Chapter 4

Using Libraries

In Chapter 3 we learned a bit about the functions, macros, variables, and constants made available when we include the header file `plib.h`. Including this single file eventually gives us access to all of the code that Microchip has provided for us. Most of this code is compiled and placed in *libraries* with the extension `.a`, like the math library `pic32mx/lib/libm.a`. These `.a` libraries are collections of `.o` object codes.

In this chapter we will explore how to make our own libraries, a big step toward modular code. We will not bother to compile into archive files, however; we will leave them as C source code. We informally refer to a "library" as a `.h` header file and an associated `.c` file without a `main` function. The functions in this "helper" C file all serve some related purpose (like the math functions in `libm.a`) and are easily reused in different projects.

One example of a library is the NU32 library consisting of `NU32.h` and `NU32.c`. The NU32 library provides initialization and communication functions for the NU32 board. The `talkingPIC.c` code in Chapter 1.6 uses the NU32 library, and we will use the NU32 library extensively throughout the book.

The basics of helper C source files and their header files are covered in Chapter A.4.16.

4.1 An Example: The Ports Header File

Here is a portion of the `pic32mx/include/peripheral/ports.h` header file associated with I/O port functions, slightly simplified and with comments added:

```
#ifndef _PORTS_H_     // include guard; if _PORTS_H_ already defined, skip to end
#define _PORTS_H_     // if not, define so preprocessor won't include again during this compile

#include <xc.h>       // definitions in xc.h are needed by ports.h

typedef enum {        // a new data type; variables of the new type IoPortId can only
                      // take values IOPORT_A, IOPORT_B, etc.
  IOPORT_A, IOPORT_B, IOPORT_C, IOPORT_D,
  IOPORT_E, IOPORT_F, IOPORT_G, IOPORT_NUM
} IoPortId;

// function prototype; the definition of this function is in
// pic32-libs/peripheral/ports/source/port_read_bits_lib.c
unsigned int PORTReadBits(IoPortId portId, unsigned int bits);

// macros and constants
#define mPORTAReadBits(_bits)  (PORTA & (unsigned int)(_bits))
#define DEBUG_JTAGPORT_ON      (1)

#endif                // end _PORTS_H_ include guard
```

This file demonstrates a typical include guard in the first two lines and last line; the defined constant used in the include guard could be anything you want, but it is common to choose the name based on the name of the `.h` file. An underscore can be used to replace the disallowed dot character, and underscores are often added at the beginning or end to help make sure the constant name does not accidentally match the name of a constant in your program. For any file that includes it, `ports.h` provides the header file `xc.h`, a new data type `IoPortId`, a prototype that allows the use of the function `PORTReadBits`, the macro `mPORTAReadBits()`, and the constant `DEBUG_JTAGPORT_ON`.

4.2 The NU32 Library

The NU32 library provides a number of functions for initializing the NU32 development board and communicating with the host computer. The `talkingPIC.c` program in Chapter 1.6 makes use of this library. By adding `NU32.c` and `NU32.h` to the same directory as our program, and by including the command `#include "NU32.h"` at the beginning of our program, we gain access to some useful functions and constants. The listing of `NU32.h` is given below.

Code Sample 4.1. `NU32.h`. The NU32 header file.

```
#ifndef __NU32_H
#define __NU32_H

#include <plib.h>

#ifdef NU32_STANDALONE              // config bits if not set by bootloader

#pragma config DEBUG = OFF          // Background Debugger disabled
#pragma config FPLLMUL = MUL_20     // PLL Multiplier: Multiply by 20
#pragma config FPLLIDIV = DIV_2     // PLL Input Divider:  Divide by 2
#pragma config FPLLODIV = DIV_1     // PLL Output Divider: Divide by 1
#pragma config FWDTEN = OFF         // WD timer: OFF
#pragma config POSCMOD = HS         // Primary Oscillator Mode: High Speed xtal
#pragma config FNOSC = PRIPLL       // Oscillator Selection: Primary oscillator w/ PLL
#pragma config FPBDIV = DIV_1       // Peripheral Bus Clock: Divide by 1
#pragma config BWP = OFF            // Boot write protect: OFF
#pragma config ICESEL = ICS_PGx2    // ICE pins configured on PGx2, Boot write protect OFF.
#pragma config FSOSCEN = OFF        // Disable second osc to get pins back
#pragma config FSRSSEL = PRIORITY_7 // Shadow Register Set for interrupt priority 7

#endif // NU32_STANDALONE

#define NU32LED1 LATAbits.LATA4
#define NU32LED2 LATAbits.LATA5
#define NU32USER PORTDbits.RD13
#define SYS_FREQ 80000000           // 80 million Hz

void NU32_Startup();
void NU32_ReadUART1(char* string,int maxLength);
void NU32_WriteUART1(const char *string);
void NU32_EnableUART1Interrupt();
void NU32_DisableUART1Interrupt();
void WriteString(UART_MODULE id, const char *string);
void PutCharacter(UART_MODULE id, const char character);

#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 test `#ifdef NU32_STANDALONE` checks to see if the C file

has defined the constant NU32_STANDALONE. If so, the device configuration bits are set by the header file; if not, the bootloader has already set them. The next three lines define the mnemonic constants NU32LED1 and NU32LED2 for the two LEDs and NU32USER for the USER button. With these we can use the C statements

```
unsigned int button;
button = NU32USER;           // button now has 0 if pressed, 1 if not
NU32LED1 = 0; NU32LED2 = 1;  // LED1 is turned on and LED2 is turned off
```

The remainder of NU32.h consists of function prototypes, described below.

void NU32_Startup() This function configures the prefetch cache module and flash wait cycles for maximum performance, enables interrupts, disables JTAG debugging so RA4 and RA5 are available as digital I/O, configures RA4 and RA5 as outputs for the LEDs, and enables UART1 and UART3 for RS-232 serial communication with the host computer. UART3 is used to talk to the bootloader communication utility on the host, while UART1 is meant for communication by the user's program with the host (as in talkingPIC.c). The communication is configured for 230,400 baud (bits per second), eight data bits, no parity, one stop bit, and hardware flow control with CTS/RTS. NU32_Startup() should be called at the beginning of main.

Example usage:

```
NU32_Startup();
```

void NU32_ReadUART1(char *string, int maxLength) This function takes string (a pointer to the first element of an array of char) and maxLength, the maximum length of string input from the user. It fills string with characters received from the host via UART1 until a newline \n or carriage return \r is received. If the string exceeds maxLength, the new characters simply wrap around to the beginning of the string.

Example usage:

```
char message[100], str[100];
int i;
NU32_ReadUART1(message, 100);
sscanf(message, "%s %d", str, &i);  // if message expected to have a string and int
```

void NU32_WriteUART1(const char *string) This function writes a string over UART1.

Example usage:

```
char msg[100];
sprintf(msg,"The value is %d.\n",22);
NU32_WriteUART1(msg);
```

void NU32_EnableUART1Interrupt() This function causes UART1 to generate an interrupt when it receives a character from the host. We will discuss interrupts in detail in Chapter 6. For now, suffice to say that an interrupt causes the CPU to stop what it is doing and jump to an *interrupt service routine* (ISR). The ISR must be written by the user to read the character, clear the interrupt flag, and do something with the data. UART1 is typically used either in interrupt mode or in NU32_ReadUART1 / NU32_WriteUART1 mode.

Example usage:

```
// the user-defined interrupt service routine
void __ISR(_UART_1_VECTOR, IPL2SOFT) IntUart1Handler(void) {
  char data;

  if (INTGetFlag(INT_SOURCE_UART_RX(UART1))) {  // RX interrupt?
    data = UARTGetDataByte(UART1);              // get the data
```

```
    PutCharacter(UART1,data);                    // do something, your choice
    INTClearFlag(INT_SOURCE_UART_RX(UART1));     // clear interrupt flag
  }
  if (INTGetFlag(INT_SOURCE_UART_TX(UART1))) {  // ignore TX interrupts
    INTClearFlag(INT_SOURCE_UART_TX(UART1));
  }
}

int main() {
  // ...
  NU32_EnableUART1Interrupt();
  // ...
}
```

void NU32_DisableUART1Interrupt() Disable the UART1 interrupt to return to the `NU32_ReadUART1` / `NU32_WriteUART1` mode.

Example usage:

```
NU32_DisableUART1Interrupt();
```

void WriteString(UART_MODULE id, const char *string) This is an alternative to `NU32_WriteUART1` that can write to any of the UARTs.

Example usage:

```
WriteString(UART2, "here's a string!");
```

void PutCharacter(UART_MODULE id, const char character) This puts a single character out to any of the UARTs.

Example usage:

```
char ch = 'k';
PutCharacter(UART4,ch);
```

If you are using `NU32_ReadUART1` or `NU32_WriteUART1`, your program will hang if an open serial port is not connecting the host to the PIC32's UART1.

4.3 Bootloaded and Standalone Programs Throughout the Book

Throughout the remainder of this book, the first two lines of all C files with a `main` function will be

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.h, config bits, constants, funcs for startup and UART
```

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

```
NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
```

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 define `NU32_STANDALONE` or call `NU32_Startup()`.

Even if the program does not need any of the functions in the NU32 library, we use the lines above for consistency. This allows the same code to build correctly whether it is built to be bootloaded (do not uncomment the first line) or standalone (uncomment the first line). It also makes explicit that the program performs some initialization of the NU32. Whether the program is bootloaded or standalone, including `"NU32.h"` and executing `NU32_Startup()` does the following things:

- the constants `NU32LED1`, `NU32LED2`, `NU32USER`, and `SYS_FREQ` are made available

- access is given to the NU32 library commands described above
- the prefetch cache is enabled and the flash wait cycles are set to the minimum (this is redundant for bootloaded programs)
- pins RA4 and RA5 are configured as outputs to control LED1 and LED2
- interrupts are enabled and UART1 and UART3 are set up for communication with the host
- if `NU32_STANDALONE` is defined, the device configuration bits are set in `NU32.h`

As always, if the project is built to be bootloaded, it must contain the `NU32bootloaded.ld` linker file. If standalone, the default linker script should be used.

If you are writing programs for another development board, simply replace the `NU32.h` file, the `#include "NU32.h"` statement, and the `NU32_Startup();` statement with the equivalents for your board. Microchip often uses a generic file called `HardwareProfile.h` that includes a specific header file depending on a constant you have defined to the preprocessor (much like `NU32_STANDALONE` in our examples).

A C source file without a `main` function, i.e., a library helper file `helper.c`, should include `plib.h` if any of Microchip's SFR definitions or functions are used, and `NU32.h` if any of the constants or function prototypes in `NU32.h` are used. Alternatively, it could simply include `helper.h` which then includes `plib.h` and `NU32.h`.

4.4 An LCD Library

A dot matrix LCD screen is a cheap and portable device to display information to the user. Such LCD screens cost less than $10 each. In this section we give a simple library to interface the NU32 with a 16×2 LCD screen equipped with a Hitachi HD44780 LCD controller, one of the most common LCD controllers.

The pinout of the HD44780 is given below:

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

The LCD is powered by VCC and GND, where VCC could be between 3.3 V and 5 V; let's assume 5 V. The R/W, or Read/Write, input determines whether the LCD is in write mode (R/W = 0) or read mode (R/W = 1). Since we will only write to the LCD, we connect R/W to GND. BL+, the anode of the backlight for the LCD, is connected to VCC and BL−, the cathode of the backlight, is connected through a resistor R1 to ground. The VO, or contrast adjustment, input is connected to GND through a resistor R2. The resistors R1 and R2 should be chosen for proper brightness and contrast according to the particular LCD model. Reasonable first guesses are R1 = 100 ohms and R2 = 1000 ohms.

The LCD library assumes the rest of the pins are connected to the NU32 according to the following table:

RS	CLK	D0	D1	D2	D3	D4	D5	D6	D7
G12	G15	E0	E1	E2	E3	E4	E5	E6	E7

The RS pin tells the HD44780 whether the bits on pins D0..D7 are data or a command. On a falling edge of CLK, the information on the RS and D0..D7 pins is clocked into the HD44780 and the LCD screen is updated. Based on the connections above, the LCD library provides three functions: `LCDSetup()`, which initializes the necessary Port E and Port G pins as outputs and clears the screen; `LCDClear(int line)`, which clears line 1, 2, or both lines if `line` is any other value; and `LCDWriteString(char *str, int row, int col)` which writes the string `str` starting at `row` 1 or 2 and `col` 1 to 16. The header file and C file that make these functions available are given below.

Code Sample 4.2. `LCD.h`. The LCD library header file.

```
#ifndef LCD_H
#define LCD_H

// Initialize the LCD
void LCDSetup();

// Write a string to the LCD starting at (row, col).  row and col start at 1.
void LCDWriteString(char* str, int row, int col);

// Clears the designated line in the LCD.  0 clears both lines.
void LCDClear(int line);

#endif
```

Code Sample 4.3. `LCD.c`. The LCD library C source code.

```
#include <plib.h>
#include "LCD.h"
#define CLK LATGbits.LATG15         // CLK falling edge sends data to controller

void LCDWriteChar(char c);          // write a char to the LCD's cursor position
void LCDcommand(int command, int d7, int d6, int d5, int d4, int d3, int d2, int d1, int d0);
void wait();                        // make sure clock pulses long enough

void LCDSetup() {
  TRISGbits.TRISG12 = 0;            // G12 is output to LCD RS pin (cmd or data)
  TRISGbits.TRISG15 = 0;            // G15 is output to LCD CLK line
  TRISECLR = 0xFF;                  // RE0..7 are outputs to LCD D0..7
  LCDcommand(0, 0,0,1,1,1,0,0,0);   // initialize 2 lines
  LCDcommand(0, 0,0,0,0,0,0,0,1);   // clear screen
  LCDcommand(0, 0,0,0,0,0,1,1,0);   // cursor moves right
  LCDcommand(0, 0,0,0,0,1,1,0,0);   // restore screen
}

void LCDWriteChar(char c) {         // send 8 bits of data for char c
  LCDcommand(1, c>>7&1, c>>6&1, c>>5&1, c>>4&1, c>>3&1, c>>2&1, c>>1&1, c&1);
}

void LCDWriteString(char *str, int row, int col) {
  row--; col--;                     // LCD uses rows 0-1, cols 0-15
  LCDcommand(0,1,row,0,0,col>>3&1,col>>2&1,col>>1&1,col&1);
  while(*str) LCDWriteChar(*str++); // increment string pointer after char sent
}

void LCDClear(int line) {
  switch(line) {
    case 1:                         // clear line 1
        LCDWriteString("                ", 1, 1);
        break;
    case 2:                         // clear line 2
        LCDWriteString("                ", 2, 1);
        break;
    default:                        // clear both lines
        LCDcommand(0,0,0,0,0,0,0,0,1);
  }
}
```

```
void LCDcommand(int command, int d7, int d6, int d5, int d4, int d3, int d2, int d1, int d0) {
  LATGbits.LATG12 = command;        // 0 for command, 1 for data
  LATECLR = 0xFF;                   // clear bits E0-7, then write data to them
  LATE = LATE | d0 | (d1<<1) | (d2<<2) | (d3<<3) | (d4<<4) | (d5<<5) | (d6<<6) | (d7<<7);
  CLK = 1; wait();                  // set CLK high and wait
  CLK = 0; wait();                  // data sent to HD44780 on CLK falling edge
}

void wait() {                       // ensure pulse is long enough
  int i = 0;
  for (; i<10000; i++);
}
```

The program `LCDwrite.c` uses both the NU32 and LCD libraries to accept a string from the user's host computer and write it to the LCD. To build the executable, you need the source files `LCDwrite.c`, `NU32.c`, and `LCD.c`; the header files `NU32.h` and `LCD.h`; and the linker script `NU32bootloaded.ld` (if the program is to be bootloaded). After building, loading, and running the program, it writes the following string to the host computer:

```
String to send to LCD:
```

If the user responds `Echo!!`, the LCD prints

```
Echo!!__________
__Command___1___
```

where the underscores represent blank spaces. As the user sends more strings, the Command number increments. The code listing is given below.

Code Sample 4.4. `LCDwrite.c`. Takes input from the user and prints it to the LCD screen.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.h, config bits, constants, funcs for startup and UART
#include "LCD.h"

int main() {
  char msg[20];
  int i=1;

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  LCDSetup();
  while (1) {
    NU32_WriteUART1("String to send to LCD: ");
    NU32_ReadUART1(msg,16);            // get msg string from the user
    LCDClear(0);                       // clear LCD screen
    LCDWriteString(msg,1,1);           // write msg at row 1 col 1
    sprintf(msg,"Command %3d",i++);    // increment i after making string
    LCDWriteString(msg,2,3);           // write new msg at row 2 col 3
  }
}
```

4.5 Microchip Libraries

Microchip provides a number of libraries for PIC32s. Understanding these libraries is rather confusing (as we began to see in Chapter 3), in part because they are written to support a large number of models of PIC32s,

and in part because of the requirement to maintain some backward compatibility, so that code written a few years ago does not become obsolete with new library releases.

First, a little history. Historically, microcontrollers were primarily programmed in assembly language, so that the interaction between the code and the hardware is quite direct: typically a single assembly command is executed each clock cycle, and there are no hidden steps. For complex software projects, however, assembly code is cumbersome, in part because it is not particularly portable; the assembly language for one microcontroller may be different from another's.

The C language, while still being relatively low-level, provides a standard allowing a level of portability and abstraction. Much of your C code can work for a variety of microcontrollers, provided you have a compiler for your particular microcontroller. Still, if your code directly manipulates a particular SFR that doesn't exist on another microcontroller model, portability is broken.

Microchip software addresses this issue by providing a range of software services that allows the code that you write (the user's application) to be portable across 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.

Microchip's most recent software release, Harmony, provides middleware and device drivers that work with peripheral libraries. This permits the most abstract programming model, partially insulating the programmer from the hardware details. In this book, however, we only make use of low-level peripheral libraries, SFR variable declarations, etc., in the XC32 distribution, which is independent of Harmony. The philosophy is to stay close to the hardware level, similar to assembly language programming, but with the benefits of the more portable, higher-level C language. This approach is further supported by the fact that the PIC32 hardware is currently better documented (in the Data Sheets and Reference Manual) than the middleware and device driver software.

The Microchip peripheral libraries in the XC32 distribution, accessible by `#include <plib.h>`, consist of a large number of functions, macros, constants, SFR variable declarations, and data types to simplify programming of the PIC32. These can be found in the header files in `pic32mx/include/peripheral` and in C source code in `pic32-libs/peripheral` (more specifically, the `.a` libraries built from that C source code).

Where possible and convenient, code in this book directly manipulates the SFR variables. Code statements that directly manipulate SFRs are accompanied by comments to make the purpose clear to the reader. An example is the following line from `simplePIC.c`:

```
LATAINV = 0x0030;    // toggle the two lights
```

In some cases, however, higher-level library functions and macros are used, particularly when they conveniently perform several steps or contain assembly code that would otherwise be cumbersome to write ourselves. An example is in `simplePIC_standalone.c`:

```
SYSTEMConfig(SYS_FREQ, SYS_CFG_ALL); // cache on, PBCLK setup, min flash wait
```

This macro, in `pic32mx/include/peripheral/system.h`, performs a number of steps to turn on the prefetch cache and to configure the peripheral bus clock and the flash wait states.

Finding the definition of a particular library function, variable, macro, constant, or data type in the Microchip code is not easy due to the large number of files and the long chains of includes. Fortunately the MPLAB X IDE simplifies this problem. For any function, constant, etc., in your program, right-click on the symbol in the source listing and choose **Navigate > Go to Declaration**. The IDE will open the file where the symbol is declared and take you to the declaration.

Another thing you can try is **Window > Classes**. This will open a window with all peripheral library data types, SFR variable declarations, and function prototypes. Double-clicking on one will take you to the file where it is declared.

4.6 Chapter Summary

- A library is often considered a `.a` archive of `.o` object codes and an associated `.h` header file that give user programs access to function prototypes, constants, macros, data types, and variables associated

with the object codes. In this chapter we call a `.c` helper file and an associated `.h` file a library.

- For a helper library, the `helper.h` header file can be included by both `helper.c` as well as the `main` C file which uses the helper library. The header file `helper.h` should contain function prototypes, constants, etc., that are meant to be public. Function prototypes and variables that are meant to be private to `helper.c` should be defined in `helper.c`, not `helper.h`.
- 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, provided by the header files, are needed. The function calls are resolved to the proper virtual address when the multiple object codes are linked. If more than one object code has a `main()` function, 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.
- The Microchip "peripheral library" consists of a large set of header files and object code to be used with your programs. Some peripheral library functions and macros, such as those that execute several steps or manipulate the CPU's CP0 register, are particularly useful. Other functions and macros are less useful; it may be easier to set the values of SFRs based on reading the Data Sheet and Reference Manual than to find the equivalent peripheral library functions.

4.7 Exercises

1. Explain what can go wrong if a header file contains the global variable definition `int i=2;` if that header file is included by more than one C file in the same project.
2. Identify which, if any, functions, constants, and global variables in `NU32.c` are private to `NU32.c`.
3. You will create your own libraries.
   (a) Strip out all the comments from `invest.c` in the Appendix. 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_ReadUART1` and `NU32_WriteUART1`. 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) Now break `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 safety of future users of the `helper` library, make sure to 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) Now break `invest.c` into three files: `main.c`, `io.c`, and `calculate.c`. Any function which deals with 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 on your header files. Verify that your code works. Turn in your code and a separate paragraph justifying your choice of header files.

   If you prefer, you are welcome to first solve the tasks using a C installation on your computer, then modify the input/output functions for the NU32.
4. 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 doesn't behave as expected. Say you're 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.

5. Take a look at `pic32mx/include/peripheral/uart.h`, a header file for the UART library with associated C code at `pic32-libs/peripheral/uart/source/uart_lib.c`. From the header file, give one example of as many of these definitions that you can find: constant, macro (a `#define` that takes at least one argument), new data type, function prototype, and inline function. (An inline function can be defined right in the header file; it does not need to be a prototype for a function in a C file. Code calling the inline function is replaced by the definition in the header file. This can potentially save a small amount of time associated with jumping to and returning from a true function.)