- Including the file `xc.h` gives our program access to variables, data types, and constants that significantly simplify programming by allowing us to access SFRs easily from C code without needing to specify addresses directly.
- The included file `pic32mx/include/proc/p32mx795f512h.h` contains variable declarations, like TRISF, that allow us to read from and write to the SFRs. We have several options for manipulating these SFRs. For TRISF, for example, we can directly assign the bits with `TRISF=0x3`, or we can use bitwise operations like `&` and `|`. Many SFRs have associated CLR, SET, and INV registers which can be used to efficiently clear, set, or invert certain bits. Finally, particular bits or groups of bits can be accessed using bit fields. For example, we access bit 3 of TRISF using TRISFbits.TRISF3. The names of the SFRs and bit fields follow the names in the Data Sheet (particularly the Memory Organization section) and Reference Manual.
- All programs are linked with `pic32mx/lib/proc/32MX795F512H/crt0_mips32r2.o` to produce the final `.hex` file. This C run-time startup code executes first, doing things like initializing global variables in RAM, before jumping to the `main` function. Other linked object code includes `processor.o`, with the VAs of the SFRs.
- Upon reset, the PIC32 jumps to the boot flash address 0xBFC00000. For a PIC32 with a bootloader, the `crt0_mips32r2` of the bootloader is installed at this address. When the bootloader completes, it jumps to an address where the bootloader has previously installed a bootloaded executable.
- When the bootloader was installed with a device programmer, the programmer set the Device Configuration Registers. In addition to loading or running executables, the bootloader enables the prefetch cache module and minimizes the number of CPU wait cycles for instructions to load from flash.
- A bootloaded program is linked with a custom linker script, like `NU32bootloaded.ld`, to make sure the flash addresses for the instructions do not conflict with the bootloader's, and to make sure that the program is placed at the address where the bootloader jumps.

3.10 Exercises

1. Convert the following virtual addresses to physical addresses, and indicate whether the address is cacheable or not, and whether it resides in RAM, flash, SFRs, or boot flash. (a) 0x80000020. (b) 0xA0000020. (c) 0xBF800001. (d) 0x9FC00111. (e) 0x9D001000.
2. Look at the linker script used with programs for the NU32. Where does the bootloader install your program in virtual memory? (Hint: look at the `_RESET_ADDR`.)
3. Refer to the Memory Organization section of the Data Sheet and Figure 2.1.
   a. Referring to the Data Sheet, indicate which bits, 0-31, can be used as input/outputs for each of Ports B through G. For the PIC32MX795F512H in Figure 2.1, indicate which pin corresponds to bit 0 of port E (this is referred to as RE0).

   b. The SFR INTCON refers to "interrupt control." Which bits, 0-31, of this SFR are unimplemented? Of the bits that are implemented, give the numbers of the bits and their names.
4. Modify `simplePIC.c` so that both lights are on or off at the same time, instead of opposite each other. Turn in only the code that changed.
5. Modify `simplePIC.c` so that the function `delay` takes an `int cycles` as an argument. The `for` loop in `delay` executes `cycles` times, not a fixed value of 1,000,000. Then modify `main` so that the first time it calls `delay`, it passes a value equal to `MAXCYCLES`. The next time it calls `delay` with a value decreased by `DELTACYCLES`, and so on, until the value is less than zero, at which time it resets the value to `MAXCYCLES`. Use `#define` to define the constants `MAXCYCLES` as 1,000,000 and `DELTACYCLES` as 100,000. Turn in your code.
6. Give the VAs and reset values of the following SFRs. (a) I2C3CON. (b) TRISC.
7. The `processor.o` file linked with your `simplePIC` project is much larger than your final `.hex` file. Explain how that is possible.
8. The building of a typical PIC32 program makes use of a number of files in the XC32 compiler distribution. Let us look at a few of them.
   a. Look at the assembly startup code `pic32-libs/libpic32/startup/crt0.S`. Although we are not studying assembly code, the comments help you understand what the startup code does. Based on the comments, you can see that this code clears the RAM addresses where uninitialized global variables are stored, for example. Find and list the line(s) of code that call the user's `main` function when the C runtime startup completes.
   b. Using the command `xc32-nm -n processor.o`, give the names and addresses of the five SFRs with the highest addresses.
   c. Open the file `p32mx795f512h.h` and go to the declaration of the SFR SPI2STAT and its associated bit field data type __SPI2STATbits_t. How many bit fields are defined? What are their names and sizes? Do these coincide with the Data Sheet?
9. Give three C commands, using TRISDSET, TRISDCLR, and TRISDINV, that set bits 2 and 3 of TRISD to 1, clear bits 1 and 5, and flip bits 0 and 4.

Further Reading

MPLAB XC32 C/C++ compiler user's guide. (2012). Microchip Technology Inc.
MPLAB XC32 linker and utilities user guide. (2013). Microchip Technology Inc.
PIC32 family reference manual. Section 32: Configuration. (2013). Microchip Technology Inc.

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.