` command to build the project. Notice that it issues commands similar to:

```
> xc32-gcc -g -O1 -x c -c -mprocessor=32MX795F512H -o talkingPIC.o talkingPIC.c
> xc32-gcc -g -O1 -x c -c -mprocessor=32MX795F512H -o NU32.o NU32.c
> xc32-gcc -mprocessor=32MX795F512H -o out.elf  talkingPIC.o  NU32.o
     -Wl,--script="NU32bootloaded.ld",-Map=out.map
> xc32-bin2hex out.elf
> xc32-objdump -S out.elf > out.dis
```

The first two commands compile the C files necessary to create `talkingPIC` using the following options:

- `-g`: Include debugging information, extra data added into the object file that helps us to inspect the generated files later.
- `-O1`: Sets optimization level one. We discuss optimization in Chapter 5.
- `-x c`: Tells the compiler to treat input files as C language files. Typically the compiler can detect the proper language based on the file extension, but we use this here to be certain.
- `-c`: Compile and assemble only, do not link. The output of this command is just an object (`.o`) file because the linker is not invoked to create an `.elf` file.

Thus the first two commands create two object files: `talkingPIC.o`, which contains the `main` function, and `NU32.o`, which includes helper functions that `talkingPIC.c` calls. The third command tells the compiler to invoke the linker, because all the "source" files specified are actually object (`.o`) files. We do not invoke the linker `xc32-ld` directly because the compiler automatically tells the linker to link against some standard libraries that we need. Notice that `make` always names its output `out.elf`, regardless of what you name the source files.

Some additional options that `make` provides to the linker are specified after the `Wl` flag:

- `--script`: Tells the linker to use the `NU32bootloaded.ld` linker script.
- `-Map`: This option is passed to the linker and tells it to produce a map file, which details the program's memory usage. Chapter 5 explains map files.

The next command produces the hex file. The final line, `xc32-objdump`, disassembles `out.elf`, saving the results in `out.dis`. This file contains interspersed C code and assembly instructions, allowing you to inspect the assembly instructions that the compiler produces from your C code.

4.2 The NU32 Library

The NU32 library provides several functions that make programming the PIC32 easier. Not only does `talkingPIC.c` use this library, but so do most examples in this book. The `<PIC32>/skeleton` directory contains the NU32 library files, `NU32.c` and `NU32.h`; you copy this directory to create a new project. The `Makefile` automatically links all files in the directory, thus `NU32.c` will be included in your project. By writing `#include "NU32.h"` at the beginning of the program, we can access the library. We list `NU32.h` below:

Code Sample 4.1 `NU32.h`**. The NU32 Header File.**

```
#ifndef NU32__H__
#define NU32__H__

#include <xc.h>                       // processor SFR definitions
#include <sys/attribs.h>              // __ISR macro

#define NU32_LED1 LATFbits.LATF0      // LED1 on the NU32 board
#define NU32_LED2 LATFbits.LATF1      // LED2 on the NU32 board
#define NU32_USER PORTDbits.RD7       // USER button on the NU32 board
#define NU32_SYS_FREQ 80000000ul      // 80 million Hz

void NU32_Startup(void);
void NU32_ReadUART3(char * string, int maxLength);
void NU32_WriteUART3(const char * string);

#endif // NU32__H__
```

The `NU32__H__` include guard, consisting of the first two lines and the last line, ensure that `NU32.h` is not included twice when compiling any single C file. The next two lines include Microchip-provided headers that you would otherwise need to include in most programs. The next three lines define aliases for SFRs that control the two LEDs (`NU32_LED1` and `NU32_LED2`) and the USER button (NU32_USER) on the NU32 board. Using these aliases allows us to write code like

```
int button = NU32_USER; // button now has 0 if pressed, 1 if not
NU32_LED1 = 0;          // turn LED1 on
NU32_LED2 = 1;          // turn LED2 off
```

which is easier than remembering which PIC32 pin is connected to these devices. The header also defines the NU32_SYS_FREQ constant, which contains the frequency, in Hz, at which the PIC32 operates. The rest of NU32.h consists of function prototypes, described below.

void NU32_Startup(void) Call NU32_Startup() at the beginning of main to configure the PIC32 and the NU32 library. You will learn about the details of this function as the book progresses, but here is an overview. First, the function configures the prefetch cache module and flash wait cycles for maximum performance. Next, it configures the PIC32 for multi-vector interrupt mode. Then it disables JTAG debugging so that the associated pins are available for other functions. The pins RF0 and RF1 are then configured as digital outputs, to control LED1 and LED2. The function then configures UART3 so that the PIC32 can communicate with your computer. Configuring UART3 allows you to use NU32_WriteUART3() and NU32_ReadUART3() to send strings between the PIC32 and the computer. The communication occurs at 230,400 baud (bits per second), with eight data bits, no parity, one stop bit, and hardware flow control with CTS/RTS. We discuss the details of UART communication in Chapter 11. Finally, it enables interrupts (see Chapter 6). You may notice that these tasks, such as configuring the prefetch cache, are also performed by the bootloader. We do this because NU32_Startup() also works with standalone code, in which case these actions would be required, not redundant.
void NU32_ReadUART3(char * string, int maxLength) This function takes a character array string and a maximum input length maxLength. It fills string with characters received from the host via UART3 until a newline \n or carriage return \r is received. If the string exceeds maxLength, the new characters wrap around to the beginning of the string. Note that this function will not exit unless it receives a \n or a \r.
Example:

```
char message[100] = {}, str[100] = {};
int i = 0;
NU32_ReadUART3(message, 100);
sscanf(message, "%s %d", str, &i);  // if message is expected to have a string and int
```

void NU32_WriteUART3(const char * string) This function sends a string over UART3. The function does not complete until the transmission has finished. Thus, if the host computer is not reading the UART and asserting flow control, the function will wait to send its data.
Example:

```
char msg[100] = {};
sprintf(msg,"The value is %d.\r\n",22);
NU32_WriteUART3(msg);
```

4.3 Bootloaded Programs

Throughout the rest of this book, all C files with a main function will begin with

```
#include "NU32.h"          // constants, funcs for startup and UART
```

and the first line of code (other than local variable definitions) in `main` will be

```
NU32_Startup();
```

While other C files and header files might include `NU32.h` to gain access to its contents and function prototypes, no file except the C file with the `main` function should call `NU32_Startup()`.

For bootloaded programs, the configuration bits are set by the bootloader. However, `NU32.c` includes the configuration bit settings. This provides a convenient reference and also allows you to use the same code for both bootloaded and standalone applications (see Chapter 3.6).

4.4 An LCD Library

Dot matrix LCD screens are inexpensive portable devices that can display information to the user. LCD screens often come with an integrated controller that simplifies communication with the LCD. We now discuss a library that allows the PIC32 to control a Hitachi HD44780 (or compatible) LCD controller connected to a 16x2 LCD screen.[2] You can purchase the screen and controller as a pre-built module. The data sheet for this controller is available on the book's website.

The HD44780 has 16 pins: ground (GND), power (VCC), contrast (VO), backlight anode (A), backlight cathode (K), register select (RS), read/write (RW), enable strobe (E), and 8 data pins (D0-D7). We show the pins below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GND	VCC	VO	RS	R/W	E	D0	D1	D2	D3	D4	D5	D6	D7	A	K

Connect the LCD as shown in Figure 4.1.

The LCD is powered by VCC (5 V) and GND. The resistors R1 and R2 determine the LCD's brightness and contrast, respectively. Good guesses for these values are $R1 = 100\ \Omega$ and $R2 = 1000\ \Omega$, but you should consult the data sheet and experiment. The remaining pins are for communication. The R/W pin controls the communication direction. From the PIC32's perspective, $R/W = 0$ means write and $R/W = 1$ means read. The RS pin indicates whether the

[2] Many LCD controllers are compatible with the HD44780. We used the Samsung KS006U for the examples in this chapter.

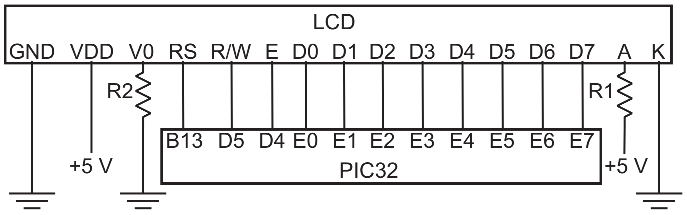
LCD
GND VDD V0 RS R/W E D0 D1 D2 D3 D4 D5 D6 D7 A K
R2
R1
B13 D5 D4 E0 E1 E2 E3 E4 E5 E6 E7
PIC32
+5 V
+5 V

Figure 4.1
Circuit diagram for the LCD.

PIC32 is sending data (e.g., text) or a command (e.g., clear screen). The pins D0-D7 carry the actual data between the devices; after setting data on these pins the PIC32 pulses the enable strobe (E) signal to tell the LCD that the data is ready. For every pulse of E, the LCD receives or sends eight bits of data simultaneously (in *parallel*). We delve into this parallel communication scheme more deeply in Chapter 14, where we discuss the parallel master port (PMP), the peripheral that coordinates the signals between the PIC32 and the LCD.

Now we present the LCD library by looking at its interface. The LCD controller has many features, such as the ability to horizontally scroll text, display custom characters, display a larger font on a single line, and display a cursor. The LCD library contains many functions that enable access to these features; however, we only discuss the basics.

Code Sample 4.2 `LCD.h`**. The LCD Library Header File.**

```
#ifndef LCD_H
#define LCD_H
// LCD control library for Hitachi HD44780-compatible LCDs.

void LCD_Setup(void);                        // Initialize the LCD
void LCD_Clear(void);                        // Clear the screen, return to position (0,0)
void LCD_Move(int line, int col);            // Move position to the given line and column
void LCD_WriteChar(char c);                  // Write a character at the current position
void LCD_WriteString(const char * string);   // Write string starting at current position
void LCD_Home(void);                         // Move to (0,0) and reset any scrolling
void LCD_Entry(int id, int s);               // Control display motion after sending a char
void LCD_Display(int d, int c, int b);       // Turn display on/off and set cursor settings
void LCD_Shift(int sc, int rl);              // Shift the position of the display
void LCD_Function(int n, int f);             // Set number of lines (0,1) and the font size
void LCD_CustomChar(unsigned char val, const char data[7]); // Write custom char to CGRAM
void LCD_Write(int rs, unsigned char db70); // Write a command to the LCD
void LCD_CMove(unsigned char addr);          // Move to the given address in CGRAM
unsigned char LCD_Read(int rs);              // Read a value from the LCD
#endif
```

LCD_Setup(void) Initializes the LCD, putting it into two-line mode and clearing the screen. You should call this at the beginning of `main()`, after you call `NU32_Startup()`.
LCD_Clear(void) Clears the screen and returns the cursor to line zero, column zero.
LCD_Move(int line, int col) Causes subsequent text to appear at the given line and column. After calling `LCD_Setup()`, the LCD has two lines and 16 columns. Remember, just like C arrays, numbering starts at zero!
LCD_WriteChar(unsigned char s) Write a character to the current cursor position. The cursor position will then be incremented.
LCD_WriteString(const char * str) Displays the string, starting at the current position. Remember, the LCD does not understand control characters like `'\n'`; you must use `LCD_Move` to access the second line.

The program `LCDwrite.c` uses both the NU32 and LCD libraries to accept a string from your computer and write it to the LCD. To build the executable, copy the `<PIC32>/skeleton` directory and then add the files `LCDwrite.c`, `LCD.c`, and `LCD.h`. After building, loading, and running the program, open the terminal emulator. You can now converse with your LCD! The terminal emulator will ask

```
What do you want to write?
```

If you respond `Echo!!`, the LCD prints

```
Echo!!__________
___Received_1___
```

where the underscores represent blank spaces. As you send more strings, the Received number increments. The code is given below.

Code Sample 4.3 `LCDwrite.c`**. Takes Input from the User and Prints It to the LCD Screen.**

```
#include "NU32.h"          // constants, funcs for startup and UART
#include "LCD.h"

#define MSG_LEN 20

int main() {
  char msg[MSG_LEN];
  int nreceived = 1;

  NU32_Startup();         // cache on, interrupts on, LED/button init, UART init

  LCD_Setup();

  while (1) {
    NU32_WriteUART3("What do you want to write? ");
    NU32_ReadUART3(msg, MSG_LEN);            // get the response
    LCD_Clear();                             // clear LCD screen
    LCD_Move(0,0);
    LCD_WriteString(msg);                    // write msg at row 0 col 0
    sprintf(msg, "Received %d", nreceived);  // display how many messages received
    ++nreceived;
    LCD_Move(1,3);
    LCD_WriteString(msg);                    // write new msg at row 1 col 3
    NU32_WriteUART3("\r\n");
  }
  return 0;
}
```

4.5 Microchip Libraries

Microchip provides several libraries for PIC32s. Understanding these libraries is rather confusing (as we began to see in Chapter 3), partially because they are written to support many PIC32 models, and partially because of the requirement to maintain backwards compatibility, so that code written years ago does not become obsolete with new library releases.

Historically, people primarily programmed microcontrollers in assembly language, where the interaction between the code and the hardware is quite direct: typically the CPU executes one assembly instruction per clock cycle, without any hidden steps. For complex software projects, however, assembly language becomes cumbersome because it is processor-specific and lacks convenient higher-level constructs.

The C language, although still low-level, provides some portability and abstraction. Much of your C code will work for different microcontrollers with different CPUs, provided you have a compiler for the particular CPU. Still, if your code directly manipulates a particular SFR that does not exist on another microcontroller model, portability is broken.

Microchip's recent software release, Harmony addresses this issue by providing functions that allow your code to work for many PIC32 models. In a simplified hierarchical view, the user's application may call Microchip *middleware* libraries, which provide a high level of abstraction and keep the user somewhat insulated from the hardware details. The middleware libraries may interface with lower-level *device drivers*. Device drivers may interface with still lower-level *peripheral libraries*. These peripheral libraries then, finally, read or write the SFRs associated with your particular PIC32.

Our philosophy is to stay close to the hardware, similar to assembly language programming, but with the benefits of the easier higher-level C language. This approach allows you to directly translate from the PIC32 hardware documentation to C code because the SFRs are accessed from C using the same names as the hardware documentation. If unsure of how to access an SFR from C code, open the processor-specific header file `<xc32dir>/<xc32ver>/pic32mx/proc/p32mx795f512h.h`, search for the SFR name, and read the declarations related to that SFR. Overall, we believe that this low-level approach to programming the PIC32 should provide you with a strong foundation in microcontroller programming. Additionally, after programming using SFRs directly, you should be able to understand the documentation for any Microchip-provided software and, if you desire, use it in your own projects. Finally, we believe that programming at the SFR level translates better to other microcontrollers: Harmony is Microchip-specific, but concepts such as SFRs are widespread.

4.6 Your Libraries

Now that you have seen how some libraries function, you can create your own libraries. As you program, try to think about the interconnections between parts of your code. If you find that some functions are independent of other functions, you may want to code them in separate `.c` and `.h` files. Splitting projects into multiple files that contain related functions helps increase program modularity. By leaving some definitions out of the header file and declaring functions and variables in your C code as `static` (meaning that they cannot be used outside the C file), you can hide the implementation details of your code from other code. Once you divide your code into independent modules, you can think about which of those modules might be useful in other projects; these files can then be used as libraries.

4.7 Chapter Summary

- A library is a `.a` archive of `.o` object files and associated `.h` header files that give programs access to function prototypes, constants, macros, data types, and variables associated with the library. Libraries can also be distributed in source code form and need not be compiled into archive format prior to being used; in this way they are much like code that you write and split amongst multiple C files. We often call a "library" a `.c` file and its associated `.h` file.
- For a project with multiple C files, each C file is compiled and assembled independently with the aid of its included header files. Compiling a C file does not require the actual definitions of helper functions in other helper C files; only the prototypes are needed. The function calls are resolved to the proper virtual addresses when the multiple objects are linked. If multiple object files have functions with the same name, and these functions are not `static` (private) to the particular file, the linker will fail.
- The NU32 library provides functions for initializing the PIC32 and communicating with the host computer. The LCD library provides functions to write to a 16×2 character dot matrix LCD screen.

4.8 Exercises

1. Identify which functions, constants, and global variables in `NU32.c` are private to `NU32.c` and which are meant to be used in other C files.

2. You will create your own libraries.
   a. Remove the comments from `invest.c` in Appendix A. Now modify it to work on the NU32 using the NU32 library. You will need to replace all instances of `printf` and `scanf` with appropriate combinations of `sprintf`, `sscanf`, `NU32_ReadUART3` and `NU32_WriteUART3`. Verify that you can provide data to the PIC32 with your keyboard and display the results on your computer screen. Turn in your code for all the files, with comments where you altered the input and output statements.
   b. Split `invest.c` into two C files, `main.c` and `helper.c`, and one header file, `helper.h`. `helper.c` contains all functions other than `main`. Which constants, function prototypes, data type definitions, etc., should go in each file? Build your project and verify that it works. For the safety of future `helper` library users, put an include guard in `helper.h`. Turn in your code and a separate paragraph justifying your choice for where to put the various definitions.
   c. Break `invest.c` into three files: `main.c`, `io.c`, and `calculate.c`. Any function which handles input or output should be in `io.c`. Think about which prototypes, data types, etc., are needed for each C file and come up with a good choice of a set of header files and how to include them. Again, for safety, use include guards in your header files. Verify that your code works. Turn in your code and a separate paragraph justifying your choice of header files.
3. When you try to build and run a program, you could run into (at least) three different kinds of errors: a compiler error, a linker error, or a run-time error. A compiler or linker error would prevent the building of an executable, while a run-time error would only become evident when the program does not behave as expected. Say you are building a program with no global variables and two C files, exactly one of which has a `main()` function. For each of the three types of errors, give simple code that would lead to it.
4. Write a function, `void LCD_ClearLine(int ln)`, that clears a single line of the LCD (either line zero or line one). You can clear a line by writing enough space (`' '`) characters to fill it.
5. Write a function, `void LCD_print(const char *)`, that writes a string to the LCD and interprets control characters. The function should start writing from position (0,0). A carriage return (`'\r'`) should reset the cursor to the beginning of the line, and a line feed (`'\n'`) should move the cursor to the other line.

Further Reading

32-Bit language tools libraries. (2012). Microchip Technology Inc.

HD44780U (LCD-II) dot matrix liquid crystal display controller/driver. HITACHI.

KS0066U 16COM/40SEG driver and controller for dot matrix LCD. Samsung.