Contents

Part I

Quickstart

Chapter 1

Quickstart

Edit, compile, run, repeat: familiar to generations of C programmers, this mantra applies to programming in C, regardless of platform. Architecture. Program loading. Input and Output. These details differ between your computer and the PIC32. Architecture refers to processor type: your computer's x86-64 CPU and the PIC32's MIPS32 CPU understand different machine code and therefore require different compilers. Your computer's operating system allows you to seamlessly run programs; the PIC32's *bootloader* writes programs it receives from your computer to flash memory and executes them when the PIC32 resets.[1] You interact directly with your computer via the screen and keyboard; you interact indirectly with the PIC32 using a *terminal emulator* to relay information between your computer and the microcontroller. As you can see, programming the PIC32 requires attention to details that you probably ignore when programming your computer.

Armed with an overview of the differences between computer programming and microcontroller programming, you are ready to get your hands dirty. The rest of this chapter will guide you through gathering the hardware and installing the software necessary to program the PIC32. You will then verify your setup by running two programs on the PIC32. By the end of the chapter, you will be able to compile and run programs for the PIC32 as easily as you compile and run programs for your computer!

1.1 What You Need

This section explains the hardware and software that you need to program the PIC32. Links to purchase the hardware and download the software are provided at the book's website, `http://hades.mech.northwestern.edu/index.php/Pic32book`.

1.1.1 Hardware

Although PIC32 microcontrollers integrate many devices on a single chip, they also require external circuitry to function. The NU32 development board, shown in Figure 1.1, provides this circuitry and more: buttons, LEDs, breakout pins, USB ports, and virtual USB serial ports. The examples in this book assume that you use this board. You will also need the following hardware:

1. **Computer with a USB port.** The host computer is used to create PIC32 programs. The examples in this book work with the Linux, Windows, and Mac operating systems.

2. **USB A to mini–B cable.** This cable carries signals between the NU32 board and your computer.

3. **AC/DC adapter (6 Volts).** This cable provides power to the PIC32 and NU32 board.

[1] Your computer also has a bootloader. It runs when you turn the computer on and loads the operating system. Also, operating systems are available for the PIC32, but we will not use them in this book.

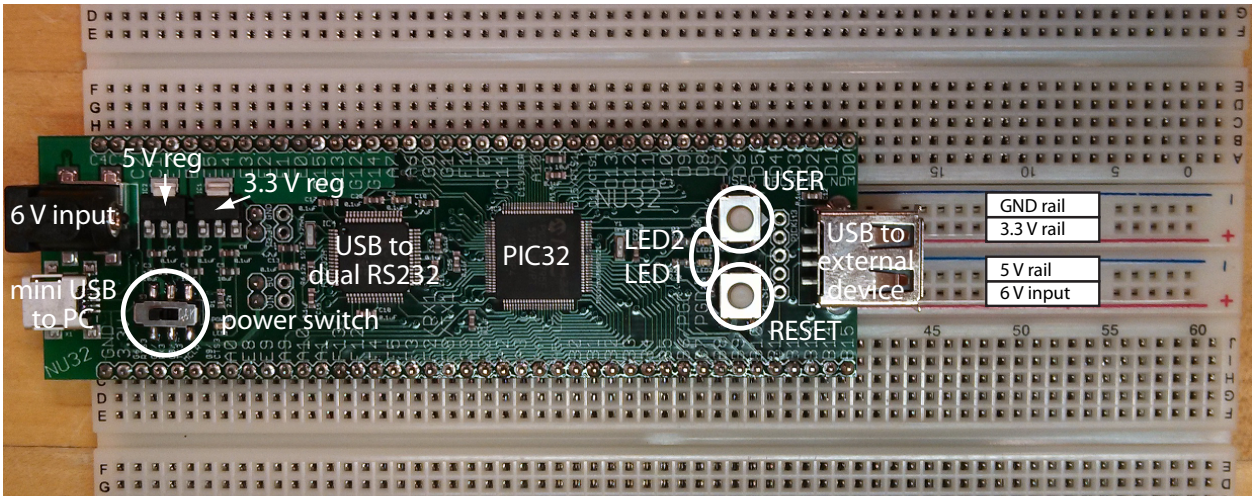


Figure 1.1: A photo of the NU32 development board.

1.1.2 Software

Programming the PIC32 requires various software. You should be familiar with some of the software from programming your computer in C; if not, refer to Appendix A. For your convenience, we have aggregated the software you need at the book's website, http://hades.mech.northwestern.edu/index.php/Pic32book. You should download and install all of the following software.

1. **The command prompt** allows you to control your computer using a text-based interface. This program, `cmd.exe` on Windows, `Terminal` on Mac, and `bash` on Linux, comes with your operating system so you should not need to install it. See Appendix A for more information about the command line.

2. **A text editor** allows you to create text files, such as those containing C source code. See Appendix A for more information.

3. **A native C compiler** converts human-readable C source code files into machine code that your computer can execute. We suggest the free GNU compiler, `gcc`, which is available for Windows, Mac, and Linux. See Appendix A for more information.

4. **Make** simplifies the build process by automatically executing the instructions required to convert source code into executables. After manually typing all of the commands necessary create your first program, you will appreciate `make`.

5. **The Microchip XC32 compiler** converts C source files into machine code that the PIC32 understands. This compiler is known as a *cross compiler* because it runs on one processor architecture (e.g., x86-64 CPU) and creates machine code for another (e.g., MIPS32). This compiler installation also includes C libraries to help you control PIC32-specific features. Note where you install the compiler; we will refer to this directory as `<xc32dir>`. If you are asked during installation whether you would like to add xc32 to your path variable, do so.

6. **MPLAB Harmony** is Microchip's collection of libraries and drivers that simplify the task of writing code targeting multiple PIC32 models. We will use this library only in the more advanced chapters; however, you should install it now. Note the installation directory, which we will refer to as `<harmony>`.

7. **The FTDI Virtual COM Port Driver** allows you to use a USB port as a "virtual serial communication (COM) port" to talk to the NU32 board. This driver is already included with most Linux distributions, but Windows and Mac users will need to install it.

8. **A terminal emulator** provides a simple interface to a COM port on your computer, sending keyboard input to the PIC32 and displaying output from the PIC32. For Windows we recommend `PuTTY`, and for Linux/Mac you can use the built-in `screen` program. On Windows, remember where you download PuTTY; we refer to this directory as `<puttyPath>`.

9. **The PIC32 quickstart code** contains source code and other support files to help you program the PIC32. Download `PIC32quickstart.zip` from the book's website, extract it, and put it in a directory that you create. We will refer to this directory as `<PIC32>`. In `<PIC32>` you will keep the quickstart code, plus all of the PIC32 code you write, so make sure the directory name makes sense to you. For example, depending on your operating system, `<PIC32>` could be `/Users/kevin/PIC32` or `C:\Users\kevin\Documents\PIC32`. In `<PIC32>`, you should have the following three files and one directory:

   - `nu32utility.c`: a program for your computer, used to load PIC32 executable programs from your computer to the PIC32
   - `simplePIC.c, talkingPIC.c`: PIC32 sample programs that we will test in this chapter
   - `skeleton`: a directory containing
     - `Makefile`: a file that will help us compile future PIC32 programs
     - `NU32.c`, `NU32.h`: a library of useful functions for the NU32 board
     - `NU32bootloaded.ld`: a linker script used when compiling programs for the PIC32

   We will learn more about each of these shortly.

You should now have code in the following directories (plus, if you are a Windows user, you will have PuTTY in the directory `<puttyPath>`):

- `<xc32dir>`. **You will never modify code in this directory.** Microchip wrote this code, and there is no reason for you to change it. Depending on your operating system, your `<xc32dir>` could look something like the following:
  - `/Applications/microchip/xc32`
  - `C:\Program Files (x86)\Microchip\xc32`
- `<harmony>`. **You will never modify code in this directory.** Depending on your operating system, your `<harmony>` could look something like the following:
  - `/Users/kevin/microchip/harmony`
  - `C:\microchip\harmony`
- `<PIC32>`. Where PIC32 quickstart code, and code you will write, is stored, as described above.

Now that you have installed all of the necessary software, it is time to program the PIC32. By following these instructions, not only will you run your first PIC32 program, you will also verify that all of the software and hardware is functioning properly. Do not worry too much about what all the commands mean, we will explain the details in subsequent chapters.

Notation: Wherever we write `<something>`, replace it with the value relevant to your computer. On Windows, use a backslash (`\`) and on Linux/Mac use a slash (`/`) to separate the directories in a path. At the command line, place paths that contain spaces between quotation marks (i.e. `"C:\Program Files"`). Enter the text following a `>` at the command line. Use a single line, even if the command spans multiple lines in the book.

1.2 Compiling The Bootloader Utility

The bootloader utility, located at `<PIC32>/nu32utility.c`, sends compiled code to the PIC32. To use the bootloader utility you must compile it. Navigate to the `<PIC32>` directory by typing:

```
> cd <PIC32>
```

Verify that `<PIC32>/nu32utility.c` exists by executing the following command, which lists all the files in a directory:

- **Windows**
  `> dir`
- **Linux/Mac**
  `> ls`

Next, compile the bootloader utility using the native C compiler `gcc`:

- **Windows**
  `> gcc nu32utility.c -o nu32utility -lwinmm`
- **Linux/Mac**
  `> gcc nu32utility.c -o nu32utility`

When you successfully complete this step the file `nu32utility` will be created. Verify that it exists by listing the files in `<PIC32>`.

1.3 Compiling Your First Program

The first program you will load onto your PIC32 is `<PIC32>/simplePIC.c`, which is listed below. We will scrutinize the source code in Ch. 3, but reading it now will help you understand how it works. Essentially, after some setup, the code enters an infinite loop that alternates between delaying and toggling two LEDs. The delay loops infinitely while the `USER` button is pressed.

Code Sample 1.1. `simplePIC.c`. Blinking lights on the NU32, unless the `USER` button is pressed.

```
#include <xc.h>         // Load the proper header for the processor

void delay(void);

int main(void) {
  DDPCONbits.JTAGEN = 0; // Disable JTAG, make pins 4 and 5 of Port A available.
  TRISA = 0xFFCF;        // Pins 4 and 5 of Port A are LED1 and LED2.  Clear
                         // bits 4/5 to zero, for output.  Others are inputs.
  LATAbits.LATA4 = 0;    // Turn LED1 on and LED2 off.  These pins sink ...
  LATAbits.LATA5 = 1;    // ... current on NU32, so "high" = "off."

  while(1) {
    delay();
    LATAINV = 0x0030;    // toggle LED1 and LED2
  }
  return 0;
}

void delay(void) {
  int j;
  for (j = 0; j < 1000000; j++) { // number is 1 million
    while(!PORTDbits.RD13) {
        ;   // Pin D13 is the USER switch, low if pressed.
```

```
    }
  }
}
```

To compile this program you will use the `xc32-gcc` cross compiler, which compiles code for the PIC32's MIPS32 processor. This compiler and other Microchip tools are located at `<xc32dir>/<xc32ver>/bin`, where `<xc32ver>` refers to the xc32 version (e.g. 1.34). To find `<xc32ver>` list the contents of the Microchip XC32 directory, e.g.,

```
> ls <xc32dir>
```

The subdirectory displayed is your `<xc32ver>` value. If you happen to have installed two or more versions of XC32, you will always use the most recent version (the largest version number).

Next you will compile `simplePIC.c` and create the executable *hex file*. This is a two-step process; first you create the `simplePIC.elf` file and then you create the `simplePIC.hex` file. This will be discussed more in Chapter 3.

```
> <xc32dir>/<xc32ver>/bin/xc32-gcc -mprocessor=32MX795F512L
    -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
> <xc32dir>/<xc32ver>/bin/xc32-bin2hex simplePIC.elf
```

The `-Wl` is "-W ell" not "-W one." You can list the contents of `<PIC32>` to make sure both `simplePIC.elf` and `simplePIC.hex` were created. The hex file contains PIC32 machine code in a format that the PIC32 understands.

If, when you installed XC32, you selected to have `xc32` added to your path, then in the two commands above you could have simply typed

```
> xc32-gcc -mprocessor=32MX795F512L
   -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
> xc32-bin2hex simplePIC.elf
```

and your operating system would be able to find these programs without needing the full paths to them.

Next, you will load `simplePIC.hex` onto the PIC32 using the bootloader utility.

1.4 Loading Your First Program

Loading a program onto the PIC32 from your computer requires communication between the two devices. When the PIC32 is powered and connected to a USB port, your computer creates two new serial communication (COM) ports. Depending on your specific system setup, these COM ports will have different names. Therefore, we will determine the names of your COM ports through experimentation. First, with the PIC32 unplugged, execute the following command to enumerate the current COM ports, and take note of the names that are listed:

- **Windows**:
  `> mode`
- **Mac**:
  `> ls /dev/tty.*`
- **Linux**:
  `> ls /dev/ttyUSB*`

Next, plug-in the NU32 board to the wall using the AC adapter, turn the power switch on, and verify that the red "power" LED illuminates. Connect the USB cable from the NU32's mini B USB jack (next to the power jack) to a USB port on the host computer. Repeat the steps above, and note that two new COM ports appear. If they do not appear, make sure that you installed the FTDI driver from the Section 1.1.2. The names of the ports will differ depending on the operating system; therefore we have listed some typical names:

- **Windows**: `COM4 COM5`

- **Mac**: `/dev/tty.usbserial-00001024A /dev/tty.usbserial-00001024B`
- **Linux**: `/dev/ttyUSB0 /dev/ttyUSB1`

Your computer, upon detecting the NU32 board, has created both of these ports. Your programs use the lower port, `<COMportA>`, and the bootloader uses the higher port, `<COMportB>`.[2].[3][4]

After identifying the COM ports, place the PIC32 into *program receive mode*. Locate the `RESET` button and the `USER` button on the NU32 board (Figure 1.1). The `RESET` button is next to pins B8, B9, B1, on the board's bottom left (the power jack is the board's top) and the `USER` button is next to pins D5, D6, and D7 on the board's bottom right. Press and hold both buttons, release `RESET`, and then release `USER`. After completing this sequence, the PIC32 will flash LED1, indicating that it has entered program receive mode.

Assuming that you are still in the `<PIC32>` directory, type

- **Windows**
  `nu32utility <COMportB> simplePIC.hex`
- **Linux/Mac**
  `> ./nu32utility <COMportB> simplePIC.hex`

to start the loading process. After the utility finishes, `LED1` and `LED2` will flash back and forth. Hold `USER` and notice that the LEDs stop flashing. Release `USER` and watch the flashing resume. Turn the PIC32 off and then on. The LEDs resume blinking because you have written the program to the PIC32's nonvolatile flash memory. Congratulations, you have successfully programmed the PIC32!

1.5 The Build Process

As you just witnessed, building an executable for the PIC32 requires several steps. Fortunately, you can use `make` to simplify this otherwise tedious and error-prone procedure. Using `make` requires a `Makefile`, which contains instructions for building the executable. We have provided a `Makefile` in `<PIC32>/skeleton`. Prior to using `make`, you need to modify `<PIC32>/skeleton/Makefile` so that it contains the paths and COM port specific to your system.

Aside from the paths you have already used, you need your terminal emulator's location, `<termEmu>`, and the Harmony version, `<harmVer>`. On Windows, `<termEmu>` is `<puttyPath>/putty.exe` and for Linux/Mac, `<termEmu>` is `screen`. To find Harmony's version, `<harmVer>`, list the contents of the `<harmony>` directory. Edit `<PIC32>/skeleton/Makefile` and update the first six lines as indicated below.

```
XC32PATH=<xc32dir>/<xc32ver>/bin
HARMONYPATH=<harmony>/<harmVer>
NU32PATH=<PIC32>
PORTA=<COMPortA>
PORTB=<COMPortB>
TERMEMU=<termEmu>
```

In the Makefile, do not surround paths with quotation marks, even if they contain spaces.

If your computer has more than one USB port, you should always use the same USB port to connect to your NU32. This is because the names of the COM ports that are created when you connect your NU32 may change if you use a different USB port. Since you are now creating a `Makefile` that you will use for all your projects in the future, you want to make sure that `COMPortA` and `COMPortB` are always correct.

After saving the `Makefile`, you can use the skeleton directory to easily create new PIC32 programs. The skeleton directory contains not only the `Makefile`, but also the NU32 library (`NU32.h and NU32.c`), and the linker script `NU32bootloaded.ld`, all of which will be used extensively throughout the book. The `Makefile` will automatically compile and link all `.c` files in the same directory into a single executable; therefore, your project directory should contain all the C files you need and none that you do not want!

[2] Windows: write the ports as `\\.\COMx` rather than `COMx`
[3] Mac: the bootloader port ends with "B".
[4] Linux: To avoid needing to execute commands as root, add yourself to the group that owns the COM port.

Each new project you create will have its own directory in <PIC32>, e.g., <PIC32>/<projectdir>. We now explain how to use the <PIC32>/skeleton directory to create a new project, using <PIC32>/talkingPIC.c as an example. For this example, we will name the project talkingPIC, so <projectdir> is talkingPIC. By following this procedure, you will have access to the NU32 library and will be able to avoid repeating the previous setup steps. First copy the <PIC32>/skeleton directory to the new project directory:

- **Windows**
  ```
  > mkdir <projectdir>
  > copy <PIC32>\skeleton\*.* <projectdir>
  ```
- **Linux/Mac**
  ```
  > cp -R <PIC32>/skeleton <projectdir>
  ```

Now copy the project source files, in this case just talkingPIC.c, to $<$ PIC32 $>$ / $<$ projectdir $>$, and change to that directory:

- **Windows**
  ```
  > copy talkingPIC.c <projectdir>
  > cd <projectdir>
  ```
- **Linux/Mac**
  ```
  > cp talkingPIC.c <projectdir>
  > cd <projectdir>
  ```

Before explaining how to use make, we will examine talkingPIC.c, which accepts input from and prints output to a terminal emulator running on the host computer. These capabilities facilitate user interaction and debugging. The source code for talkingPIC.c is listed below:

Code Sample 1.2. talkingPIC.c. The PIC32 echoes any messages sent to it from the host keyboard back to the host screen.

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

#define MAX_MESSAGE_LENGTH 200

int main(void) {
  char message[MAX_MESSAGE_LENGTH];

  NU32_Startup(); // cache on, interrupts on, LED/button init, UART init
  sprintf(message, "Hello World! \r\n");
  NU32_WriteUART1(message);
  while (1) {
    sprintf(message, "Hello?  What do you want to tell me?  ");
    NU32_WriteUART1(message);
    NU32_ReadUART1(message, MAX_MESSAGE_LENGTH);
    NU32_WriteUART1(message);
    NU32_WriteUART1("\r\n");
    NU32_LED1 = !NU32_LED1; // toggle the LEDs
    NU32_LED2 = !NU32_LED2;
  }
  return 0;
}
```

Notice the calls to sprintf. This function acts like printf except it writes to a string rather than the screen. The NU32 library function NU32_WriteUART1 sends the output of sprintf from the PIC32 to the computer via the serial port. Likewise, NU32_ReadUART1 allows the PIC32 to read data from the computer

via the serial port. If you removed the calls to NU32_WriteUART1, replaced NU32_ReadUART1 with scanf, and replaced sprintf with printf, you would be left with a rather simple program that prints what you type!

Now that you know how talkingPIC.c works, it is time to see it in action. First, make sure you are in the <projectdir> and build the project using make.

```
> make
```

This compiles and assembles all .c files into .o object files, links them into a single out.elf file, and turns that out.elf file into an executable out.hex file. You can do a directory listing to see all of these files.

Next, put the PIC32 into program receive mode (the RESET and USER buttons) and execute

```
> make write
```

to invoke the bootloader utility nu32utility and program the PIC32 with out.hex. When LED1 stops flashing, the PIC32 has been programmed.

To communicate with talkingPIC, you must connect to the PIC32 using your terminal emulator. Recall that the terminal emulator communicates with the PIC32 using <COMportA>. Enter the following command:

- **Windows**
  <puttyPath>\putty -serial <COMportA> -sercfg 230400
- **Linux/Mac**
  screen <COMportA> 230400

PuTTY will launch in a new window, whereas screen will use the command prompt window. The number 230400 in the above commands is the baud, the speed at which the PIC32 and computer communicate.

After connecting, press RESET to restart the program. You should see a message in the terminal. Start typing, and notice that no characters appear until you hit ENTER. This behavior may seem strange, but it occurs because the terminal emulator only displays the text it receives from the PIC32. The PIC32 does not send any text to your computer until it receives a special *control character*, which you generate by pressing ENTER.[5]

For example, when the PIC32 prompts you with a question

```
Hello?  What do you want to tell me?
```

and you type Everything! ENTER, the PIC32 will receive Everything!\r, write Everything!\r\n to the terminal emulator, and ask you for more input.

When you are done conversing with the PIC32, you can exit the terminal emulator. To exit screen type

```
CTRL-a k y
```

Note that CTRL and a should be pressed simultaneously. To exit PuTTY make sure the command prompt window is focused and type

```
CTRL-c
```

Rather then memorizing these rather long commands to connect to the serial port, you can use the Makefile.

To connect PuTTY to the PIC32 type

```
> make putty
```

To use screen type

```
> make screen
```

Your system is now configured for PIC32 programming. Although the build process may seem opaque, do not worry. For now it is only important that you can successfully compile programs and load them onto the PIC32. Later chapters will explain the details of the build process.

[5]Depending on the terminal emulator, ENTER may generate a carriage return (\r), newline (\n) or both. The terminal emulator moves the cursor to the leftmost column when it recieves a \r and to the next line when it receives a \n.

1.6 Chapter Summary

- To start a new project, copy the `<PIC32>/skeleton` directory to a new location, `<projectdir>`, and add your source code.
- In the directory `<projectdir>`, use `make` to compile your code.
- Put the PIC32 into program receive mode by pressing the `USER` and `RESET` button simultaneously, then releasing the `RESET` button, and finally releasing the `USER` button. Then use `make write` to load your program.
- Use a terminal emulator to communicate with programs running on the PIC32. Typing `make putty` or `make screen` will launch the appropriate terminal emulator and connect it to the PIC32.

Part II

Fundamentals

Chapter 2

Looking Under the Hood: Hardware

Now that you have some programs running, it's time to look under the hood. We begin with the PIC32 hardware by examining the PIC32MX795F512L in detail. We then describe the NU32 development board.

2.1 The PIC32

2.1.1 Pin Functions and Special Function Registers (SFRs)

The PIC32MX795F512L requires a supply voltage between 2.3 and 3.6 V and features a maximum clock frequency of 80 MHz, 512 KB of program memory (flash), and 128 KB of data memory (RAM). Its peripherals include a 10-bit analog-to-digital converter (ADC), many digital I/O pins, USB 2.0, Ethernet, two CAN modules, five I^2C and four SPI synchronous serial communication modules, six UARTs for RS-232 or RS-485 asynchronous serial communication, five 16-bit counter/timers (configurable to give two 32-bit timers and one 16-bit timer), five pulse-width modulation outputs, and several pins that can generate interrupts based on external signals. Whew. Don't worry if you don't know what all of these peripherals do, much of this book is dedicated to expaining them.

To cram so much functionality into only 100 pins, many pins serve multiple functions. See the pinout diagram for the PIC32MX795F512L (Figure 2.1). As an example, pin 21 can be an analog input, a comparator input, a change notification input (which can generate an interrupt when an input changes state), or a digital input or output.

Table 2.1 summarizes the pin functions; we indicated some of the most useful for embedded control in **bold**.

Which function a particular pin actually serves is determined by *Special Function Registers* (SFRs). Each SFR is a 32-bit word that sits at a memory address. The values of the SFR bits, 0 (cleared) or 1 (set), control the functions of the pins as well as other functions of the PIC32.

For example, pin 78 in Figure 2.1 can serve as OC4 (output compare 4) or RD3 (digital I/O number 3 on port D). Let's say we want to use it as a digital output. We can modify the SFRs that control this pin to disable the OC4 function and to choose the RD3 function as digital output instead of digital input. The PIC32MX5xx/6xx/7xx Data Sheet explains the memory addresses and meanings of the SFRs. Be careful, because it includes information for many different PIC32 models. Looking at the data sheet section on Output Compare reveals that the 32-bit SFR named "OC4CON" determines whether OC4 is enabled. Specifically, for bits numbered 0 ... 31, we see that bit 15 is responsible for enabling or disabling OC4. We refer to this bit as OC4CON$\langle 15 \rangle$. If it is cleared (0), OC4 is disabled, and if it is set (1), OC4 is enabled. So we clear this bit to 0. (Bits can be "cleared to 0" or simply "cleared," or "set to 1" or simply "set.") Now, referring to the I/O Ports section of the Data Sheet, we see that the input/output direction of Port D is controlled by the SFR TRISD, and bits 0-15 correspond to RD0-15. Bit 3 of the SFR TRISD, i.e., TRISD$\langle 3 \rangle$, should be cleared to 0 to make RD3 (pin 78) a digital output.

In fact, according to the Memory Organization section of the Data Sheet, OC4CON$\langle 15 \rangle$ is cleared by default on reset. On the other hand, TRISD$\langle 3 \rangle$ is set to 1 on reset, making pin 78 a digital input by default.

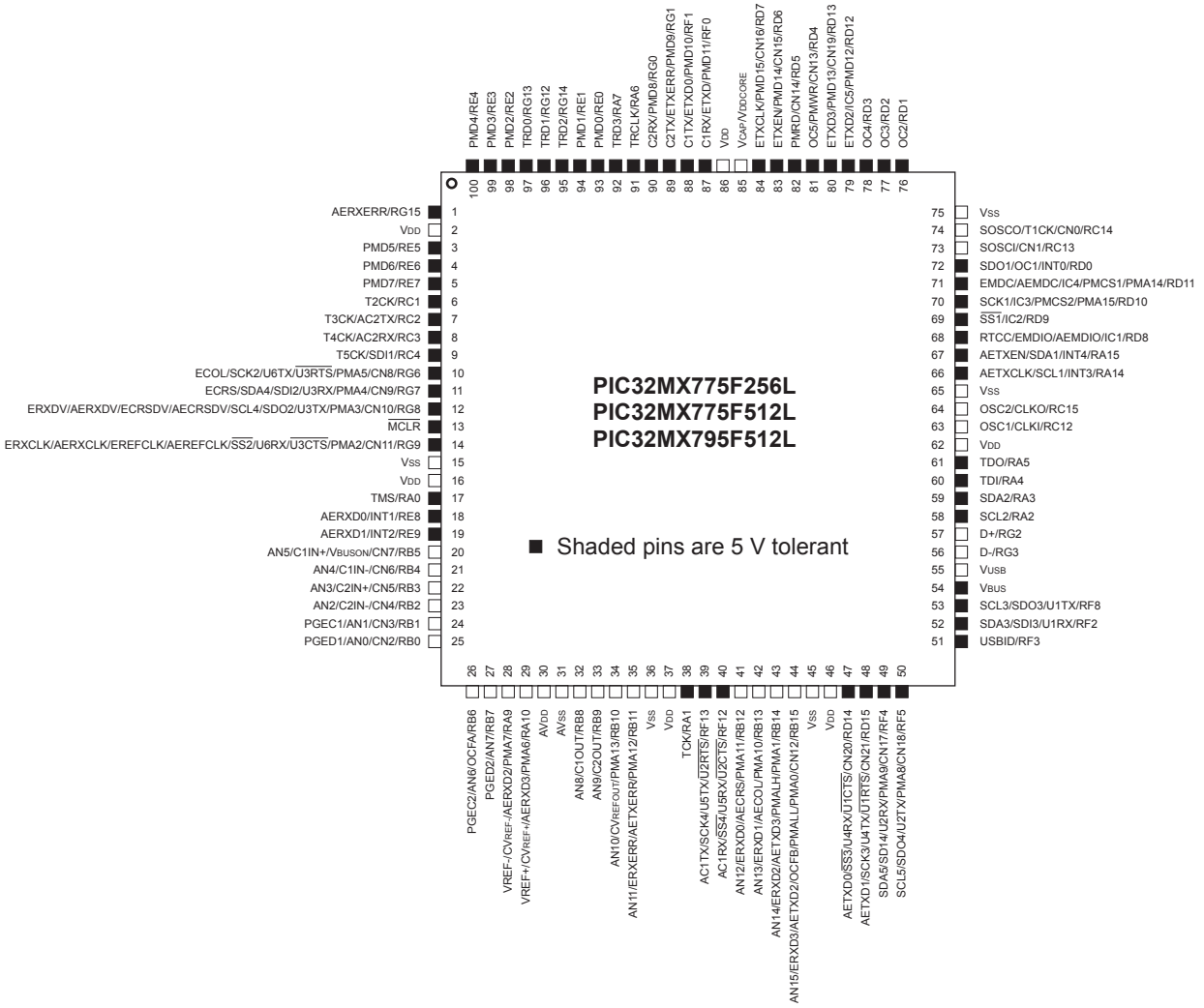


Figure 2.1: The pinout of the PIC32MX795F512L used on the NU32.

(This is for safety, to make sure the PIC32 does not impose an unwanted voltage on external circuitry upon startup.)

We will see SFRs again and again as we learn about the PIC32.

2.1.2 PIC32 Architecture

Figure 2.2 depicts the PIC32's architecture. Of course there is a CPU, program memory, and data memory. Perhaps most interesting to us, though, is the plethora of *peripherals*, which are what make microcontrollers useful for embedded control. From left to right, top to bottom, these peripherals consist of PORTA . . . PORTG, which are digital I/O ports; 22 change notification (CN) pins that generate interrupts when input signals change; five 16-bit counters (which can be used as one 16-bit counter and two 32-bit counters by chaining) that can be used for a variety of counting operations, and timing operations by counting clock ticks; five pins for output pulse-width modulation (PWM) pulse trains (or "output compare" OC); five pins for "input capture" (IC) which are used to capture timer values or trigger interrupts on rising or falling inputs; four SPI and five I^2C synchronous serial communication modules; a "parallel master port" (PMP) for parallel communication; an analog-to-digital converter (ADC) multiplexed to 16 input pins; six UARTs for asynchronous serial communication (e.g., RS-232, RS-485); a real-time clock and calendar (RTCC) that can maintain accurate

Pin Label	Function
ANx (x=0-15)	analog-to-digital (ADC) inputs
AVDD, AVSS	positive supply and ground reference for ADC
CxIN-, CxIN+, CxOUT (x=1,2)	comparator negative and positive input and output
CxRX, CxTx (x=1,2)	CAN receive and transmit pins
CLKI, CLKO	clock input and output (for particular clock modes)
CNx (x=0-21)	change notification; voltage changes on these pins can generate interrupts
CVREF-, CVREF+, CVREFOUT	comparator reference voltage low and high inputs, output
D+, D-	USB communication lines
EMUCx, EMUDx (x=1,2)	used by an in-circuit emulator (ICE)
ENVREG	enable for on-chip voltage regulator that provides 1.8 V to internal core (on the NU32 board it is set to VDD to enable the regulator)
ICx (x=1-5)	input capture pins for measuring frequencies and pulse widths
INTx (x=0-4)	voltage changes on these pins can generate interrupts
$\overline{\text{MCLR}}$	master clear reset pin, resets PIC when low
OCx (x=1-5)	output compare pins, usually used to generate pulse trains (pulse-width modulation) or individual pulses
OCFA, OCFB	fault protection for output compare pins; if a fault occurs, they can be used to make OC outputs be high impedance (neither high nor low)
OSC1, OSC2	crystal or resonator connections for different clock modes
PGCx, PGDx (x=1,2)	used with in-circuit debugger (ICD)
PMALL, PMALH	latch enable for parallel master port
PMAx (x=0-15)	parallel master port address
PMDx (x=0-15)	parallel master port data
PMENB, PMRD, PMWR	enable and read/write strobes for parallel master port
Rxy (x=A-G, y=0-15)	digital I/O pins
RTCC	real-time clock alarm output
SCLx, SDAx (x=1-5)	I^2C serial clock and data input/output for I^2C synchronous serial communication modules
SCKx, SDIx, SDOx (x=1-4)	serial clock, serial data in, out for SPI synchronous serial communication modules
$\overline{\textbf{SS1}}$, $\overline{\textbf{SS2}}$	slave select (active low) for SPI communication
TxCK (x=1-5)	input pins for counters when counting external pulses
TCK, TDI, TDO, TMS	used for JTAG debugging
TRCLK, TRDx (x=0-3)	used for instruction trace controller
$\overline{\textbf{UxCTS}}$, $\overline{\textbf{UxRTS}}$, **UxRX, UxTX (x=1-6)**	UART clear to send, request to send, receive input, and transmit output for UART modules
VDD	positive voltage supply for peripheral digital logic and I/O pins (3.3 V on NU32)
VDDCAP	capacitor filter for internal 1.8 V regulator when ENVREG enabled
VDDCORE	external 1.8 V supply when ENVREG disabled
VREF-, VREF+	can be used as negative and positive limit for ADC
VSS	ground for logic and I/O
VBUS	monitors USB bus power
VUSB	power for USB transceiver
VBUSON	output to control supply for VBUS
USBID	USB on-the-go (OTG) detect

Table 2.1: Some of the pin functions on the PIC32. Commonly used functions for embedded control are in **bold**. See Section 1 of the Data Sheet for more information.

year-month-day-time without using the CPU; and two comparators, each of which determines which of two analog inputs has a higher voltage.

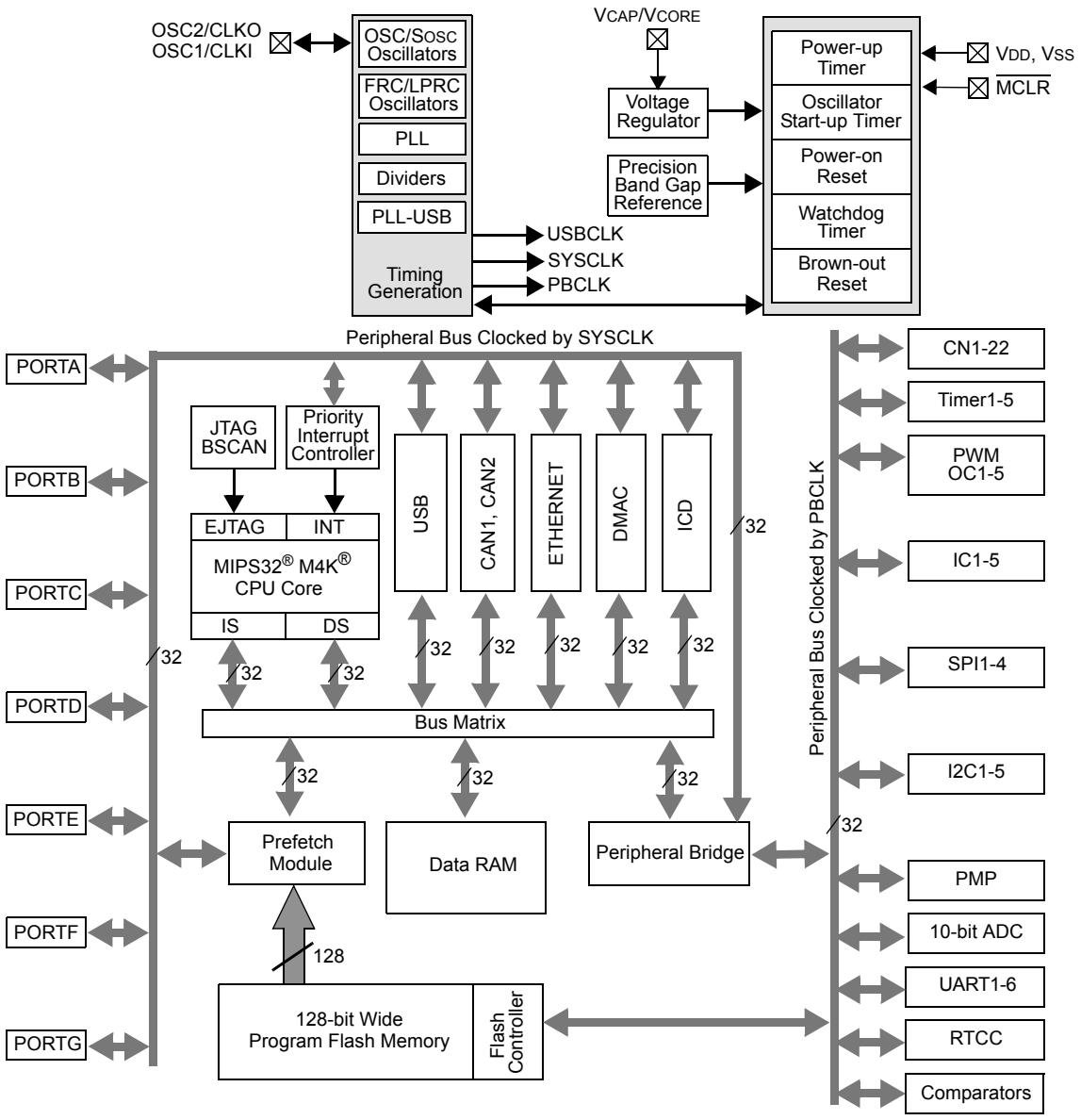


Figure 2.2: The PIC32MX5XX/6XX/7XX architecture.

Note that the peripherals are on two different buses: one is clocked by the system clock SYSCLK, and the other is clocked by the peripheral bus clock PBCLK. A third clock, USBCLK, is used for USB communication. The timing generation block that creates these clock signals and other elements of the architecture in Figure 2.2 are briefly described below.

CPU The central processing unit runs the whole show. It fetches program instructions over its "instruction side" (IS) bus, reads in data over its "data side" (DS) bus, executes the instructions, and writes out the results over the DS bus. The CPU can be clocked by SYSCLK at up to 80 MHz, meaning it can execute one instruction every 12.5 nanoseconds. The CPU is capable of multiplying a 32-bit integer by a 16-bit integer in one cycle, or a 32-bit integer by a 32-bit integer in two cycles. There is no floating point unit (FPU), so floating point math is carried out in a series of steps in software, meaning floating point operations are much slower than integer math.

The CPU also communicates with the interrupt controller, described below.

The CPU is based on the MIPS32® M4K® microprocessor core licensed from Imagination Technologies. The CPU operates at 1.8 V (provided by a voltage regulator internal to the PIC32, as it's used on the NU32 board).

Bus Matrix The CPU communicates with other units through the 32-bit bus matrix. Depending on the memory address specified by the CPU, the CPU can read data from, or write data to, program memory (flash), data memory (RAM), or SFRs. The memory map is discussed in Section 2.1.3.

Interrupt Controller The job of the interrupt controller is to present "interrupt requests" to the CPU. An interrupt request (IRQ) may be generated by a variety of sources, such as a changing input on a change notification pin or by the elapsing of a specified time on one of the timers. If the CPU accepts the request, it will suspend whatever it is doing and jump to an *interrupt service routine* (ISR), a function defined in the program. After completing the ISR, program control returns to where it was suspended. Interrupts are an extremely important concept in embedded control.

Memory: Program Flash and Data RAM The PIC32 has two types of memory: flash and RAM. Flash is generally more plentiful on PIC32's (e.g., 512 KB flash vs. 128 KB RAM on the PIC32MX795F512L), nonvolatile (meaning that its contents are preserved when powered off, unlike RAM), but slower to read and write than RAM. Your program is stored in flash memory and your temporary data is stored in RAM. When you power cycle your PIC32, your program is still there but your data in RAM is lost.[1]

Because flash is slow, with a max speed of 30 MHz for the PIC32MX795F512L, reading a program instruction from flash may take three CPU cycles when operating at 80 MHz (see Electrical Characteristics in the Data Sheet). One job of the prefetch cache module (below) is to minimize or eliminate the need for the CPU to wait around for program instructions to load.

Prefetch Cache Module You might be familiar with the term *cache* from your web browser. Your browser's cache stores recent documents or pages you have accessed over the web, so the next time you request them, your browser can provide a local copy immediately, instead of waiting for the download.

The *prefetch cache module* operates similarly—it stores recently executed program instructions, which are likely to be executed again soon (as in a program loop), and, in linear code with no branches, it can even run ahead of the current instruction and predictively *prefetch* future instructions into the cache. In both cases, the goal is to have the next instruction requested by the CPU already in the cache. When the CPU requests an instruction, the cache is first checked. If the instruction at that memory address is in the cache (a cache *hit*), the prefetch module provides the instruction to the CPU immediately. If there is a *miss*, the slower load from flash memory begins.

In some cases, the prefetch module can provide the CPU with one instruction per cycle, hiding the delays due to slow flash access. The module can cache all instructions in small program loops, so that flash memory does not have to be accessed while executing the loop. For linear code, the 128-bit wide data path between the prefetch module and flash memory allows the prefetch module to run ahead of execution despite the slow flash load times.

The prefetch cache module can also store constant data.

Clocks and Timing Generation There are three clocks on the PIC32: SYSCLK, PBCLK, and USBCLK. USBCLK is a 48 MHz clock used for USB communication. SYSCLK clocks the CPU at a maximum frequency of 80 MHz, adjustable all the way down to 0 Hz. Higher frequency means more calculations per second but higher power usage, approximately proportional to frequency. PBCLK is used by a number of the peripherals, and its frequency is set to SYSCLK's frequency divided by 1, 2, 4, or 8. You might want to set PBCLK's frequency lower than SYSCLK's if you want to save power. If PBCLK's frequency is less than SYSCLK's, then programs with back-to-back peripheral operations will cause the CPU to wait cycles before issuing the second peripheral command to ensure that the first one has completed.

[1] It is also possible to store program instructions in RAM, and to store data in flash, but we set that aside for now.

All clocks are derived either from an oscillator internal to the PIC32 or an external resonator or oscillator provided by the user. High-speed operation requires an external circuit, so the NU32 provides an external 8 MHz resonator as a clock source. The NU32 software sets the PIC32's configuration bits (see Section 2.1.4) to use a phase-locked loop (PLL) on the PIC32 to multiply this frequency by a factor of 10, generating a SYSCLK of 80 MHz. The PBCLK is set to the same frequency. The USBCLK is also derived from the 8 MHz resonator by a PLL multiplying the frequency by 6.

Digital Input and Output A digital I/O pin configured as an input can be used to detect whether the input voltage is low or high. On the NU32, the PIC32 is powered by 3.3 V, so voltages close to 0 V are considered low and those close to 3.3 V are considered high. Some input pins can tolerate up to 5.5 V, while voltages over 3.3 V on other pins could damage the PIC32 (see Figure 2.1 for the pins that can tolerate 5.5 V).

A digital I/O pin configured as an output can produce a voltage of 0 or 3.3 V. An output pin can also be configured as *open drain*. In this configuration, the pin is connected by an external pull-up resistor to a voltage of up to 5.5 V. This allows the pin's output transistor to either sink current (to pull the voltage down to 0 V) or turn off (allowing the voltage to be pulled up as high as 5.5 V). This increases the range of output voltages the pin can produce.

Counter/Timers The PIC32 has five 16-bit counters. Each can count from 0 up to $2^{16} - 1$, or any preset value less than $2^{16} - 1$ that we choose, before rolling over. Counters can be configured to count external events, such as pulses on a TxCK pin, or internal events, like PBCLK ticks. In the latter case, we refer to the counter as a *timer*. The counter can be configured to generate an interrupt when it rolls over. This allows the execution of an ISR on exact timing intervals.

Two 16-bit counters can be configured to make a single 32-bit counter. This can be done with two different pairs of counters, giving one 16-bit counter and two 32-bit counters.

Analog Input The PIC32 has a single analog-to-digital converter (ADC), but 16 different pins can be connected to it, one at a time. This allows up to 16 analog voltage values (typically sensor inputs) to be monitored. The ADC can be programmed to continuously read in data from a sequence of input pins, or to read in a single value when requested. Input voltages must be between 0 and 3.3 V. The ADC has 10 bits of resolution, allowing it to distinguish $2^{10} = 1024$ different voltage levels. Conversions are theoretically possible at a maximum rate of 1 million samples per second on the PIC32MX795F512L.

Output Compare Output compare pins are used to generate a single pulse of specified duration, or a continuous pulse train of specified duty cycle and frequency. They work with timers to generate the precise timing. A common use of output compare pins is to generate PWM (pulse-width modulated) signals as control signals for motors. Pulse trains can also be low-pass filtered to generate approximate analog outputs. (There are no analog outputs on the PIC32.)

Input Capture A changing input on an input capture pin can be used to store the current time measured by a timer. This allows precise measurements of input pulse widths and signal frequencies. Optionally, the input capture pin can generate an interrupt.

Change Notification A change notification pin can be used to generate an interrupt when the input voltage changes from low to high or vice-versa.

Comparators A comparator is used to compare which of two analog input voltages is larger. A comparator can generate an interrupt when one of the inputs exceeds the other.

Real-Time Clock and Calendar The RTCC module is used to maintain accurate time, day, month, and year over extended periods of time while using little power and requiring no attention from the CPU. It uses a separate clock, allowing it to run even when the PIC32 is in sleep mode.

Parallel Master Port The PMP module is used to read data from and write data to external parallel devices with 8-bit and 16-bit data buses.

DMA Controller The Direct Memory Access controller is useful for data transfer without involving the CPU. For example, DMA can allow an external device to dump data through a UART directly into PIC32 RAM.

SPI Serial Communication The Serial Peripheral Interface bus provides a simple method for serial communication between a master device (typically a microcontroller) and one or more slave devices. Each slave device has four communication pins: a Clock (set by the master), Data In (from the master), Data Out (to the master), and Select. The slave is selected for communication if the master holds its Select pin low. The master device controls the Clock, has a Data In and a Data Out line, and one Select line for each slave it can talk to. Communication rates can be up to tens of megabits per second.

I^2C Serial Communication The Inter-Integrated Circuit protocol I^2C (pronounced "I squared C") is a somewhat more complicated serial communication standard that allows several devices to communicate over only two shared lines. Any of the devices can be the master at any given time. The maximum data rate is less than for SPI.

UART Serial Communication The Universal Asynchronous Receiver Transmitter module provides another method for serial communication between two devices. There is no clock line, hence "asynchronous," but the two devices communicating must be set to the same communication rate. Each of the two devices has a Receive Data line and a Transmit Data line, and typically a Request to Send line (to ask for permission to send data) and a Clear to Send line (to indicate that the device is ready to receive data). Typical data rates are 9600 bits per second (9600 baud) up to hundreds of thousands of bits per second.

USB The Universal Serial Bus is a popular asynchronous communication protocol. One master communicates with one or more slaves over a four-line bus: +5 V, ground, D+ and D− (differential data signals). The PIC32MX795F512L implements USB 2.0 full-speed and low-speed options, and can communicate at theoretical data rates of up to several megabits per second.

CAN Controller Area Networks are heavily used in electrically noisy environments (particularly industrial and automotive environments) to allow many devices to communicate over a single two-wire bus. Data rates of up to 1 megabit per second are possible.

Ethernet The ethernet module uses an external PHY chip (physical layer protocol transceiver chip) and direct memory access (DMA) to offload from the CPU the heavy processing requirements of ethernet communication. The NU32 board does not include a PHY chip.

Watchdog Timer If the Watchdog Timer is used by your program, your program must periodically reset the timer counter. Otherwise, when the counter reaches a specified value, the PIC32 will reset. This is a way to have the PIC32 restart if your program has entered an unexpected state where it doesn't pet the watchdog.

2.1.3 The Physical Memory Map

The CPU accesses the peripherals, data, and program instructions in the same way: by writing a memory address to the bus. The PIC32's memory addresses are 32-bits long, and each address refers to a byte in the *memory map*. This means that the memory map of the PIC32 consists of 4 GB (four gigabytes, or 2^{32} bytes). Of course most of these addresses are meaningless; there are not nearly that many things to address.

The PIC32's memory map consists of four main components: RAM, flash, peripheral SFRs that we write to (to control the peripherals or send outputs) or read from (to get sensor input, for example), and *boot flash*. Of these, we have not yet seen "boot flash." This is extra flash memory, 12 KB on the PIC32MX795F512L,

that contains program instructions that are executed immediately upon reset of the PIC32.[2] The boot flash instructions typically perform PIC32 initialization and then call the program installed in program flash. For the PIC32 on the NU32 board, the boot flash contains a "program receive" program that communicates with your computer when you load a new program on the PIC32. More on this in Chapter 3.

The following table illustrates the PIC32's *physical* memory map. It consists of a block of "RAMsize" bytes of RAM (128 KB for the PIC32MX795F512L), "flashsize" bytes of flash (512 KB for the PIC32MX795F512L), 1 MB for the peripheral SFRs, and "bootsize" for the boot flash (12 KB for the PIC32MX795F512L):

Physical Memory Start Address	**Size (bytes)**	**Region**
0x00000000	RAMsize (128 KB)	Data RAM
0x1D000000	flashsize (512 KB)	Program Flash
0x1F800000	1 MB	Peripheral SFRs
0x1FC00000	bootsize (12 KB)	Boot Flash

The memory regions are not contiguous. For example, the first address of program flash is 480 MB after the first address of data RAM. An attempt to access an address between the data RAM segment and the program flash segment would generate an error.

It is also possible to allocate a portion of RAM to hold program instructions.

In Chapter 3, when we discuss programming the PIC32, we will introduce the *virtual* memory map and its relationship to the physical memory map.

2.1.4 Configuration Bits

The last four 32-bit words of the boot flash are the Device Configuration Registers, DEVCFG0 to DEVCFG3, containing the *configuration bits*. The values in these configuration bits choose a number of important properties of how the PIC32 will function. You can learn more about configuration bits in the Special Features section of the Data Sheet. For example, DEVCFG1 and DEVCFG2 contain configuration bits that determine the frequency multiplier converting the external resonator frequency to the SYSCLK frequency, as well as bits that determine the ratio between the SYSCLK and PBCLK frequencies.

2.2 The NU32 Development Board

The NU32 development board is shown in Figure 2.3, and the pinout is given in Table 2.2. The main purpose of the NU32 board is to provide easy breadboard access to 82 of the 100 PIC32MX795F512L pins. The NU32 acts like a big 84-pin DIP chip and plugs into two standard prototyping breadboards, straddling the long rails used for power, as shown in Figure 2.3.

Beyond simply breaking out the pins, the NU32 provides a few other things that make it easy to get started with the PIC32. For example, to power the PIC32, the NU32 provides a barrel jack that accepts a 2.1 mm inner diameter, 5.5 mm outer diameter "center positive" power plug. The plug should provide DC 6 V or more; the NU32 comes with a 6 V wall wart capable of providing 1 amp. The PIC32 requires a supply voltage VDD between 2.3 and 3.6 V, and the NU32 provides a 3.3 V voltage regulator providing a stable voltage source for the PIC32 and other electronics on board. Since it is often convenient to have a 5 V supply available, the NU32 also has a 5 V regulator. The power plug's raw input voltage and ground, as well as the regulated 3.3 V and 5 V supplies, are made available to the user on the power rails running down the center of the NU32, as illustrated in Figure 2.3. Since the power jack is directly connected to the 6 V and GND rails, you could power the NU32 by putting 6 V and GND on these rails directly and not connecting the power jack.

The 3.3 V regulator is capable of providing up to 800 mA and the 5 V regulator is capable of providing up to 1 amp. However, the wall wart can only provide 1 amp total, and in practice you should stay well under each of these limits. For example, you should not plan to draw more than 200-300 mA or so from any of the power rails. Even if you use a higher current power supply, such as a battery, you should respect these

[2] The last four 32-bit words of the boot flash memory region are Device Configuration Registers. See Section 2.1.4.

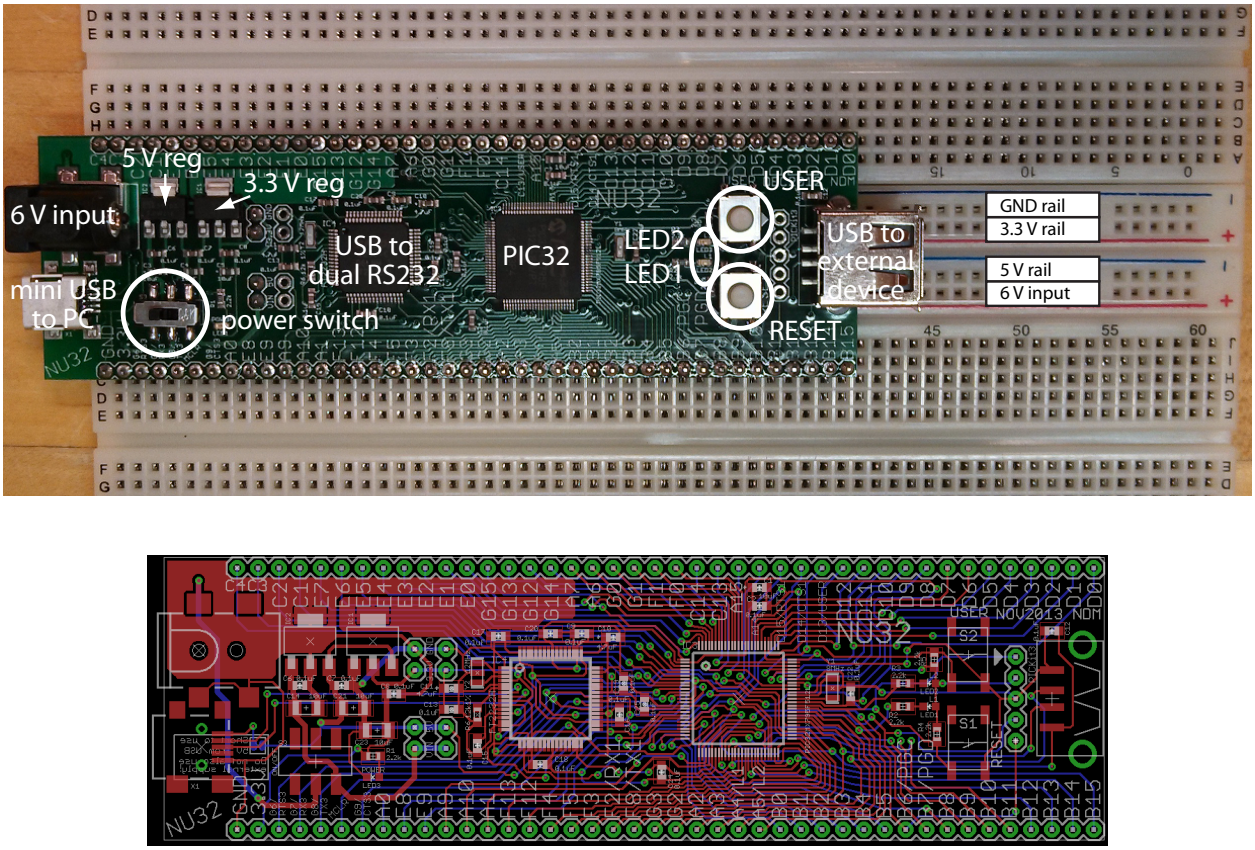


Figure 2.3: The NU32 development board: photo and PCB silkscreen.

limits, as the current has to flow through the relatively thin traces of the PCB. It is also not recommended to use high voltage supplies greater than 9 V or so, as the regulators will heat up.

Since motors tend to draw lots of current (even small motors may draw hundreds of mA up to several amps), do not try to power them using power from the NU32 rails. Use a separate battery or power supply instead.

In addition to the voltage regulators, the NU32 provides an 8 MHz resonator as the source of the PIC32's 80 MHz clock signal. It also has a mini B USB jack to connect your computer's USB port to a dual USB-to-RS-232 FTDI chip that allows your PIC32 to speak RS-232 to your computer's USB port. Two RS-232 channels share the single USB cable—one dedicated to programming the PIC32 and the other allowing communication with the host computer while a program is running on your PIC32.

A standard A USB jack is provided to allow the PIC32 to talk to another external device, like a smartphone.

The NU32 board also has a power switch which connects or disconnects the input power supply to the voltage regulators, and two LEDs and two buttons (labeled USER and RESET) allowing very simple input and output. The two LEDs, LED1 and LED2, are connected at one end by a resistor to 3.3 V and the other end to digital outputs RA4 and RA5, respectively, so that they are off when those outputs are high and on when they are low. The USER and RESET buttons are attached to the digital input RD13 and $\overline{\text{MCLR}}$ pins, respectively, and both buttons are configured to give 0 V to these inputs when pressed and 3.3 V otherwise. See Figure 2.4.

While the NU32 comes with a bootloader installed in its flash memory, you have the option to use a programmer to install a standalone program. The five plated through-holes near the USB jack align with the pins of devices such as the PICkit 3 programmer (Figure 2.5).

Function	PIC32			PIC32	Function
GND		GND	C4	9 √	T5CK/SDI1/C4
3.3 V		3.3 V	C3	8 √	T4CK/C3
SCK2/U6TX/U3RTS/CN8/G6	√ 10	**G6/RTS3**	C2	7 √	T3CK/C2
SDA4/SDI2/U3RX/CN9/G7	√ 11	**G7/RX3**	C1	6 √	T2CK/C1
SCL4/SDO2/U3TX/CN10/G8	√ 12	**G8/TX3**	E7	5 √	PMD7/E7
MCLR	√ 13	MCLR	E6	4 √	PMD6/E6
SS2/U6RX/U3CTS/CN11/G9	√ 14	**G9/CTS3**	E5	3 √	PMD5/E5
A0	√ 17	A0	E4	100 √	PMD4/E4
INT1/E8	√ 18	E8	E3	99 √	PMD3/E3
INT2/E9	√ 19	E9	E2	98 √	PMD2/E2
VREF-/CVREF-/A9	28	A9	E1	94 √	PMD1/E1
VREF+/CVREF+/A10	29	A10	E0	93 √	PMD0/E0
A1	√ 38	A1	G15	1 √	G15
SCK4/U5TX/U2RTS/F13	√ 39	F13	G13	97 √	G13
SS4/U5RX/U2CTS/F12	√ 40	F12	G12	96 √	G12
SDA5/SDI4/U2RX/CN17/F4	√ 49	F4	G14	95 √	G14
SCL5/SDO4/U2TX/CN18/F5	√ 50	F5	A7	92 √	A7
USBID/F3	√ 51	F3	A6	91 √	A6
SDA3/SDI3/U1RX/F2	√ 52	**F2/RX1**	G0	90 √	C2RX/PMD8/G0
SCL3/SD03/U1TX/F8	√ 53	**F8/TX1**	G1	89 √	C2TX/PMD9/G1
D-/G3	56	G3	F1	88 √	C1TX/PMD10/F1
D+/G2	57	G2	F0	87 √	C1RX/PMD11/F0
SCL2/A2	√ 58	A2	C14	74	T1CK/CN0/C14
SDA2/A3	√ 59	A3	C13	73	CN1/C13
A4	√ 60	**A4/L1**	A15	67 √	SDA1/INT4/A15
A5	√ 61	**A5/L2**	A14	66 √	SCL1/INT3/A14
PGED1/AN0/CN2/B0	25	B0	**D15/RTS1**	48 √	SCK3/U4TX/U1RTS/CN21/D15
PGEC1/AN1/CN3/B1	24	B1	**D14/CTS1**	47 √	SS3/U4RX/U1CTS/CN20/D14
AN2/C2IN-/CN4/B2	23	B2	**D13/USER**	80 √	PMD13/CN19/D13
AN3/C2IN+/CN5/B3	22	B3	D12	79 √	IC5/PMD12/D12
AN4/C1IN-/CN6/B4	21	B4	D11	71 √	IC4/D11
AN5/C1IN+/CN7/B5	20	B5	D10	70 √	SCK1/IC3/D10
PGEC2/AN6/OCFA/B6	26	**B6/PGC**	D9	69 √	SS1/IC2/D9
PGED2/AN7/B7	27	**B7/PGD**	D8	68 √	RTCC/IC1/D8
AN8/C1OUT/B8	32	B8	D7	84 √	PMD15/CN16/D7
AN9/C2OUT/B9	33	B9	D6	83 √	PMD14/CN15/D6
AN10/CVREFOUT/B10	34	B10	D5	82 √	CN14/D5
AN11/B11	35	B11	D4	81 √	OC5/CN13/D4
AN12/B12	41	B12	D3	78 √	OC4/D3
AN13/B13	42	B13	D2	77 √	OC3/D2
AN14/B14	43	B14	D1	76 √	OC2/D1
AN15/OCFB/CN12/B15	44	B15	D0	72 √	SDO1/OC1/INT0/D0

Table 2.2: The NU32 pinout (in green, with power jack at top) with PIC32MX795F512L pin numbers. Board pins in **bold** should only be used with care, as they are used for other functions by the NU32. Pins marked with a √ are 5.5 V tolerant. Not all pin functions are listed; see Figure 2.1 or the PIC32 Data Sheet.

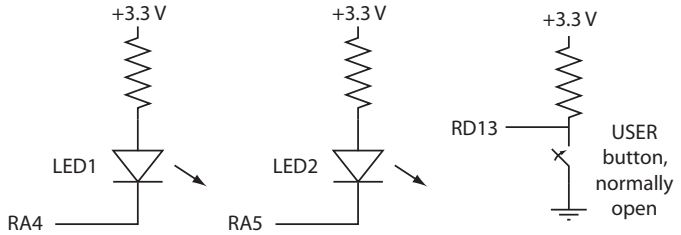


Figure 2.4: The NU32 connection of pins RA4, RA5, and RD13 to LED1, LED2, and the USER button, respectively.

2.3 Chapter Summary

- The PIC32 features a 32-bit data bus and a CPU capable of performing some 32-bit operations in a single clock cycle.

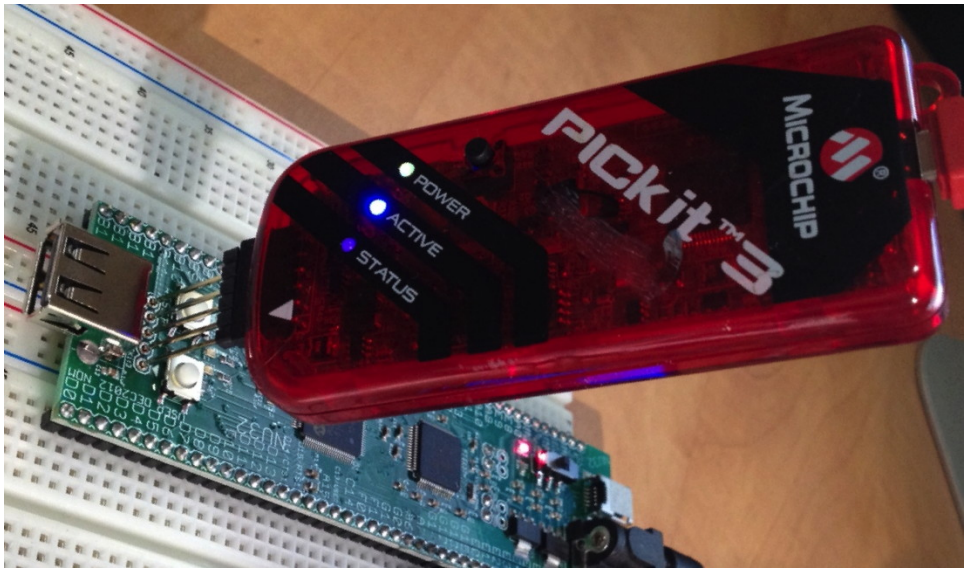


Figure 2.5: Attaching the PICkit 3 programmer to the NU32 board.

- In addition to nonvolatile flash program memory and RAM data memory, the PIC32 provides peripherals particularly useful for embedded control, including analog inputs, digital I/O, PWM outputs, counter/timers, inputs that generate interrupts or measure pulse widths or frequencies, and pins for a variety of communication protocols, including RS-232, USB, ethernet, CAN, I^2C, and SPI.
- The functions performed by the pins and peripherals are determined by Special Function Registers. SFR settings also determine other aspects of the behavior of the PIC32.
- The PIC32 has three main clocks: the SYSCLK that clocks the CPU, the PBCLK that clocks peripherals, and the USBCLK that clocks USB communication.
- Physical memory addresses are specified by 32 bits. The physical memory map contains four regions: data RAM, program flash, SFRs, and boot flash. RAM can be accessed in one clock cycle, while flash access may be slower. The prefetch cache module can be used to minimize delays in accessing program instructions.
- Four 32-bit configuration words, DEVCFG0 to DEVCFG3, set behavior of the PIC32 that should not be changed during execution. For example, these configuration bits determine how an external clock frequency is multiplied or divided to create the PIC32 clocks.
- The NU32 development board provides voltage regulators for power, includes a resonator for clocking, breaks out the PIC32 pins to a solderless breadboard, provides a couple of LEDs and buttons for simple input and output, and makes USB/RS-232 communication and programming simple.

2.4 Exercises

You will need to refer to the PIC32MX5XX/6XX/7XX Data Sheet and PIC32 Reference Manual to answer some questions.

1. Search for the "Microchip flash products parametric chart" or navigate to it from the Microchip homepage. You should see a listing of all the PICs made by Microchip. Set the page to show all specs and limit the display to 32-bit PICs.

(a) Find PIC32s that meets the following specs: at least 128 KB of flash, at least 32 KB of RAM, and at least 80 MHz max CPU speed. (You can choose a range of settings within a single parameter by shift-clicking or ctrl-clicking.) What is the cheapest PIC32 that meets these specs, and what is its volume price? How many ADC, UART, SPI, and I^2C channels does it have? How many timers?

(b) What is the cheapest PIC32 overall? How much flash and RAM does it have, and what is its maximum clock speed?

(c) Among all PIC32's with 512 KB flash and 128 KB RAM, which is the cheapest? How does it differ from the PIC32MX795F512L?

2. Based on C syntax for bitwise operators and bit-shifting, calculate the following and give your results in hexadecimal.

   (a) `0x37 | 0xA8`

   (b) `0x37 & 0xA8`

   (c) `~0x37`

   (d) `0x37>>3`

3. Describe the four functions that pin 22 of the PIC32MX795F512L can have. Is it 5 V tolerant?

4. Referring to the Data Sheet section on I/O Ports, what is the name of the SFR you have to modify if you want to change pins on PORTC from output to input?

5. The SFR CM1CON controls comparator behavior. Referring to the Memory Organization section of the Data Sheet, what is the reset value of CM1CON in hexadecimal?

6. In one sentence each, without going into detail, explain the basic function of the following items shown in the PIC32 architecture block diagram Figure 2.2: SYSCLK, PBCLK, PORTA...G (and indicate which of these can be used for analog input on the NU32's PIC32), Timer 1-5, 10-bit ADC, PWM OC1-5, Data RAM, Program Flash Memory, and Prefetch Cache Module.

7. List the peripherals that are *not* clocked by PBCLK.

8. If the ADC is measuring values between 0 and 3.3 V, what is the largest voltage difference that it may not be able to detect? (It's a 10-bit ADC.)

9. Refer to the Reference Manual chapter on the Prefetch Cache. What is the maximum size of a program loop, in bytes, that can be completely stored in the cache?

10. Explain why the path between flash memory and the prefetch cache module is 128 bits wide instead of 32, 64, or 256 bits.

11. Explain how a digital output could be configured to swing between 0 and 4 V, even though the PIC32 is powered by 3.3 V.

12. PIC32's have increased their flash and RAM over the years. What is the maximum amount of flash memory a PIC32 can have before the current choice of base addresses in the physical memory map (for RAM, flash, peripherals, and boot flash) would have to be changed? What is the maximum amount of RAM? Give your answers in bytes in hexadecimal.

13. Check out the Special Features section of the Data Sheet.

    (a) If you want your PBCLK frequency to be half the frequency of SYSCLK, which bits of which Device Configuration Register do you have to modify? What values do you give those bits?

    (b) Which bit(s) of which SFR set the watchdog timer to be enabled? Which bit(s) set the postscale that determines the time interval during which the watchdog must be reset to prevent it from restarting the PIC32? What values would you give these bits to enable the watchdog and to set the time interval to be the maximum?

(c) The SYSCLK for a PIC32 can be generated in a number of ways. This is discussed in the Oscillator chapter in the Reference Manual and the Oscillator Configuration section in the Data Sheet. The PIC32 on the NU32 uses the (external) primary oscillator in HS mode with the phase-locked loop (PLL) module. Which bits of which device configuration register enable the primary oscillator and turn on the PLL module?

14. Your NU32 board provides four power rails: GND, regulated 3.3 V, regulated 5 V, and the unregulated input voltage (e.g., 6 V). You plan to put a load from the 5 V output to ground. If the load is modeled as a resistor, what is the smallest resistance that would be safe? An approximate answer is fine. In a sentence, explain how you arrived at the answer.

15. The NU32 could be powered by different voltages. Give a reasonable range of voltages that could be used, minimum to maximum, and explain the reason for the limits.

16. Two buttons and two LEDs are interfaced to the PIC32 on the NU32. Which pins are they connected to? Give the actual pin numbers, 1-100, as well as the name of the pin function as it is used on the NU32. For example, pin 57 on the PIC32MX795F512L could have the function D+ (USB data line) or RG2 (Port G digital input/output), but only one of these functions could be active at a given time.

Chapter 3

Looking Under The Hood: Software

In this chapter we explore how a simple C program interacts with the hardware described in the previous chapter. We begin by introducing the virtual memory map and its relationship to the physical memory map. We then use the `simplePIC.c` program from Chapter 1 to explore the compilation process and the XC32 compiler installation.

3.1 The Virtual Memory Map

In the previous chapter we learned about the PIC32's physical memory map, which allows the CPU to access any SFR or any location in data RAM, program flash, or boot flash, using a 32-bit address. The PIC32 doesn't actually have 2^{32} bytes, or 4 GB worth of SFRs and memory; therefore, many physical addresses are invalid.

In this chapter we focus on the *virtual memory map*. Software refers to memory and SFRs using virtual addresses (VAs) rather than physical addresses (PAs). The fixed mapping translation (FMT) unit in the CPU converts VAs into PAs using the following formula:

$$\text{PA} = \text{VA} \ \& \ \text{0x1FFFFFFF}$$

This bitwise AND operation clears the three most significant bits of the address; thus multiple VAs map to the same PA.

If the PIC32 just discards the first three bits, why bother having them? Well, the CPU and the prefetch cache module we learned about in the previous chapter use them. If the first three bits of the virtual address are 0b100 (corresponding to an 8 or 9 as the most significant hex digit of the VA), then that instruction can be cached. If the first three bits are 0b101 (corresponding to an A or B as the most significant hex digit of the VA), then it cannot be cached. Thus the segment of virtual memory 0x80000000 to 0x9FFFFFFF is cacheable, while the segment 0xA0000000 to 0xBFFFFFFF is noncacheable. The cacheable segment is called KSEG0 (for "kernel segment") and the noncacheable segment is called KSEG1.[1]

Figure 3.1 illustrates the relationship between the physical and virtual memory maps. Note that the SFRs are excluded from the KSEG0 cacheable virtual memory segment. SFRS correspond to physical devices (e.g., peripherals); therefore their values cannot be cached. Otherwise, the CPU could read outdated SFR values because the state of the SFR could change between when it was cached and when it was needed by the CPU. For instance, if port B were configured as a digitial input port, the SFR PORTB would contain the current input values of some pins. The voltage on these pins could change at any time; therefore, the only way to retrieve a reliable value is to read directly from the SFR rather than from the cache.

Also note that program flash and data RAM can be accessed using either cacheable or noncacheable VAs. Typically, you can ignore this detail because the PIC32 will be configured to access program flash via the cache (since flash memory is slow), and data RAM without the cache (since RAM is fast).

[1] Another cacheable segment, USEG (for "user segment") is available in the lower half of virtual memory. This memory segment is for "user programs" that run under an operating system installed in a kernel segment. For safety, programs in the user segment cannot access SFRs or boot flash. We will never use the user segment.

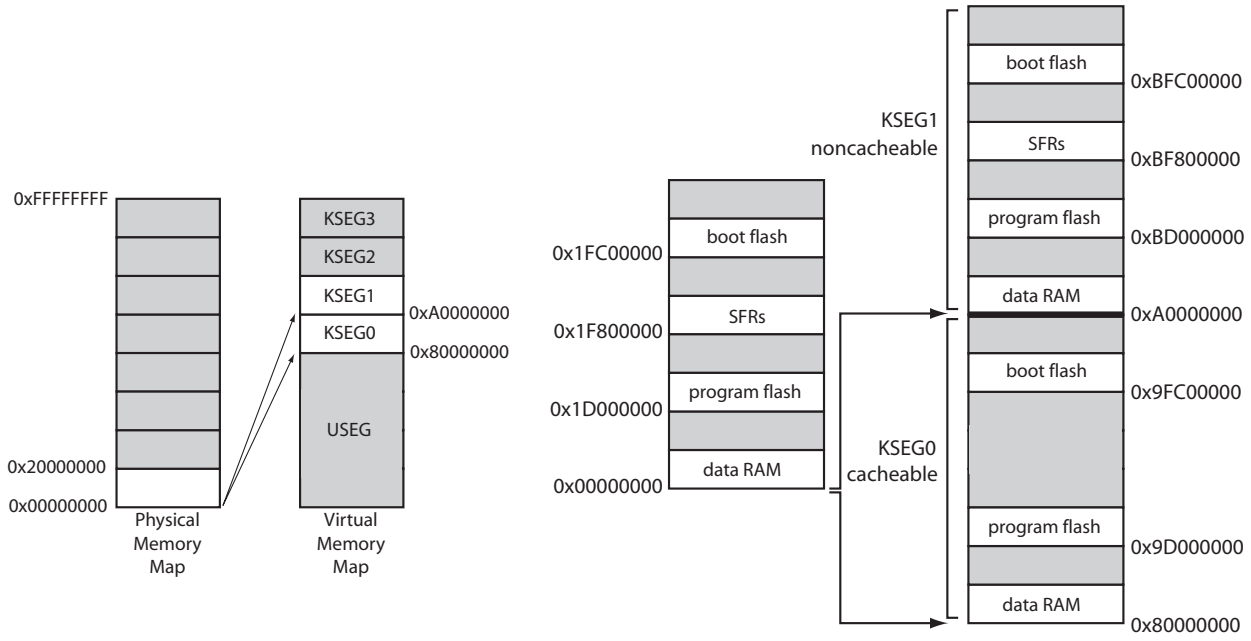


Figure 3.1: (Left) The 4 GB physical and virtual memory maps are divided into 512 MB segments. The mapping of the valid physical memory addresses to the virtual memory regions KSEG0 and KSEG1 is illustrated. The PIC32 does not use the virtual memory segments KSEG2 and KSEG3, which are allowed by the MIPS architecture, and we will not use the user segment USEG, which sits in the bottom half of the virtual memory map. (Right) Physical addresses mapped to virtual addresses in cacheable memory (KSEG0) and noncacheable memory (KSEG1). Note that SFRs are not cacheable. The last four words of boot flash, 0xBFC02FF0 to 0xBFC02FFF in KSEG1, correspond to the device configuration words DEVCFG0 to DEVCFG3. Memory regions are not drawn to scale.

Going forward, we will use virtual addresses like 0x9D000000 and 0xBD000000, and you should realize that these refer to the same physical address. Since virtual addresses start at 0x80000000, and all physical addresses are below 0x20000000, there is no possibility of confusing whether we are talking about a VA or a PA.

3.2 An Example: `simplePIC.c`

Let's build the `simplePIC.c` bootloaded executable from Chapter 1. For convenience, here is the program again:

Code Sample 3.1. `simplePIC.c`. Blinking lights, unless the USER button is pressed.

```
#include <xc.h>          // Load the proper header for the processor

void delay(void);

int main(void) {
  DDPCONbits.JTAGEN = 0; // Disable JTAG, make pins 4 and 5 of Port A available.
  TRISA = 0xFFCF;        // Pins 4 and 5 of Port A are LED1 and LED2.  Clear
                         // bits 4/5 to zero, for output.  Others are inputs.
  LATAbits.LATA4 = 0;    // Turn LED1 on and LED2 off.  These pins sink ...
  LATAbits.LATA5 = 1;    // ... current on NU32, so "high" = "off."
```

```
  while(1) {
    delay();
    LATAINV = 0x0030;    // toggle LED1 and LED2
  }
  return 0;
}

void delay(void) {
  int j;
  for (j = 0; j < 1000000; j++) { // number is 1 million
    while(!PORTDbits.RD13) {
        ;   // Pin D13 is the USER switch, low if pressed.
    }
  }
}
```

Navigate to the `<PIC32>` directory. Following the same procedure as in Chapter **??**, build `simplePIC.hex` and load it onto your NU32. We have reprinted the instructions here (you may need to specify the full path to these commands):

```
> xc32-gcc -mprocessor=32MX795F512L
   -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
> xc32-bin2hex simplePIC.elf
> nu32utility <COMPortB> simplePIC.hex
```

When you have the program running, the NU32's two LEDs should alternate on and off and stop while you press the USER button.

Look at the source code: the program refers to SFRs named TRISA, LATAINV, etc. These names align with the SFR names in the Data Sheet and Reference Manual sections on input/output (I/O) ports. We will consult the Data Sheet and Reference Manual often when programming the PIC32. We will explain the use of these SFRs shortly.

3.3 What Happens When You Build?

First, let's begin to understand what happens when you create `simplePIC.hex` from tt simplePIC.c. Refer to Figure 3.2.

First the **preprocessor** removes comments and inserts `#include`d header files. It also handles other preprocessor instructions such as `#define`. You can have multiple `.c` C source files and `.h` header files, but only one C file is allowed to have a `main` function. The other files may contain helper functions. We will learn more about this in Chapter 4.

Then the **compiler** turns the C files into MIPS32 assembly language files, machine commands specific to the PIC32's MIPS32 CPU. Basic C code will not vary between processor architectures, but assembly language may be completely different. These assembly files are readable by a text editor, and it is possible to program the PIC32 directly in assembly language.

The **assembler** turns the assembly files into machine-level *relocatable object code*. This code cannot be inspected with a text editor. The code is called relocatable because the final memory addresses of the program instructions and data used in the code are not yet specified. The **archiver** is a utility that allows you to package several related `.o` object files into a single `.a` library file. We will not be making our own archived libraries, but we will certainly be using `.a` libraries that have already been made by Microchip!

Finally, the **linker** takes one or more object files and combines them into a single executable file, with all program instructions assigned to specific memory locations. The linker uses a linker script that has information about the amount of RAM and flash on your particular PIC32, as well as directions about where in virtual memory to place the data and instructions. The result is an executable and linkable format (`.elf`) file, a standard executable file format. This file contains useful debugging information as well as information that allows tools such as `xc32-objdump` to *disassemble* the file, which converts it back into assembly code (Section 3.8). This extra information adds up; building `simplePIC.c` results in a `.elf` file that is hundreds of

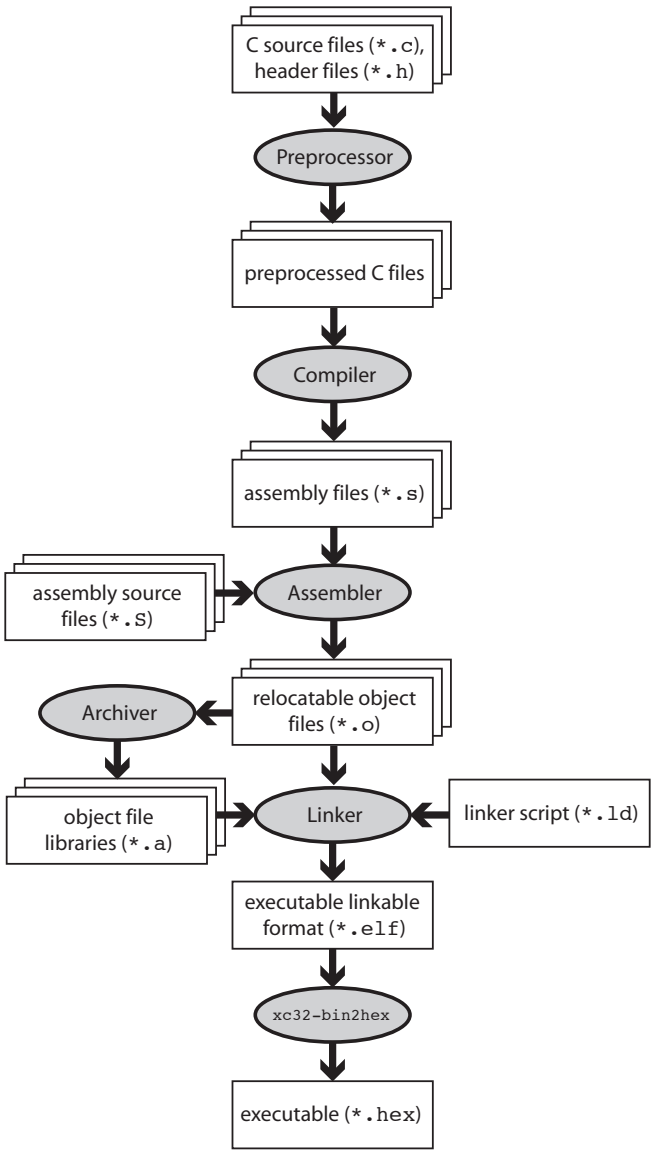


Figure 3.2: The "compilation" process.

kilobytes! A final step creates a stripped-down `.hex` file of less than 10 KB. This is an ASCII representation of your executable suitable for sending to the bootloader program on your PIC32 (more on this in the next section) that writes the program into flash on your PIC32.

Although the entire process consists of several steps, it is often referred to as "compiling" for short. "Build" or "make" is more accurate.

3.4 What Happens When You Reset the PIC32?

Your program is running. You hit the RESET button on the NU32. What happens next?

First the CPU jumps to the beginning of boot flash, address 0xBFC00000, and starts executing instructions.[2] For the NU32, the boot flash contains the *bootloader*, a program used to load other programs onto the

[2]If you are just powering on your PIC32, it will wait a short period while electrical transients die out, clocks synchronize, etc., before jumping to 0xBFC00000.

Virtual Address (BF88_#)	Register Name	Bit Range	Bits																All Resets
			31/15	30/14	29/13	28/12	27/11	26/10	25/9	24/8	23/7	22/6	21/5	20/4	19/3	18/2	17/1	16/0	
6000	TRISA	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	TRISA15	TRISA14	—	—	—	TRISA10	TRISA9	—	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	C6FF

Figure 3.3: Port A registers, taken from the PIC32 Data Sheet.

PIC32. The bootloader first checks to see if you are pressing the USER button. If so, it knows that you want to reprogram the PIC32, so it attempts to communicate with the bootloader utility (`nu32utility`) on your computer. With communication established, the bootloader receives the executable `.hex` file and writes it to the PIC32's program flash (see exercise 2). We refer to the virtual address where your program is installed as `_RESET_ADDR`.

Note: The PIC32's reset address 0xBFC00000 is hardwired and cannot be changed. The address where the bootloader writes your program, however, can be changed in software.

Now assume that you weren't pressing the USER button when you reset the PIC32. Then the bootloader jumps to the address `_RESET_ADDR` and begins executing the program you previously installed there. Notice that our program, `simplePIC.c`, is an infinite loop, so it never stops executing. That is the desired behavior in embedded control. If your program exits, the PIC32 will just sit in a tight loop, doing nothing until it is reset.

3.5 Understanding `simplePIC.c`

Let's return to understanding `simplePIC.c`. The `main` function initializes values of DDPCONbits, TRISA, and LATAbits, then enters an infinite `while` loop. Each time through the loop it calls `delay()` and then assigns a value to LATAINV. The `delay` function executes a `for` loop that iterates one million times. During each iteration it enters a `while` loop, which checks the value of (!PORTDbits.RD13). If PORTDbits.RD13 is 0 (FALSE), then the expression (!PORTDbits.RD13) evaluates to TRUE, and the program remains here, doing nothing except checking the expression (!PORTDbits.RD13). When this expression evaluates to FALSE, the `while` loop exits, and the program continues with the `for` loop. After the `for` loop finishes, control returns to `main`.

Special Function Registers (SFRs) The main difference between `simplePIC.c` and programs that you may have written for your computer is how it interacts with the outside world. Rather than via keyboard or mouse, `simplePIC.c` accesses SFRs like TRISA, LATA, and PORTD, all of which correspond to peripherals. [3] Specifically, TRISA and LATA correspond to port A, an I/O port, and PORTD corresponds to port D, another I/O port. I/O ports allow the PIC32 to read and set the digital voltage on a pin. To discover what these SFRs control we start by consulting the table in Section 1 of the Data Sheet, which lists the pinout I/O descriptions. We see that port A, with pins named RA0 to RA15, has 12 pins, and port C, with pins named RC1 to RC15, has 8 pins. Port B, has 16 pins, labeled RB0 to RB15.

We now turn to the Data Sheet section on I/O Ports to for more information. We find that TRISA, short for "tri-state A," controls the direction, input or output, of the pins on port A. Each port A pin has a corresponding bit in TRISA. If this bit is 0, the pin is an output. If the bit is a 1, the pin is an input. ($0 = \mathrm{O_{utput}}$ and $1 = \mathrm{I_{nput}}$.) We can make some pins inputs and some outputs, or we can make them all have the same direction.

If you're curious about which direction the pins are by default, you can consult the Memory Organization section of the Data Sheet. Tables there list the VAs of many of the SFRs, as well as the values they default to

[3] DDPCON corresponds to JTAG debugging, which we do not use in this book. The `DDPCONbits.JTAGEN = 0` command disables the JTAG debugger so that pins RA4 and RA5 are available for digital I/O. See the Special Features section of the Data Sheet.

upon reset. There are a lot of SFRs! After some searching, you will find that TRISA sits at virtual address 0xBF886000, and its default value upon reset is 0x0000C6FF. (We've reproduced part of this table for you in Figure 3.3.) In binary, this would be

$$0x0000C6FF = 0000\ 0000\ 0000\ 0000\ 1100\ 0110\ 1111\ 1111.$$

The four most significant hex digits (two bytes, or 16 bits) are all 0. This is because those bits, technically, don't exist. Microchip calls them "unimplemented." No I/O port has more than 16 pins, so we don't need those bits, which are numbered 16–31. (The 32 bits are numbered 0–31.) Of the remaining bits, since the 0'th bit (least significant bit) is the rightmost bit, we see that bits 0–7, 9–10, and 14–15 are 1, while the rest are 0. The bits set to 1 correspond precisely to the pins we have available: RA0-7, RA9-10, and RA14-15 (there is no RA8), meaning that they are inputs. I/O pins are configured as inputs on reset for safety reasons; when we power on the PIC32, each pin will take its default direction before the program can change it. If an output pin were connected to an external circuit that is also trying to control the voltage on the pin, the two devices would fight each other, with damage to one or the other a possibility. No such problems arise if the pin is configured as an input by default.

So now we understand that the instruction

```
TRISA = 0xFFCF;
```

clears bits 4 and 5 to 0, implicitly clears bits 16–31 to 0 (which is ignored, since the bits are not implemented), and sets the rest of the bits to 1. It doesn't matter that we try to set some unimplemented bits to 1; those bits are ignored. The result is that port A pins 4 and 5, or RA4 and RA5 for short, are now outputs.

Our PIC32 C compiler allows the use of binary (base 2) representations of unsigned integers using 0b at the beginning of the number, so if you don't get lost counting bits, you could have equally written

```
TRISA = 0b1111111111001111;
```

The equivalent in base 10 would be

```
TRISA = 65487;
```

Another option would have been to use the instructions

```
TRISAbits.TRISA4 = 0; TRISAbits.TRISA5 = 0;
```

This allows us to change individual bits without worrying about specifying the other bits. We see this kind of notation later in the program, with LATAbits.LATA4 and PORTDbits.RD13, for example.

The two other basic SFRs in this program are LATA and PORTD. Again consulting the I/O Ports section of the Data Sheet, we see that LATA, short for "latch A," is used to write values to the output pins. Thus

```
LATAbits.LATA5 = 1;
```

sets pin RA5 high. Finally, PORTD contains the digital inputs on the port D pins. (Notice we didn't configure port D as input; we relied on the fact that it's the default.) PORTDbits.RD13 is 0 if 0 V is present on pin RD13 and 1 if approximately 3.3 V is present. Note that we use the latch when writing pins and the port when reading pins, for reasons explained in Ch. **??**.

Pins RA4, RA5, and RD13 on the NU32 Figure 2.4 shows how pins RA4, RA5, and RD13 are wired on the NU32 board. LED1 (LED2) is on if RA4 (RA5) is 0 and off if it is 1. When the USER button is pressed, RD13 registers a 0, and otherwise it registers a 1.

The result of these electronics and the `simplePIC.c` program is that the LEDs flash alternately, but remain unchanging while you press the USER button.

CLR, SET, and INV SFRs So far we have ignored the instruction

```
LATAINV = 0x0030;
```

Again consulting the Memory Organization section of the Data Sheet, we see that associated with the SFR LATA are three more SFRs, called LATACLR, LATASET, and LATAINV. (Indeed, many SFRs have corresponding CLR, SET, and INV SFRs.) These SFRs are used to easily change some of the bits of LATA without affecting the others. A write to these registers causes a one-time change to LATA's bits, but only in the bits corresponding to bits on the right-hand side that have a value of 1. For example,

```
LATAINV = 0x30;      // flips (inverts) bits 4 and 5 of LATA; all others unchanged
LATAINV = 0b110000;  // same as above
LATASET = 0x0005;    // sets bits 0 and 2 of LATA to 1; all others unchanged
LATACLR = 0x0002;    // clears bit 1 of LATA to 0;  all others unchanged
```

A less efficient way to toggle bits 4 and 5 of LATA is

```
LATAbits.LATA4 = !LATAbits.LATA4; LATAbits.LATA5 = !LATAbits.LATA5;
```

We'll look at efficiency in Chapter 5.

You can return to the table in the Data Sheet to see the VA addresses of the CLR, SET, and INV registers. They are always offset from their base register by 4, 8, and 12 bytes, respectively. Since LATA is at 0xBF886020, LATACLR, LATASET, and LATAINV are at 0xBF886024, 0xBF886028, and 0xBF88602C, respectively.

You should now understand how `simplePIC.c` works. But we have been ignoring the fact that we never declared TRISA, etc., before we started using them. We know you can't do that in C; these SFRs must be declared somewhere. The only place they could be declared is in the included file `xc.h`. We've ignored that `#include <xc.h>` statement until now. Time to take a look.[4]

3.5.1 Down the Rabbit Hole

Where do we find `xc.h`? If our program had the preprocessor command `#include "xc.h"`, the preprocessor would look for `xc.h` in the same directory as the C file including it. But we had `#include <xc.h>`, and the `<...>` notation means that the preprocessor will look in directories specified in the *include path*. For us, the default include path means that the compiler finds `xc.h` sitting at

```
<xc32dir>/<xc32ver>/pic32mx/include/xc.h
```

You should substitute your install directory in place of `<xc32dir>/<xc32ver>`.

Including `xc.h` gives us access to many data types, variables, and constants that MIcrochip has provided for our convenience. In particular, it provides variable declarations for SFRs like TRISA, allowing us to access the SFRs from C.

Before we open `xc.h`, let's look at the directory structure of the XC32 compiler installation. There's a lot here! We certainly don't need to understand all of it at this point, but let's try to get a sense of what's going on. Let's start at the level of your XC32 install directory and summarize what's in the nested set of directories, without being exhaustive.

1. `bin`: Contains the actual executable programs that do the compiling, assembling, linking, etc. For example, `xc32-gcc` is the C compiler.
2. `docs`: Some manuals, including the XC32 C Compiler User's Guide, and other documentation.
3. `examples`: Some sample code.
4. `lib`: Contains some `.h` header files and `.a` library archives containing general C object code.

[4]Microchip often changes the software it distributes, so there may be differences in details, but the essence of what we describe here will be the same.

5. `pic32-libs`: This directory contains the source code (`.c` C files, `.h` header files, and `.S` assembly files) needed to create numerous Microchip-provided libraries. These files are provided for reference and are not included directly in any of your code.

6. `pic32mx`: This directory has several files we are interested in because many of them end up in your project.

   (a) `lib`: This directory consists mostly of PIC32 object code and libraries that are linked with our compiled and assembled source code. For some of these libraries, source code exists in `pic32-libs`; for others we have only the object code libraries. Some important files in this directory include:

      i. `proc/32MX79512L/crt0_mips32r2.o`: The linker combines this object code with your program's object code when it creates the `.elf` file. The linker ensures that this "C Runtime Startup" code is executed first, since it performs various initializations your code needs to run, such as initializing the values of global variables. Different PIC32 models have different versions of this file under the appropriate `proc/<processor>` directory. You can find readable assembly source code at `pic32-libs/libpic32/startup/crt0.S`.

      ii. `libc.a`: Implementations of functions that are part of the C standard library.

      iii. `libdsp.a`: This library contains MIPS implementations of finite and infinite impulse response filters, the fast Fourier transform, and various vector math functions.

      iv. `proc/32MX795F512L/processor.o`: This object file gives the SFR virtual memory addresses for your particular PIC32. We can't look at it directly with a text editor, but there are utilities that allow us to examine it. For example, from the command line you could use the `xc32-nm` program in the top-level `bin` directory to see all the SFR VAs:

```
> xc32-nm processor.o
bf809040 A AD1CHS
...
bf886000 A TRISA
bf886004 A TRISACLR
bf88600c A TRISAINV
bf886008 A TRISASET
...
```

      All of the SFRs are printed out, in alphabetical order, with their corresponding VA. The spacing between SFRs is four, since there are four bytes (32 bits) in an SFR. The "A" means that these are absolute addresses. The linker must use these addresses when making final address assignments because the SFR's are implemented in hardware and can't be moved! The listing above indicates that TRISA is located at VA 0xBF886000, agreeing with the Memory Organization section of the Data Sheet.

      v. `proc/32MX795F512L/configuration.data`: This file describes some constants used in setting the configuration bits in DEVCFG0 to DEVCFG3 (Chapter 2.1.4). These bits are set by the bootloader (Section 3.6), so you do not need to worry about them in your programs. It is possible to use a programmer device to load programs onto the PIC32 without having a bootloader pre-installed on the PIC32 (that's how the bootloader got there in the first place!), in which case you would need to worry about these bits. See Appendix **??** for more information about programs that do not use a bootloader.

   (b) `include`: This directory contains a number of `.h` header files.

      i. `cp0defs.h`: This file defines a number of constants and macros that allow us to access functions of coprocessor 0 (CP0) on the MIPS32 M4K CPU. In particular, it allows us to read and set the *core timer* clock that ticks once every two SYSCLK cycles using macros like `_CP0_GET_COUNT()` (see Chapters 5 and 6 for more details). More information on CP0 can be found in the "CPU for Devices with the M4K Core" section of the Reference Manual.

      ii. `sys/attribs.h`: In the directory `sys`, the file `attribs.h` defines the macro syntax `__ISR` that we will use for interrupt service routines starting in Chapter 6.

iii. `xc.h`: This is the file we've been looking for. The most important purpose of `xc.h` is to include the appropriate processor-specific header file, in our case `include/proc/p32mx795f512l.h`. It does this by checking if `__32MX795F512L__` is defined:

```
#elif defined(__32MX795F512L__)
#include <proc/p32mx795f512l.h>
```

If you look at the command for compiling `simplePIC.c`, you may have noticed the option `-mprocessor=32MX795F512L`. This option defines the constant `__32MX795F512L__` to the compiler, allowing `xc.h` to function properly.

iv. `proc/p32mx795f512l.h`: Open this file in your text editor. Whoa! This file is over 40,000 lines long! It must be important. Time to look at it in more detail.

3.5.2 The Header File `p32mx795f512l.h`

The first 30% of `p32mx795f512l.h`, about 14,000 lines, consists of code like this, with line numbers added to the left for reference:

```
 1  extern volatile unsigned int        TRISA __attribute__((section("sfrs")));
 2  typedef union {
 3    struct {
 4      unsigned TRISA0:1;   // TRISA0 is bit 0 (1 bit long), interpreted as an unsigned int
 5      unsigned TRISA1:1;   // bits are in order, so the next bit, bit 1, is called TRISA1
 6      unsigned TRISA2:1;   // ...
 7      unsigned TRISA3:1;
 8      unsigned TRISA4:1;
 9      unsigned TRISA5:1;
10      unsigned TRISA6:1;
11      unsigned TRISA7:1;
12      unsigned :1;         // don't give a name to bit 8; it's unimplemented
13      unsigned TRISA9:1;   // bit 9 is called TRISA9
14      unsigned TRISA10:1;
15      unsigned :3;         // skip 3 bits, 11-13
16      unsigned TRISA14:1;
17      unsigned TRISA15:1;  // later bits are not given names
18    };
19    struct {
20      unsigned w:32;       // w refers to all 32 bits; the 16 above, and 16 more unimplemented bits
21    };
22  } __TRISAbits_t;
23  extern volatile __TRISAbits_t TRISAbits __asm__ ("TRISA") __attribute__((section("sfrs")));
24  extern volatile unsigned int        TRISACLR __attribute__((section("sfrs")));
25  extern volatile unsigned int        TRISASET __attribute__((section("sfrs")));
26  extern volatile unsigned int        TRISAINV __attribute__((section("sfrs")));
```

The first line, beginning `extern`, declares the variable TRISA as an `unsigned int`. The keyword `extern` means that no RAM has to be allocated for it; memory to hold the variable has been allocated for it elsewhere. In a typical C program, memory for the variable has been allocated by another C file using syntax without the `extern`, like `volatile unsigned int TRISA;`. In this case, however, no RAM has to be allocated for TRISA because it refers to an SFR, not a word in RAM. The `processor.o` file is the one that actually defines the VA of the symbol TRISA, as mentioned earlier.

The `volatile` keyword, applied to all the SFRs, means that the value of this variable could change without the CPU knowing it. Thus the compiler should generate assembly code to reload TRISA into the CPU registers every time it is used, rather than assuming that its value is unchanged just because no C code has modified it.

Finally, the `__attribute__` syntax tells the linker that TRISA is in the `sfrs` section of memory.

The next section of code, lines 2–22, defines a new data type called `__TRISAbits_t`. Next, in line 23, a variable named TRISAbits is declared of type `__TRISAbits_t`. Again, since it is an `extern` variable, no memory is allocated, and the `__asm__ ("TRISA")` syntax means that TRISAbits is at the same VA as TRISA.

It is worth understanding the new data type __TRISAbits_t. It is a union of two structs. The union means that the two structs share the same memory, a 32-bit word in this case. Each struct is called a *bit field*, which gives names to specific groups of bits within the 32-bit word. Thus declaring a variable TRISAbits of type __TRISAbits_t allows us to use syntax like TRISAbits.TRISA0 to refer to bit 0 of TRISA.

A named set of bits in a bit field need not be one bit long; for example, TRISAbits.w refers to the entire unsigned int TRISA, created from all 32 bits. The type __RTCALRMbits_t defined earlier in the file by

```
typedef union {
  struct {
    unsigned ARPT:8;
    unsigned AMASK:4;
 ...
} __RTCALRMbits_t;
```

has a first field ARPT that is 8 bits long and a second field AMASK that is 4 bits long. Since RTCALRM is a variable of type __RTCALRMbits_t, a C statement of the form RTCALRMbits.AMASK = 0xB would put the values 1, 0, 1, 1 in bits 11, 10, 9, 8, respectively, of RTCALRM.

After the declaration of TRISA and TRISAbits, lines 24–26 contain declarations of TRISACLR, TRISASET, and TRISAINV. These declarations allow simplePIC.c, which uses these variables, to compile successfully. When the object code of simplePIC.c is linked with the processor.o object code, references to these variables are resolved to the proper SFR VAs.

With these declarations in p32mx795f512l.h, the simplePIC.c statements

```
TRISA = 0xFFCF;
LATAINV = 0x0030;
while(!PORTDbits.RD13)
```

finally make sense; these statements write values to, or read values from, SFRs at VAs specified by processor.o. You can see that p32mx795f512l.h declares a lot of SFRs, but no RAM has to be allocated for them; they exist at fixed addresses in the PIC32's hardware.

The next 9% of p32mx795f512l.h is the extern variable declaration of the same SFRs, without the bit field types, for assembly language. The VAs of each of the SFRs is given, making this a handy reference.

Starting at about 17,800 lines into the file, we see constant definitions like the following:

```
#define _T1CON_TCS_POSITION                 0x00000001
#define _T1CON_TCS_MASK                     0x00000002
#define _T1CON_TCS_LENGTH                   0x00000001

#define _T1CON_TCKPS_POSITION               0x00000004
#define _T1CON_TCKPS_MASK                   0x00000030
#define _T1CON_TCKPS_LENGTH                 0x00000002
```

These refer to the Timer 1 SFR T1CON. Consulting the information about T1CON in the Timer1 section of the Data Sheet, we see that bit 1, called TCS, controls whether Timer 1's clock input comes from the T1CK input pin or from PBCLK. Bits 4 and 5, called TCKPS for "timer clock prescaler," control how many times the input clock has to "tick" before Timer 1 is incremented (e.g., TCKPS = 0b10 means there is one clock increment per 64 input ticks). The constants defined above are for convenience in accessing these bits. The POSITION constants indicate the least significant bit location in TCS or TCKPS in T1CON—one for TCS and four for TCKPS. The LENGTH constants indicate that TCS consists of one bit and TCKPS consists of two bits. Finally, the MASK constants can be used to determine the values of the bits we care about. For example:

```
unsigned int tckpsval = (T1CON & _T1CON_TCKPS_MASK) >> _T1CON_TCKPS_POSITION;
// AND MASKing clears all bits except 5 and 4, which are unchanged and shifted to positions 1 and 0
```

The definitions of the POSITION, LENGTH, and MASK constants take up most of the rest of the file. Of course, there is also a T1CONbits defined that allows you to access these bits directly (e.g. T1CONbits.TCS). We recommend that you use this method, as it is typically clearer and less error prone than performing direct bit manipulations.

At the end, some more constants are defined, like below:

```
#define _ADC10
#define _ADC10_BASE_ADDRESS    0xBF809000
#define _ADC_IRQ               33
#define _ADC_VECTOR            27
```

The first is merely a flag indicating to other `.h` and `.c` files that the 10-bit ADC is present on this PIC32. The second indicates the first address of 22 consecutive SFRs related to the ADC (see the Memory Organization section of the Data Sheet). The third and fourth relate to interrupts. The PIC32MX's CPU is capable of being interrupted by up to 96 different events, such as a change of voltage on an input line or a timer rollover event. Upon receiving these interrupts, it can call up to 64 different interrupt service routines, each identified by a "vector" corresponding to its address. These two lines say that the ADC's "interrupt request" line is 33 (out of 0 to 95), and its corresponding interrupt service routine is at vector 27 (out of 0 to 63). Interrupts are covered in Chapter 6.

Finally, `p32mx795f512l.h` concludes by including `ppic32mx.h`, which contains legacy code that is no longer needed but remains for backward compatibility with old programs.

3.5.3 Other Microchip Software: Harmony

Installed in your Harmony directory is an extensive and complex set of libraries and sample code written by Microchip. Because of the complexity and abstraction it introduces, we will avoid using Harmony functions until later in the book, when our programs are complex enough that low-level access to the peripherals through SFRs no longer suffices to take full advantage of the PIC32's capabilities.[5]

3.5.4 The `NU32bootloaded.ld` Linker Script

To create the executable `.hex` file, we needed the C source file `simplePIC.c` and the linker script `NU32bootloaded.ld`. Examining `NU32bootloaded.ld` with a text editor, we see the following line near the beginning:

```
INPUT("processor.o")
```

This line tells the linker to load the `processor.o` file specific to your PIC32. This allows the linker to resolve references to SFRs (declared as `extern` variables in `p32mx795f512l.h`) to actual addresses.

The rest of the `NU32bootloaded.ld` linker script has information such as the amount of program flash and data memory available, as well as the virtual addresses where program elements and global data should be placed. Below is a portion of `NU32bootloaded.ld`:

```
_RESET_ADDR             = (0xBD000000 + 0x1000 + 0x970);

/*************************************************************************
 * NOTE:  What is called boot_mem and program_mem below do not directly
 * correspond to boot flash and program flash.  For instance, here
 * kseg0_boot_mem and kseg1_boot_mem both live in program flash memory.
 * (We leave the boot flash solely to the bootloader.)
 * The boot_mem names below tell the linker where the startup codes should
 * go (here, in program flash).  The first 0x1000 + 0x970 + 0x490 = 0x1E00
 * of program flash memory is allocated to the interrupt vector table and
 * startup codes.  The remaining 0x7E200 is allocated to the user's program.
 *************************************************************************/
MEMORY
{
  /* interrupt vector table */
  exception_mem             : ORIGIN = 0x9D000000, LENGTH = 0x1000
  /* Start-up code sections; some cacheable, some not */
  kseg0_boot_mem            : ORIGIN = (0x9D000000 + 0x1000), LENGTH = 0x970
  kseg1_boot_mem            : ORIGIN = (0xBD000000 + 0x1000 + 0x970), LENGTH = 0x490
```

[5] Even though most sample code in the book does not use Harmony, the `Makefile` asks the linker to link with Harmony files, just to ensure that the same `make` process works whether or not you use Harmony. So you should install Harmony.

```
  /* User's program is in program flash, kseg0_program_mem, all cacheable       */
  /* 512 KB flash = 0x80000, or 0x1000 + 0x970 + 0x940 + 0x7E200                */
  kseg0_program_mem    (rx)  : ORIGIN = (0x9D000000 + 0x1000 + 0x970 + 0x490), LENGTH = 0x7E200
  debug_exec_mem             : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  /* Device Configuration Registers (configuration bits) */
  config3                    : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                    : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                    : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                    : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  configsfrs                 : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
  /* all SFRS */
  sfrs                       : ORIGIN = 0xBF800000, LENGTH = 0x100000
  /* PIC32MX795F512L has 128 KB RAM, or 0x20000 */
  kseg1_data_mem       (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x20000
}
```

Converting virtual to physical addresses, we see that the cacheable interrupt vector table (we will learn more about this in Chapter 6) in `exception_mem` is placed in a memory region of length 0x1000 bytes beginning at PA 0x1D000000 and running to 0x1D000FFF; cacheable startup code in `kseg0_boot_mem` is placed at PAs 0x1D001000 to 0x1D00196F; noncacheable startup code in `kseg1_boot_mem` is placed at PAs 0x1D001970 to 0x1D001DFF; and cacheable program code in `kseg0_program_mem` is allocated the rest of program flash, PAs 0x1D001E00 to 0x1D07FFFF. This program code includes the code we write plus other code that is linked.

The linker script for the NU32 bootloader placed the bootloader completely in the 12 KB boot flash with little room to spare. Therefore, the linker script for our bootloaded programs should place the programs solely in program flash. This is why the `boot_mem` sections above are defined to be in program flash. The label `boot_mem` simply tells the linker where the startup code should be placed, just as the label `kseg0_program_mem` tells the linker where the program code should be placed. (For the bootloader program, `kseg0_program_mem` was in boot flash.)

If the LENGTH of any given memory region is not large enough to hold all the program instructions or data for that region, the linker will fail.

Upon reset, the PIC32 always jumps to 0xBFC00000, where the first instruction of the startup code for the bootloader resides. The last thing the bootloader does is jump to VA 0xBD001970. Since the first instruction in the startup code for our bootloaded program is installed at the first address in `kseg1_boot_mem`, `NU32bootloaded.ld` *must* define the ORIGIN of `kseg1_boot_mem` at this address. This address is also known as `_RESET_ADDR` in `NU32bootloaded.ld`.

3.6 Bootloaded Programs vs. Standalone Programs

It is important to keep in mind that your executable is being installed on the PIC32 by another executable: the bootloader. The bootloader has been pre-installed in the boot flash portion of flash memory. This program, which always runs first when the PIC32 is reset, has already defined some of the behavior of the PIC32, so you didn't need to specify it in `simplePIC.c`. In particular, the bootloader code turns on the prefetch cache module (to allow faster performance) and sets a number of other important properties of the PIC32 with XC32-specific preprocessor commands such as

```
#pragma config FWDTEN = OFF         // watchdog timer OFF
#pragma config FNOSC = PRIPLL       // SYSCLK uses the primary oscillator with phase-locked loop (PLL)
#pragma config POSCMOD = HS         // use the high speed crystal mode for the primary oscillator
#pragma config FPLLIDIV = DIV_2     // PLL Input Divider:  Divide by 2
#pragma config FPLLMUL = MUL_20     // PLL Multiplier: Multiply by 20
#pragma config FPLLODIV = DIV_1     // PLL Output Divider: Divide by 1
#pragma config FPBDIV = DIV_1       // Peripheral Bus Clock: Divide by 1
#pragma config FSRSSEL = PRIORITY_7 // Shadow Register Set for interrupt priority 7
```

The commands above

- turn the PIC32's watchdog timer off;
- configures the PIC32's clock generation circuit to take the external 8 MHz resonator as input, divide this input frequency to a phase-locked loop (PLL) circuit by 2, multiply the frequency by 20, and divide the output frequency by 1, creating a SYSCLK of $8/2 * 20/1$ MHz = 80 MHz;
- set the PBCLK frequency to be SYSCLK divided by 1 (80 MHz); and
- sets the shadow register set to be used for interrupts of priority level 7 (see Chapter 6).

If you decide not to use a bootloader program on the PIC32, and instead use a programmer device like the PICkit 3 (Figure 2.5) to install a standalone program, you would have to make sure that your program appropriately sets the configuration bits with commands such as those above, turns on the prefetch cache module, etc. More information on this process can be found in Appendix **??**.

3.7 Build Summary

Recall that what we colloquially refer to as "compiling" actually consists of multiple steps. You initiated these steps by invoking the compiler, `xc32-gcc`, at the command line:

```
> xc32-gcc -mprocessor=32MX795F512L
   -o simplePIC.elf -Wl,--script=skeleton/NU32bootloaded.ld simplePIC.c
```

This step creates the `.elf` file, which then needs to be converted into a `.hex` file that the bootloader understands:

```
> xc32-bin2hex simplePIC.elf
```

The compiler requires multiple *command line options* to work. It accepts arguments, as detailed in the XC32 Users Manual, and some important ones are displayed by typing `xc32-gcc --help`. The arguments we used were

- `-mprocessor=32MX795F512L` Tells the compiler what PIC32 model to target. This also causes the compiler to define `__32MX795F512L__` so that the processor model can be detected in header files such as `xc.h`.
- `-o simplePIC.elf` Specifies that the final output will be named `simplePIC.elf`.
- `-Wl,` Tells the compiler that what follows are a comma-separated list of options for the linker.
- `--script=skeleton/NU32bootloaded.ld` A linker option that specifies the linker script to use.
- `simplePIC.c` The C files that you want compiled and linked are listed. In this case the whole program is in just one file.

Another option that may be useful when exploring what the compiler does is `--save-temps`. This option will save all of the intermediate files generated during the build process.

Here is what happens when you build and load `simplePIC.c`.

- **Preprocessing.** The preprocessor (`xc32-cpp`), among other duties, handles include files. By including `xc.h` at the beginning of your program, we get access to variables for all the SFRs. The output of the preprocessor is a `.i` file, which by default is not saved.
- **Compiling.** After the preprocessor, the compiler (`xc32-gcc`) turns your C code into assembly language specific to the PIC32. For convenience, (`xc32-gcc`) automatically invokes the other commands required in the build process. The result of the compilation step is an assembly language `.S` file, containing a human-readable version of instructions specific to a MIPS32 processor. This output is also not saved by default.

- **Assembling.** The assembler (`xc32-as`) converts the human-readable assembly code into object files (`.o`) that contain machine code instructions. These files cannot be executed directly, however, because addresses have not been resolved. This step yields `simplePIC.o`

- **Linking.** The object code `simplePIC.o` is linked with the `crt0_mips32r2.o` C run-time startup library, which performs functions such as initializing global variables, and the `processor.o` object code, which contains the SFR VAs. The linker script `NU32bootloaded.ld` provides information to the linker on the allowable absolute virtual addresses for the program instructions and data, as required by the bootloader and the specific PIC32 model. Linking yields a self-contained executable in `.elf` format.

- **Hex file.** The `xc32-bin2hex` utility converts `.elf` files into `.hex` files. The `.hex` is a different format for the executable then the `.elf` file that the bootloader understands and can load into the PIC32's program memory.

- **Installing the program.** The last step is to use the NU32 bootloader and the host computer's bootloader utility to install the executable. By resetting the PIC32 while holding the USER button, the bootloader enters a mode where it tries to communicate with the bootload communication utility on the host computer. When it receives the executable from the host, it writes the program instructions to the virtual memory addresses specified by the linker. Now every time the PIC32 is reset without holding the USER button, the bootloader exits and jumps to the newly installed program.

3.8 Useful Command Line Utilities

The `bin` directory of the XC32 installation contains a number of useful command line utilities. These can be used directly at the command line and many are invoked by the `Makefile`. We have already seen the first two of these utilities, as described in Section 3.7:

xc32-gcc The XC32 version of the `gcc` compiler is used to compile, assemble, and link, creating the executable `.elf` file.

xc32-bin2hex Converts a `.elf` file to a `.hex` file suitable for placing directly into PIC32 flash memory.

xc32-ar The archiver can be used to create an archive, list the contents of an archive, or extract object files from an archive. Example uses include:

```
xc32-ar -t lib.a         // list the object files in lib.a (in current directory)
xc32-ar -x lib.a code.o  // extract code.o from lib.a to the current directory
```

xc32-as The assembler.

xc32-ld This is the actual linker called by `xc32-gcc`.

xc32-nm Prints the symbols (e.g., global variables) in an object file. Examples:

```
xc32-nm processor.o      // list the symbols in alphabetical order
xc32-nm -n processor.o   // list the symbols in numerical order of their VAs
```

xc32-objdump Displays the assembly code corresponding to an object or `.elf` file. This process is called *disassembly*. Example:

```
xc32-objdump -S file.elf > file.dis   // send output to the file file.dis
```

xc32-readelf Displays a lot of information about the `.elf` file. Example:

```
xc32-readelf -a filename.elf  // output is dominated by SFR definitions
```

These utilities correspond to standard "GNU binary utilities" of the same name without the preceding `xc32-`. To learn the options available for a command called `xc32-cmdname`, you can type `xc32-cmdname --help` or read about them in the XC32 compiler reference manual.

3.9 Chapter Summary

OK, that's a lot to digest. Don't worry, you can view much of this chapter as reference material; you don't have to memorize it to program the PIC32!

- Software refers almost exclusively to the virtual memory map. Virtual addresses map directly to physical addresses by PA = VA & 0x1FFFFFFF.
- Building an executable `.hex` file from a source file consists of the following steps: preprocessing, compiling, assembling, linking, and converting the `.elf` file to a `.hex` file.
- 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/p32mx795f512l.h` contains variable declarations, like TRISA, that allow us to read from and write to the SFRs. We have several options for manipulating these SFRs. For TRISA, for example, we can directly assign the bits with `TRISA=0x30`, 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 TRISA using TRISAbits.TRISA3. 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/32MX795F512L/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.
- The bootloader sets the Device Configuration Registers, turns on 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 A through G. For the PIC32MX795F512L 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) I2C2CON. (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's 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 `p32mx795f512l.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 TRISASET, TRISACLR, and TRISAINV, that set bits 2 and 3 of TRISA to 1, clear bits 1 and 5, and flip bits 0 and 4.

Chapter 4

Using Libraries

You've used libraries all your life—well, at least as long as you've 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 `pic32mx795f512l.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's revisit `talkingPIC.c`, and examine how it includes libraries during the build process.

4.1 Talking PIC

The `talkingPIC.c` program, which you compiled in Chapter 1, relies heavily on libraries. All the calls starting with `NU32_` require the NU32 library and calls to `sprintf` use the C standard library `libc.a`. Recall from Chapter 1 that the `Makefile` will compile and link all `.c` files in the directory. Since the project directory, `<PIC32>/talkingPIC.c` 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:

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

```
> xc32-gcc -g -O1 -x c -c -mprocessor=32MX795F512L
 -I"<harmonyDir>/<harmonyVer>/framework" -I"<harmonyDir>/<harmonyVer>/framework/peripheral" -o talkingPIC.o talkin
> xc32-gcc -g -O1 -x c -c -mprocessor=32MX795F512L
 -I"<harmonyDir>/<harmonyVer>/framework" -I"<harmonyDir>/<harmonyVer>/framework/peripheral" -o NU32.o NU32.c
> xc32-gcc -mprocessor=32MX795F512L -o out.elf  talkingPIC.o  NU32.o
 -L"<harmonyDir>/<harmonyVer>/bin/framework/peripheral"  -l:PIC32MX795F512L_peripherals.a
   -Wl,--script="NU32bootloaded.ld",-Map=out.map
> xc32-bin2hex out.elf
> xc32-objdump -S out.elf > out.dis
```

The first two commands compile the modules necessary to create `talkingPIC`, using certain options:

- `-g` Include debugging information. This is 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 only, do not link. The output of this command is just an object (`.o`) file because the linker is not invoked to create the `.elf` file.
- `-I` Gives the compiler additional directories to search for include files. The particular directories contain the Harmony include files which will be needed in later chapters.

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 don't 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:

- `-L` Adds the following directory to the library search path; we have added the path to the Harmony peripheral library.
- `-l:` Tells the linker to include the following library when linking, in this case, the Harmony peripheral library `PIC32MX795F512L_peripherals.a`.
- `-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 the 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

#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 NU32_LED1 LATAbits.LATA4    // LED1 on the NU32 board
#define NU32_LED2 LATAbits.LATA5    // LED2 on the NU32 board
#define NU32_USER PORTDbits.RD13    // user button on the NU32 board
#define NU32_SYS_FREQ 80000000ul    // 80 million Hz

void NU32_Startup(void);
void NU32_ReadUART1(char * string, int maxLength);
void NU32_WriteUART1(const char * string);
unsigned int NU32_ReadCoreTimer(void);
void NU32_WriteCoreTimer(unsigned int value);
unsigned int NU32_EnableInterrupts(void);
unsigned int NU32_DisableInterrupts(void);
void NU32_EnableCache(void);
void NU32_DisableCache(void);
#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 header file sets the device configuration bits (see Appendix **??**; if not, the bootloader has already set them. The next few lines include Microchip-provided headers that you would otherwise need to include in most programs. You have already seen `xc.h`; we discuss `sys/attribs.h` in Chapter 6. The next three lines define aliases for SFRs the control the two LEDs (`NU32_LED1` and `NU32_LED2`) and the USER button (`NU32_USER`) on the NU32 board. Using these aliases allow 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 the PIC32 pin that are 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 setup 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 it can use RA4 and RA5 as digital outputs for LED1 and LED2. The function then configures UART1 so that the PIC32 can communicate with your computer. Configuring UART1 allows you to use `NU32_WriteUART1()` and `NU32_ReadUART1()` 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: all details of UART communication that we discuss in Chapter **??**. Finally, it enables interrupts (see Chapter 6).

void NU32_ReadUART1(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 UART1 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_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 sends a string over UART1. The function does not complete until the transmission has finished. Thus, if the host computer is not reading the UART, the function will wait to send its data.

Example:

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

unsigned int NU32_ReadCoreTimer(void) The core timer increments a register every other CPU cycle (see Chapter 5). This function returns the value of that counter.

void NU32_WriteCoreTimer(unsigned int value) This function sets the core timer counter to the provided value.

unsigned int NU32_EnableInterrupts(void) Enables all interrupts on the CPU. It returns the coprocessor 0 (CP0) STATUS register. This register is part of the CPU and contains information about the state of the processor, including whether interrupts are enabled.

unsigned int NU32_DisableInterrupts(void) Disables all interrupts on the CPU, returning the value of the CP0 STATUS register prior to disabling interrupts.

void NU32_EnableCache(void) This function makes memory in KSEG0 cacheable, allowing program instructions to be stored in cache.

void NU32_DisableCache(void) This function makes memory in KSEG0 uncacheable. This means that instructions stored in program flash will be read directly from flash memory rather than the cache.

4.3 Bootloaded Programs

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

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.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, 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 NU32 library functions, we use the lines above for consistency. This convention allows the same code to build correctly regardless of whether it is bootloaded (do not uncomment the first line) or standalone (uncomment the first line, see Appendix **??**). Including `"NU32.h"` and executing `NU32_Startup()` does the following:

- includes `<xc.h>`, providing SFR definitions
- includes `<sys/attribs.h>`, which is used when declaring interrupt service routines (ISRs) (see Chapter 6)
- defines the constants `NU32_LED1`, `NU32_LED2`, `NU32_USER`, and `NU32_SYS_FREQ`
- declares the NU32 library functions described above
- enables the prefetch cache and sets the minimum flash wait cycles
- configures pins RA4 and RA5 as outputs to control LED1 and LED2
- enables and configures UART1
- sets the device configuration bits, if `NU32_STANDALONE` is defined

4.4 An LCD Library

Dot matrix LCD screens are inexpensive portable devices that can display information to the user. LCD controllers allow you to more easily display text on the screen; often a screen comes packaged with a controller. We now discuss a library that allows the PIC32 to control a Hitachi HD44780 (or compatible) LCD controller connected to a 16x2 LCD screen. You can purchase these components and the associated support hardware as a pre-built module. The data sheet for this controller is available on the book's website, http://hades.mech.northwestern.edu/index.php/Pic32book.

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 **??**.

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 = 100Ω and R2 = 1000Ω, 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 PIC32 is sending data (i.e. text) or a command (i.e. 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

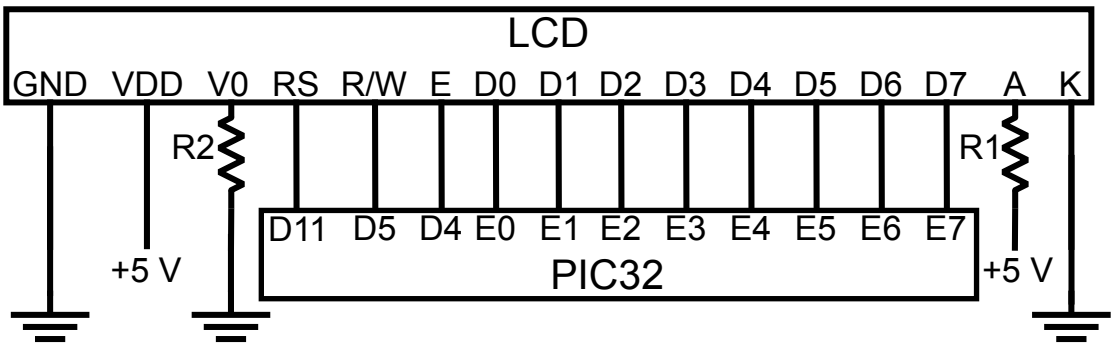


Figure 4.1: Circuit diagram for the LCD.

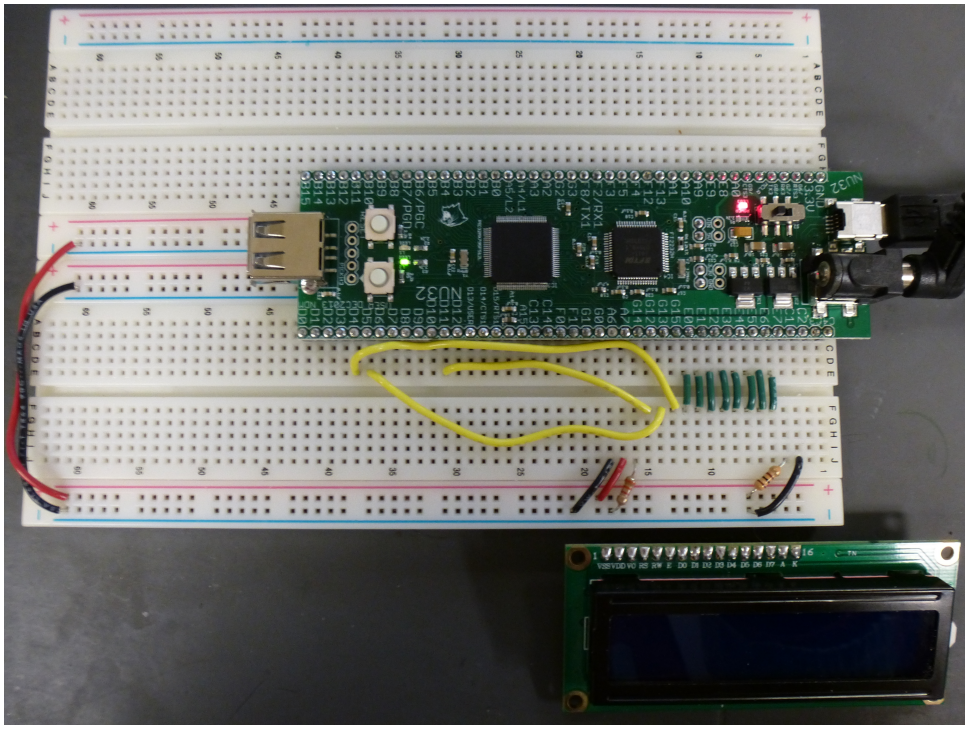


Figure 4.2: The NU32 board and the LCD screen.

sends 8 bits of data simultaneously, or in *parallel*. We delve into this parallel communication scheme more deeply in Ch. **??**, where we discuss the parallel master port (PMP), the peripheral that properly 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. More details can be found in **??** which discusses the implementation details.

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

```
#ifndef LCD_H
#define LCD_H

void LCD_Setup(void);                       // Initialize the LCD
void LCD_Clear(void);                       // Clear the screen and return to position (0,0)
```

```
void LCD_Move(int line, int col);           // Move the 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 a string, starting at the current position

void LCD_Home(void);                        // Move to (0,0) and reset any scrolling
void LCD_Entry(int id, int s);              // Control how the display moves after sending a character
void LCD_Display(int d, int c, int b);      // Turn the display on/off and change cursor settings
void LCD_Shift(int sc, int rl);             // Shift the position of the display
void LCD_Function(int n, int f);            // Set the number of lines (0,1) and the font size
void LCD_CustomChar(unsigned char val, const char data[7]); // Write a custom character 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 2 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 the user's host 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!

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

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

```
Echo!!__________
____Received___1___
```

where the underscores represent blank spaces. As the user sends 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.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // config bits, 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_WriteUART1("What do you want to write? ");
    NU32_ReadUART1(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 0 col 2
    NU32_WriteUART1("\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 requires manipulating a specific processor directly and doesn't contain convenient constructs like loops, if statements, or functions.

The C language, although still low-level, provides a level of portability and abstraction. Much of your C code works for multiple microcontrollers, provided you have a compiler for the 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 software that allows 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.

Microchip's most recent software release, Harmony, provides middleware, device drivers, and peripheral libraries. This permits an abstract programming model, partially insulating the programmer from hardware details. For some more complicated peripherals, we will use Harmony, which is why we include the options necessary for it in the `Makefile`. When beginning, however, we use only the SFR variable declarations and other definitions in the XC32 distribution to manipulate the hardware directly from C code. 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/p32mx795f512l.h`, search for the SFR name, and read the declarations related to that SFR. Overall, we believe that using this low-level approach to programming the PIC32 should provide you with a strong foundation in microcontroller programming. Additionally, after programing 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.

4.6 Your Libraries

Now that you've 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 as `static` (meaning that they cannot be used outside the module), 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.
- 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 address when the multiple objects are linked. If multiple object files have functions with the same name, 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. Explain what can go wrong if a header file contains the global variable definition `int i=2;` if that header file is included by multiple C files 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) Remove the comments from `invest.c` in 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) 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.

   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. 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.

6. 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.

Chapter 5

Time and Space

Of course it is a good idea to write "efficient" code. But "efficient" can mean a number of different things, such as time-efficient (runs fast), RAM-efficient (makes the most of limited RAM), flash-efficient (makes the most of limited flash), but perhaps most importantly, programmer-time-efficient (minimizes the time needed to write and debug the code, or for a future programmer to understand it). Often these interests are in competition with each other. In fact, the XC32 compiler provides a number of compilation options, some of which are not available in the free version of the compiler, that allow you to explicitly make space-time tradeoffs. As one example, the compiler could "unroll" loops. If a loop is known to be executed 20 times, for example, instead of using a small piece of code, incrementing a counter, and checking to see if the count has reached 20, the compiler could simply write the same block of code 20 times. This may save a little bit of execution time (no counter increments, no conditional tests, no branches) at the expense of using more flash to store the program.

The purpose of this chapter is to make you aware of some tools for understanding the time and space consumed by your program. These will help you squeeze the most out of your PIC32, allowing you to do more with a given PIC32 or to choose a cheaper PIC32. More importantly, though, they help you understand how your software works.

5.1 Compiler Optimization

The XC32 compiler provides five levels of optimization. Their availability depends on whether you have a license for the free version of the compiler, the Standard version, or the Pro version:

Version	Label	Description
All	`O0`	no optimization
All	`O1`	level 1: attempts to reduce both code size and execution time
Standard, Pro	`O2`	level 2: further reduces code size and execution time beyond `O1`
Pro	`O3`	level 3: maximum optimization for speed
Pro	`Os`	maximum optimization for code size

The greater the optimization, the longer it takes the compiler to produce the assembly code. You can learn more about compiler optimization in the XC32 C/C++ Compiler User's Guide.

When you issue a `make` command with the `Makefile` from the quickstart code, you see that the compiler is invoked with optimization level `O1`, using commands like

```
xc32-gcc -g -O1 -x c  ...
```

`-g -O1 -x c` are compiler flags set in the variable `CFLAGS` in the `Makefile`. The `-O1` means that optimization level 1 is being requested.

In this chapter we examine the assembly code that the compiler produces from your C code. The mapping between your C code and the assembly code is relatively direct when no optimization is used, but is less clear when optimization is invoked. (We will see an example of this in Section 5.2.3.) To create clearer assembly

code, we will find it useful to be able to `make` files with no optimization. This can be done by overriding the `CFLAGS` variable defined in the `Makefile`:

```
> make CFLAGS='-g -x c'
```

or

```
> make write CFLAGS='-g -x c'
```

In these examples, since no optimization level is being specified, the default (no optimization) is applied. Unless otherwise specified, all examples in this chapter assume that no optimization is applied.

5.2 Time and the Disassembly File

5.2.1 Timing Using a Stopwatch (or an Oscilloscope)

A direct way to time something is to toggle a digital output and look at that digital output using an oscilloscope or stopwatch. For example:

```
   ...                      // digital output RA4 has been high for some time
LATACLR = 0x10;             // clear RA4 to 0 (turn on NU32 LED1)
   ...                      // some code you want to time
LATASET = 0x10;             // set RA4 to 1 (turn off LED1)
```

The time that RA4 is low (or the NU32's LED1 is on) is approximately the duration of the code you want to measure.

If the duration is too short to catch with your scope or stopwatch, you could modify the code to something like

```
   ...                      // digital output RA4 has been high for some time
LATACLR = 0x10;             // clear RA4 to 0 (turn on NU32 LED1)
for (i=0; i<1000000; i++) { // but modify 1,000,000 to something appropriate for you
   ...                      // some code you want to time
}
LATASET = 0x10;             // set RA4 to 1 (turn off LED1)
```

Then you can divide the total time by 1,000,000.[1] Keep in mind, however, that there is overhead to implement the `for` loop (incrementing a counter, checking the inequality, etc.). We will see this in Section 5.2.3. If the code you want to time uses only a few assembly instructions, then the time you actually measure will be dominated by the implementation of the `for` loop.

5.2.2 Timing Using the Core Timer

A more accurate time can be obtained using a timer onboard the PIC32. The NU32's PIC32 has 6 timers: a 32-bit *core* timer, associated with the MIPS CPU, and five 16-bit *peripheral* timers. We can use the core timer for pure timing operations, leaving the much more flexible peripheral timers available for other tasks (see Chapter **??**). The core timer increments once for every two ticks of SYSCLK. For a SYSCLK of 80 MHz, the timer increments every 25 ns. Because the timer is 32 bits, it rolls over every $2^{32} \times 25$ ns $= 107$ seconds.

If you have compiled your program with the NU32 library `NU32.{c,h}`, you can use statements such as the following:

```
unsigned int elapsedticks, elapsedns;

NU32_WriteCoreTimer(0);               // set the core timer counter to 0
   ...                                // some code you want to time
elapsedticks = NU32_ReadCoreTimer();  // read the core timer
elapsedns = elapsedticks * 25;        // duration in ns, for 80 MHz SYSCLK
```

[1] If you use optimization in compiling your program, however, the compiler might recognize that you are not doing anything with the results of the loop, and not generate assembly code for the loop at all!

Writing to and reading from the core timer takes a few processor cycles, and the timer only counts every 2 ticks of SYSCLK. To minimize the uncertainty introduced by these, you can execute the code several times (just copy and paste it) between the write and read of the core timer. Avoid the overhead of implementing a loop.

We can actually do a bit better. Since we are concerned about timing, let's reduce the time to write to and read from the core timer by eliminating the small overhead of calling and returning from the NU32 functions:

```
unsigned int elapsedticks, elapsedns;

_CP0_SET_COUNT(0);                // set the core timer counter to 0
   ...                            // some code you want to time
elapsedticks = _CP0_GET_COUNT();  // read the core timer
elapsedns = elapsedticks * 25;    // duration in ns, for 80 MHz SYSCLK
```

The macros `_CP0_SET_COUNT(val)` and `_CP0_GET_COUNT()` are defined in `pic32mx/include/cp0defs.h`, and are resolved to macros in `pic32mx/include/xc.h`, which call functions that are built-in to the compiler, resulting in a minimum number of assembly language commands.

In the next section we look more systematically at the assembly code created by our C code.

5.2.3 Disassembling Your Code

A convenient way to examine the time efficiency of your code is to look at the assembly code produced by the compiler. The fewer instructions, the faster your code will execute.

In Chapter 3.5, we claimed that the code

```
LATAINV = 0x30;
```

is more efficient than

```
LATAbits.LATA4 = !LATAbits.LATA4; LATAbits.LATA5 = !LATAbits.LATA5;
```

Let's examine that claim by looking at the assembly code of the following program. This program simply delays by executing a `for` loop 50 million times, then toggles RA5 (LED2 on the NU32).

Code Sample 5.1. `timing.c`. RA5 toggles (LED2 on the NU32 flashes).

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // peripheral.h, config bits, constants, functions for startup and UART
#define DELAYTIME 50000000 // 50 million

void delay(void);
void toggleLight(void);

int main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  while(1) {
    delay();
    toggleLight();
  }
}

void delay(void) {
  int i;
  for (i = 0; i < DELAYTIME; i++) {
      ; //do nothing
```

```
  }
}

void toggleLight(void) {
  LATAINV = 0x0020; // invert the LED (which is on port A5)
  // LATAbits.LATA5 = !LATAbits.LATA5;
}
```

Put `timing.c` in a project directory together with the `Makefile`, `NU32.c`, `NU32.h`, and `NU32bootloaded.ld`, and nothing else. Then use

```
> make CFLAGS='-g -x c'
```

to build with no optimization. The `Makefile` automatically disassembles the `out.elf` file to the file `out.dis`, but if you wanted to do it manually, you could type

```
> xc32-objdump -S out.elf > out.dis
```

Open `out.dis` in a text editor. You will see a listing showing the assembly code corresponding to `out.hex`. Included in the file is your C code and, below your C statements, the assembly code it generated.[2] Each assembly line has the actual virtual address where the assembly instruction is placed in memory, the 32-bit machine instruction, and the equivalent human-readable (if you know assembly!) assembly code. Let's look at the segment of the listing corresponding to the command `LATAINV = 0x20`. You should see something like

```
  LATAINV = 0x0020; // invert the LED (which is on port A5)
9d002464: 3c02bf88  lui v0,0xbf88
9d002468: 24030020  li v1,32
9d00246c: ac43602c  sw v1,24620(v0)
  // LATAbits.LATA5 = !LATAbits.LATA5;
```

We see that the `LATAINV = 0x20` command has expanded to three assembly statements. Without going into detail[3], the `li` stores the base-10 value 32 (or hex 0x20) in the CPU register `v1`, which is then written by the `sw` command to the memory address corresponding to LATAINV.

If instead we comment out the `LATAINV = 0x0020;` command and replace it with the bit manipulation version, we get the following disassembly:

```
  // LATAINV = 0x0020; // invert the LED (which is on port A5)
  LATAbits.LATA5 = !LATAbits.LATA5;
9d002464: 3c02bf88  lui v0,0xbf88
9d002468: 8c426020  lw v0,24608(v0)
9d00246c: 30420020  andi v0,v0,0x20
9d002470: 2c420001  sltiu v0,v0,1
9d002474: 304400ff  andi a0,v0,0xff
9d002478: 3c03bf88  lui v1,0xbf88
9d00247c: 8c626020  lw v0,24608(v1)
9d002480: 7c822944  ins v0,a0,0x5,0x1
9d002484: ac626020  sw v0,24608(v1)
```

The bit manipulation version requires nine assembly statements. Basically the value of LATA is being copied to a CPU register, manipulated, then stored back in LATA. In contrast, with the LATAINV syntax, there is no copying the values of LATAINV back and forth.

Although one method of manipulating the SFR bit appears three times slower than the other, we don't yet know how many CPU cycles each consumes. Assembly instructions are generally performed in a single clock cycle, but there is still the question of whether the CPU is getting one instruction per cycle. (Recall the issue of slow program flash.) We will look at this further with the prefetch cache module in Section 5.2.4 below. For now, though, let's time that delay loop that is executed 50 million times. Here is the disassembly for `delay()`, with comments added to the right:

[2] The output from the `xc32-objdump` disassembler is not perfect. While the assemmbly code should be correct, portions of your C code may be duplicated for no apparent reason.

[3] You can look up the MIPS32 assembly instruction set if you're interested.

```
void delay(void) {
9d002408: 27bdfff0  addiu sp,sp,-16    // manipulate the stack pointer on ...
9d00240c: afbe000c  sw s8,12(sp)       // ... entering the function (see text)
9d002410: 03a0f021  move s8,sp
  int i;
  for (i = 0; i < DELAYTIME; i++) {
9d002414: afc00000  sw zero,0(s8)      // initialization of i in RAM to 0
9d002418: 0b40090b  j 9d00242c         // jump to 9d00242c (skip adding 1 to i), but ...
9d00241c: 00000000  nop                // ... "no operation" executed in "delay slot" before jump
9d002420: 8fc20000  lw v0,0(s8)        // start of the loop; load RAM i into register v0
9d002424: 24420001  addiu v0,v0,1      // add 1 to v0 ...
9d002428: afc20000  sw v0,0(s8)        // ... and store it to i in RAM
9d00242c: 8fc30000  lw v1,0(s8)        // load i into register v1
9d002430: 3c0202fa  lui v0,0x2fa       // load the upper 16 bits and ...
9d002434: 3442f080  ori v0,v0,0xf080   // ... the lower 16 bits of 50,000,000 into v0
9d002438: 0062102a  slt v0,v1,v0       // store "true" (1) in v0 if v1 < v0
9d00243c: 1440fff8  bnez v0,9d002420   // if v0 does not equal 0, branch to top of loop, but ...
9d002440: 00000000  nop                // ... branch delay slot is executed before branch
      ; //do nothing
  }
}
9d002444: 03c0e821  move sp,s8         // manipulate the stack pointer on exiting
9d002448: 8fbe000c  lw s8,12(sp)
9d00244c: 27bd0010  addiu sp,sp,16
9d002450: 03e00008  jr ra              // jump to return address ra stored by jal, but ...
9d002454: 00000000  nop                // ... jump delay slot is executed before jump
```

There are nine instructions in the delay loop itself, starting with `lw v0,0(s8)` and ending with the next `nop`. When the LED comes on, these instructions are carried out 50 million times, and then the LED turns off. (There are a few other instructions to set up the loop, but the duration of these is negligible compared to the 50 million executions of the loop.) So if one instruction is executed per cycle, we would predict the light to stay on for approximately 50 million $\times$ 9 instructions $\times$ 12.5 ns/instruction = 5.625 seconds. When we time by a stopwatch, we get about 6.25 seconds, which implies 10 CPU (SYSCLK) cycles per loop. So our cache module has the CPU executing one assembly instruction almost every cycle.

In the code above there are two "jumps" (`j` for "jump" to the specified address and `jr` for "jump register" to jump to the address in the return address register `ra`, which was set by the calling function) and one "branch" (`bnez` for "branch if not equal to zero"). For MIPS32, the command after a jump or branch is executed before the jump actually occurs. This next command is said to be in the "delay slot" for the jump or branch. In all three delay slots in this code is a `nop` command, which stands for "no operation."

You might notice a few ways you could have written the assembly code for the delay function to use fewer assembly commands. This is certainly one of the advantages of coding directly in assembly: direct control of the processor instructions. The disadvantage, of course, is that MIPS32 assembly is a much lower-level language than C, requiring significantly more knowledge of MIPS32 from the programmer. Until you have already invested a great deal of time learning the assembly language, programming in assembly fails the "programmer-time-efficient" criterion! (Not to mention that `delay()` was designed to waste time, so no need to minimize assembly lines!)

Another thing you may have noticed in the disassembly of `delay()` is the manipulation of the *stack pointer* (`sp`) upon entering and exiting the function. The *stack* is an area of RAM that holds temporary local variables and parameters. When a function is called, its parameters and local variables are "pushed" onto the stack. When the function exits, the local variables are "popped" off of the stack by moving the stack pointer back to its original position before the function was called. A *stack overflow* occurs if there is not enough RAM available to the stack to hold all the local variables defined in currently-called functions. We will see the stack again in Section 5.3.

The overhead due to passing parameters and manipulating the stack pointer on entering and exiting a function should not discourage you from writing modular code. This should only be a concern when your code is fully debugged and you are trying to squeeze a final few nanoseconds out of your program execution time.

Finally, if you compiled `timing.c` with optimization level 1 (the optimization flag `-O1`, the default for the `Makefile`), you would see that `delay()` is optimized to

```
void delay(void) {
9d00220c: 3c0202fa  lui v0,0x2fa      // load the upper 16 bits and ...
9d002210: 3442f080  ori v0,v0,0xf080  // ... the lower 16 bits of 50,000,000 into v0
9d002214: 2442ffff  addiu v0,v0,-1    // subtract 1 from v0
  int i;
  for (i = 0; i < DELAYTIME; i++) {
9d002218: 1440ffff  bnez v0,9d002218  // if v0 != 0, branch back to the same line, but ...
9d00221c: 2442ffff  addiu v0,v0,-1    // ... before branch completes, subtract 1 from v0
    ; //do nothing
  }
}
9d002220: 03e00008  jr ra             // jump to return address ra stored by jal
9d002224: 00000000  nop               // no operation in jump delay slot
```

No local variables are stored in RAM, and there is no stack pointer manipulation upon entering and exiting the function. The counter variable is simply stored in a CPU register. The loop itself has only two lines instead of nine, and it has been designed to count down from 49,999,999 to zero instead of counting up. The branch delay slot is actually used to implement the counter update instead of having a wasted `nop` cycle.

More importantly, however, `delay()` is never called by the assembly code for `main` in our `-O1` optimized code! The compiler has recognized that `delay()` doesn't do anything. As a result, the LED toggles so quickly that you can't see it by eye. The LED just looks dim.[4]

5.2.4 The Prefetch Cache Module

In the previous section, we saw that our `timing.c` program was executing an assembly instruction nearly every clock cycle. This is because `NU32_Startup()` optimized performance by turning on the prefetch cache module and choosing the minimum number of CPU wait cycles for instructions loading from flash.[5]

Let's try turning off the prefetch cache module to see the effect on our program `timing.c`. The prefetch cache module performs two primary tasks: (1) it keeps recent instructions in the cache, ready if the CPU requests the instruction at that address again (allowing the cache to completely store small loops); and (2) for linear code it runs ahead so as to have the instruction ready to go when needed (prefetch). We can disable each of these functions separately, or we can disable both.

Let's start by disabling both. Modify `timing.c` in Code Sample 5.1 by adding

```
NU32_DisableCache(); // Turn off function (1), storing recent instructions in cache
CHECONCLR = 0x30;    // Turn off function (2), prefetch
```

right after `NU32_Startup()` in `main`. Everything else stays the same. The first line is an NU32 function to turn off storing recent instructions in cache. As for the second line, consulting the section on the prefetch cache module in the Reference Manual, we see that bits 4 and 5 of the SFR CHECON determine whether instructions are prefetched, and that clearing both bits disables predictive prefetch.

Recompiling `timing.c` with with no compiler optimizations and rerunning, we find that the LED stays on for approximately 17 seconds, compared to approximately 6.25 seconds before. This corresponds to 27 SYSCLK cycles per delay loop, which we saw earlier has nine assembly commands. These numbers make sense—since the prefetch cache is completely disabled, it takes three CPU cycles (one request cycle plus two wait cycles) for each instruction to get from flash to the CPU.

[4] To prevent `delay()` from being optimized away, we could have added a "no operation" `_nop();` command inside the delay loop. Or we could have used a `volatile` variable inside the loop. Or we could just use polling of the core timer to implement a desired delay.

[5] The number of "wait cycles" is the number of extra cycles the CPU is told to wait for instructions to finish loading from flash if they are not cached. Since the PIC32's flash operates at a maximum of 30 MHz and the CPU operates at 80 MHz, the number of wait cycles is configured as two in `NU32_Startup()`, to allow three total cycles for a flash instruction to load. Fewer wait cycles would result in an error in operation, and more wait cycles would slow performance unnecessarily.

If we comment out the second line, so that (1) the cache of recent instructions is off but (2) the prefetch is enabled, and rerun, we find that the LED stays on for about 8.1 seconds, or 13 SYSCLK cycles per loop, a small penalty compared to our original performance of 10 cycles. The prefetch is able to run ahead to grab future instructions, but it cannot run past the `for` loop conditional statement, since it does not know the outcome of the test.

Finally, if we comment out the first line but leave the second line uncommented, so that (1) the cache of recent instructions is on but (2) the prefetch is disabled, we recover our original performance of approximately 6.25 seconds or 10 SYSCLK cycles per loop. The reason is that the entire loop is stored in the cache, so prefetch is not necessary.

5.2.5 Math

For real-time systems, it is often critical to perform mathematical operations as quickly as possible. Mathematical expressions should be coded to minimize execution time. We will delve into the speed of various math operations in the Exercises, but here are a few rules of thumb for efficient math:

- There is no floating point unit on the PIC32MX, so all floating point math is carried out in software. Integer math is much faster than floating point math. If speed is an issue, perform all math as integer math, scaling the variables as necessary to maintain precision, and only convert to floating point when needed.
- Floating point division is slower than multiplication. If you will be dividing by a fixed value many times, consider taking the reciprocal of the value once and then using multiplication thereafter.
- Functions such as trigonometric functions, logarithms, square roots, etc. in the math library are generally slower to evaluate than arithmetic functions. Their use should be minimized when speed is an issue.
- Partial results should be stored in variables for future use to avoid performing the same computation multiple times.

5.3 Space and the Map File

The previous section focused on the time of execution. Now let's look at how much program memory (flash) and data memory (RAM) our programs use.

The linker allocates virtual addresses in program flash for all program instructions, and virtual addresses in data RAM for all global variables. The rest of RAM is allocated to the *heap* and the *stack*.

The heap is memory set aside to hold dynamically allocated memory, as allocated by `malloc` and `calloc`. These functions allow you to declare a variable size array, for example, while the program is running, instead of specifying a (possibly space-wasteful) fixed-sized array in advance.

The stack holds temporary local variables used by functions. When a function is called, space on the stack is allocated for its local variables. When the function exits, the local variables are thrown away and the space is made available again by simply moving the stack pointer. The stack grows "down" from the end of RAM—as local variables are "pushed" onto the stack, the stack pointer address decreases, and when local variables are "popped" off the stack after exiting a function, the stack pointer address increases. (See the assembly listing for `delay()` in `timing.c` in Section 5.2.3 for an example of moving the stack pointer when a function is called and when it exits.)

If your program attempts to put too many local variables on the stack (stack overflow), the error won't show up until run time. The linker does not catch this error because it does not explicitly set aside space for temporary local variables; it assumes they will be handled by the stack.

To dig a little deeper into how memory is allocated, we can ask the linker to create a "map" file when it creates the `.elf` file. The map file indicates where instructions are placed in program memory and where global variables are placed in data memory. Your `Makefile` automatically creates an `out.map` file for you by including the `-Map` option to the linker command:

```
> xc32-gcc [details omitted] -Wl,--script="NU32bootloaded.ld",-Map="out.map"
```

The map file can be opened with a text editor.

Let's look at the `out.map` file for `timing.c` as shown in Code Sample 5.1, and again compiled with no optimizations. There's a lot in this file, but here's an edited portion of it:

```
Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.text                    0x9d001e00          0x4fc        1276  App's exec code
.text.general_exception  0x9d0022fc           0xdc         220
.text                    0x9d0023d8           0xac         172  App's exec code
.text.main_entry         0x9d002484           0x4c          76
.text._bootstrap_except  0x9d0024d0           0x48          72
.text._general_exceptio  0x9d002518           0x48          72
.text                    0x9d002560           0x44          68  App's exec code
.vector_default          0x9d0025a4           0x38          56
.text                    0x9d0025dc           0x18          24  App's exec code
.dinit                   0x9d0025f4           0x10          16
.text._on_reset          0x9d002604            0x8           8
.text._on_bootstrap      0x9d00260c            0x8           8
      Total kseg0_program_mem used  :       0x814        2068  0.4% of 0x7e200

kseg0 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
         Total kseg0_boot_mem used  :           0           0  <1% of 0x970

Exception-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.app_excpt               0x9d000180           0x10          16  General-Exception
.vector_0                0x9d000200            0x8           8  Interrupt Vector 0
.vector_1                0x9d000220            0x8           8  Interrupt Vector 1

  [[[ ... snipping long list of vectors ...]]]

.vector_51               0x9d000860            0x8           8  Interrupt Vector 51
          Total exception_mem used  :       0x1b0         432  10.5% of 0x1000

kseg1 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.reset                   0xbd001970          0x1f0         496  Reset handler
.bev_excpt               0xbd001cf0           0x10          16  BEV-Exception
         Total kseg1_boot_mem used  :       0x200         512  43.8% of 0x490
        ------------------------------------------------------------------------
         Total Program Memory used  :       0xbc4        3012  0.6% of 0x80000
        ------------------------------------------------------------------------
```

The kseg0 program memory usage report tells us that 2068 (or `0x814`) bytes are used for the main part of our program. The first entry is denoted `.text`, which stands for program instructions. It is the largest single section, using 1276 bytes, described as `App's exec code`, and installed starting at VA 0x9d001e00. Searching for this address in the map file, we see that this is the code for `NU32.o`, the object code associated with the NU32 library.

Going down through the subsequent sections of kseg0 program memory, we see that the sections are packed tightly and in order of decreasing section size. The next section is `.text.general_exception`, which corresponds to a routine that is called when the CPU encounters certain types of "exceptions" (run-time

errors). This code was linked from `pic32mx/lib/libpic32.a`. The next `.text` section, also labeled `App's exec code`, is the obect code `timing.o`, 172 (or `0xac`) bytes long. Searching for `timing.o` we find the following text:

```
.text           0x9d0023d8       0xac
 .text          0x9d0023d8       0xac timing.o
                0x9d0023d8                main
                0x9d002408                delay
                0x9d002458                toggleLight
```

Our functions `main`, `delay`, and `toggleLight` of `timing.o` are stored consecutively in memory. The addresses agree with our disassembly file from Section 5.2.3.

Continuing, the kseg0 boot memory report indicates that no code is placed in this memory region. The exception memory report indicates that placeholders for instructions corresponding to interrupts occupy 432 bytes. Finally, the kseg1 boot memory report indicates that the C runtime startup code installed reset functions that occupy 512 bytes. The address of the `.reset` section is the address that the bootloader (already installed in the 12 K boot flash) jumps to.

In all, 3012 bytes of the 512 KB of program memory are used.

Continuing further in the map file, we see

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
         Total kseg1_data_mem used  :           0           0  <1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :           0           0  <1% of 0x20000
        ------------------------------------------------------------------------


Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
heap                     0xa0000008           0           0  Reserved for heap
stack                    0xa0000020        0x1ffd8      131032  Reserved for stack
```

There are no global variables, so no kseg1 data memory is used. The heap size is zero, so essentially all data memory is reserved for the stack.

Now let's modify our program by adding some useless global variables, just to see what happens to the map file. Let's add the following lines just before `main`:

```
char my_cat_string[] = "2 cats!";
int my_int = 1;
char my_message_string[] = "Here's a long message stored in a character array.";
char my_small_string[6], my_big_string[97];
```

Rebuilding and examining the new map file, we see the following for the data memory report:

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.sdata                   0xa0000000            0xc          12  Small init data
.sbss                    0xa000000c            0x6           6  Small uninit data
.bss                     0xa0000014           0x64         100  Uninitialized data
.data                    0xa0000078           0x34          52  Initialized data
         Total kseg1_data_mem used  :          0xaa         170  0.1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :          0xaa         170  0.1% of 0x20000
        ------------------------------------------------------------------------
```

Our global variables now occupy 170 bytes of data RAM. The global variables have been placed in four different data memory sections, depending on whether the variable is small or large (according to a command line option or `xc32-gcc` default) and whether or not it is initialized:

section name	data type	variables stored there
`.sdata`	small initialized data	`my_cat_string`, `my_int`
`.sbss`	small uninitialized data	`my_small_string`
`.bss`	larger uninitialized data	`my_big_string`
`.data`	larger initialized data	`my_message_string`

Searching for the `.sdata` section further in the map file, we see

```
 .sdata         0xa0000000        0xc timing.o
                0xa0000000                my_cat_string
                0xa0000008                my_int
                0xa000000c                _sdata_end = .
```

Even though the string `my_cat_string` uses only 7 bytes, the variable `my_int` starts 8 bytes after the start of `my_cat_string`. This is because variables are aligned on four-byte boundaries. Similarly, the strings `my_message_string`, `my_small_string`, and `my_big_string` occupy memory to the next four-byte boundary. You are not saving memory by defining a string as 5 bytes instead of 8 bytes.

Apart from the addition of these sections to the data memory usage report, we see that the global variables reduce the data memory available for the stack, and the `.dinit` (global data initialization, from the C runtime startup code) section of the kseg0 program memory report has grown to 112 bytes, meaning that our total kseg0 program memory used is now 2164 bytes instead of 2068.

Now let's make one last change. Let's move the definition

```
char my_cat_string[] = "2 cats!";
```

inside the `main` function, so that `my_cat_string` is now local to `main`. Building the program again, we find in the data memory report that the initialized global variable section `.sdata` has shrunk by 8 bytes, as expected.

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.sdata                   0xa0000000             0x4           4  Small init data
.sbss                    0xa0000004             0x6           6  Small uninit data
.bss                     0xa000000c            0x64         100  Uninitialized data
.data                    0xa0000070            0x34          52  Initialized data
         Total kseg1_data_mem used  :         0xa2         162  0.1% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :         0xa2         162  0.1% of 0x20000
        ------------------------------------------------------------------------
```

Now looking at the kseg0 program memory report

```
kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.text                    0x9d001e00           0x4fc        1276  App's exec code
.text.general_exception  0x9d0022fc            0xdc         220
.text                    0x9d0023d8            0xc4         196  App's exec code
.dinit                   0x9d00249c            0x60          96

  [[[ ... snipping long kseg0_program_mem report ...]]]

.rodata                  0x9d00266c             0x8           8  Read-only const
.text._on_reset          0x9d002674             0x8           8
.text._on_bootstrap      0x9d00267c             0x8           8
      Total kseg0_program_mem used  :        0x884        2180  0.4% of 0x7e200
```

we see that `timing.o` is now 196 bytes as compared to 172 before. This is because the initialization of `my_cat_string` is now taken care of by assembly commands in the code, not by the global variable initialization in `.dinit`. A new section, `.rodata` for "read only data," appears in the program memory usage, corresponding to the string `"2 cats!"`. The global data initialization section `.dinit` shrinks from 112 bytes to 96 bytes since it is no longer responsible for initializing `my_cat_string`.

Finally, we might wish to reserve some RAM for dynamic memory allocation using `malloc` or `calloc`. By default, the heap size is set to zero. To set a nonzero heap size, we can pass a linker option to `xc32-gcc`:

```
xc32-gcc [details omitted] -Wl,--script="NU32bootloaded.ld",-Map="out.map",--defsym=_min_heap_size=4096
```

This defines a heap of 4 KB. After building, the map file shows

```
Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  ------------------------  -----------
heap                      0xa00000a8         0x1000       4096  Reserved for heap
stack                     0xa00010c0        0x1ef30     126768  Reserved for stack
```

The heap is allocated at low RAM addresses, close after the global variables, starting in this case at address 0xa00000a8. The stack occupies most of the rest of RAM.

5.4 Chapter Summary

- The CPU's core timer increments once every two ticks of the SYSCLK, or every 25 ns for an 80 MHz SYSCLK. The commands `NU32_WriteCoreTimer(0);` and `unsigned int dt = NU32_ReadCoreTimer();` can be used to measure the execution time of the code in between to within a few SYSCLK cycles.
- To generate a disassembly listing at the command line, use `xc32-objdump -S filemane.elf > filename.dis`.
- With the prefetch cache module fully enabled, your PIC32 should be able to execute an assembly instruction nearly every cycle. The prefetch allows instructions to be fetched in advance for linear code, but the prefetch cannot run past conditional statements. For small loops, the entire loop can be stored in the cache.
- The linker assigns specific program flash VAs to all program instructions and data RAM VAs to all global variables. The remainder of RAM is allocated to the heap, for dynamic memory allocation, and to the stack, for function parameters and temporary local variables. The heap is zero bytes by default.
- A map file provides a detailed summary of memory usage. To generate a map file at the command line, use the `-Map` option to the linker, e.g.,

  ```
  xc32-gcc [details omitted] -Wl,-Map="out.map"
  ```

- Global variables can be initialized (assigned a value when they are defined) or uninitialized. Initialized global variables are stored in RAM memory sections `.data` and `.sdata` and uninitialized globals are stored in RAM memory sections `.bss` and `.sbss`. Sections beginning with `.s` mean that the variables are "small." When the program is executed, initialized global variables are assigned their values by C runtime startup code, and uninitialized global variables are set to zero.
- Global variables are packed tightly at the beginning of data RAM, 0xA0000000. The heap comes immediately after. The stack begins at the high end of RAM and grows "down" toward lower RAM addresses. Stack overflow occurs if the stack pointer attempts to move into an area reserved for the heap or global variables.

5.5 Exercises

Unless otherwise specified, compile with no optimizations for all problems.

1. Describe two examples of how you can write code differently to either make it execute faster or use less program memory.

2. Compile and run `timing.c`, Code Sample 5.1, with no optimizations (`make CFLAGS='-g -x c'`). With a stopwatch, verify the time taken by the delay loop. Do your results agree with Section 5.2.3?

3. You will look at the disassembly for two programs with a similar function.

   (a) Write a short program that uses `NU32_WriteCoreTimer(0)` and `elapsed = NU32_ReadCoreTimer()` to time a few C statements. Disassemble your executable and look at it. If you assume that one assembly instruction is executed per clock cycle, how many SYSCLK cycles does it take to complete the `NU32_WriteCoreTimer` command? How many cycles does it take to complete the `NU32_ReadCoreTimer` command? Approximately how much error will you have in your estimate of the timed code? (It's not a sum of the two.)

   (b) Now replace the `NU32_WriteCoreTimer(0)` and `elapsed = NU32_ReadCoreTimer()` with the `cp0defs.h` macros, as in Section 5.2.2. Disassemble and look at the code, and answer the same questions.

4. To write time-efficient code, it is important to understand that some mathematical operations are faster than others. We will look at the disassembly of code that performs simple arithmetic operations on different data types. Create a program with the following local variables in `main`:

   ```
   char c1=5, c2=6, c3;
   int i1=5, i2=6, i3;
   long long int j1=5, j2=6, j3;
   float f1=1.01, f2=2.02, f3;
   long double d1=1.01, d2=2.02, d3;
   ```

   Now write code that performs add, subtract, multiply, and divide for each of the five data types, i.e., for `char`s:

   ```
   c3 = c1+c2;
   c3 = c1-c2;
   c3 = c1*c2;
   c3 = c1/c2;
   ```

   Build the program with no optimization and look at the disassembly. For each of the statements, you'll notice that some of the assembly code involves simply loading the variables from RAM into CPU registers and storing the result (also in a register) back to RAM. Also, while some of the statements are completed by a few assembly commands in sequence, others result in a jump to a software subroutine to complete the calculation. (These subroutines are provided with our C installation and included in the linking process.) Answer the following questions.

   (a) Which combinations of data types and arithmetic functions result in a jump to a subroutine? From your disassembly file, copy the C statement and the assembly commands it expands to (including the jump) for one example.

   (b) For those statements that do not result in a jump to a subroutine, which combination(s) of data types and arithmetic functions result in the fewest assembly commands? From your disassembly, copy the C statement and its assembly commands for one of these examples. Is the smallest data type, `char`, involved in it? If not, what is the purpose of extra assembly command(s) for the `char` data type vs. the `int` data type? (Hint: the assembly command `ANDI` takes the bitwise AND of the second argument with the third argument, a constant, and stores the result in the first argument. Or you may wish to look up a MIPS32 assembly instruction reference.)

(c) Fill in the following table. Each cell should have two numbers: the number of assembly commands for the specified operation and data type, and the ratio of this number (greater than or equal to 1.0) to the smallest number of assembly commands in the table. For example, if addition of two `int`s takes four assembly commands, and this is the fewest in the table, then the entry in that cell would be 1.0 (4). This has been filled in below, but you should change it if you get a different result. If a statement results in a jump to a subroutine, write J in that cell.

	`char`	`int`	`long long`	`float`	`long double`
+		1.0 (4)			
−					
*					
/					

(d) From the disassembly, find out the name of any math subroutine that has been added to your assembly code. Now create a map file of the program. Where are the math subroutines installed in virtual memory? Approximately how much program memory is used by each of the subroutines? You can use evidence from the disassembly file and/or the map file. (Hint: You can search backward from the end of your map file for the name of any math subroutines.)

5. Let's look at the assembly code for bit manipulation. Create a program with the following local variables:

```
unsigned int u1=33, u2=17, u3;
```

and look at the assembly commands for the following statements:

```
u3 = u1 & u2;     // bitwise AND
u3 = u1 | u2;     // bitwise OR
u3 = u2 << 4;     // shift left 4 spaces, or multiply by 2^4 = 16
u3 = u1 >> 3;     // shift right 3 spaces, or divide by 2^3 = 8
```

How many commands does each use? For unsigned integers, bit-shifting left and right make for computationally efficient multiplies and divides, respectively, by powers of 2.

6. Use the core timer to calculate a table similar to that in Problem 4, except with entries corresponding to the actual execution time in terms of SYSCLK cycles. So if a calculation takes 15 cycles, and the fastest calculation is 10 cycles, the entry would be 1.5 (15). This table should contain all 20 entries, even for those that jump to subroutines. (Note: subroutines often have conditional statements, meaning that the calculation could terminate faster for some operands than for others. You can report the results for the variable values given in Problem 4.)

   To minimize uncertainty due to the setup and reading time of the core timer, and the fact that the timer only increments once every two SYSCLK cycles, each math statement could be repeated ten or more times (no loops) between setting the timer to zero and reading the timer. The average number of cycles, rounded down, should be the number of cycles for each statement. Use the NU32 communication routines, or any other communication routines, to report the answers back to your computer.

7. Certain math library functions can take quite a bit longer to execute than simple arithmetic functions. Examples include trigonometric functions, logarithms, square roots, etc. Make a program with the following local variables:

```
float f1=2.07, f2;        // four bytes for each float
long double d1=2.07, d2;  // eight bytes for each long double
```

Also be sure to put `#include <math.h>` at the top of your program to make the math function prototypes available.

(a) Using methods similar to those in Problem 6, measure how long it takes to perform each of `f2 = cosf(f1)`, `f2 = sqrtf(f1)`, `d2 = cos(d1)`, and `d2 = sqrt(d1)`.

(b) Copy and paste the disassembly from a `f2 = cosf(f1)` statement and a `d2 = cos(d1)` statement into your solution set and compare them. Based on the comparison of the assembly codes, comment on the advantages and disadvantages of using the eight-byte `long double` floating point representation compared to the four-byte `float` representation when you compute a cosine with the PIC32 compiler.

(c) Make a map file for this program, and search for the references to the math library `libm.a` in the map file. There are several `libm.a` files in your C installation, but which one was used by the linker when you built your program? Give the directory.

8. Explain what stack overflow is, and give a short code snippet (not a full program) that would result in stack overflow on your PIC32.

9. In the map file of the original `timing.c` program, there are several `App's exec code`, one corresponding to `timing.o`. Explain briefly what each of the others are for. Provide evidence for your answer from the map file.

10. Create a map file for `simplePIC.c` from Chapter 3. (a) How many bytes does `simplePIC.o` use? (b) Where are the functions `main` and `delay` placed in virtual memory? Are instructions at these locations cacheable? (c) Search the map file for the `.reset` section. Where is it in virtual memory? Is it consistent with your `NU32bootloaded.ld` linker file? (d) Now augment the program by defining `short int`, `long int`, `long long int`, `float`, `double`, and `long double` global variables. Provide evidence from the map file indicating how much memory each data type uses.

11. Assume your program defines a global `int` array `int glob[5000]`. Now what is the maximum size of an array of `int`s that you can define as a local variable for your particular PIC32?

12. Provide global variable definitions (not an entire program) so that the map file has data sections `.sdata` of 16 bytes, `.sbss` of 24 bytes, `.data` of 0 bytes, and `.bss` of 200 bytes.

13. If you define a global variable and you want to set its initial value, is it "better" to initialize it when the variable is defined or to initialize it in a function? Explain any pros and cons.

Chapter 6

Interrupts

Say the PIC32 is attending to some mundane task when an important event occurs. For example, the user has pressed a button. We want the PIC32 to respond immediately. To do so, we have this event generate an interrupt, or *interrupt request* (IRQ), which interrupts the program and sends the CPU to execute some other code, called the *interrupt service routine* (ISR). Once the ISR has completed, the CPU returns to its original task.

Interrupts are a key concept in embedded real-time control, and they can arise from many different events. This chapter provides a summary of PIC32 interrupt handling.

6.1 Overview

Interrupts can be generated by the processor core, peripherals, and external inputs. Example events include

- a digital input changing its value,
- information arriving on a communication port,
- the completion of some task a PIC32 peripheral was executing in parallel with the CPU, and
- the elapsing of a specified amount of time.

As an example, to guarantee performance in real-time control applications, sensors must be read and new control signals calculated at a known fixed rate. For a robot arm, a common control loop frequency is 1 kHz. So we could configure one of the PIC32's counter/timers to use the peripheral bus clock as input and roll over every 80,000 ticks $\times$ 12.5 ns/tick = 1 ms. This roll-over event generates the interrupt that calls the feedback control ISR, which reads sensors and produces output. In this case, we would have to make sure that the control ISR is efficient code that always executes in less than 1 ms. (To check this, you could use the core timer to measure the time between entering and exiting the ISR.)

Imagine the PIC32 is controlling the robot arm to hold steady at a particular position when it receives a message from the user over the UART, asking the arm to move to a new position. The arrival of data on the UART generates an interrupt, and the corresponding ISR reads in the information and stores it in global variables representing the desired state. These desired states are used in the feedback control ISR.

So what happens if the PIC32 is in the middle of executing the control ISR when the communication interrupt is generated? Or if the PIC32 is in the middle of the communication ISR and a control interrupt is generated? We have to choose which has higher priority. If a high priority interrupt occurs while a low priority ISR is executing, the CPU will jump to the high priority ISR, complete it, and then return to finish the low priority ISR. If a low priority interrupt occurs while a high priority ISR is executing, the low priority ISR will remain pending until the high priority ISR is finished executing. When it is finished, the CPU jumps to the low priority ISR.

In our example, communication could be slow, and we might not have a guarantee as to the duration. To ensure the stability of the robot arm, we would probably choose the control interrupt to have higher priority

than the communication interrupt. But we would also have to ensure that the control ISR executes quickly enough to allow time for communication and other processes.

Every time an interrupt is generated, the CPU must save the contents of the internal CPU registers, called the "context," to the stack (data RAM). It then uses its registers in the execution of the ISR. After the ISR completes, it copies the context from RAM back to its registers and continues where it left off before the interrupt. The copying of register data back and forth is called "context save and restore."

6.2 Details

The address of an ISR in virtual memory is determined by the *interrupt vector* associated with the IRQ. The PIC32MX supports up to 64 unique interrupt vectors (and therefore 64 ISRs). For `timing.c` in Chapter 5.3, the virtual addresses of the interrupt vectors can be seen in this edited exception memory listing from the map file (an interrupt is also known as an "exception"):

```
.vector_0               0x9d000200            0x8           8  Interrupt Vector 0
.vector_1               0x9d000220            0x8           8  Interrupt Vector 1

  [[[ ... snipping long list of vectors ...]]]

.vector_51              0x9d000860            0x8           8  Interrupt Vector 51
```

If an ISR has been written for the core timer (interrupt vector 0), the code at 0x9D000200 simply contains a jump to the location in program memory that actually holds the ISR.

Although the PIC32 can have only 64 interrupt vectors, it has up to 96 events (or IRQs) that generate an interrupt. Therefore some of the IRQs share the same interrupt vector and ISR.

Before interrupts can be used, the CPU has to be enabled to process them in either "single vector mode" or "multi-vector mode." In single vector mode, all interrupts jump to the same ISR. This is the default setting on reset of the PIC32. In multi-vector mode, different interrupt vectors are used for different IRQs. We typically use multi-vector mode, which is set by `NU32_Startup()`.

After interrupts have been enabled, the CPU jumps to an ISR when three conditions are satisfied: (1) the specific IRQ has been enabled by setting a bit to 1 in the SFR IECx (one of three Interrupt Enable Control SFRs, with x equal to 0, 1, or 2); (2) some event causes a 1 to be written to the same bit of the SFR IFSx (Interrupt Flag Status); and (3) the priority of the interrupt vector, as represented in the SFR IPCy (one of 16 Interrupt Priority Control SFRs, y=0...15), is greater than the current priority of the CPU. If the first two conditions are satisfied, but not the third, the interrupt waits until the CPU's priority drops lower.

The "x" in the IECx and IFSx SFRs above can be 0, 1, or 2, corresponding to (3 SFRs) $\times$ (32 bits) = 96 interrupt sources. The "y" in IPCy takes values 0...15, and each of the IPCy registers contains the priority level for four different interrupt vectors, i.e., (16 SFRs) $\times$ (four vectors per register) = 64 interrupt vectors. The priority level for each of the 64 vectors is represented by five bits: three indicating the priority (taking values 0 to 7, or 0b000 to 0b111; a priority of 0 indicates that the interrupt is disabled) and two indicating the subpriority (taking values 0 to 3). Thus each IPCy has 20 relevant bits—five for each of the four interrupt vectors—and 12 unused bits.

The list of interrupt sources (IRQs) and their corresponding bit locations in the IECx and IFSx SFRs, as well as the bit locations in IPCy of their corresponding interrupt vectors, are are given in the table below, reproduced from the Interrupts section of the Data Sheet. Consulting this table, we see that the change notification's (CN) interrupt has x=1 (for the IRQ) and y=6 (for the vector), so information about this interrupt is stored in IFS1, IEC1, and IPC6. Specifically, IEC1$\langle 0 \rangle$ is its interrupt enable bit, IFS1$\langle 0 \rangle$ is its interrupt flag status bit, IPC6$\langle 20{:}18 \rangle$ are the three priority bits for its interrupt vector, and IPC6$\langle 17{:}16 \rangle$ are the two subpriority bits.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source(1)	IRQ Number	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3 – External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4 – Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4 – Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>
OC4 – Output Compare 4	18	18	IFS0<18>	IEC0<18>	IPC4<20:18>	IPC4<17:16>
INT4 – External Interrupt 4	19	19	IFS0<19>	IEC0<19>	IPC4<28:26>	IPC4<25:24>
T5 – Timer5	20	20	IFS0<20>	IEC0<20>	IPC5<4:2>	IPC5<1:0>
IC5 – Input Capture 5	21	21	IFS0<21>	IEC0<21>	IPC5<12:10>	IPC5<9:8>
OC5 – Output Compare 5	22	22	IFS0<22>	IEC0<22>	IPC5<20:18>	IPC5<17:16>
SPI1E – SPI1 Fault	23	23	IFS0<23>	IEC0<23>	IPC5<28:26>	IPC5<25:24>
SPI1RX – SPI1 Receive Done	24	23	IFS0<24>	IEC0<24>	IPC5<28:26>	IPC5<25:24>
SPI1TX – SPI1 Transfer Done	25	23	IFS0<25>	IEC0<25>	IPC5<28:26>	IPC5<25:24>
U1E – UART1 Error SPI3E – SPI3 Fault I2C3B – I2C3 Bus Collision Event	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
U1RX – UART1 Receiver SPI3RX – SPI3 Receive Done I2C3S – I2C3 Slave Event	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>
U1TX – UART1 Transmitter SPI3TX – SPI3 Transfer Done I2C3M – I2C3 Master Event	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
I2C1B – I2C1 Bus Collision Event	29	25	IFS0<29>	IEC0<29>	IPC6<12:10>	IPC6<9:8>
I2C1S – I2C1 Slave Event	30	25	IFS0<30>	IEC0<30>	IPC6<12:10>	IPC6<9:8>
I2C1M – I2C1 Master Event	31	25	IFS0<31>	IEC0<31>	IPC6<12:10>	IPC6<9:8>
CN – Input Change Interrupt	32	26	IFS1<0>	IEC1<0>	IPC6<20:18>	IPC6<17:16>

Note 1: Not all interrupt sources are available on all devices. See **TABLE 1: "PIC32 USB and CAN – Features"**, **TABLE 2: "PIC32 USB and Ethernet – Features"** and **TABLE 3: "PIC32 USB, Ethernet and CAN – Features"** for the list of available peripherals.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source(1)	IRQ Number	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
AD1 – ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>
PMP – Parallel Master Port	34	28	IFS1<2>	IEC1<2>	IPC7<4:2>	IPC7<1:0>
CMP1 – Comparator Interrupt	35	29	IFS1<3>	IEC1<3>	IPC7<12:10>	IPC7<9:8>
CMP2 – Comparator Interrupt	36	30	IFS1<4>	IEC1<4>	IPC7<20:18>	IPC7<17:16>
U3E – UART2A Error SPI2E – SPI2 Fault I2C4B – I2C4 Bus Collision Event	37	31	IFS1<5>	IEC1<5>	IPC7<28:26>	IPC7<25:24>
U3RX – UART2A Receiver SPI2RX – SPI2 Receive Done I2C4S – I2C4 Slave Event	38	31	IFS1<6>	IEC1<6>	IPC7<28:26>	IPC7<25:24>
U3TX – UART2A Transmitter SPI2TX – SPI2 Transfer Done IC4M – I2C4 Master Event	39	31	IFS1<7>	IEC1<7>	IPC7<28:26>	IPC7<25:24>
U2E – UART3A Error SPI4E – SPI4 Fault I2C5B – I2C5 Bus Collision Event	40	32	IFS1<8>	IEC1<8>	IPC8<4:2>	IPC8<1:0>
U2RX – UART3A Receiver SPI4RX – SPI4 Receive Done I2C5S – I2C5 Slave Event	41	32	IFS1<9>	IEC1<9>	IPC8<4:2>	IPC8<1:0>
U2TX – UART3A Transmitter SPI4TX – SPI4 Transfer Done IC5M – I2C5 Master Event	42	32	IFS1<10>	IEC1<10>	IPC8<4:2>	IPC8<1:0>
I2C2B – I2C2 Bus Collision Event	43	33	IFS1<11>	IEC1<11>	IPC8<12:10>	IPC8<9:8>
I2C2S – I2C2 Slave Event	44	33	IFS1<12>	IEC1<12>	IPC8<12:10>	IPC8<9:8>
I2C2M – I2C2 Master Event	45	33	IFS1<13>	IEC1<13>	IPC8<12:10>	IPC8<9:8>
FSCM – Fail-Safe Clock Monitor	46	34	IFS1<14>	IEC1<14>	IPC8<20:18>	IPC8<17:16>
RTCC – Real-Time Clock and Calendar	47	35	IFS1<15>	IEC1<15>	IPC8<28:26>	IPC8<25:24>
DMA0 – DMA Channel 0	48	36	IFS1<16>	IEC1<16>	IPC9<4:2>	IPC9<1:0>
DMA1 – DMA Channel 1	49	37	IFS1<17>	IEC1<17>	IPC9<12:10>	IPC9<9:8>
DMA2 – DMA Channel 2	50	38	IFS1<18>	IEC1<18>	IPC9<20:18>	IPC9<17:16>
DMA3 – DMA Channel 3	51	39	IFS1<19>	IEC1<19>	IPC9<28:26>	IPC9<25:24>
DMA4 – DMA Channel 4	52	40	IFS1<20>	IEC1<20>	IPC10<4:2>	IPC10<1:0>
DMA5 – DMA Channel 5	53	41	IFS1<21>	IEC1<21>	IPC10<12:10>	IPC10<9:8>
DMA6 – DMA Channel 6	54	42	IFS1<22>	IEC1<22>	IPC10<20:18>	IPC10<17:16>
DMA7 – DMA Channel 7	55	43	IFS1<23>	IEC1<23>	IPC10<28:26>	IPC10<25:24>
FCE – Flash Control Event	56	44	IFS1<24>	IEC1<24>	IPC11<4:2>	IPC11<1:0>
USB – USB Interrupt	57	45	IFS1<25>	IEC1<25>	IPC11<12:10>	IPC11<9:8>
CAN1 – Control Area Network 1	58	46	IFS1<26>	IEC1<26>	IPC11<20:18>	IPC11<17:16>
CAN2 – Control Area Network 2	59	47	IFS1<27>	IEC1<27>	IPC11<28:26>	IPC11<25:24>
ETH – Ethernet Interrupt	60	48	IFS1<28>	IEC1<28>	IPC12<4:2>	IPC12<1:0>
IC1E – Input Capture 1 Error	61	5	IFS1<29>	IEC1<29>	IPC1<12:10>	IPC1<9:8>
IC2E – Input Capture 2 Error	62	9	IFS1<30>	IEC1<30>	IPC2<12:10>	IPC2<9:8>

Note 1: Not all interrupt sources are available on all devices. See **TABLE 1: "PIC32 USB and CAN – Features"**, **TABLE 2: "PIC32 USB and Ethernet – Features"** and **TABLE 3: "PIC32 USB, Ethernet and CAN – Features"** for the list of available peripherals.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)

Interrupt Source(1)	IRQ Number	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
IC3E – Input Capture 3 Error	63	13	IFS1<31>	IEC1<31>	IPC3<12:10>	IPC3<9:8>
IC4E – Input Capture 4 Error	64	17	IFS2<0>	IEC2<0>	IPC4<12:10>	IPC4<9:8>
IC4E – Input Capture 5 Error	65	21	IFS2<1>	IEC2<1>	IPC5<12:10>	IPC5<9:8>
PMPE – Parallel Master Port Error	66	28	IFS2<2>	IEC2<2>	IPC7<4:2>	IPC7<1:0>
U4E – UART4 Error	67	49	IFS2<3>	IEC2<3>	IPC12<12:10>	IPC12<9:8>
U4RX – UART4 Receiver	68	49	IFS2<4>	IEC2<4>	IPC12<12:10>	IPC12<9:8>
U4TX – UART4 Transmitter	69	49	IFS2<5>	IEC2<5>	IPC12<12:10>	IPC12<9:8>
U6E – UART6 Error	70	50	IFS2<6>	IEC2<6>	IPC12<20:18>	IPC12<17:16>
U6RX – UART6 Receiver	71	50	IFS2<7>	IEC2<7>	IPC12<20:18>	IPC12<17:16>
U6TX – UART6 Transmitter	72	50	IFS2<8>	IEC2<8>	IPC12<20:18>	IPC12<17:16>
U5E – UART5 Error	73	51	IFS2<9>	IEC2<9>	IPC12<28:26>	IPC12<25:24>
U5RX – UART5 Receiver	74	51	IFS2<10>	IEC2<10>	IPC12<28:26>	IPC12<25:24>
U5TX – UART5 Transmitter	75	51	IFS2<11>	IEC2<11>	IPC12<28:26>	IPC12<25:24>
(Reserved)	—	—	—	—	—	—
Lowest Natural Order Priority						

Note 1: Not all interrupt sources are available on all devices. See **TABLE 1: "PIC32 USB and CAN – Features"**, **TABLE 2: "PIC32 USB and Ethernet – Features"** and **TABLE 3: "PIC32 USB, Ethernet and CAN – Features"** for the list of available peripherals.

As mentioned earlier, some IRQs share the same vector. For example, IRQs 26, 27, and 28, each corresponding to UART1 events, all share vector number 24. Priorities and subpriorities are associated with interrupt vectors, not IRQs.

If the CPU is currently processing an ISR at a particular priority level, and it receives an interrupt request for a vector (and therefore ISR) at the same priority, it will complete its current ISR before servicing the other IRQ, regardless of the subpriority. When the CPU has multiple interrupts pending at a higher priority level than it is currently operating at, the CPU first processes the one with the highest priority level. If there are more than one at the same highest priority level, the CPU first processes the one with the highest subpriority. If interrupts have the same priority and subpriority, then their priority is resolved using the "natural order priority" table given above, where vectors earlier in the table have higher priority.

If the priority of an interrupt vector is zero, then the interrupt is disabled. There are seven enabled priority levels.

Every ISR should clear the interrupt flag (clear the appropriate bit of IFSx to zero), indicating that the interrupt has been serviced. By doing so, after the ISR completes, the CPU is free to return to the program state when the ISR was called.

When setting up an interrupt, you set a bit in IECx to 1 indicating the interrupt is enabled (all bits are set to zero upon reset) and assign values to the associated IPCy priority bits. (These priority bits default to zero upon reset, which will keep the interrupt disabled.) You will generally never write code setting an IFSx bit to 1. Instead, when you set up the device that generates the interrupt (e.g., a UART or counter/timer), you configure it to set the interrupt flag IFSx upon the appropriate event.

The Shadow Register Set The PIC32MX's CPU provides an internal *shadow register set* (SRS), which is a full extra set of registers. You can take advantage of this extra register set to avoid the time needed for context save and restore. When processing an ISR using the SRS, the CPU simply switches to this extra set of internal registers. When it finishes the ISR, it switches back to its original register set, without needing to save and restore them. We see examples of this in Section 6.4. Obviously we cannot allow one ISR using the SRS to be interrupted by another ISR using the SRS! An easy way to avoid this is to have only one ISR written to use the SRS.

The Device Configuration Register DEVCFG3 determines which priority level is assigned to the shadow register set. The preprocessor command

```
#pragma config FSRSSEL = PRIORITY_7
```

implemented in `NU32.h` and the NU32 bootloader sets the shadow register set to priority level 7. This choice makes sense; the highest priority interrupt should spend the least time jumping to and from the ISR.

External Interrupt Inputs The PIC32 has five digital inputs, INT0–INT4, that can be used to generate interrupts on rising or falling edges. The enable and flag status bits are in IFS0 and IEC0, respectively, at bits 3, 7, 11, 15, and 19 for INT0, INT1, INT2, INT3, and INT4, respectively. The priority and subpriority bits are in IPCy⟨28:26⟩ and IPCy⟨25:24⟩ for the input INTy. The SFR INTCON bits 0...4 determine whether the associated interrupt is triggered on a falling edge (bit cleared to 0) or rising edge (bit set to 1).

Special Function Registers

The SFRs associated with interrupts are summarized below. For full details, consult the Reference Manual.

INTCON The interrupt control SFR determines whether the interrupt controller operates in single vector or multi-vector mode. It also determines whether the five external interrupt pins INT0–INT4 generate an interrupt on a rising edge or a falling edge.

INTSTAT The interrupt status SFR is read-only and contains information on the address and priority level of the latest IRQ given to the CPU when in single vector mode. We will not need it.

IPTMR The interrupt proximity timer SFR can be used to implement a delay to queue up interrupt requests before presenting them to the CPU. For example, upon receiving an interrupt request, the timer starts counting clock cycles, queuing up any subsequent interrupt requests, until IPTMR cycles have passed. By default, this timer is turned off by INTCON, and we will typically leave it that way.

IECx, x = 0, 1, or 2 Three 32-bit interrupt enable control SFRs for up to 96 interrupt sources. A 1 enables the interrupt, a 0 disables it.

IFSx, x = 0, 1, or 2 The three 32-bit interrupt flag status SFRs represent the status of up to 96 interrupt sources. A 1 indicates an interrupt has been requested, a 0 indicates no interrupt is requested.

IPCy, y = 0 to 15 Each of the sixteen interrupt priority control SFRs contains 5-bit priority and subpriority values for 4 different interrupt vectors (64 vectors total).

In this chapter only, we reproduce some SFR information from the Data Sheet. You should always consult the appropriate sections from the Reference Manual and the Data Sheet for more information. The Memory Organization section of the Data Sheet indicates which of the SFRs in the Reference Manual are present on your PIC32.

REGISTER 7-1: INTCON: INTERRUPT CONTROL REGISTER

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
23:16	U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
	—	—	—	—	—	—	—	SS0
15:8	U-0	U-0	U-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
	—	—	—	MVEC	—	TPC<2:0>		
7:0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

- INTCON⟨16⟩, or INTCONbits.SS0: 1 = use the shadow register set when in single vector mode, 0 = do not use

- INTCON⟨12⟩, or INTCONbits.MVEC: 1 = interrupt controller in multi-vector mode, 0 = single vector mode

- INTCON⟨10:8⟩, or INTCONbits.TPC: control bits for the IPTMR (we leave it at the default of 000 = IPTMR off)

- INTCON⟨x⟩, for x = 0 to 4, or INTCONbits.INTxEP: 1 = external interrupt pin x triggers on a rising edge, 0 = triggers on a falling edge

REGISTER 7-4: IFSx: INTERRUPT FLAG STATUS REGISTER

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS31	IFS30	IFS29	IFS28	IFS27	IFS26	IFS25	IFS24
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS23	IFS22	IFS21	IFS20	IFS19	IFS18	IFS17	IFS16
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS15	IFS14	IFS13	IFS12	IFS11	IFS10	IFS09	IFS08
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IFS07	IFS06	IFS05	IFS04	IFS03	IFS02	IFS01	IFS00

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

IFSx⟨bit⟩: 1 = interrupt has been requested, 0 = no interrupt has been requested. See the Interrupt Controller section of the Data Sheet, or the table reproduced earlier, for the the register number x in IFSx, and the bit number, for a particular IRQ source. For example, the change notification interrupt request flag status bit is IFS1⟨0⟩.

REGISTER 7-5: IECx: INTERRUPT ENABLE CONTROL REGISTER

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC31	IEC30	IEC29	IEC28	IEC27	IEC26	IEC25	IEC24
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC23	IEC22	IEC21	IEC20	IEC19	IEC18	IEC17	IEC16
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC15	IEC14	IEC13	IEC12	IEC11	IEC10	IEC09	IEC08
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IEC07	IEC06	IEC05	IEC04	IEC03	IEC02	IEC01	IEC00

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

IECx⟨bit⟩: 1 = interrupt has been enabled so that requests are allowed, 0 = interrupt is disabled. See the Interrupt Controller section of the Data Sheet, or the table reproduced earlier, for the the register number x in IECx, and the bit number, for a particular IRQ source. For example, the change notification interrupt enable bit is IEC1⟨0⟩.

REGISTER 7-6: IPCx: INTERRUPT PRIORITY CONTROL REGISTER

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP03<2:0>			IS03<1:0>	
23:16	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP02<2:0>			IS02<1:0>	
15:8	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP01<2:0>			IS01<1:0>	
7:0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	IP00<2:0>			IS00<1:0>	

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Each IPCy, y = 0 to 15, contains five priority and subpriority bits for each of four different interrupt vectors. For example, consulting the table, we see that IPC6⟨20:18⟩ are the three priority bits for the change notification interrupt vector, and IPC6⟨17:16⟩ are its two subpriority bits.

6.3 Steps to Set Up and Use an Interrupt

Among the other things it does, `NU32_Startup()` enables the CPU to receive interrupts, and sets the mode to multi-vector mode by setting INTCONbits.MVEC to 1. After this, there are seven basic steps to set up and use an interrupt. We recommend your program execute steps 2–7 in the order given below. The details of the syntax are left to the examples in Section 6.4.

1. Write an ISR with a priority level 1–7 using the syntax `IPLnSOFT`, for `n`=1...7, or `IPL7SRS`. `SOFT` indicates software context save and restore, and `SRS` makes use of the shadow register set. (The bootloader on the NU32 allows only priority level 7 to use the SRS.) No subpriority is specified in the ISR function. The ISR must clear the appropriate interrupt flag IFSx⟨bit⟩.
2. Disable interrupts at the CPU to prevent spurious generation of interrupts while you are configuring. Although interrupts are disabled by default on reset, `NU32_Startup()` enables them.
3. Configure the device (e.g., peripheral) to generate interrupts on the appropriate event. This often involves configuring the SFRs of the particular peripheral.
4. Configure the interrupt priority and subpriority in IPCy. The IPCy priority should match the priority of the ISR defined in Step 1.
5. Clear the interrupt flag status bit to 0 in IFSx.
6. Set the interrupt enable bit to 1 in IECx.
7. Reenable interrupts at the CPU.

6.4 Sample Code

6.4.1 Core Timer Interrupt

Let's toggle a digital output once per second based on an interrupt from the CPU's core timer. To do this, we place a value in the CPU's CP0_COMPARE register, and whenever the core timer counter value is equal to CP0_COMPARE, an interrupt is generated. In the interrupt routine, the core timer counter is reset to 0. Since the core timer runs at half the frequency of the system clock, setting CP0_COMPARE to 40,000,000 toggles the digital output once per second.

To make the effect visible, let's toggle pin RA5, which corresponds to LED2 on the NU32 board. We'll use priority level 7, subpriority 0, and the shadow register set.

Code Sample 6.1. INT_core_timer.c. A core timer interrupt using the shadow register set.

```
//#define NU32_STANDALONE   // uncomment if program is standalone, not bootloaded
#include "NU32.h"           // peripheral.h, config bits, constants, funcs for startup and UART
#define CORE_TICKS 40000000 // 40 M ticks (one second)

void __ISR(_CORE_TIMER_VECTOR, IPL7SRS) CoreTimerISR(void) {  // step 1: the ISR
  IFS0CLR = 1;                      // clear CT int flag IFS0<0>, same as IFS0bits.CTIF=0
  LATAINV = 0x20;                   // invert pin RA5 only
  _CP0_SET_COUNT(0);                // set core timer counter to 0
  _CP0_SET_COMPARE(CORE_TICKS);     // must set CP0_COMPARE again after interrupt
}

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

  __builtin_disable_interrupts();   // step 2: disable interrupts at CPU
  _CP0_SET_COMPARE(CORE_TICKS);     // step 3: CP0_COMPARE register set to 40 M
  IPC0bits.CTIP = 7;                // step 4: interrupt priority
  IPC0bits.CTIS = 0;                // step 4: subp is 0, which is the default
  IFS0bits.CTIF = 0;                // step 5: clear CT interrupt flag
  IEC0bits.CTIE = 1;                // step 6: enable core timer interrupt
  __builtin_enable_interrupts();    // step 7: CPU interrupts enabled

  _CP0_SET_COUNT(0);                // set core timer counter to 0
  while(1) {
      ; // do nothing
  }
}
```

Following our seven steps to use an interrupt, we have:

Step 1. The ISR.

```
void __ISR(_CORE_TIMER_VECTOR, IPL7SRS) CoreTimerISR(void) {  // step 1: the ISR
  IFS0CLR = 1;                      // clear CT int flag IFS0<0>, same as IFS0bits.CTIF=0
  ...
}
```

We are allowed to call our ISR whatever we want, and in this example we call it `CoreTimerISR`. The `__ISR` syntax is XC32-specific (not a C standard) and tells the compiler and linker that this function should be treated as an interrupt handler. The two arguments to this syntax are the interrupt vector for the core timer, called `_CORE_TIMER_VECTOR` (defined as 0 in `p32mx795f512l.h`, which agrees with the table in this chapter), and the interrupt priority level. The interrupt priority level is specified using the syntax `IPLnSRS` or `IPLnSOFT`, where `n` is 1 to 7, `SRS` indicates that the shadow register set should be used, and `SOFT` indicates that software context save and restore should be used. Use `IPL7SRS` if you'd like to use the shadow register set, as in this example, since the device configuration registers on the NU32's PIC32 specify priority level 7 for the shadow register set. You don't specify subpriority in the ISR.

The ISR must clear the interrupt flag in IFS0⟨0⟩, which the table tells us corresponds to the core timer interrupt. Specifically for the core timer, a write to CP0_COMPARE is also needed to clear the interrupt.

Step 2. Disabling interrupts. Since `NU32_Startup()` enables interrupts, we disable them before configuring the core timer interrupt.

```
  __builtin_disable_interrupts();   // step 2: disable interrupts at CPU
```

Disabling interrupts before configuring the device that generates interrupts is good general practice, to avoid unwanted interrupts during configuration. In many cases it is not strictly necessary, however.

Step 3. Configuring the core timer to interrupt.

```
_CP0_SET_COMPARE(CORE_TICKS);     // step 3: CP0_COMPARE register set to 40 M
```

This line sets the core timer's CP0_COMPARE value so that an interrupt is generated when the core timer counter reaches `CORE_TICKS`. If the interrupt were to be generated by a peripheral, we should consult the appropriate section of this book, or the Reference Manual, to set the SFRs to generate an IRQ on the appropriate event.

Step 4. Configuring interrupt priority.

```
IPC0bits.CTIP = 7;                // step 4: interrupt priority
IPC0bits.CTIS = 0;                // step 4: subp is 0, which is the default
```

These two commands set the appropriate bits in IPCy (y=0, according to the table). Consulting the file `p32mx795f512l.h` or the Memory Organization section of the Data Sheet shows us that the priority and subpriority bits of IPC0 are called IPC0bits.CTIP and IPC0bits.CTIS, respectively. Alternatively, we could have used any other means to manipulate the bits IPC0⟨4:2⟩ and IPC0⟨1:0⟩, as indicated in the table, while leaving all other bits unchanged. The priority must agree with the ISR priority. It is not strictly necessary to set the subpriority, which defaults to zero in any case.

Step 5. Clearing the interrupt flag status bit.

```
IFS0bits.CTIF = 0;                // step 5: clear CT interrupt flag
```

This command clears the appropriate bit in IFSx (x=0 here). An alternative would be `IFS0CLR = 1;`, to clear the zeroth bit of IFS0, as used in the ISR.

Step 6. Enabling the core timer interrupt.

```
IEC0bits.CTIE = 1;                // step 6: enable core timer interrupt
```

This command sets the appropriate bit in IECx (x=0 here). An alternative would be `IEC0SET = 1;` to set the zeroth bit of IEC0.

Step 7. Reenable interrupts at the CPU.

```
__builtin_enable_interrupts();    // step 7: CPU interrupts enabled
```

6.4.2 External Interrupt

Code Sample 6.2 causes the NU32's LEDs to turn on briefly on a falling edge on the external interrupt input pin INT0. The IRQ associated with INT0 is 3, and the flag, enable, priority, and subpriority bits can be found in the table. In this example we use interrupt priority level 2, with software context save and restore.

You can test this program with the NU32 by connecting a wire from the D13/USER pin to the D0/INT0 pin. When the USER button is pressed, there is a falling edge on digital input D13 (see the wiring diagram in Figure 2.4) and therefore INT0, which causes the LEDs to flash. You might also notice the issue of switch *bounce*: when you release the button, nominally creating a rising edge, you might see the LEDs flash again. This is because there may be a brief period of chattering when mechanical contact between two conductors is established or broken, creating spurious rising and falling edges. This can be addressed by a *debouncing* circuit or software; see Problem 17.

Code Sample 6.2. `INT_ext_int.c`. Using an external interrupt to flash LEDs on the NU32.

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

void __ISR(_EXTERNAL_0_VECTOR, IPL2SOFT) Ext0ISR(void) { // step 1: the ISR
  LATACLR = 0x30;                 // clear RA4 and RA5 low (LED1 and LED2 on)
  _CP0_SET_COUNT(0);
  while(_CP0_GET_COUNT() < 10000000) {
      ;                           // delay for 10 M core ticks, 0.25 s
  }
  LATASET = 0x30;                 // set pins RA4 and RA5 high (LED1 and LED2 off)
  IFS0CLR = 1 << 3;               // clear interrupt flag IFS0<3>
}

void main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts(); // step 2: disable interrupts
  INTCONCLR = 0x1;                // step 3: INT0 triggers on falling edge
  IPC0CLR = 0x1F << 24;           // step 4: clear the 5 pri and subpri bits
  IPC0 |= 9 << 24;                // step 4: set priority to 2, subpriority to 1
  IFS0bits.INT0IF = 0;            // step 5: clear the int flag, or IFS0CLR=1<<3
  IEC0SET = 1 << 3;               // step 6: enable INT0 by setting IEC0<3>
  __builtin_enable_interrupts();  // step 7: enable interrupts
  while(1) {
      ; // do nothing, loop forever
  }
}
```

6.4.3 Speedup Due to the Shadow Register Set

This last example measures the amount of time it takes to enter and exit an ISR using context save and restore vs. the SRS. We write two identical ISRs; the only difference is that one uses `IPL7SOFT` and the other uses `IPL7SRS`. The two ISRs are based on the external interrupts INT0 and INT1, respectively. To get precise timing, however, we trigger interrupts in software by setting the appropriate bit of the appropriate SFR.

After setting up the interrupts, the program `INT_timing.c` enters an infinite loop. First the core timer is reset to zero, then the interrupt flag is set for INT0. The `main` function then waits until the ISR clears the flag. The first thing the ISR for INT0 does is log the core timer counter; then it toggles LED2 and clears the interrupt flag; and the last thing it does before exiting is log the time again. Finally, the `main` function logs the time when control is returned. The timing results are written back to the host computer over a serial link using the NU32 library. Then the `main` function does the same for INT1 and goes back to the beginning of the infinite loop.

These are the results (which are repeated over and over):

```
IPL7SOFT in  16 out  22 total  32 time  800 ns
 IPL7SRS in  14 out  20 total  29 time  725 ns
```

For context save and restore, it takes 16 core clock ticks (about 32 SYSCLK ticks) to begin executing statements in the ISR; the last ISR statement completes about 6 (12) ticks later; and finally control is returned to `main` approximately 32 (64) total ticks, or 800 nanoseconds, after the interrupt flag is set. For the SRS, the first ISR statement is executed after about 14 (28) ticks; the ISR runs in the same amount of time (6 core timer ticks, or 12 SYSCLK ticks); and a total of approximately 29 (58) ticks, or 725 nanoseconds, elapse between the time the interrupt flag is set and control is returned to `main`.

While the numbers may be different for other ISRs and `main` functions, depending on the amount of register context to be saved and restored, a few things can be noted:

- The ISR is not entered immediately after the flag is set. It takes time to respond to the interrupt request, and instructions in `main` may be executed after the flag is set.

- The SRS saves time in entering and exiting the ISR, approximately 75 nanoseconds total in this case.
- Simple ISRs can be completed in less than a microsecond after the interrupt event occurs.

The sample code is below.

Code Sample 6.3. INT_timing.c. Timing the shadow register set vs. typical context save and restore.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded""
#include "NU32.h"          // peripheral.h, config bits, constants, funcs for startup and UART
#define DELAYTIME 40000000 // 40 million core clock ticks, or 1 second

void delay();

volatile unsigned int Entered = 0, Exited = 0;  // note the qualifier "volatile"

void __ISR(_EXTERNAL_0_VECTOR, IPL7SOFT) Ext0ISR(void) {
  Entered = _CP0_GET_COUNT();       // record time ISR begins
  IFS0CLR = 1 << 3;                 // clear the interrupt flag
  NU32_LED2 = 1;                    // turn off LED2
  Exited = _CP0_GET_COUNT();        // record time ISR ends
}

void __ISR(_EXTERNAL_1_VECTOR, IPL7SRS) Ext1ISR(void) {
  Entered = _CP0_GET_COUNT();       // record time ISR begins
  IFS0CLR = 1 << 7;                 // clear the interrupt flag
  NU32_LED2 = 0;                    // turn on LED2
  Exited = _CP0_GET_COUNT();        // record time ISR ends
}

void main(void) {
  unsigned int dt = 0;
  unsigned int encopy, excopy;      // local copies of globals Entered, Exited
  char msg[128] = {};

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts();   // step 2: disable interrupts at CPU
  INTCONSET = 0x3;                  // step 3: INT0 and INT1 trigger on rising edge
  IPC0CLR = 31 << 24;               // step 4: clear 5 priority and subp bits for INT0
  IPC0 |= 28 << 24;                 // step 4: set INT0 to priority 7 subpriority 0
  IPC1CLR = 0x1F << 24;             // step 4: clear 5 priority and subp bits for INT1
  IPC1 |= 0x1C << 24;               // step 4: set INT1 to priority 7 subpriority 0
  IFS0bits.INT0IF = 0;              // step 5: clear INT0 flag status
  IFS0bits.INT1IF = 0;              // step 5: clear INT1 flag status
  IEC0SET = 0x88;                   // step 6: enable INT0 and INT1 interrupts
  __builtin_enable_interrupts();    // step 7: enable interrupts
  while(1) {
    delay();                        // delay, so results sent back at reasonable rate
    _CP0_SET_COUNT(0);              // start timing
    IFS0bits.INT0IF = 1;            // artificially set the INT0 interrupt flag
    while(IFS0bits.INT0IF) {
        ;                           // wait until the ISR clears the flag
    }
    dt = _CP0_GET_COUNT();          // get elapsed time
    __builtin_disable_interrupts(); // good practice before using vars shared w/ISR
    encopy = Entered;               // copy the shared variables to local copies ...
    excopy = Exited;                // ... so the time interrupts are off is short
    __builtin_enable_interrupts();  // turn interrupts back on quickly!
    sprintf(msg,"IPL7SOFT in %3d out %3d total %3d time %4d ns\r\n",encopy,excopy,dt,dt*25);
```

```
    NU32_WriteUART1(msg);           // send times to the host

    delay();                        // same as above, except for INT1
    _CP0_SET_COUNT(0);
    IFS0bits.INT1IF = 1;            // trigger INT1 interrupt
    while(IFS0bits.INT1IF) {
        ;                           // wait until the ISR clears the flag
    }
    dt = _CP0_GET_COUNT();
    __builtin_disable_interrupts();
    encopy = Entered;
    excopy = Exited;
    __builtin_enable_interrupts();
    sprintf(msg," IPL7SRS in %3d out %3d total %3d time %4d ns\r\n",encopy,excopy,dt,dt*25);
    NU32_WriteUART1(msg);
  }
}

void delay() {
  _CP0_SET_COUNT(0);
  while(_CP0_GET_COUNT() < DELAYTIME) {
      ;
  }
}
```

6.4.4 Sharing Variables with ISRs

Code Sample 6.3 was our first to share variables between an ISR and other functions. Namely, `Entered` and `Exited` are used in both ISRs as well as `main`. This code demonstrates two good practices when sharing variables with ISRs:

(1) Using the type qualifier `volatile`. By putting the qualifier `volatile` in front of the type in the global variable definition

```
volatile unsigned int Entered = 0, Exited = 0;
```

we are telling the compiler that external processes (namely, an ISR that may be triggered at an unknown time) may need the values of the variables, or may change the values of the variables, at any time. Therefore, any optimizations the compiler is performing should not take shortcuts in generating assembly code associated with a `volatile` variable. For example, if you had code of the form

```
int i = 0;    // global variable shared by functions and an ISR

void myFunc(void) {
  i = 1;
  // some other code that doesn't use or affect i
  i = 2;
}
```

a compiler running optimizations might not generate any code for `i = 1;` at all, believing that the value 1 for `i` is never used. If an external interrupt triggered during execution of the code between `i = 1;` and `i = 2;`, however, and the ISR made use of the value of `i`, it would use the wrong value (perhaps the originally initialized value `i = 0`).

To correct this, the declaration of the global variable `i` should be

```
volatile int i = 0;
```

The `volatile` qualifier assures that the compiler will emit full assembly code for any reads or writes of `i` in RAM. The compiler does not assume anything about the value of `i` or whether it is changed or used by processes that it does not know about. This is why all SFRs are declared as `volatile` in `p32mx795f512l.h`; their values can be changed by processes external to the CPU.

(2) Enabling and disabling interrupts. Consider a scenario where the main-line code and an ISR share a 64-bit `long double` variable. To load the variable into two of the CPU's 32-bit registers, one assembly instruction first loads the most significant 32 bits into one register. Then the process is interrupted, the ISR modifies the variable in RAM, and control returns to the main code. At that point, the next assembly instruction loads the lower 32 bits of the new value of the `long double` in RAM into the other CPU register. Now the CPU registers have neither the variable value from before nor after the ISR.

To prevent data corruption like this, interrupts can be disabled before reading or writing the shared variables, then reenabled afterward. If an IRQ is generated during the time that the CPU is ignoring interrupts, the IRQ will simply wait until the CPU is accepting interrupts again.

Interrupts should not be disabled for long, as this defeats the purpose of interrupts. In the sample code, the time that interrupts are disabled is minimized by simply copying the shared variables to local copies during this period. This avoids having the interrupts disabled during the `sprintf` command, which can take many CPU cycles.

In many cases it is not necessary to disable interrupts before using shared variables. (For example, it was not necessary in the sample code above.) It is good practice if you are unsure, however. Just minimize the time that interrupts are disabled.

6.5 Chapter Summary

- The PIC32MX supports 96 interrupt requests (IRQs) and 64 interrupt vectors, and therefore up to 64 interrupt service routines (ISRs). Therefore, some IRQs share the same ISR. For example, all IRQs related to UART1 (data received, data transmitted, error) have the same interrupt vector.

- The PIC32 can be configured to operate in single vector mode (all IRQs result in a jump to the same ISR) or in multi-vector mode. `NU32_Startup()` puts the NU32 in multi-vector mode.

- Priorities and subpriorities are associated with interrupt vectors, and therefore ISRs, not IRQs. The priority of a vector is defined in an SFR IPCy, y=0...15. In the definition of the associated ISR, the same priority `n` should be specified using `IPLnSOFT` or `IPLnSRS`. `SOFT` indicates that software context save and restore is performed, while `SRS` means that the shadow register set is used instead, reducing ISR entry and exit time. The PIC32 on the NU32 is configured by its device configuration registers to make the SRS available only on priority level 7, so the SRS can only be used with `IPL7SRS`.

- When an interrupt is generated, it is serviced immediately if its priority is higher than the current priority. Otherwise it waits until the current ISR is finished.

- In addition to configuring the CPU to accept interrupts, enabling specific interrupts, and setting their priority, the specific peripherals (such as counter/timers, UARTs, change notification pins, etc.) must be configured to generate interrupt requests on the appropriate events. These configurations are left for the chapters covering those peripherals.

- The seven steps to use an interrupt, after putting the CPU in multi-vector mode, are: (1) write the ISR; (2) disable interrupts; (3) configure a device or peripheral to generate interrupts; (4) set the ISR priority and subpriority; (5) clear the interrupt flag; (6) enable the IRQ; and (7) enable interrupts at the CPU.

- If a variable is shared with an ISR, it is a good idea to (1) define that variable with the type qualifier `volatile` and (2) turn off interrupts before reading or writing it if there is a danger the process could be interrupted. If interrupts are disabled, they should be disabled for as short a period as possible.

6.6 Exercises

1. Interrupts can be used to implement a fixed frequency control loop (e.g., 1 kHz). Another way is *polling*: keep checking the core timer, and when some number of ticks has passed, jump to the control routine. Polling can also be used to check for changes on input pins and other events. Give pros and cons (if any) of using interrupts vs. polling.

2. You are watching TV. Give an analogy to an IRQ and ISR for your mental attention in this situation. Also give an analogy to polling.

3. What is the relationship between an interrupt vector and an ISR? What is the maximum number of ISRs that the PIC32 can handle?

4. (a) What happens if an IRQ is generated for an ISR at priority level 4, subpriority level 2 while the CPU is in normal execution (not executing an ISR)? (b) What happens if that IRQ is generated while the CPU is executing a priority level 2, subpriority level 3 ISR? (c) What happens if that IRQ is generated while the CPU is executing a priority level 4, subpriority level 0 ISR? (d) What happens if that IRQ is generated while the CPU is executing a priority level 6, subpriority level 0 ISR?

5. An interrupt asks the CPU to stop what it's doing, attend to something else, and then return to what it was doing. When the CPU is asked to stop what it's doing, it needs to remember the "context" of what it was working on, i.e., the values currently stored in the CPU registers. (a) Assuming no shadow register set, what is the first thing the CPU must do before executing the ISR and the last thing it must do upon completing the ISR? (b) How are things different if a shadow register set is used?

6. What is the peripheral and interrupt vector number associated with IRQ 35? What are the SFRs and bit numbers controlling its interrupt enable, interrupt flag status, and priority and subpriority? Does IRQ 35 share the interrupt vector with any other IRQ?

7. What peripherals and IRQs are associated with interrupt vector 24? What are the SFRs and bit numbers controlling the priority and subpriority of the vector and the interrupt enable and flag status of the associated IRQs?

8. For the problems below, use only the SFRs IECx, IFSx, IPCy, and INTCON, and their CLR, SET, and INV registers (do not use other registers, nor the bit fields as in IFS0bits.INT0IF). Give valid C bit manipulation commands to perform the operations without changing any uninvolved bits. Also indicate, in English, what you are trying to do, in case you have the right idea but wrong C statements. Do not use any constants defined in Microchip XC32 files; just use numbers.

   (a) Enable the Timer2 interrupt, set its flag status to 0, and set its vector's priority and subpriority to 5 and 2, respectively.

   (b) Enable the Real-Time Clock and Calendar interrupt, set its flag status to 0, and set its vector's priority and subpriority to 6 and 1, respectively.

   (c) Enable the UART4 receiver interrupt, set its flag status to 0, and set its vector's priority and subpriority to 7 and 3, respectively.

   (d) Enable the INT2 external input interrupt, set its flag status to 0, set its vector's priority and subpriority to 3 and 2, and configure it to trigger on a rising edge.

9. Edit steps 3, 4, and 6 of Code Sample 6.2 so that each line correctly uses the "bits" forms of the SFRs. In other words, the left-hand sides of the statements should use a form similar to that used in step 5, except using INTCONbits, IPC0bits, and IEC0bits.

10. Consulting the `p32mx795f512l.h` file, give the names of the constants, and the numerical values, associated with the following IRQs: (a) Input Capture 5. (b) SPI3 receive done. (c) USB interrupt.

11. Consulting the `p32mx795f512l.h` file, give the names of the constants, and the numerical values, associated wih the following interrupt vectors: (a) Input Capture 5. (b) SPI3 receive done. (c) USB interrupt.

12. True or false? When the PIC32 is in single vector interrupt mode, only one IRQ can trigger an ISR. Explain your answer.

13. Give the numerical value of the SFR INTCON, in hexadecimal, when it is configured for single vector mode using the shadow register set; and external interrupt input INT3 triggers on a rising edge while the rest of the external inputs trigger on a falling edge. The Interrupt Proximity Timer bits are left as the default.

14. So far we have only seen interrupts generated by the core timer and the external interrupt inputs, because we first have to learn something about the other peripherals to complete Step 3 of the seven-step interrupt setup procedure. Let's jump ahead and see how the Change Notification peripheral could be configured in Step 3. Consulting the Reference Manual chapter on I/O Ports, name the SFR and bit number that has to be manipulated to enable Change Notification pins to generate interrupts.

15. Consult Code Sample 6.3. It uses a few different kinds of syntax to perform bit manipulation on SFRs. For each line of code labeled step 3 to step 6, explain in one sentence what that line of code does and what the purpose is.

16. Build `INT_timing.c` and open its disassembly file `out.dis` with a text editor. Starting at the top of the file, you see the startup code inserted by `crt0.o`. Continuing down, you see the "bootstrap exception" section `.bev_excpt`, which handles any exceptions that might occur while executing boot code; the "general exception" section `.app_excpt`, which the CPU could be set to jump to on an interrupt instead of using the interrupt table; and finally the interrupt vector sections, labeled `.vector_x`, where `x` can take values from 0 to 51 (12 of the possible 64 vectors are not used by the PIC32MX). Each of these exception vectors simply jumps to another address. (Note that `j`, `jal`, and `jr` are all jump statements in assembly. Jumps are not executed immediately; the next assembly statement, in the *jump delay slot*, executes before the jump completes. The jump `j` jumps to the address specified. `jal` jumps to the address specified, usually corresponding to a function, and stores in a CPU register `ra` a return address two instructions [eight bytes] later. `jr` jumps to an address stored in a register, often `ra` to return from a function.)

    (a) What addresses do the `.vector_x` sections jump to? What is installed at these addresses?

    (b) Find the `Ext0ISR` and `Ext1ISR` functions. How many assembly commands are before the first `_CP0_GET_COUNT()` command in each function? How many assembly commands are after the last `_CP0_GET_COUNT()` command in each function? What is the purpose of the commands that account for the majority of the difference in the number of commands? (Note that `sw`, short for "store word," copies a 32-bit CPU register to RAM, and `lw`, short for "load word," copies a 32-bit word from RAM to a CPU register.) Explain why the two functions are different even though their C code is essentially identical.

17. Modify Code Sample 6.2 so the USER button is debounced. How can you change the ISR so the LEDs do not flash if the falling edge comes at the beginning of a very brief, spurious down pulse? Verify that your solution works.

18. Using your solution to debouncing the USER button (Problem 17), write a stopwatch program using an ISR based on INT2. Connect a wire from the USER button pin to the INT2 pin so you can use the USER button as your timing button. Using the NU32 library, your program should send the following message to the user's screen: `Press the USER button to start the timer.` When the USER button has been pressed, it should send the following message: `Press the USER button again to stop the timer.` When the user presses the button again, it should send a message such as `12.505 seconds elapsed.` The ISR should either (1) start the core timer at 0 counts or (2) read the current timer count, depending on whether the program is in the "waiting to begin timing" state or the "timing state." Use priority level 7 and the shadow register set. Verify that the timing is accurate. The stopwatch only has to be accurate for periods of less than the core timer's rollover time.

    You could also try using polling in your `main` function to write out the current elapsed time (when the program is in the "timing state") to the user's screen every second so the user can see the running time.

19. Modify the previous program to use a 16×2 LCD screen instead of the host computer's display.

20. Write a program that interrupts at a frequency defined interactively by the user. The `main` function is an infinite loop that uses the NU32 library to ask the user to specify the integer variable `InterruptPeriod`. If the user enters a number greater than an appropriate minimum and less than an appropriate maximum, this becomes the number of core clock ticks between core timer interrupts. The ISR simply toggles the LEDs, so the `InterruptPeriod` is visible. Set the vector priority to 3 and subpriority to 0.

21. (a) Write a program that has two ISRs, one for the core timer and one for the debounced input INT2. The core timer interrupts every four seconds, and the ISR simply turns on LED1 for two seconds, turns it off, and exits. The INT2 interrupt turns LED2 on and keeps it on until the user lets go of the button. Choose interrupt priority level 1 for the core timer and 5 for INT2. Run the program, experiment with button presses, and see if it agrees with what you expect. (b) Modify the program so the two priority levels are switched. Run the program, experiment with button presses, and see if it agrees with what you expect.

Appendix A

A Crash Course in C

This appendix gives a brief introduction to C for beginners who have some programming experience in a high-level language such as MATLAB. It is not intended as a complete reference; there are lots of great C resources and references out there for a more complete picture. This appendix is also not specific to the Microchip PIC. In fact, I recommend that you start by programming your laptop or desktop so you can experiment with C without needing extra equipment like a PIC32 board.

A.1 Quick Start in C

To get started with C, you need three things: a desktop or laptop, a text editor, and a C compiler. You use the text editor to create your C program, which is a plain text file with a name ending with the postfix `.c`, such as `myprog.c`. Then you use the C compiler to convert this program into machine code that your computer can execute. There are many free C compilers available. I recommend the `gcc` C compiler, which is part of the free GNU Compiler Collection (GCC, found at `http://gcc.gnu.org`). GCC is available for Windows, Mac OS, and Linux. For Windows, you can download the GCC collection in MinGW.[1] (If the installation asks you about what tools to install, make sure to include the `make` tools.) For Mac OS, you can download the full Xcode environment from the Apple Developers site. This installation is multiple gigabytes, however; you can instead opt to install only the "Command Line Tools for Xcode," which is much smaller and more than sufficient for getting started with C (and for this appendix).

Many C installations come with an Integrated Development Environment (IDE) complete with text editor, menus, graphical tools, and other things to assist you with your programming projects. Each IDE is different, however, and the things we cover in this appendix do not require a sophisticated IDE. Therefore we will use only *command line tools*, meaning that we initiate the compilation of the program, and run the program, by typing at the command line. In Mac OS, the command line can be accessed from the Terminal program. In Windows, you can access the command line (also known as MS-DOS or Disk Operating System) by searching for `cmd` or `command prompt`.

To work from the command line, it is useful to learn a few command line instructions. The Mac operating system is built on top of Unix, which is almost identical to Linux, so Mac/Unix/Linux use the same syntax. Windows is similar but slightly different. See the table of a few useful commands below. You can find more information online on how to use these commands as well as others by searching for command line commands in Unix, Linux, or DOS (disk operating system, for Windows).

[1] You are also welcome to use Visual C from Microsoft. The command line compile command will look a bit different than what you see in this appendix.

function	Mac/Unix/Linux	Windows
show current directory	`pwd`	`cd`
list directory contents	`ls`	`dir`
make subdirectory `newdir`	`mkdir newdir`	`mkdir newdir`
change to subdirectory `newdir`	`cd newdir`	`cd newdir`
move "up" to parent directory	`cd ..`	`cd ..`
copy `file` to `filenew`	`cp file filenew`	`copy file filenew`
delete file `file`	`rm file`	`del file`
delete directory `dir`	`rmdir dir`	`rmdir dir`
help on using command `cmd`	`man cmd`	`cmd ?`

Following the long-established programming tradition, your first C program will simply print "Hello world!" to the screen. Use your text editor to create the file `HelloWorld.c`:

```
#include <stdio.h>
int main(void) {
  printf("Hello world!\n");
  return(0);
}
```

Your text editor could be Notepad in Windows or TextEdit on the Mac. You could even use Microsoft Word if you insisted. I personally prefer emacs, but it's not easy to get started with! Text editors packaged with IDEs help enforce a consistent look to your programs. Whichever editor you use, you should save your file as plain text, not rich text or any other formatted text.

To compile your program, navigate from the command line to the directory where the program sits. Then, assuming your command prompt appears as `>`, type the following at the prompt:

```
> gcc HelloWorld.c -o HelloWorld
```

This should create the executable file `HelloWorld` in the same directory. (The argument after the `-o` output flag is the name of the executable file to be created from `HelloWorld.c`.) Now to execute the program, type

```
> HelloWorld
```

The response should be

```
Hello world!
>
```

If the response is instead `command not found` or similar, your computer didn't know where to look for the executable `HelloWorld`. On Mac/Unix/Linux, you can type

```
> ./HelloWorld
```

where the "`.`" is shorthand for "current directory," telling your computer to look in the current directory for `HelloWorld`.

If you've succeeded in getting this far, you have a working C installation and you are ready for the rest of this appendix. If not, time to get help from friends or the web.

A.2 Overview

If you are familiar with a high-level language like MATLAB, you have some idea of loops, functions, program modularity, etc. You'll see that C syntax is different, but that's not a big deal. Let's start instead by focusing on important concepts you must master in C which you don't have to worry about in MATLAB:

- **Memory, addresses, and pointers.** A variable is stored at a particular *address* in memory as 0's and 1's. In C, unlike MATLAB, it is often useful to have access to the memory address where a variable is located. We will learn how to generate a *pointer* to a variable, which contains the address of the variable, and how to access the contents of an address, i.e., the *pointee*.

- **Data types.** In MATLAB, you can simply type `a = 1; b = [1.2 3.1416]; c = [1 2; 3 4]; s = 'a string'`. MATLAB figures out that `a` is a scalar, `b` is a vector with two elements, `c` is a 2×2 matrix, and `s` is a string, and automatically keeps track of the type of the variable (e.g., a list of numbers for a vector or a list of characters for a string) and sets aside, or *allocates*, enough memory to store them. In C, on the other hand, you have to first *define* the variable before you ever use it. For a vector, for example, you have to say what *data type* the elements of the vector will be—integers or numbers with a decimal point (floating point)—and how long the vector will be. This allows the C compiler to allocate enough memory to hold the vector, and to know that the binary representations (0's and 1's) at those locations in memory should be interpreted as integers or floating point numbers.

- **Compiling.** MATLAB programs are typically run as *interpreted* programs—the commands are interpreted, converted to machine-level code, and executed while the program is running. C programs, on the other hand, are *compiled* in advance. This process consists of several steps, but the point is to turn your C program into machine-executable code before the program is ever run. The compiler can identify some errors and warn you about other questionable code. Compiled code typically runs faster than interpreted code, since the translation to machine code is done in advance.

Each of these concepts is described in Section A.3 without going into detail on C syntax. In Section A.4 we will look at sample programs to introduce the syntax, then follow up with a more detailed explanation of the syntax.

A.3 Important Concepts in C

We begin our discussion of C with this caveat:

Important! C consists of an evolving set of standards for a programming language, and any specific C installation is an "implementation" of C. While C standards require certain behavior from all implementations, a number of details are left as implementation-dependent. For example, the number of bytes used for some data types is not fully standard. C wonks like to point out when certain behavior is required and when it is implementation-dependent. While it is good to know that differences may exist from one implementation to another, in this appendix I will often blur the line between what is required and what is common. I prefer to keep this introduction succinct instead of overly precise.

A.3.1 Data Types

Binary and hexadecimal. On a computer, programs and data are represented by sequences of 0's and 1's. A 0 or 1 may be represented by two different voltage levels (low and high) held by a capacitor and controlled by a transistor, for example. Each of these units of memory is called a **bit**.

A sequence of 0's and 1's may be interpreted as a base-2 or **binary** number, just as a sequence of digits in the range 0 to 9 is commonly treated as a base-10 or **decimal** number. In the decimal numbering system, a multi-digit number like 793 is interpreted as $7 * 10^2 + 9 * 10^1 + 3 * 10^0$; the rightmost column is the 10^0 (or 1's) column, the next column to the left is the 10^1 (or 10's) column, the next column to the left is the 10^2 (or 100's) column, and so on. Similarly, the rightmost column of a binary number is the 2^0 column, the next column to the left is the 2^1 column, etc. Converting the binary number `00111011` to its decimal representation, we get

$$0 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 32 + 16 + 8 + 2 + 1 = 59.$$

We can clarify that a sequence of digits is base-2 by writing it as $\texttt{00111011}_2$ or `0b00111011`, where the `b` stands for "binary."

To convert a base-10 number x to binary:

1. Initialize the binary result to all zeros and k to a large integer, such that 2^k is known to be larger than x.

2. If $2^k \leq x$, place a 1 in the 2^k column of the binary number and set x to $x - 2^k$.

3. If $x = 0$ or $k = 0$, we're finished. Else set k to $k - 1$ and go to line 2.

The leftmost digit in a multi-digit number is called the **most significant digit**, and the rightmost digit, corresponding to the 1's column, is called the **least significant digit**. For binary representations, these are often called the **most significant bit (msb)** and **least significant bit (lsb)**, respectively.

Compared to base-10, base-2 has a more obvious connection to the actual hardware representation. Binary can be inconvenient for human reading and writing, however, due to the large number of digits. Therefore we often use base-16, or **hexadecimal** (hex), representations. A single hex digit represents four binary digits using the numbers 0..9 and the letters A..F:

base-2	base-16	base-10	base-2	base-16	base-10
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Thus we can write the eight-digit binary number `00111011`, or `0011 1011`, more succinctly in hex as `3B`, or $\texttt{3B}_{16}$ or `0x3B` to clarify that it is a hex number. The corresponding decimal number is $3 * 16^1 + 11 * 16^0 = 59$.

Bits, bytes, and data types. Bits of memory are grouped together in groups of eight called **bytes**. A byte can be written equivalently in binary or hex (e.g., `00111011` or `3B`), and can represent values between 0 and $2^8 - 1 = 255$ in base-10. Sometimes the four bits represented by a single hex digit are referred to as a **nibble**. (Get it?)

A **word** is a grouping of multiple bytes. The number of bytes depends on the processor, but four-byte words are common, as with the PIC32. A word `01001101 11111010 10000011 11000111` in binary can be written in hex as `4DFA83C7`. The msb is the leftmost bit of the leftmost byte, a `0` in this case.

A byte is the smallest unit of memory that has its own **address**. The address of the byte is a number that represents where the byte is in memory. Suppose your computer has 4 gigabytes (GB)[2], or $4 \times 2^{30} = 2^{32}$ bytes, of RAM. Then to find the value stored in a particular byte, you need at least 32 binary digits (8 hex digits or 4 bytes) to specify the address.

An example showing the first eight addresses in memory is shown below.

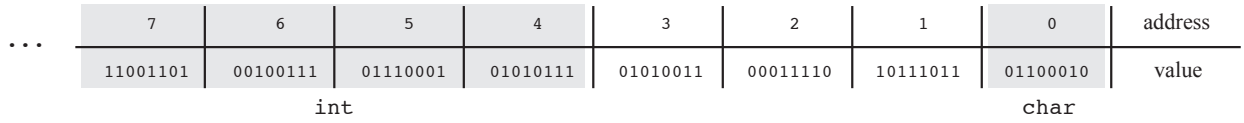
...	7	6	5	4	3	2	1	0	address
	11001101	00100111	01110001	01010111	01010011	00011110	10111011	01100010	value

Now assume that the byte at address 4 is part of the representation of a variable. Do these 0's and 1's represent an integer, or part of an integer? A number with a fractional component? Something else?

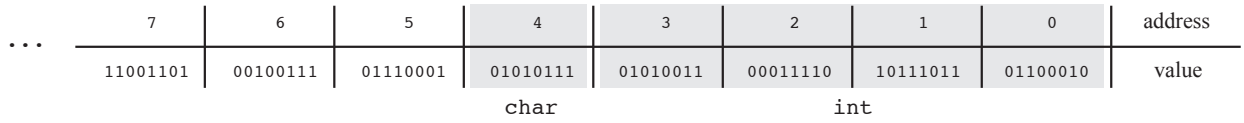
The answer lies in the **data type** of the variable at that address. In C, before you use a variable, you have to *define* it and its type. This tells the compiler how many bytes to set aside for the variable and how to write or interpret 0's and 1's at the address(es) used by that variable. The most common data types come in two flavors: integers and floating point numbers (numbers with a decimal point). Of the integers, the two most common types are `char`[3], often used to represent keyboard characters, and `int`. Of the floating point numbers, the two most common types are `float` and `double`. As we will see shortly, a `char` uses 1 byte and an `int` usually uses 4, so two possible interpretations of the data held in the eight memory addresses could be

[2] In common usage, a kilobyte (KB) is $2^{10} = 1024$ bytes, a megabyte (MB) is $2^{20} = 1,048,576$ bytes, a gigabyte is $2^{30} = 1,073,741,824$ bytes, and a terabyte (TB) is $2^{40} = 1,099,511,627,776$ bytes. To remove confusion with the common SI prefixes that use powers of 10 instead of powers of 2, these are sometimes referred to instead as kibibyte, mebibyte, gibibyte, and tebibyte, where the "bi" refers to "binary."

[3] `char` is derived from the word "character." People pronounce `char` variously as "car" (as in "driving the car"), "care" (a shortening of "character"), and "char" (as in charcoal), and some just punt and say "character." Up to you.



where byte 0 is used to represent a `char` and bytes 4-7 are used to represent an `int`, or



where bytes 0-3 are used to represent an `int` and byte 4 represents a `char`. Fortunately we don't usually have to worry about how variables are packed into memory.

Below are descriptions of the common data types. While the number of bytes used for each type is not the same for every processor, the numbers given are common. (Differences for the PIC32 are noted in Table A.1.) Example syntax for defining variables is also given. Note that most C statements end with a semicolon.

`char`

Example definition:
```
char ch;
```

This syntax defines a variable named `ch` to be of type `char`. `char`s are the smallest common data type, using only one byte. They are often used to represent keyboard characters. You can do a web search for "ASCII table" (pronounced "ask-key") to find the American Standard Code for Information Interchange, which maps the values 0 to 127 to keyboard characters and other things. (The values 128 to 255 may map to an "extended" ASCII table.) For example, the values 48 to 57 map to the characters '0' to '9', 65 to 90 map to the uppercase letters 'A' to 'Z', and 97 to 122 map to the lowercase letters 'a' to 'z'. The assignments

```
ch = 'a';
```

and

```
ch = 97;
```

are equivalent, as C equates characters inside single quotes to their ASCII table numerical value.

Depending on the C implementation, `char` may be treated by default as `unsigned`, i.e., taking values from 0 to 255, or `signed`, taking values from −128 to 127. If you plan to use the `char` to represent a standard ASCII character, you don't need to worry about this. If you plan to use the `char` data type for integer math on small integers, however, you may want to use the specifier `signed` or `unsigned`, as appropriate. For example, we could use the following definitions, where everything after `//` is a comment:

```
unsigned char ch1; // ch1 can take values 0 to 255
signed char ch2;   // ch2 can take values -128 to 127
```

`int` (also known as `signed int` or `signed`)

Example definition:
```
int i,j;
signed int k;
signed n;
```

`int`s are typically four bytes (32 bits) long, taking values from $-(2^{31})$ to $2^{31}-1$ (approximately ±2 billion). In the example syntax, each of `i`, `j`, `k`, and `n` are defined to be the same data type.

We can use specifiers to get the following integer data types: `unsigned int` or simply `unsigned`, a four-byte integer taking nonnegative values from 0 to $2^{32}-1$; `short int`, `short`, `signed short`, or `signed`

type	# bytes on my laptop	# bytes on PIC32
char	1	1
short int	2	2
int	4	4
long int	8	4
long long int	8	8
float	4	4
double	8	4
long double	16	8

Table A.1: Data type sizes on two different machines.

`short int`, a two-byte integer taking values from $-(2^{15})$ to $2^{15}-1$ (i.e., $-32,768$ to $32,767$); `unsigned short int` or `unsigned short`, a two-byte integer taking nonnegative values from 0 to $2^{16}-1$ (i.e., 0 to $65,535$); `long int`, `long`, `signed long`, or `signed long int`, often consisting of eight bytes and representing values from $-(2^{63})$ to $2^{63}-1$; and `unsigned long int` or `unsigned long`, an eight-byte integer taking nonnegative values from 0 to $2^{64}-1$. A `long long int` data type may also be available.

`float`

Example definition:
```
float x;
```

This syntax defines the variable `x` to be a four-byte "single-precision" floating point number.

`double`

Example definition:
```
double x;
```

This syntax defines the variable `x` to be an eight-byte "double-precision" floating point number. The data type `long double` (quadruple precision) may also be available, using 16 bytes (128 bits). These types allow the representation of larger numbers, to more decimal places, than single-precision `float`s.

The sizes of the data types, both on my laptop and the PIC32, are summarized in Table A.1. Note the differences; C does not enforce a strict standard.

Using the data types. If your program calls for floating point calculations, you can choose between `float`, `double`, and `long double` data types. The advantages of smaller types are that they use less memory and computations with them (e.g., multiplies, square roots, etc.) may be faster. The advantage of the larger types is the greater precision in the representation (e.g., smaller roundoff error).

If your program calls for integer calculations, you are better off using integer data types than floating point data types due to the higher speed of integer math and the ability to represent a larger range of integers for the same number of bytes.[4] You can decide whether to use `signed` or `unsigned char`s, or {`signed`/`unsigned`} {`short`/`long`} `int`s. The considerations are memory usage, possibly the time of the computations[5], and whether or not the type can represent a sufficient range of integer values. For example, if you decide to use `unsigned char`s for integer math to save on memory, and you add two of them with values 100 and 240 and assign to a third `unsigned char`, you will get a result of 84 due to *integer overflow*. This example is illustrated in the program `overflow.c` in Section A.4.

As we will see shortly, functions have data types, just like variables. For example, a function that calculates

[4] Just as a four-byte `float` can represent fractional values that a four-byte `int` cannot, a four-byte `int` can represent more integers than a four-byte `float` can. See the type conversion example program `typecast.c` in Section A.4 for an example.

[5] Computations with smaller data types are not always faster than with larger data types. It depends on the architecture.

the sum of two `double`s and returns a `double` should be defined as type `double`. Functions that don't return a value are defined of type `void`.

Representations of data types. A simple representation for integers is the *sign and magnitude* representation. In this representation, the msb represents the sign of the number (0 = positive, 1 = negative), and the remaining bits represent the magnitude of the number. The sign and magnitude method represents zero twice (positive and negative zero) and is not often used.

A much more common representation for integers is called *two's complement*. This method also uses the msb as a sign bit, but it only has a single representation of zero. The two's complement representation of an 8-bit `char` is given below:

binary	`signed char`, base-10	`unsigned char`, base-10
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
⋮		
01111111	127	127
10000000	−128	128
10000001	−127	129
⋮		
11111111	−1	255

As the binary representation is incremented, the two's complement (signed) interpretation of the binary number also increments, until it "wraps around" to the most negative value when the msb becomes 1 and all other bits are 0. The signed value then resumes incrementing until it reaches −1 when all bits are 1.

Another representation choice is *endianness*. The *little-endian* representation of an `int` stores the least significant byte at `ADDRESS` and the most significant byte at `ADDRESS+3`, while the *big-endian* convention is the opposite.[6] The convention used depends on the processor. For definiteness in this appendix, we will use the little-endian representation, which is also used by the PIC32.

`float`s, `double`s, and `long double`s are commonly represented in the IEEE 754 floating point format $(-1)^s * b * 2^c$, where one bit is used to represent the sign ($s = 0$ or 1); $m = 23/52/112$ bits are used to represent the significand b in the range 1 to $2 - 2^{-m}$; and $n = 8/11/15$ bits are used to represent the exponent c in the range $-(2^{n-1}) + 2$ to $2^{n-1} - 1$, where n and m depend on whether the type uses 4/8/16 bytes. Certain exponent and significand combinations are reserved for representing zero, positive and negative infinity, and "not a number" (`NaN`).

It is rare that you need to worry about the specific bit-level representation of the different data types: endianness, two's complement, IEEE 754, etc. You tell the compiler to store values and retrieve values, and it takes care of implementing the representations.

A.3.2 Memory, Addresses, and Pointers

Consider the following C syntax:

```
int i;
int *ip;
```

or equivalently

```
int i, *ip;
```

[6] These phrases come from *Gulliver's Travels*, where Lilliputians fanatically divide themselves according to which end of a soft-boiled egg they crack open.

These definitions appear to define the variables `i` and `*ip` of type `int`. The character `*` is not allowed as part of a variable name, however. The variable name is actually `ip`, and the special character `*` means that `ip` is a **pointer** to something of type `int`. The purpose of a pointer is to hold the address of a variable it "points" to. I often use the words "address" and "pointer" interchangeably.

When the compiler sees the definition `int i`, it allocates four bytes of memory to hold the integer `i`. When the compiler sees the definition `int *ip`, it creates the variable `ip` and allocates to it whatever amount of memory is needed to hold an address. The compiler also remembers the data type that `ip` points to, `int` in this case, so if later you use `ip` in a context that requires a pointer to a different variable type, the compiler will generate a warning or an error. Technically, the type of `ip` is "pointer to type `int`."

Important! Defining a pointer only allocates memory to hold the pointer. It does **not** allocate memory for a pointee variable to be pointed at. Also, simply defining a pointer does not initialize it to point to anything valid.

When we have a variable and we want the address of it, we apply the **reference operator** to the variable, which returns a "reference" (i.e., a pointer to the variable, or the address). In C, the reference operator is written `&`. Thus the following command makes sense:

```
ip = &i;  // ip now holds the address of i
```

The reference operator always returns the lowest address of a multi-byte type. For example, if the four-byte `int i` occupies addresses 0x0004 to 0x0007 in memory, `&i` will return 0x0004.[7]

If we have a pointer (an address) and we want the contents at that address, we apply the **dereference operator** to the pointer. In C, the dereference operator is written `*`. Thus the following command makes sense:

```
i = *ip;  // i now holds the contents at the address ip
```

However, you should never dereference a pointer until it has been initialized to point at something using a statement such as `ip = &i`.

As an analogy, consider the pages of a book. A page number can be considered a pointer, while the text on the page can be considered the contents of a variable. So the notation `&text` would return the page number (pointer or address) of the text, while `*page_number` would return the text on that page (but only after `page_number` is initialized to point at a page of text).

Even though we are focusing on the concept of pointers, and not C syntax, let's go ahead and look at some sample C code, remembering that everything after `//` on the same line is a comment:

```
int i,j,*ip; // define i, j as type int, as well as ip as type "pointer to type int"
ip = &i;     // set ip to the address of i (& references i)
i = 100;     // put the value 100 in the location allocated by the compiler for i
j = *ip;     // set j to the contents of the address ip (* dereferences ip), i.e., 100
j = j+2;     // add 2 to j, making j equal to 102
i = *(&j);   // & references j to get the address, then * gets contents; i is set to 102
*(&j) = 200; // content of the address of j (j itself) is set to 200; i is unchanged
```

The use of pointers can be powerful, but also dangerous. For example, you may accidentally try to access an illegal memory location. The compiler is unlikely to recognize this during compilation, and you may end up with a "segmentation fault" when you execute the code.[8] This kind of bug can be difficult to track down, and dealing with it is a C rite of passage. More on pointers in Section A.4.8.

A.3.3 Compiling

The process loosely referred to as "compilation" actually consists of four steps:

[7] This is the right way to think about it conceptually, but in fact the computer may automatically translate the value of `&i` to an actual physical address.

[8] A good name for a program like this is `coredumper.c`.

1. **Preprocessing.** The preprocessor takes the `program.c` source code and produces an equivalent `.c` source code, performing operations such as stripping out comments. The preprocessor is discussed in more detail in Section A.4.3.

2. **Compiling.** The compiler turns the preprocessed code into *assembly* code for the specific processor. This process converts the code from standard C syntax into a set of commands that can be understood natively by the processor. The compiler can be configured with a number of options that impact the assembly code generated. For example, the compiler can be instructed to generate assembly code that trades off time of execution with the amount of memory needed to store the code. Assembly code generated by a compiler can be inspected with a standard text editor. In fact, coding directly in assembly is still a popular, if painful, way to program microcontrollers.

3. **Assembling.** The assembler takes the assembly code and produces processor-dependent machine-level binary *object* code. This code cannot be examined using a text editor. Object code is called *relocatable*, in that the exact memory addresses for the data and program statements are not specified.

4. **Linking.** The linker takes one or more object codes and produces a single executable file. For example, if your code includes pre-compiled libraries, such as printout functions in the `stdio` library (described in Sections A.4.3 and A.4.15), this code is included in the final executable. The data and program statements in the various object codes are assigned to specific memory locations.

In our `HelloWorld.c` program, this entire process is initiated by the single command line statement

```
> gcc HelloWorld.c -o HelloWorld
```

If our `HelloWorld.c` program used any mathematical functions in Section A.4.7, the compilation would be initiated by

```
> gcc HelloWorld.c -o HelloWorld -lm
```

where the `-lm` flag tells the linker to link the math library, which may not be linked by default like other libraries are.

If you want to see the intermediate results of the preprocessing, compiling, and assembling steps, Problem 40 gives an example.

For more complex projects requiring compilation of several files into a single executable or specifying various options to the compiler, it is common to create a `makefile` that specifies how the compilation is to be done, and to then use the command `make` to actually create the executable. The use of `makefile`s is beyond the scope of this appendix. Section A.4.16 gives a simple example of compiling multiple C files to make a single executable program.

A.4 C Syntax

So far we have seen only glimpses of C syntax. Let's begin our study of C syntax with a few simple programs. We will then jump to a more complex program, `invest.c`, that demonstrates many of the major elements of C structure and syntax. If you can understand `invest.c` and can create programs using similar elements, you are well on your way to mastering C. We will defer the more detailed descriptions of the syntax until after introducing `invest.c`.

Printing to screen. Because it is the simplest way to see the results of a program, as well as the most useful tool for debugging, let's start with the function `printf` for printing to the screen. We have already seen it in `HelloWorld.c`. Here's a slightly more interesting example. Let's call this program file `printout.c`.

```
#include <stdio.h>

int main(void) {
```

```
  int i; float f; double d; char c;

  i = 32; f = 4.278; d = 4.278; c = 'k'; // or, by ASCII table, c = 107;
  printf("Formatted output:\n");
  printf(" i = %4d  c = '%c'\n",i,c);
  printf(" f = %19.17f\n",f);
  printf(" d = %19.17lf\n",d);
  return(0);
}
```

The `17lf` in the last `printf` statement is "seventeen ell eff."

The first line of the program

```
#include <stdio.h>
```

tells the preprocessor that the program will use functions from the "standard input and output" library, one of many code libraries provided in standard C installations that extend the power of the language. The `stdio.h` function used in `printout.c` is `printf`, covered in more detail in Section A.4.15.

The next line

```
int main(void) {
```

starts the block of code that defines the `main` function. The `main` code block is closed by the final closing brace }. Each C program has exactly one `main` function. The type of `main` is `int`, meaning that the function should end by returning a value of type `int`. In our case, it returns a `0`, which indicates that the program has terminated successfully.

The next line defines and allocates memory for four variables of four different types, while the line after assigns values to those variables. The `printf` lines will be discussed after we look at the output.

Now that you have created `printout.c`, you can create the executable file `printout` and run it from the command line. Make sure you are in the directory containing `printout.c`, then type the following:

```
> gcc printout.c -o printout
> printout
```

(Again, you may have to use `./printout` to tell your computer to look in the current directory.) On my laptop, here is the output:

```
Formatted output:
  i =   32  c = 'k'
  f = 4.27799987792968750
  d = 4.27799999999999958
```

The main point of this program is to demonstrate formatted output from the code

```
  printf("Formatted output:\n");
  printf(" i = %4d  c = '%c'\n",i,c);
  printf(" f = %19.17f\n",f);
  printf(" d = %19.17lf\n",d);
```

Inside a `printf` statement, everything inside the double quotes is printed to the screen, but some character sequences have special meaning. The `\n` sequence creates a newline (carriage return). The `%` is a special character, indicating that some data will be printed, and for each `%` in the double quotes, there must be a variable or other expression in the comma-separated list at the end of the `printf` statement. The `%4d` means that an `int` type variable is expected, and it will be displayed right-justified using 4 spaces. (If the number is more than 4 digits, it will take as much space as is needed.) The `%c` means that a `char` is expected. The `%19.17f` means that a `float` will be printed right-justified over 19 spaces with 17 spaces after the decimal point. The `%19.17lf` means that a `double` (or "long float") will be printed right-justified over 19 spaces, with 17 after the decimal point. More details on `printf` can be found in Section A.4.15.

The output of the program also shows that neither the `float f` nor the `double d` can represent 4.278 exactly, though the double-precision representation comes closer.

Data sizes. Since we have focused on data types, our next program measures how much memory is used by different data types. Create a file called `datasizes.c` that looks like the following:

```
#include <stdio.h>

int main(void) {
  char a, *bp; short c; int d; long e;
  float f; double g; long double h, *ip;

  printf("Size of char:                  %2ld bytes\n",sizeof(a));// "% 2 ell d"
  printf("Size of char pointer:          %2ld bytes\n",sizeof(bp));
  printf("Size of short int:             %2ld bytes\n",sizeof(c));
  printf("Size of int:                   %2ld bytes\n",sizeof(d));
  printf("Size of long int:              %2ld bytes\n",sizeof(e));
  printf("Size of float:                 %2ld bytes\n",sizeof(f));
  printf("Size of double:                %2ld bytes\n",sizeof(g));
  printf("Size of long double:           %2ld bytes\n",sizeof(h));
  printf("Size of long double pointer:   %2ld bytes\n",sizeof(ip));
  return(0);
}
```

The first two lines in the `main` function define nine variables, telling the compiler to allocate space for these variables. Two of these variables are pointers. The `sizeof()` operator returns the number of bytes allocated in memory for its argument.

Here is the output of the program:

```
Size of char:                   1 bytes
Size of char pointer:           8 bytes
Size of short int:              2 bytes
Size of int:                    4 bytes
Size of long int:               8 bytes
Size of float:                  4 bytes
Size of double:                 8 bytes
Size of long double:           16 bytes
Size of long double pointer:    8 bytes
```

We see that, on my laptop, `int`s and `float`s use 4 bytes, `short int`s 2 bytes, `long int`s and `double`s 8 bytes, and `long double`s 16 bytes. Regardless of whether it points to a `char` or a `long double`, a pointer (address) uses 8 bytes, meaning we can address a maximum of $(2^8)^8 = 256^8$ bytes of memory. Considering that corresponds to almost 18 quintillion bytes, or 18 billion gigabytes, we should have enough available addresses for a laptop!

Overflow. Now let's try the program `overflow.c`, which demonstrates the issue of integer overflow mentioned in Section A.3.1.

```
#include <stdio.h>

int main(void) {
  char i = 100, j = 240, sum;
  unsigned char iu = 100, ju = 240, sumu;
  signed char is = 100, js = 240, sums;

  sum = i+j; sumu = iu+ju; sums = is+js;
  printf("char:          %d + %d = %3d or ASCII %c\n",i,j,sum,sum);
  printf("unsigned char: %d + %d = %3d or ASCII %c\n",iu,ju,sumu,sumu);
  printf("signed char:   %d + %d = %3d or ASCII %c\n",is,js,sums,sums);
  return(0);
}
```

In this program we initialize the values of some of the variables when they are defined. You might also notice that we are assigning a `signed char` a value of 240, even though the range for that data type is −128 to 127. So something fishy is going on. When I compile and run the program, I get the output

```
char:           100 + -16 =  84 or ASCII T
unsigned char:  100 + 240 =  84 or ASCII T
signed char:    100 + -16 =  84 or ASCII T
```

One thing we notice is that, with my C compiler at least, `char`s are the same as `signed char`s. Another thing is that even though we assigned the value of 240 to `js` and `j`, they contain the value −16. This is because the binary representation of 240 has a 1 in the 2^7 column, but for the two's complement representation of a `signed char`, this column indicates whether the value is positive or negative. Finally, we notice that the `unsigned char ju` is successfully assigned the value 240 (since its range is 0 to 255), but the addition of `iu` and `ju` leads to an overflow. The correct sum, 340, has a 1 in the 2^8 (or 256) column, but this column is not included in the 8 bits of the `unsigned char`. Therefore we see only the remainder of the number, 84. The number 84 is assigned the character T in the standard ASCII table.

Type conversion. Continuing our focus on the importance of understanding data types, we try one more simple program that illustrates what can happen when you mix data types in a mathematical expression. This is also our first program that uses a helper function beyond the `main` function. Call this program `typecast.c`.

```
#include <stdio.h>

void printRatio(int numer, int denom) {
  double ratio;

  ratio = numer/denom;
  printf("Ratio, %1d/%1d:                        %5.2f\n",numer,denom,ratio);
  ratio = numer/((double) denom);
  printf("Ratio, %1d/((double) %1d):             %5.2f\n",numer,denom,ratio);
  ratio = ((double) numer)/((double) denom);
  printf("Ratio, ((double) %1d)/((double) %1d):  %5.2f\n",numer,denom,ratio);
}

int main(void) {
  int num = 5, den = 2;

  printRatio(num,den);
  return(0);
}
```

The helper function `printRatio` is of type `void` since it does not return a value. It takes two `int`s as input arguments and calculates their ratio in three different ways. In the first, the two `int`s are divided and the result is assigned to a `double`. In the second, the integer `denom` is **typecast** or **cast** as a `double` before the division occurs, so an `int` is divided by a `double` and the result is assigned to a `double`.[9] In the third, both the numerator and denominator are cast as `double`s before the division, so two `double`s are divided and the result is assigned to a `double`.

The `main` function simply defines two variables, `num` and `den`, and passes their values to `printRatio`, where those values are copied to `numer` and `denom`, respectively. The variables `num` and `den` are only available to `main`, and the variables `numer` and `denom` are only available to `printRatio`, since they are defined inside those functions.

Execution of any C program always begins with the `main` function, regardless of where it appears in the file.

After compiling and running, we get the output

[9] The typecasting does not change the variable `denom` itself; it simply creates a temporary `double` version of `denom` which is lost as soon as the division is complete.

```
Ratio, 5/2:                          2.00
Ratio, 5/((double) 2):               2.50
Ratio, ((double) 5)/((double) 2):    2.50
```

The first answer is "wrong," while the other two answers are correct. Why?

The first division, `numer/denom`, is an *integer* division. When the compiler sees that there are `int`s on either side of the divide sign, it assumes you want integer math and produces a result that is an `int` by simply truncating any remainder (rounding toward zero). This value, 2, is then converted to the floating point number 2.0 to be assigned to the variable `ratio`. On the other hand, the expression `numer/((double) denom)`, by virtue of the parentheses, first produces a `double` version of `denom` before performing the division. The compiler recognizes that you are dividing two different data types, so it temporarily **coerces** the `int` to a `double` so it can perform a floating point division. This is equivalent to the third and final division, except that the typecast of the numerator to `double` is explicit in the code for the third division.

Thus we have two kinds of type conversions:

- **Implicit** type conversion, or **coercion**. This occurs, for example, when a type has to be converted to carry out a variable assignment or to allow a mathematical operation. For example, dividing an `int` by a `double` will cause the compiler to treat the `int` as a `double` before carrying out the division.
- **Explicit** type conversion. An explicit type conversion is coded using a casting operator, e.g., `(double) <expression>` or `(char) <expression>`, where `<expression>` may be a variable or mathematical expression.

Certain type conversions may result in a change of value. For example, assigning the value of a `float` to an `int` results in truncation of the fractional portion; assigning a `double` to a `float` may result in roundoff error; and assigning an `int` to a `char` may result in overflow. Here's a less obvious example:

```
float f;
int i = 16777217;
f = i;            // f now has the value 16,777,216, not 16,777,217!
```

It turns out that $16{,}777{,}217 = 2^{24} + 1$ is the smallest positive integer that cannot be represented by a 32-bit `float`. On the other hand, a 32-bit `int` can represent all integers in the range -2^{31} to $2^{31} - 1$.

Some type conversions, called **promotions**, never result in a change of value because the new type can represent all possible values of the original type. Examples include converting a `char` to an `int` or a `float` to a `long double`.

We will see more on use of parentheses (Section A.4.1), the scope of variables (Section A.4.5), and defining and calling helper functions (Section A.4.6).

A more complete example: `invest.c`. Until now we have been dipping our toes in the C pool. Now let's dive in headfirst.

Our next program is called `invest.c`, which takes an initial investment amount, an expected annual return rate, and a number of years, and returns the growth of the investment over the years. After performing one set of calculations, it prompts the user for another scenario, and continues this way until the data entered is invalid. The data is invalid if, for example, the initial investment is negative or the number of years to track is outside the allowed range.

The real purpose of `invest.c`, however, is to demonstrate the syntax and a number of useful features of C.

Here's an example of compiling and running the program. The only data entered by the user are the three numbers corresponding to the initial investment, the growth rate, and the number of years.

```
> gcc invest.c -o invest
> invest
Enter investment, growth rate, number of yrs (up to 100): 100.00 1.05 5
Valid input?  1

RESULTS:
```

```
Year   0:      100.00
Year   1:      105.00
Year   2:      110.25
Year   3:      115.76
Year   4:      121.55
Year   5:      127.63

Enter investment, growth rate, number of yrs (up to 100): 100.00 1.05 200
Valid input?  0
Invalid input; exiting.
>
```

Before we look at the full `invest.c` program, let's review two principles that should be adhered to when writing a longer program: modularity and readability.

- *Modularity.* You should break your program into a set of functions that perform specific, well-defined tasks, with a small number of inputs and outputs. As a rule of thumb, no function should be longer than about 20 lines. (Experienced programmers often break this rule of thumb, but if you are a novice and are regularly breaking this rule, you're likely not thinking modularly.) Almost all variables you define should be "local" to (i.e., only recognizable by) their particular function. Global variables, which can be accessed by all functions, should be minimized or avoided altogether, since they break modularity, allowing one function to affect the operation of another without the information passing through the well-defined "pipes" (input arguments to a function or its returned results). If you find yourself typing the same (or similar) code more than once, that's a good sign you should figure out how to write a single function and just call that function from multiple places. Modularity makes it much easier to develop large programs and track down the inevitable bugs.

- *Readability.* You should use comments to help other programmers, and even yourself, understand the purpose of the code you have written. Variable and function names should be chosen to indicate their purpose. Be consistent in how you name variables and functions. Any "magic number" (constant) used in your code should be given a name and defined at the beginning of the program, so if you ever want to change this number, you can just change it at one place in the program instead of every place it is used. Global variables and constants should be written in a way that easily distinguishes them from more common local variables; for example, you could WRITE CONSTANTS IN UPPERCASE and Capitalize Globals. You should use whitespace (blank lines, spaces, tabbing, etc.) consistently to make it easy to read the program. Use a fixed-width font (e.g., `Courier`) so that the spacing/tabbing is consistent. Modularity (above) also improves readability.

The program `invest.c` demonstrates readable modular code using the structure and syntax of a typical C program. The line numbers to the left are not part of the program; they are there for reference. In the program's comments, you will see references of the form `==SecA.4.3==` that indicate where you can find more information in the review of syntax that follows the program.

```
1  /*****************************************************************************
2   * PROGRAM COMMENTS (PURPOSE, HISTORY)
3   *****************************************************************************/
4
5  /*
6   * invest.c
7   *
8   * This program takes an initial investment amount, an expected annual
9   * return rate, and the number of years, and calculates the growth of
10  * the investment.  The main point of this program, though, is to
11  * demonstrate some C syntax.
12  *
13  * References to further reading are indicated by ==SecA.B.C==
14  *
```

```
 * HISTORY:
 * Dec 20, 2011   Created by Kevin Lynch
 * Jan 4, 2012    Modified by Kevin Lynch (small changes, altered comments)
 */

/*****************************************************************************
 * PREPROCESSOR COMMANDS   ==SecA.4.3==
 *****************************************************************************/

#include <stdio.h>      // input/output library
#define MAX_YEARS 100   // Constant indicating max number of years to track

/*****************************************************************************
 * DATA TYPE DEFINITIONS (HERE, A STRUCT)  ==SecA.4.4==
 *****************************************************************************/

typedef struct {
  double inv0;                    // initial investment
  double growth;                  // growth rate, where 1.0 = zero growth
  int years;                      // number of years to track
  double invarray[MAX_YEARS+1];   // investment array   ==SecA.4.9==
} Investment;                     // the new data type is called Investment

/*****************************************************************************
 * GLOBAL VARIABLES   ==SecA.4.2, A.4.5==
 *****************************************************************************/

// no global variables in this program

/*****************************************************************************
 * HELPER FUNCTION PROTOTYPES  ==SecA.4.2==
 *****************************************************************************/

int getUserInput(Investment *invp);     // invp is a pointer to type ...
void calculateGrowth(Investment *invp); // ... Investment ==SecA.4.6, A.4.8==
void sendOutput(double *arr, int years);

/*****************************************************************************
 * MAIN FUNCTION   ==SecA.4.2==
 *****************************************************************************/

int main(void) {

  Investment inv;                // variable definition, ==SecA.4.5==

  while(getUserInput(&inv)) {    // while loop ==SecA.4.14==
    inv.invarray[0] = inv.inv0;  // struct access ==SecA.4.4==
    calculateGrowth(&inv);       // & referencing (pointers) ==SecA.4.6, A.4.8==
    sendOutput(inv.invarray,     // passing a pointer to an array ==SecA.4.9==
               inv.years);       // passing a value, not a pointer ==SecA.4.6==
  }
  return(0);                     // return value of main ==SecA.4.6==
} // ***** END main *****

/*****************************************************************************
 * HELPER FUNCTIONS   ==SecA.4.2==
 *****************************************************************************/

/* calculateGrowth
```

```
 *
 * This optimistically-named function fills the array with the investment
 * value over the years, given the parameters in *invp.
 */
void calculateGrowth(Investment *invp) {

  int i;

  // for loop ==SecA.4.14==
  for (i=1; i <= invp->years; i=i+1) {   // relational operators ==SecA.4.10==
                                         // struct access ==SecA.4.4==
    invp->invarray[i] = invp->growth * invp->invarray[i-1];
  }
} // ***** END calculateGrowth *****


/* getUserInput
 *
 * This reads the user's input into the struct pointed at by invp,
 * and returns TRUE (1) if the input is valid, FALSE (0) if not.
 */
int getUserInput(Investment *invp) {

  int valid;      // int used as a boolean ==SecA.4.10==

  // I/O functions in stdio.h ==SecA.4.15==
  printf("Enter investment, growth rate, number of yrs (up to %d): ",MAX_YEARS);
  scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));

  // logical operators ==SecA.4.11==
  valid = (invp->inv0 > 0) && (invp->growth > 0) &&
    (invp->years > 0) && (invp->years <= MAX_YEARS);
  printf("Valid input?  %d\n",valid);

  // if-else ==SecA.4.13==
  if (!valid) { // ! is logical NOT ==SecA.4.11==
    printf("Invalid input; exiting.\n");
  }
  return(valid);
} // ***** END getUserInput *****


/* sendOutput
 *
 * This function takes the array of investment values (a pointer to the first
 * element, which is a double) and the number of years (an int).  We could
 * have just passed a pointer to the entire investment record, but we decided
 * to demonstrate some different syntax.
 */
void sendOutput(double *arr, int yrs) {

  int i;
  char outstring[100];  // defining a string ==SecA.4.9==

  printf("\nRESULTS:\n\n");
  for (i=0; i<=yrs; i++) {  // ++, +=, math in ==SecA.4.7==
    sprintf(outstring,"Year %3d:  %10.2f\n",i,arr[i]);
    printf("%s",outstring);
  }
```

```
133   printf("\n");
134 } // ***** END sendOutput *****
```

A.4.1 Basic Syntax

Comments. Everything after a `/*` and before the next `*/` is a comment. Comments are stripped out in the preprocessing step of compilation. They are only there to make the purpose of the program, function, loop, or statement clear to yourself or other programmers. Keep the comments neat and concise for program readability. Some programmers use extra asterisks or other characters to make the comments pretty (see the examples in `invest.c`), but all that matters is that `/*` starts the comment and the next `*/` ends it.

If your comment is short, you can use `//` instead. Everything after `//` and before the next carriage return will be ignored.

Semicolons. A code statement must be completed by a semicolon. Some exceptions to this rule include preprocessor commands (see `PREPROCESSOR COMMANDS` in the program and Section A.4.3) and statements that end with blocks of code enclosed by braces { }. A single code statement may extend over multiple lines of the program listing until it is terminated by a semicolon (see, for example, the assignment to `valid` in the function `getUserInput`).

Braces and blocks of code. Blocks of code are enclosed in braces { }. Examples include entire functions (see the definition of the `main` function and the helper functions), blocks of code executed inside of a `while` loop (in the `main` function) or `for` loop (in the `calculateGrowth` and `sendOutput` functions), as well as other examples. In `invest.c`, braces are placed as shown here

```
while (<expression>) {
  /* block of code */
}
```

but this style is equivalent

```
while (<expression>)
  {
    /* block of code */
  }
```

as is this

```
while (<expression>) { /* block of code */ }
```

Which brings us to...

Whitespace. Whitespace, such as spaces, tabs, and carriage returns, is only required where it is needed to recognize keywords and other syntax. The whole program `invest.c` could be written without carriage returns after the semicolons, for example. Indentations and carriage returns should be used consistently, however, to make the program readable. Carriage returns should be used after each semicolon, statements within the same code block should be left-justified with each other, and statements in a code block nested within another code block should be indented with respect to the parent code block. Text editors should use a fixed-width font so that alignment is clear. Most IDE editors provide fixed-width fonts and automatic indentation to enhance readability.

Parentheses. C has a set of rules defining the order in which operations in an expression are evaluated, much like standard math rules that say $3 + 5 * 2$ evaluates to $3 + (10) = 13$, not $(8) * 2 = 16$. If you are uncertain of the default order of evaluation, use parentheses `( )` to enclose sub-expressions to enforce the evaluation order you want. More deeply nested parenthetical expressions will be evaluated first. For example, $3 + (40/(4 * (3 + 2)))$ evaluates to $3 + (40/(4 * 5)) = 3 + (40/20) = 3 + 2 = 5$. Parentheses can be used to control the order of evaluation for non-mathematical statements, too. An example is shown in `getUserInput` of `invest.c`:

```
valid = (invp->inv0 > 0) && (invp->growth > 0) &&
        (invp->years > 0) && (invp->years <= MAX_YEARS);
```

Each relational expression using `>` and `<=` (Section A.4.10) is evaluated before applying the logical AND operators `&&` (Section A.4.11).

A.4.2 Program Structure

`invest.c` demonstrates a typical structure for a program written in one `.c` file. When you write larger programs, you may wish to break your program into multiple files. In this case, exactly one of these files must contain a `main` function, and you have some choices as to which variables and functions in one file are visible to other files. In this appendix we will focus on programs that consist of a single file (apart from any C libraries you may include, as we will discuss in Section A.4.3). Section A.4.16 gives a simple example of a program broken up into multiple C files.

Let's consider the seven major sections of the program in order of appearance. `PROGRAM COMMENTS` describe the purpose of the program and its revision history. `PREPROCESSOR COMMANDS` define constants and "header" files that should be included, giving the program access to library functions that extend the power of the C language. This is described in more detail in Section A.4.3. In some programs, it may be helpful to define a new data type, as shown in `DATA TYPE DEFINITIONS`. In `invest.c`, several variables are packaged together in a single record or `struct` data type, as described in Section A.4.4. Any `GLOBAL VARIABLES` are then defined. These are variables that are available for use by all functions in the program. Because of this special status, the names of global variables could be Capitalized or otherwise written in a way to remind the programmer that they are not local variables (Section A.4.5).

The next section of the program contains the `HELPER FUNCTION PROTOTYPES` of the various helper functions. A prototype of a function declares the name of the function that will be defined later, its data type, and the data types of the **arguments** passed to the function. As an example, the function `printRatio` is of type `void`, since it does not return a value, and takes two arguments, each of type `int`. The function `getUserInput` takes a single argument which is a pointer to a variable of type `Investment`, a data type which is defined a few lines above, and returns an `int`.

The next section of the program, `MAIN FUNCTION`, is where the `main` function is defined. Every program has exactly one `main` function, and it is where the program starts execution. The `main` function is of type `int`, and by convention it returns a 0 if it executes successfully, and otherwise returns a nonzero value. In `invest.c`, `main` takes no arguments, hence the `void` in the argument list. On the other hand, when a program is run from the command line, it is possible to specify arguments to `main`. For example, we could have written `invest.c` to be run with a command such as this:

```
> invest 100.0 1.05 5
```

To allow this, `main` would have been defined with the following syntax:

```
int main(int argc, char **argv) {
```

Then when the program is invoked as above, the integer `argc` would be set to 4, the number of whitespace-separated strings on the command line, and `argv` would point to a vector of 4 strings, where the string `argv[0]` is 'invest', `argv[1]` is '100.0', etc. You can learn more about arrays and strings in Section A.4.9.

Finally, the last section of the program is the definition of the `HELPER FUNCTIONS` whose prototypes were given earlier. It is not strictly necessary that the helper functions have prototypes, but if not, every function should be defined before it is used by any other function. For example, none of the helper functions uses

another helper function, so they could have all been defined before the `main` function, in any order, and their function prototypes eliminated. The names of the variables in a function prototype and in the actual definition of the function need not be the same; for example, the prototype of `sendOutput` uses variables named `arr` and `years`, whereas the actual function definition uses `arr` and `yrs`. What matters is that the prototype and actual function definition have the same number of arguments, of the same types, and in the same order. In fact, in the arguments of the function prototypes, you can leave out variable names altogether, and just keep the comma separated list of argument data types.

A.4.3 Preprocessor Commands

In the preprocessing stage of compilation, all comments are stripped out of the program. In addition, the preprocessor encounters the following preprocessor commands, recognizable by the `#` character:

```
#include <stdio.h>      // input/output library
#define MAX_YEARS 100   // Constant indicating max number of years to track
```

Constants. The second line defines the constant `MAX_YEARS` to be equal to 100. The preprocessor searches for each instance of `MAX_YEARS` in the program and replaces it with 100. If we later decide that the maximum number of years to track investments should be 200, we can simply change the definition of this constant, in one place, instead of in several places. Since `MAX_YEARS` is a constant, not a variable, it can never be assigned another value somewhere else in the program. To indicate that it not a variable, a common convention is to write constants in UPPERCASE. This is not required by C, however.

Included libraries. The first line of the preprocessor commands in `invest.c` indicates that the program will use the standard C input/output library. The file `stdio.h` is called a **header** file for the library. This file is readable by a text editor and contains a number of constants that are made available to the program, as well as a set of function prototypes for input and output functions. The preprocessor replacea the `#include <stdio.h>` command with the header file `stdio.h`.[10] Examples of function prototypes that are included are

```
int printf(const char *Format, ...);
int sprintf(char *Buffer, const char *Format, ...);
int scanf(const char *Format, ...);
```

Each of these three functions is used in `invest.c`. If the program were compiled without including `stdio.h`, the compiler would generate a warning or an error due to the lack of function prototypes. See Section A.4.15 for more information on using the `stdio` input and output functions.

During the linking stage, the object code of `invest.c` is linked with the object code for `printf`, `sprintf`, and `scanf` in your C installation. Libraries like `stdio` provide access to functions beyond the basic C syntax. Other useful libraries are briefly described in Section A.4.15.

Macros. One more use of the preprocessor is to define simple function-like *macros* that you may use in more than one place in your program. Here's an example that converts radians to degrees:

```
#define RAD_TO_DEG(x) ((x) * 57.29578)
```

The preprocessor will search for any instance of `RAD_TO_DEG(x)` in the program, where `x` can be any expression, and replace it with `((x) * 57.29578)`. For example, the initial code

```
angle_deg = RAD_TO_DEG(angle_rad);
```

is replaced by

[10] This assumes that our preprocessor can find the header file somewhere in the "include path" of directories to search for header files. If the header file `header.h` sits in the same directory as `invest.c`, we would write `#include "header.h"` instead of `#include <header.h>`.

```
angle_deg = ((angle_rad) * 57.29578);
```

Note the importance of the outer parentheses in the macro definition. If we had instead used the preprocessor command

```
#define RAD_TO_DEG(x) (x) * 57.29578   // don't do this!
```

then the code

```
answer = 1.0 / RAD_TO_DEG(3.14);
```

would be replaced by

```
answer = 1.0 / (3.14) * 57.29578;
```

which is very different from

```
answer = 1.0 / ((3.14) * 57.29578);
```

Moral: if the expression you are defining is anything other than a single constant, enclose it in parentheses, to tell the compiler to evaluate the expression first. You can even enclose a single constant in parentheses; it doesn't cost you anything.

As a second example, the macro

```
#define MAX(A,B)      ((A) > (B) ? (A):(B))
```

returns the maximum of two arguments. The `?` is the *ternary operator* in C, which has the form

```
<test> ? return_value_if_test_is_true : return_value_if_test_is_false
```

The preprocessor replaces

```
maxval = MAX(13+7,val2);
```

with

```
maxval = ((13+7) > (val2) ? (13+7):(val2));
```

Why define a macro instead of just writing a function? One reason is that the macro may execute slightly faster, since no passing of control to another function and no passing of variables is needed.

A.4.4 Defining Structs and Data Types

In most programs you write, you will do just fine with the data types `int, char, float, double`, and variations. Occasionally, though, you will find it useful to define a new data type. You can do this with the following command:

```
typedef <type> newtype;
```

where `<type>` is a standard C data type and `newtype` is the name of your new data type, which will be the same as `<type>`. Then you can define a new variable `x` of type `newtype` by

```
newtype x;
```

For example, you could write

```
typedef int days_of_the_month;
days_of_the_month day;
```

You might find it satisfying that your variable `day` (taking values 1 to 31) is of type `days_of_the_month`, but the compiler will still treat it as an `int`.

A more useful example is when you have several variables that are always used together. You might like to package these variables together into a single record, as we do with the investment information in `invest.c`. This packaging can be done with a `struct`. The `invest.c` code

```
typedef struct {
  double inv0;                    // initial investment
  double growth;                  // growth rate, where 1.0 = zero growth
  int years;                      // number of years to track
  double invarray[MAX_YEARS+1];   // investment values
} Investment;                     // the new data type is called Investment
```

replaces the data type `int` in our previous `typedef` example with `struct {...}`. This syntax creates a new data type `Investment` with a record structure, with *fields* named `inv0` and `growth` of type `double`, `years` of type `int`, and `invarray`, an array of `doubles`. (Arrays are discussed in Section A.4.9.) With this new type definition, we can define a variable named `inv` of type `Investment`:

```
Investment inv;
```

This definition allocates sufficient memory to hold the two `doubles`, the `int`, and the array of `doubles`. We can access the contents of the `struct` using the "`.`" operator:

```
int yrs;
yrs = inv.years;
inv.growth = 1.1;
```

An example of this kind of usage is seen in `main`.

Referring to the discussion of pointers in Sections A.3.2 and A.4.8, if we are working with a pointer `invp` that points to `inv`, we can use the "`->`" operator to access the contents of the record `inv`:

```
Investment inv;     // allocate memory for inv, an investment record
Investment *invp;   // invp will point to something of type Investment
int yrs;
invp = &inv;        // invp points to inv
inv.years = 5;      // setting one of the fields of inv
yrs = invp->years;  // inv.years, (*invp).years, and invp->years are all identical
invp->growth = 1.1;
```

Examples of this usage are seen in `calculateGrowth()` and `getUserInput()`.

A.4.5 Defining Variables

Variable names. Variable names can consist of uppercase and lowercase letters, numbers, and underscore characters '_'. You should generally use a letter as the first character; `var`, `Var2`, and `Global_Var` are all valid names, but `2var` is not. C is case sensitive, so the variable names `var` and `VAR` are different. A variable name cannot conflict with a reserved keyword in C, like `int` or `for`. Names should be succinct but descriptive. The variable names `i`, `j`, and `k` are often used for integers, and pointers often begin with `ptr_`, such as `ptr_var`, or end with `p`, such as `varp`, to remind you that they are pointers. These are all to personal taste, however.

Scope. The **scope** of a variable refers to where it can be used in the program. A variable may be *global*, i.e., usable by any function, or *local* to a specific function or piece of a function. A global variable is one that is defined in the `GLOBAL VARIABLES` section, outside of and before any function that uses it. Such variables can be referred to or altered in any function.[11] Because of this special status, global variables are often Capitalized. Global variable usage should be minimized for program modularity and readability.

[11] You could also define a variable outside of any function definition but *after* some of the function definitions. This quasi-global variable would be available to all functions defined after the variable is defined, but not those before. This practice is discouraged, as it makes the code harder to read.

A local variable is one that is defined in a function. Such a variable is only usable inside that function, after the definition.[12] If you choose a local variable name `var` that is also the name of a global variable, inside that function `var` will refer to the local variable, and the global variable will not be available. It is not good practice to choose local variable names to be the same as global variable names, as it makes the program confusing to understand.

A local variable can be defined in the argument list of a function definition, as in `sendOutput` at the end of `invest.c`:

```
void sendOutput(double *arr, int yrs) {    // ...
```

Otherwise, local variables are defined at the beginning of the function code block by syntax similar to that shown in the function `main`.

```
int main(void) {
  Investment inv;    // Investment is a variable type we defined
  // ... rest of the main function ...
```

Since this definition appears within the function, `inv` is local to `main`. Had this definition appeared before any function definition, `inv` would be a global variable.

Definition and initialization. When a variable is defined, memory for the variable is allocated. In general, you cannot assume anything about the contents of the variable until you have initialized it. For example, if you want to define a `float x` with value 0.0, the command

```
float x;
```

is insufficient. The memory allocated may have random 0's and 1's already in it, and the allocation of memory does not generally change the current contents of the memory. Instead, you can use

```
float x = 0.0;
```

to initialize the value of `x` when you define it. Equivalently, you could use

```
float x;
x = 0.0;
```

Static local variables. Each time a function is called, its local variables are allocated space in memory. When the function completes, its local variables are thrown away, freeing memory. If you want to keep the results of some calculation by the function after the function completes, you could either return the results from the function or store them in a global variable. An alternative is to use the `static` modifier in the local variable definition, as in the following program:

```
#include <stdio.h>

void myFunc(void) {
  static char ch='d';   // this local variable is static, allocated and initialized
                        // only once during the entire program
  printf("ch value is %d, ASCII character %c\n",ch,ch);
  ch = ch+1;
}

int main(void) {
  myFunc();
  myFunc();
  myFunc();
  return 0;
}
```

[12] Since we recommend that each function be brief, you can define all local variables in that function at the beginning of the function, so we can see in one place what local variables the function uses. Some programmers prefer instead to define variables just before their first use, to minimize their scope. Older C specifications required that all local variables be defined at the beginning of a code block enclosed by braces { }.

The `static` modifier in the definition of `ch` in `myFunc` means that `ch` is only allocated, and initialized to `'d'`, the first time `myFunc` is called during the execution of the program. This allocation persists after the function is exited, and the value of `ch` is remembered. The output of this program is

```
ch value is 100, ASCII character d
ch value is 101, ASCII character e
ch value is 102, ASCII character f
```

Numerical values. Just as you can assign an integer a base-10 value using commands like `ch=100`, you can assign a number written in hexadecimal notation by putting "`0x`" at the beginning of the digit sequence, e.g.,

```
unsigned char ch = 0x4D;
```

This form may be convenient when you want to directly control bit values. This is often useful in microcontroller applications.

A.4.6 Defining and Calling Functions

A function definition consists of the function's data type, the function name, a list of arguments that the function takes as input, and a block of code. Allowable function names follow the same rules as variables. The function name should make clear the purpose of the function, such as `getUserInput` in `invest.c`.

If the function does not return a value, it is defined as type `void`, as with `calculateGrowth`. If it does return a value, such as `getUserInput` which returns an `int`, the function should end with the command

```
return(val);
```

or

```
return val;
```

where `val` is a variable of the same type as the function. The `main` function is of type `int` and should return 0 upon successful completion.

The function definition

```
void sendOutput(double *arr, int yrs) {  // ...
```

indicates that `sendOutput` returns nothing and takes two arguments, a pointer to type `double` and an `int`. When the function is called with the statement

```
sendOutput(inv.invarray, inv.years);
```

the `invarray` and `years` fields of the `inv` structure in `main` are copied to `sendOutput`, which now has its own local copies of these variables, stored in `arr` and `yrs`. The difference is that `yrs` is simply data, while `arr` is a pointer, specifically the address of the first element of `invarray`, i.e., `&(inv.invarray[0])`. (Arrays will be discussed in more detail in Section A.4.9.) Since `sendOutput` now has the memory address of the beginning of this array, *it can directly access, and potentially change, the original array seen by* `main`. On the other hand, `sendOutput` cannot by itself change the value of `inv.years` in `main`, since it only has a copy of that value, not the actual memory address of `main`'s `inv.years`. `sendOutput` takes advantage of its direct access to the `inv.invarray` to print out all the values stored there, eliminating the need to copy all the values of the array from `main` to `sendOutput`.

The function `calculateGrowth`, which is called with a pointer to `main`'s `inv` data structure, takes advantage of its direct access to the `invarray` field to change the values stored there.

When a function is called with a pointer argument, it is sometimes called a *call by reference*; the call sends a reference (address, or pointer) to data. When a function is called with non-pointer data, it is sometimes called a *call by value*; data is copied over, but not an address.

If a function takes no arguments and returns no value, we can define it as `void myFunc(void)` or `void myFunc()`. The function is called using

```
myFunc();
```

A.4.7 Math

Standard *binary* math operators (operators on two operands) include `+, -, *,` and `/`. These operators take two operands and return a result, as in

```
ratio = a/b;
```

If the operands are the same type, then the CPU carries out a division (or add, subtract, multiply) specific for that type and produces a result of the same type. In particular, if the operands are integers, the result will be an integer, even for division (fractions are rounded toward zero). If one operand is an integer type and the other is a floating point type, the integer type will generally be coerced to a floating point to allow the operation (see the `typecast.c` program description of Section A.4).

The modulo operator `%` takes two integers and returns the remainder of their division, i.e.,

```
int i;
i = 16%7;  // i is now equal to 2
```

C also provides `+=, -=, *=, /=, %=` to simplify some expressions, as shown below:

```
x = x * 2;  y = y + 7;    // this line of code is equivalent...
x *= 2;     y += 7;       // ...to this one
```

Since adding one to an integer or subtracting one from an integer are common operations in loops, these have a further simplification. For an integer `i`, we can write

```
i++;  // adds 1 to i, equivalent to i = i+1;
i--;  // equivalent to i = i-1;
```

In fact we also have the syntax `++i` and `--i`. If the `++` or `--` come in front of the variable, the variable is modified before it is used in the rest of the expression. If they come after, the variable is modified after the expression has been evaluated. So

```
int i=5,j;
j = (++i)*2;   // after this line, i is 6 and j is 12
```

but

```
int i=5,j;
j = (i++)*2;   // after this line, i is 6 and j is 10
```

But your code would be much more readable if you just wrote `i++` before or after the `j=i*2` line.

If your program includes the C math library with the preprocessor command `#include <math.h>`, you have access to a much larger set of mathematical operations, some of which are listed here:

```
int     abs      (int x);               // integer absolute value
double  fabs     (double x);            // floating point absolute value
double  cos      (double x);            // all trig functions work in radians, not degrees
double  sin      (double x);
double  tan      (double x);
double  acos     (double x);            // inverse cosine
double  asin     (double x);
double  atan     (double x);
double  atan2    (double y, double x);  // two-argument arctangent
double  exp      (double x);            // base e exponential
double  log      (double x);            // natural logarithm
double  log2     (double x);            // base 2 logarithm
double  log10    (double x);            // base 10 logarithm
double  pow      (double x, double y);  // raise x to the power of y
double  sqrt     (double x);            // square root of x
```

These functions also have versions for `float`s. The names of those functions are identical, except with an '`f`' appended to the end, e.g., `cosf`.

When compiling programs using `math.h`, remember to include the linker flag `-lm`, e.g.,

```
gcc myprog.c -o myprog -lm
```

The math library is not linked by default like most other libraries.

A.4.8 Pointers

It's a good idea to review the introduction to pointers in Section A.3.2 and the discussion of call by reference in Section A.4.6. In summary, the operator `&` references a variable, returning a pointer to (the address of) that variable, and the operator `*` dereferences a pointer, returning the contents of the address.

These statements define a variable `x` of type `float` and a pointer `ptr` to a variable of type `float`:

```
float x;
float *ptr;
```

At this point, the assignment

```
*ptr = 10.3;
```

would result in an error, because the pointer `ptr` does not currently point to anything. The following code would be valid:

```
ptr = &x;         // assign ptr to the address of x; x is the "pointee" of ptr
*ptr = 10.3;      // set the contents at address ptr to 10.3; now x is equal to 10.3
*(&x) = 4 + *ptr; // the * and & on the left cancel each other; x is set to 14.3
```

Since `ptr` is an address, it is an integer (technically the type is "pointer to type float"), and we can add and subtract integers from it. For example, say that `ptr` contains the value n, and then we execute the statement

```
ptr = ptr + 1;    // equivalent to ptr++;
```

If we now examined `ptr`, we would find that it has the value $n + 4$. Why? Because the compiler knows that `ptr` points to the type `float`, so when we add 1 to `ptr`, the assumption is that we want to increment one `float` in memory, not one byte. Since a `float` occupies four bytes, the address `ptr` must increase by 4 to point to the next `float`. The ability to increment a pointer in this way can be useful when dealing with arrays, next.

A.4.9 Arrays and Strings

One-dimensional arrays. An array of five `float`s can be defined by

```
float arr[5];
```

We could also initialize the array at the time we define it:

```
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
```

Each of these definitions allocates five `float`s in memory, accessed by `arr[0]` (initialized to 0.0 above) through `arr[4]` (initialized to 40.0). The assignment

```
arr[5] = 3.2;
```

is a mistake, since only `arr[0..4]` have been allocated. This statement would likely compile just fine, because compilers typically do not check for indexing arrays out of bounds. The best result at this point would be for your program to crash, to alert you to the fact that you are overwriting memory that may be allocated for another purpose. More insidiously, the program could seem to run just fine, but with difficult-to-debug erratic behavior. Bottom line: never access arrays out of bounds!

In the expression `arr[i]`, `i` is an integer called the *index*, and `arr[i]` is of type `float`. The variable `arr` by itself is actually a pointer to the first element of the array, equivalent to `&(arr[0])`. The address `&(arr[i])` is located at the address `arr` plus `i*4` bytes, since the elements of the array are stored consecutively, and a `float` uses four bytes. Both `arr[i]` and `*(arr+i)` are correct syntax to access the `i`'th element of the array. Since the compiler knows that `arr` is a pointer to the four-byte type `float`, the address represented by `(arr+i)` is `i*4` bytes higher than the address `arr`.

Consider the following code snippet:

```
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
float *ptr;
ptr = arr + 3;
// arr[0] contains 0.0 and ptr[0] = arr[3] = 30.0
// arr[0] is equivalent to *arr; ptr[0] is equivalent to *ptr and *(arr+3);
// ptr is equivalent to &(arr[3])
```

If we'd like to pass the array `arr` to a function that initializes each element of the array, we could call

```
arrayInit(arr,5);
```

or

```
arrayInit(&(arr[0]),5);
```

The function definition for `arrayInit` might look like

```
void arrayInit(float *vals, int length) {

  int i;

  for (i=0; i<length; i++) vals[i] = i*10.0;
  // equivalently, we could substitute the line below for the line above
  // for (i=0; i<length; i++) {*vals = i*10.0; vals++;}
}
```

The pointer `vals` in `arrayInit` is set to point to the same location as `arr` in the calling function. Therefore `vals[i]` refers to the same memory contents that `arr[i]` does.

Note that `arr` does not carry any information on the length of the array. This is why we have to separately send the length of the array to `arrayInit`.

Strings. A string is an array of `char`s. The definition

```
char s[100];
```

allocates memory for 100 `char`s, `s[0]` to `s[99]`. We could initialize the array with

```
char s[100] = "cat"; // note the double quotes
```

This places a '`c`' (integer value 99) in `s[0]`, an '`a`' (integer value 97) in `s[1]`, a '`t`' (integer value 116) in `s[2]`, and a value of 0 in `s[3]`, corresponding to the NULL character and indicating the end of the string. (You could also do this, less elegantly, by initializing just those four elements using braces as we did with the `float` array above.)

You notice that we allocated more memory than was needed to hold "cat." Perhaps we will append something to the string in future, so we might want to allocate that extra space just in case. But if not, we could have initialized the string using

```
char s[] = "cat";
```

and the compiler would only assign the minimum memory needed.

The function `sendOutput` in `invest.c` shows an example of constructing a string using `sprintf`, a function provided by `stdio.h`. Other functions for manipulating strings are provided in `string.h`. Both of these libraries are described briefly in Section A.4.15.

Multi-dimensional arrays. The definition

```
int mat[2][3];
```

allocates memory for 6 `int`s, `mat[0][0]` to `mat[1][2]`, which can be thought of as a two-dimensional array, or matrix. These occupy a contiguous region of memory, with `mat[0][0]` at the lowest memory location, followed by `mat[0][1]`, `mat[0][2]`, `mat[1][0]`, `mat[1][1]`, and `mat[1][2]`. This matrix can be initialized using nested braces,

```
int mat[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Higher-dimensional arrays can be created by simply adding more indexes. In memory, a "row" of the rightmost index is completed before incrementing the next index to the left.

Static vs. dynamic memory allocation. A command of the form `float arr[5]` is called *static memory allocation*. This means that the size of the array is known at compile time. Another option is *dynamic memory allocation*, where the size of the array can be chosen at run time.[13] With the C library `stdlib.h` included using the preprocessor command `#include <stdlib.h>`, the syntax

```
float *arr; // arr is a pointer to float, but no memory has been allocated for the array
int i=5;
arr = (float *) malloc(i * sizeof(float));  // allocate the memory
```

allocates `arr[0..4]`, and

```
free(arr);
```

releases the memory when it is no longer needed.[14] If `malloc` cannot allocate the requested memory, perhaps because the computer is out of memory, it returns a NULL pointer (i.e., `arr` will have value 0).

A.4.10 Relational Operators and TRUE/FALSE Expressions

`==`	equal
`!=`	not equal
`>, >=`	greater than, greater than or equal to
`<, <=`	less than, less than or equal to

Relational operators operate on two values and evaluate to 0 or 1. A 0 indicates that the expression is FALSE and a 1 indicates that the expression is TRUE. For example, the expression `(3>=2)` is TRUE, so it evaluates to 1, while `(3<2)` evaluates to 0, or FALSE.

The most common mistake is using `=` to test for equality instead of `==`. For example, using the `if` conditional syntax (Section A.4.13), the test

```
int i=2;
if (i=3) printf("Test is true.");
```

[13] Dynamic memory is allocated from the *heap*, a portion of memory set aside for dynamic allocation (and therefore is not available for statically allocated variables and program code). You may have to adjust linker options setting the size of the heap.

[14] Bookkeeping has kept track of the size of the block associated with the address `arr`, so you don't need to tell `free` how much memory to release.

will always evaluate to TRUE, because the expression `(i=3)` assigns the value of 3 to `i`, and the expression evaluates to 3. Any nonzero value is treated as logical TRUE. If the condition is written `(i==3)`, it will operate as intended, evaluating to 0 (FALSE).

Be aware of potential pitfalls in checking equality of floating point numbers. Cconsider the following program:

```
#include <stdio.h>
#define VALUE 3.1
int main(void) {
  float  x = VALUE;
  double y = VALUE;
  if (x==VALUE) printf("x is equal to %lf.\n",VALUE);
  else printf("x is not equal to %lf!\n",VALUE);
  if (y==VALUE) printf("y is equal to %lf.\n",VALUE);
  else printf("y is not equal to %lf!\n",VALUE);
  return 0;
}
```

You might be surprised to see that your program says that `x` is not equal while `y` is! In fact, neither `x` nor `y` are exactly 3.1 due to roundoff error in the floating point representation. However, by default, the constant 3.1 is treated as a `double`, so the `double y` carries the identical (wrong) value. If you want a constant to be treated explicitly as a `float`, you can write it as 3.1F, and if you want it to be treated as a `long double`, you can write it as 3.1L.

A.4.11 Logical Operators

Logical operators include AND, OR, and NOT, written as `&&`, `||`, and `!`, respectively. Here are some examples:

```
    (3>2) && (4!=0)  // (TRUE) AND (TRUE) evaluates to TRUE
    (3>2) || (4==0)  // (TRUE) OR (FALSE) evaluates to TRUE
   !(3>2) || (4==0)  // NOT(TRUE) OR (FALSE) evaluates to FALSE
```

Another example is given in `getUserInput`, where four expressions are AND'ed. As always, if you are unsure of the order of evaluating a string of logical expressions, use parentheses to enforce the order you want.

A.4.12 Bitwise Operators

~	bitwise NOT
&	bitwise AND
|	bitwise OR
∧	bitwise XOR
>>	shift bits to the right (shifting in 0's from the left)
<<	shift bits to the left (shifting in 0's from the right)

Bitwise operators act directly on the bits of the operand(s), as in the following example:

```
unsigned char a=0xC, b=0x6, c;  // in binary, a is 0b00001100 and b is 0b00000110
c = ~a;     // NOT; c is 0xF3 or 0b11110011
c = a & b;  // AND; c is 0x04 or 0b00000100
c = a | b;  // OR;  c is 0x0E or 0b00001110
c = a ^ b;  // XOR; c is 0x0A or 0b00001010
c = a >> 3; // SHIFT RT 3; c is 0x01 or 0b00000001, one 1 is shifted off the right end
c = a << 3; // SHIFT LT 3; c is 0x60 or 0b01100000, 1's shifted to more significant digits
```

Much like the math operators, we also have the assignment expressions `&=`, `|=`, `^=`, `>>=`, and `<<=`, so `a &= b` is equivalent to `a = a&b`.

A.4.13 Conditional Statements

If-Else. The basic `if-else` construct takes this form:

```
if (<expression>) {
  // execute this code block if <expression> is TRUE, then exit
}
else {
  // execute this code block if <expression> is FALSE
}
```

If the code block is a single statement, the braces are not necessary. The `else` and the block after it can be eliminated if no action needs to be taken when `<expression>` is FALSE.

`if-else` statements can be made into arbitrarily long chains:

```
if (<expression1>) {
  // execute this code block if <expression1> is TRUE, then exit this if-else chain
}
else if (<expression2>) {
  // execute this code block if <expression2> is TRUE, then exit this if-else chain
}
else {
  // execute this code block if both expressions above are FALSE
}
```

An example `if` statement is in `getUserInput`.

Switch. If you would like to check if the value of a single expression is one of several possibilities, a `switch` may be simpler than a chain of `if-else` statements. Here is an example:

```
char ch;
// ... omitting code that sets the value of ch ...
switch (ch) {
  case 'a':      // execute these statements if ch has value 'a'
    <statement>;
    <statement>;
    break;       // exit the switch statement
  case 'b':
    // ... some statements
    break;
  case 'c':
    // ... some statements
    break;
  default:       // execute this code if none of the previous cases applied
    // ... some statements
}
```

A.4.14 Loops

`for` loop. A `for` loop has the following syntax:

```
for (<initialization>; <test>; <update>) {
  // code block
  }
```

If the code block consists of only one statement, the surrounding braces can be eliminated.

The sequence is as follows: at the beginning of the loop, the `<initialization>` statement is executed. Then the `<test>` is evaluated. If it is TRUE, then the code block is executed, the `<update>` is performed, and we return to the `<test>`. If it is FALSE, the `for` loop is exited.

The following `for` loop is in `calculateGrowth`:

```
for (i=1; i <= invp->years; i=i+1) {
  invp->invarray[i] = invp->growth*invp->invarray[i-1];
}
```

The <initialization> step sets i=1. The <test> is TRUE if i is less than or equal to the number of years we will calculate growth in the investment. If it is TRUE, the value of the investment in year i is calculated from the value in year i-1 and the growth rate. The <update> adds 1 to i. In this example, the code block is executed for i values of 1 to invp->years.

It is possible to perform more than one statement in the <initialization> and <update> steps by separating the statements by commas. For example, we could write

```
for (i=1,j=10; i <= 10; i++, j--)  { /* code */ };
```

if we want i to count up and j to count down.

while loop. A while loop has the following syntax:

```
while (<test>) {
  // code block
}
```

First, the <test> is evaluated, and if it is FALSE, the while loop is exited. If it is TRUE, the code block is executed and we return to the <test>.

In main of invest.c, the while loop executes until the function getUserInput returns 0, i.e., FALSE. getUserInput collects the user's input and returns an int that is 0 if the user's input is invalid and 1 if it is valid.

do-while loop. This is similar to a while loop, except the <test> is executed at the end of the code block.

```
do {
  // code block
} while (<test>);
```

break and continue. If anywhere in the loop's code block the command break is encountered, the program will exit the loop. If the command continue is encountered, the rest of the commands in the code block will be skipped, and control will return to the <update> in a for loop or the <test> in a while or do-while loop. Examples:

```
while (<test1>) {
  if (<test2>) break;  // jump out of the while loop
  // ...
}
```

```
while (<test1>) {
  if (<test2>) continue;  // skip the rest of the loop and go back to <test1>
  x = x+3;
}
```

A.4.15 Some Useful Libraries

Libraries can be used in your C program if you include the .h header file that defines the library function prototypes.[15] We have already seen examples of functions in header files such as stdio.h, which contains input/output functions; math.h in Section A.4.7; and stdlib.h in Section A.4.9.

It is well beyond our scope to provide details on the standard libraries in C. If you are interested, try a web search on "standard libraries in C." Here we highlight a few particularly useful functions in stdio.h, string.h, and stdlib.h.

[15] Reminder: if you include <math.h>, you should also compile your program with the -lm flag, so the math library is linked during the linking stage.

Input and Output: `stdio.h`

```
int printf(const char *Format, ...);
```

The function `printf` is used to print to the "standard output," which, for a PC, is typically the screen. It takes a formatting string `Format` and a variable number of extra arguments, determined by the formatting string, as indicated by the `...` notation. The keyword `const` means that `printf` cannot change the string `Format`.

An example comes from our program `printout.c`:

```
int i; float f; double d; char c;
i = 32; f = 4.278; d = 4.278; c = 'k';
printf("Formatted string:  i = %4d  c = '%c'\n",i,c);
printf("f = %25.23f  d = %25.23lf\n",f,d);
```

which produces the output

```
Formatted string:  i =   32  c = 'k'
f = 4.27799987792968750000000  d = 4.27799999999999958077979
```

The formatting strings consist of plain text, the special character `\n` that prints a newline, and directives of the form `%4d` and `%25.23f`. Each directive indicates that `printf` will be looking for a corresponding variable in the argument list to insert into the output. A non-exhaustive list of directives is given here:

- `%d` Print an `int`. Corresponding argument should be an `int`.
- `%u` Print an `unsigned int`. Corresponding argument should be an integer data type.
- `%ld` Print a `long int`.
- `%f` Print a `float`.
- `%lf` Print a `double`, or "long float."
- `%c` Print a character according to the ASCII table. Argument should be `char`.
- `%s` Print a string. Argument should be a pointer to a `char` (first element of a string).
- `%X` Print an `unsigned int` as a hex number.

The directive `%d` can be written instead as `%4d`, for example, meaning that four spaces are allocated to write the integer, which will be right-justified in that space with unused spaces blank. The directive `%f` can be written instead as `%6.3f`, indicating that six spaces are reserved to write out the variable, with one of those spaces being the decimal point and three of the spaces after the decimal point.

```
int sprintf(char *str, const char *Format, ...);
```

Instead of printing to the screen, `sprintf` prints to the string `str`. An example of this is in `sendOutput`.

```
int scanf(const char *Format, ...);
```

The function `scanf` is a formatted read from the "standard input," which is typically the keyboard. Arguments to `scanf` consist of a formatting string and pointers to variables where the input should be stored. Typically the formatting string consists of directives like `%d`, `%f`, etc., separated by whitespace. The directives are similar to those for `printf`, except they don't accept spacing modifiers (like the 5 in `%5d`).

For each directive, `scanf` expects to see a pointer to a variable of that type in the argument list. A very common mistake is the following:

```
int i;
scanf("%d",i);  // WRONG!  We need a pointer to the variable.
scanf("%d",&i); // RIGHT.
```

The pointer allows `scanf` to put the input into the right place in memory.

`getUserInput` uses the statement

```
scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));
```

to read in two doubles and an integer and place them into the appropriate spots in the investment data structure. `scanf` ignores the whitespace (tabs, newlines, spaces, etc.) between the inputs.

```
int sscanf(char *str, const char *Format, ...);
```

Instead of scanning from the keyboard, `scanf` scans the string pointed to by `str`.

```
FILE* fopen(const char *Path, const char *Mode);
int fclose(FILE *Stream);
int fscanf(FILE *Stream, const char *Format, ...);
int fprintf(FILE *Stream, const char *Format, ...);
```

These commands are for reading from and writing to files. Say you've got a file named `inputfile`, sitting in the same directory as the program, with information your program needs. The following code would read from it and then write to the file `outputfile`.

```
int i;
double x;
FILE *input, *output;
input = fopen("inputfile","r");    // "r" means you will read from this file
output = fopen("outputfile","w");  // "w" means you will write to this file
fscanf(input,"%d %lf",&i,&x);
fprintf(output,"I read in an integer %d and a double %lf.\n",i,x);
fclose(input);                     // these streams should be closed ...
fclose(output);                    // ... at the end of the program
```

```
int fputc(int character, FILE *stream);
int fputs(const char *str, FILE *stream);
int fgetc(FILE *stream);
char* fgets(char *str, int num, FILE *stream);
int puts(const char *str);
char* gets(char *str);
```

These commands get a character or string from a file, write (put) a character or string to a file, put a string to the screen, or get a string from the keyboard.

String Manipulation: `string.h`

```
char* strcpy(char *destination, const char *source);
```

Given two strings, `char destination[100],source[100]`, we cannot simply copy one to the other using the assignment `destination = source`. Instead we use `strcpy(destination,source)`, which copies the string `source` (until reaching the string terminator character, integer value 0) to `destination`. The string `destination` must have enough memory allocated to hold the source string.

```
char* strcat(char *destination, const char *source);
```

Appends the string in `source` to the end of the string `destination`.

```
int strcmp(const char *s1, const char *s2);
```

Returns 0 if the two strings are identical, a positive integer if the first unequal character in `s1` is greater than `s2`, and a negative integer if the first unequal character in `s1` is less than `s2`.

```
size_t strlen(const char *s);
```

The type `size_t` is an unsigned integer type. `strlen` returns the length of the string `s`, where the end of

the string is indicated by the string terminator character (value 0).

```
void* memset(void *s, int c, size_t len);
```

`memset` writes `len` bytes of the value `c` (converted to an `unsigned char`) starting at the beginning of the string `s`. So

```
char s[10];
memset(s,'c',5);
```

would fill the first five characters of the string `s` with the character `'c'` (or integer value 99). This can be a convenient way to initialize a string.

General Purpose Functions in `stdlib.h`

```
void* malloc(size_t objectSize)
```

`malloc` is used for dynamic memory allocation. An example use is in Section A.4.9.

```
void free(void *objptr)
```

`free` is used to release memory allocated by `malloc`. An example use is in Section A.4.9.

```
int rand()
```

It is sometimes useful to generate random numbers, particularly for games. The code

```
int i;
i = rand();
```

places in `i` a pseudo-random number between 0 and `RAND_MAX`, a constant which is defined in `stdlib.h` (2,147,483,647 on my laptop). To convert this to an integer between 1 and 10, you could follow with

```
i = 1 + (int) ((10.0*i)/(RAND_MAX+1.0));
```

One drawback of the code above is that calling `rand` multiple times will lead to the same sequence of random numbers every time the program is run. The usual solution is to "seed" the random number algorithm with a different number each time, and this different number is often taken from a system clock. The `srand` function is used to seed `rand`, as in the example below:

```
#include <stdio.h>   // allows use of printf()
#include <stdlib.h>  // allows use of rand() and srand()
#include <time.h>    // allows use of time()

int main(void) {
  int i;
  srand(time(NULL)); // seed the random number generator with the current time
  for (i=0; i<10; i++) printf("Random number:  %d\n",rand());
  return 0;
}
```

If we take out the line with `srand`, this program produces the same ten "random" numbers every time we run it. Note that this program includes the `time.h` library to allow the use of the `time` function.

```
void exit(int status)
```

When `exit` is invoked, the program exits with the exit code `status`. `stdlib.h` defines `EXIT_SUCCESS` with value 0 and `EXIT_FAILURE` with value -1, so that a typical call to `exit` might look like

```
exit(EXIT_SUCCESS);
```

A.4.16 Multiple File Programs and Libraries

Our programs have been making use of the stdio library, which provides a number of functions allowing printing to the screen and reading input from the keyboard. We gain access to these functions because of two things:

1. The preprocessor command `#include <stdio.h>` inserts the header file `stdio.h` consisting of function prototypes that declare functions such as `printf`, allowing you to use `printf` in your program.

2. The linker links in the pre-compiled library object code for stdio functions like `printf`.

Thus a library consists of a header file and object code. We could also loosely define a library to consist of a header file and a C file (without a `main` function) containing source code for the library functions.

The purpose of a library is to gather together functions that are likely to be useful in many programs, so you don't have to rewrite the code for each program. Let's look at a simple example.

A Simple Example: the rad2volume Library

Say you plan to write a number of programs that all need a function to calculate the volume of a sphere given its radius. Instead of putting the same function into a bunch of different C files, you decide to write one helper C file, `rad2volume.c`, with a function `double radius2Volume(double r)` that you make available to other C files. For good measure, you decide to make the constant `MY_PI` available also. To test your new rad2volume library consisting of `rad2volume.c` and `rad2volume.h`, you create a `main.c` file that uses it. The three files are given below.

```
// ***** file:  rad2volume.h *****
#ifndef RAD2VOLUME_H                // "include guard"; don't include twice in one compilation
#define RAD2VOLUME_H                // second line of the "include guard"

#define MY_PI 3.1415926             // constant available to files including rad2Volume.h
double radius2Volume(double r);     // prototype available to files including rad2Volume.h

#endif                              // third line, and end, of "include guard"
```

```
// ***** file:  rad2volume.c *****
#include <math.h>                   // for the function pow
#include "rad2volume.h"             // if the header is in the same directory, use "quotes"

double cuber(double x) {            // this function is not available externally
  return(pow(x,3.0));
}

double radius2Volume(double rad) {  // function definition
  return((4.0/3.0)*MY_PI*cuber(rad));
}
```

```
// ***** file:  main.c *****
#include <stdio.h>
#include "rad2volume.h"

int main(void) {
  double radius = 3.0, volume;
  volume = radius2Volume(radius);
  printf("Pi is approximated as %25.23lf.\n",MY_PI);
  printf("The volume of the sphere is %8.4lf.\n",volume);
  return 0;
}
```

The C file `rad2volume.c` contains two functions, `cuber` and `radius2Volume`. The function `cuber` is only meant for internal, private use by `rad2volume.c`, so there is no prototype in `rad2volume.h`. On the other hand, the function `radius2Volume` is meant for public use by other C files, so a prototype for `radius2Volume` is included in the library header file `rad2volume.h`. The constant `MY_PI` is also meant for public use, so it is defined in `rad2volume.h`. Now `radius2Volume` and `MY_PI` are available to any file that includes `rad2volume.h`. In this case, they are available to `main.c` and `rad2volume.c`.

Each of `main.c` and `rad2volume.c` is compiled independently to create the object codes `main.o` and `rad2volume.o`. These object codes are then linked to create the final executable. `main.c` compiles successfully because it expects that, during the linking stage, `MY_PI` and `rad2Volume` will be properly linked (in this case, to `main.o`).

Note the three lines making up the *include guard* in `rad2volume.h`. During preprocessing of a C file, if `rad2volume.h` is included, the flag `RAD2VOLUME_H` is defined. If the same C file tries to include `rad2volume.h` again, the include guard will recognize that `RAD2VOLUME_H` already exists and therefore skip the prototype and constant definition, down to the `#endif`. Without include guards, if we wrote a `.c` file including both `header1.h` and `header2.h`, for example, not knowing that `header2.h` already includes `header1.h`, we would get a compilation error due to duplicate declarations.

The two C files can be compiled into object codes using the commands

```
gcc -c rad2volume.c -o rad2volume.o
gcc -c main.c -o main.o
```

where the `-c` flag indicates that the code should be compiled and assembled, but not linked. The result is the object codes `rad2volume.o` and `main.o`. The two object codes can be linked into a final executable using

```
gcc rad2volume.o main.o -o myprog
```

Alternatively, the single command

```
gcc rad2volume.c main.c -o myprog
```

could be used in place of the preceding three lines to compile and link without writing the object files. Executing `myprog`, the output is

```
Pi is approximated as 3.14159260000000006840537.
The volume of the sphere is 113.0973.
```

Generalizing

Generalizing the simple example above, a header file defines constants, macros, new data types, and function prototypes that are needed by the files that `#include` them. A header file can be included by C source files or other header files. Figure A.1 illustrates a project consisting of one C source file with a `main` function and two helper C source files without a `main` function. (Every C project has exactly one `.c` file with a `main` function.) Each of the helper C files has its own header file, and a "library," in our simplified context, is a C helper file together with its header file. This project also has one other header file, `general.h`, without an associated C file. This header is for general constant, macro, and data type definitions that are not specific to either library nor the `main` C file. The arrows indicate that the pointed-to file includes the pointed-from header file.

Assuming all the files are in the same directory, the project in Figure A.1 can be built by the following four command-line commands, which create three object files (one for each source file) and link them together into `myprog`:

```
gcc -c main.c -o main.o
gcc -c helper1.c -o helper1.o
gcc -c helper2.c -o helper2.o
gcc main.o helper1.o helper2.o -o myprog
```

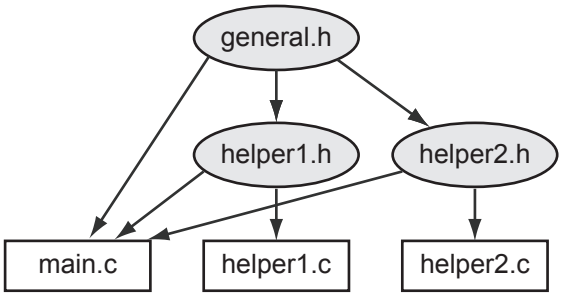


Figure A.1: An example project consisting of three C files and three header files. Arrows point from header files to files that include them.

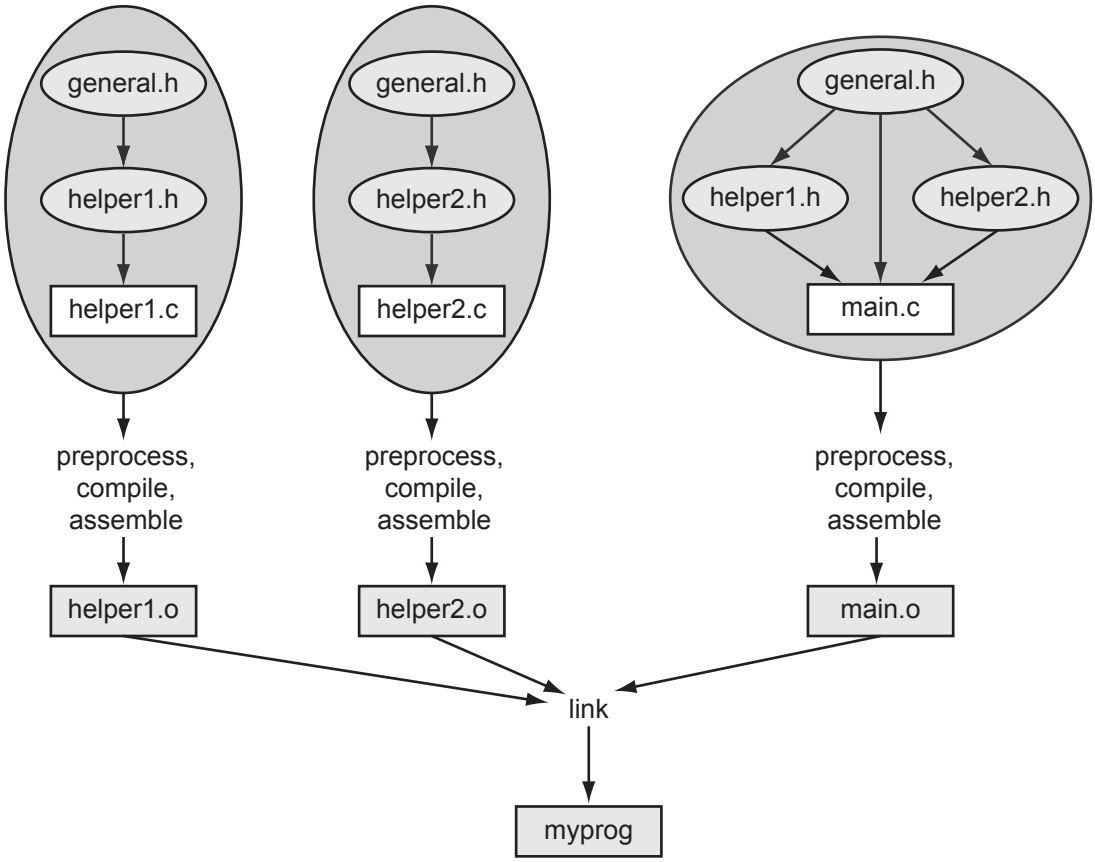


Figure A.2: The building of the project in Figure A.1.

The build is illustrated in Figure A.2. Each C file is compiled independently and requires the constants, macros, data types, and function prototypes needed to successfully compile and assemble into an object file. During compilation of a single C file, the compiler does not have (nor need) access to the source code for functions in other C files. If `main.c` uses a function in `helper1.c`, for example, it needs only a prototype of the function, provided by `helper1.h`, so the compiler knows the type of the function and what arguments it takes. Calls to the function from `main.o` are linked to the actual function in `helper1.o` at the linker stage.

Most libraries, once finalized, should not be changed, so it is usually not necessary to re-create `helper1.o` from `helper1.c`. Instead, your project can just use the object code `helper1.o` at the linking stage.

According to Figures A.1 and A.2, `main.c` has the following preprocessor commands:

```
#include "general.h"    // use "quotes" for header files that live in the same directory
#include "helper1.h"
#include "helper2.h"
```

The preprocessor replaces these commands with copies of the files `general.h`, `helper1.h`, and `helper2.h`. But when it includes `helper1.h`, it finds that `helper1.h` tries to include a second copy of `general.h` (see Figure A.1; `helper1.h` has a `#include "general.h"` command). Since `general.h` has already been copied in, it should not be copied again; otherwise we would have multiple copies of the same function prototypes, constant definitions, etc. To prevent this kind of error, header files should have include guards, as in our simple example above.

In summary, the `general.h`, `helper1.h`, and `helper2.h` header files contain definitions that are made public to files including them. We might see the following items in the `helper1.h` header file, for example:

- an include guard
- other include files
- constants and macros made public (and which may also be used by `helper1.c`)
- new data types (which may also be used by `helper1.c`)
- function prototypes of those functions in `helper1.c` which are meant to be used by other files
- possibly global variables that are *defined* (space allocated) in `helper1.c` but made available to other files by an `extern` *declaration* in `helper1.h`

If a variable, function prototype, or constant is private to one C file, you can just define and use it in that C file without including it in a header file.

A header file like `helper1.h` could also have the declaration

```
extern int Helper1_Global_Var;   // no space is allocated by this declaration
```

where `helper1.c` has the global variable definition

```
int Helper1_Global_Var;          // space is allocated by this definition
```

Then any file including `helper1.h` would have access to the global variable `Helper1_Global_Var` allocated by `helper1.c`. Global variables defined in `helper1.c` that do not have `extern` declarations in `helper1.h` are private to `helper1.c` and cannot be accessed by other files. Global variables should not be defined (space allocated) in a header file that could be included by more than one C file in a project.

Makefiles

When you are ready to build your executable, you can type the `gcc` commands at the command line, as we have seen previously. A *makefile* simplifies the process, particularly for multi-file projects, by specifying the dependencies and commands needed to build the project. A makefile for our rad2volume example is shown below, where everything after a `#` is a comment.

```
# ***** file:  makefile *****
# Comment:  This is the simplest of makefiles!

# Here is a template:
# [target]:  [dependencies]
# [tab] [command to execute]

# The thing to the left of the colon in the first line is what is created,
# and the thing(s) to the right of the colon are what it depends on.  The second
# line is the action to create the target.  If the things it depends on
# haven't changed since the target was last created, no need to do the action.
# Note:  The tab spacing in the second line is important!  You can't just use
```

```
# individual spaces.

# "make myprog" or "make" links to create the executable
myprog:  main.o rad2volume.o
        gcc main.o rad2volume.o -o myprog

# "make main.o" produces main.o object code
main.o:  main.c rad2volume.h
        gcc -c main.c -o main.o

# "make rad2volume.o" produces rad2volume.o object code
rad2volume.o:  rad2volume.c rad2volume.h
        gcc -c rad2volume -o rad2volume.o

# "make clean" throws away any object files to ensure make from scratch
clean:
        rm *.o
```

With this `makefile` in the same directory as your other files, you should be able to type the command `make [target]`[16], where `[target]` is `myprog`, `main.o`, `rad2volume.o`, or `clean`. If the `target` depends on other files, `make` will make sure those are up to date first, and if not, it will call the commands needed to make them. For example, `make myprog` triggers a check of `main.o`, which triggers a check of `main.c` and `rad2volume.h`. If either of those have changed since the last time `main.o` was made, then `main.c` is compiled and assembled to create a new `main.o` before the linking step.

The command `make` with no target specified will make the first target (which is `myprog` in this case).

Make sure your `makefile` is saved without any extensions (e.g., `.txt`) and that the commands are preceded by a tab (not spaces).

There are many more sophisticated uses of makefiles which you can learn about from other sources.

[16] In some C installations `make` is named differently, like `nmake` for Visual Studio or `mingw32-make`. If you can find no version of `make`, you may not have selected the `make` tools installation option when you performed the C installation.

A.5 Exercises

1. Install C, create the `HelloWorld.c` program, and compile and run it.

2. Explain what a pointer variable is, and how it is different from a non-pointer variable.

3. Explain the difference between interpreted and compiled code.

4. Write the following hexadecimal (base-16) numbers in eight-digit binary (base-2) and three-digit decimal (base-10). Also, for each of the eight-digit binary representations, give the value of the most significant bit. (a) 0x1E. (b) 0x32. (c) 0xFE. (d) 0xC4.

5. What is 333_{10} in binary and 1011110111_2 in hexadecimal? What is the maximum value, in decimal, that a 12-bit number can hold?

6. Assume that each byte of memory can be addressed by a 16-bit address, and every 16-bit address has a corresponding byte in memory. How many total bits of memory do you have?

7. (Consult an ASCII table.) Let `ch` be of type `char`. (a) The assignment `ch = 'k'` can be written equivalently using a number on the right side. What is that number? (b) The number for `'5'`? (c) For `'='`? (d) For `'?'`?

8. What is the range of values for an `unsigned char`, `short`, and `double` data type?

9. How many bits are used to store a `char`, `short`, `int`, `float`, and `double`?

10. Explain the difference between `unsigned` and `signed` integers.

11. (a) For integer math, give the pros and cons of using `char`s vs. `int`s. (b) For floating point math, give the pros and cons of using `float`s vs. `double`s. (c) For integer math, give the pros and cons of using `char`s vs. `float`s.

12. The following `signed short int`s, written in decimal, are stored in two bytes of memory using the two's complement representation. For each, give the four-digit hexadecimal representation of those two bytes. (a) 13. (b) 27. (c) -10. (d) -17.

13. The smallest positive integer that cannot be exactly represented by a four-byte IEEE 754 `float` is $2^{24} + 1$, or 16,777,217. Explain why.

14. Technically the data type of a pointer to a `double` is "pointer to type `double`." Of the common integer and floating point data types discussed in this chapter, which is the most similar to this pointer type? Assume pointers occupy eight bytes.

15. To keep things simple, let's assume we have a microcontroller with only $2^8 = 256$ bytes of RAM, so each address is given by a single byte. Now consider the following code defining four local variables:

    ```
    unsigned int i, j, *kp, *np;
    ```

    Let's assume that the linker places `i` in addresses 0xB0..0xB3, `j` in 0xB4..0xB7, `kp` in 0xB8, and `np` in 0xB9. The code continues as follows:

    ```
                // (a) the initial conditions, all memory contents unknown
    kp = &i;    // (b)
    j = *kp;    // (c)
    i = 0xAE;   // (d)
    np = kp;    // (e)
    *np = 0x12; // (f)
    j = *kp;    // (g)
    ```

For each of the comments (a)-(g) above, give the contents (in hexadecimal) at the address ranges 0xB0..0xB3 (the `unsigned int i`), 0xB4..0xB7 (the `unsigned int j`), 0xB8 (the pointer `kp`), and 0xB9 (the pointer `np`), at that point in the program, after executing the line containing the comment. The contents of all memory addresses are initially unknown or random, so your answer to (a) is "unknown" for all memory locations. If it matters, assume little-endian representation.

16. Invoking the `gcc` compiler with a command like `gcc myprog.c -o myprog` actually initiates four steps. What are the four steps called, and what is the output of each step?

17. What is `main`'s data type, and what is the meaning of its return value?

18. Give the `printf` statement that will print out a `double d` with eight digits to the right of the decimal point and four spaces to the left.

19. Consider three `unsigned char`s, `i`, `j`, and `k`, with values 60, 80, and 200, respectively. Let `sum` also be an `unsigned char`. For each of the following, give the value of `sum` after performing the addition. (a) `sum = i+j;` (b) `sum = i+k;` (c) `sum = j+k;`

20. For the variables defined as

```
int a=2, b=3, c;
float d=1.0, e=3.5, f;
```

give the values of the following expressions. (a) `f = a/b;` (b) `f = ((float) a)/b;` (c) `f = (float) (a/b);` (d) `c = e/d`; (e) `c = (int) (e/d);` (f) `f = ((int) e)/d;`

21. In each snippet of code in (a)-(d), there is an arithmetic error in the final assignment of `ans`. What is the final value of `ans` in each case?

(a)
```
char c = 17;
float ans = (1 / 2) * c;
```

(b)
```
unsigned int ans = -4294967295;
```

(c)
```
double d = pow(2, 16);
short ans = (short) d;
```

(d)
```
double ans = ((double) -15 * 7) / (16 / 17) + 2.0;
```

22. Truncation isn't always bad. Say you wanted to store a list of percentages rounded down to the nearest percent, but you were tight on memory and cleverly used an array of `char`s to store the values. For example, pretend you already had the following snippet of code:

```
char percent(int a, int b) {
  // assume a <= b
  char c;
  c = ???;
  return c;
}
```

You can't simply write `c = a / b`. If $\frac{a}{b} = 0.77426$ or $\frac{a}{b} = 0.778$, then the correct return value is `c` = 77. Finish the function definition by writing a one-line statement to replace `c = ???`.

23. Explain why global variables work against modularity.

24. What are the seven sections of a typical C program?

25. You've written a large program with a number of functions. Your program compiles without errors, but when you run the program with input for which you know the correct output, you discover that your program returns a wrong result. What do you do next? Describe your systematic strategy for debugging.

26. Erase all the comments in `invest.c`, recompile, and run the program to make sure it still functions correctly. You should be able to recognize what is a comment and what is not. Turn in your modified `invest.c` code.

27. The following problems refer to the program `invest.c`. For all problems, you should modify the original code (or the code without comments from the previous problem) and run it to make sure you get the expected behavior. For each problem, turn in the modified portion of the code only.

    (a) *Using* `if`, `break` *and* `exit`. Include the header file `stdlib.h` so we have access to the `exit` function (Section A.4.15). Change the `while` loop in `main` to be an infinite loop by inserting an expression `<expr>` in `while(<expr>)` that always evaluates to 1 (TRUE). (What is the simplest expression that evaluates to 1?) Now the first command inside the while loop gets the user's input. `if` the input is not valid, `exit` the program; otherwise continue. Next, change the `exit` command to a `break` command, and see the different behavior.

    (b) *Accessing fields of a* `struct`. Alter `main` and `getUserInput` to set `inv.invarray[0]` in `getUserInput`, not `main`.

    (c) *Using* `printf`. In `main`, before `sendOutput`, echo the user's input to the screen. For example, the program could print out `You entered 100.00, 1.05, and 5`.

    (d) *Altering a string.* After the `sprintf` command of `sendOutput`, try setting an element of `outstring` to 0 before the `printf` command. For example, try setting the third element of `outstring` to 0. What happens to the output when you run the program? Now try setting it to `'0'` instead and see the behavior.

    (e) *Relational operators.* In `calculateGrowth`, eliminate the use of `<=` in favor of an equivalent expression that uses `!=`.

    (f) *Math.* In `calculateGrowth`, replace `i=i+1` with an equivalent statement using `+=`.

    (g) *Data types.* Change the fields `inv0`, `growth`, and `invarray[]` to be `float` instead of `double` in the definition of the `Investment` data type. Make sure you make the correct changes everywhere else in the program.

    (h) *Pointers.* Change `sendOutput` so that the second argument is of type `int *`, i.e., a pointer to an integer, instead of an integer. Make sure you make the correct changes everywhere else in the program.

    (i) *Conditional statements.* Use an `else` statement in `getUserInput` to print `Input is valid` if the input is valid.

    (j) *Loops.* Change the `for` loop in `sendOutput` to an equivalent `while` loop.

    (k) *Logical operators.* Change the assignment of `valid` to an equivalent statement using `||` and `!`, and no `&&`.

28. Consider this array definition and initialization:

    ```
    int x[4] = {4, 3, 2, 1};
    ```

    For each of the following, give the value or write "error/unknown" if the compiler will generate an error or the value is unknown. (a) `x[1]` (b) `*x` (c) `*(x+2)` (d) `(*x)+2` (e) `*x[3]` (f) `x[4]` (g) `*(&(x[1]) + 1)`

29. For the (strange) code below, what is the final value of `i`? Explain why.

    ```
    int i,k=6;
    i = 3*(5>1) + (k=2) + (k==6);
    ```

30. As the code below is executed, give the value of `c` in hex at the seven break points indicated, (a)-(g).

```
unsigned char a=0x0D, b=0x03, c;
c = ~a;     // (a)
c = a & b;  // (b)
c = a | b;  // (c)
c = a ^ b;  // (d)
c = a >> 3; // (e)
c = a << 3; // (f)
c &= b;     // (g)
```

31. In your C installation, or by searching on the web, find a listing of the header file `stdio.h`. Find the function prototype for one function provided by the library, but not mentioned in this appendix, and describe what that function does.

32. Write a program to generate the ASCII table for values 33 to 127. The output should be two columns: the left side with the number and the right side with the corresponding character. Turn in your code and the output of the program.

33. We will write a simple *bubble sort* program to sort a string of text in ascending order according to the ASCII table values of the characters. A bubble sort works as follows. Given an array of n elements with indexes 0 to $n-1$, we start by comparing elements 0 and 1. If element 0 is greater than element 1, we swap them. If not, leave them where they are. Then we move on to elements 1 and 2 and do the same thing, etc., until finally we compare elements $n-2$ and $n-1$. After this, the largest value in the array has "bubbled" to the last position. We now go back and do the whole thing again, but this time only comparing elements 0 up to $n-2$. The next time, elements 0 to $n-3$, etc., until the last time through we only compare elements 0 and 1.

    Although this simple program `bubble.c` could be written in one function (`main`), we are going to break it into some helper functions to get used to using them. The function `getString` will get the input from the user; the function `printResult` will print the sorted result; the function `greaterThan` will check if one element is greater than another; and the function `swap` will swap two elements in the array. With these choices, we start with an outline of the program that looks like this.

```
#include <stdio.h>
#include <string.h>
#define MAXLENGTH 100       // max length of string input

void getString(char *str);  // helper prototypes
void printResult(char *str);
int greaterThan(char ch1, char ch2);
void swap(char *str, int index1, int index2);

int main(void) {
  int len;                  // length of the entered string
  char str[MAXLENGTH];      // input should be no longer than MAXLENGTH
  // here, any other variables you need

  getString(str);
  len = strlen(str);        // get length of the string, from string.h
  // put nested loops here to put the string in sorted order
  printResult(str);
  return(0);
}

// helper functions go here
```

    Here's an example of the program running. Everything after the first colon is entered by the user. Blank spaces are written using an underscore character, since `scanf` assumes that the string ends at the first whitespace.

```
Enter the string you would like to sort:  This_is_a_cool_program!
Here is the sorted string:  !T____aacghiilmoooprrss
```

Complete the following steps in order. Do not move to the next step until the current step is successful.

(a) Write the helper function `getString` to ask the user for a string and place it in the array passed to `getString`. You can use `scanf` to read in the string. Write a simple call in `main` to verify that `getString` works as you expect before moving on.

(b) Write the helper function `printResult` and verify that it works correctly.

(c) Write the helper function `greaterThan` and verify that it works correctly.

(d) Write the helper function `swap` and verify that it works correctly.

(e) Now define the other variables you need in `main` and write the nested loops to perform the sort. Verify that the whole program works as it should.

Turn in your final documented code and an example of the output of the program.

34. A more useful sorting program would take a series of names (e.g., `Doe_John`) and scores associated with them (e.g., 98) and then list the names and scores in two columns in descending order. Modify your bubble sort program to do this. The user enters a name string and a number at each prompt. The user indicates that there are no more names by entering `0 0`.

    Your program should define a constant `MAXRECORDS` which contains the maximum number of records allowable. You should define an array, `MAXRECORDS` long, of `struct` variables, where each `struct` has two fields: the name string and the score. Write your program modularly so that there is at least a `sort` function and a `readInput` function of type `int` that returns the number of records entered.

    Turn in your code and example output.

35. Modify the previous program to read the data in from a file using `fscanf` and write the results out to another file using `fprintf`. Turn in your code and example output.

36. Consider the following lines of code:

```
int i, tmp, *ptr, arr[7] = {10, 20, 30, 40, 50, 60, 70};

ptr = &arr[6];
for(i = 0; i < 4; i++) {
  tmp = arr[i];
  arr[i] = *ptr;
  *ptr = tmp;
  ptr--;
}
```

    (a) How many elements does the array `arr` have?

    (b) How would you access the middle element of `arr` and assign its value to the variable `tmp`? Do this two ways, once indexing into the array using `[]` and the other with the dereferencing operator and some pointer arithmetic. Your answer should only be in terms of the variables `arr` and `tmp`.

    (c) What are the contents of the array `arr` before and after the loop?

37. The following questions pertain to the code below. For your responses, you only need to write down the changes you would make using valid C code. You should verify that your modifications actually compile and run correctly. Do not submit a full C program for this question. Only write the changes you would make using legitimate C syntax.

```
#include <stdio.h>
#define MAX 10

void MyFcn(int max);

int main(void) {
  MyFcn(5);
  return(0);
}

void MyFcn(int max) {
  int i;
  double arr[MAX];

  if(max > MAX) {
    printf("The range requested is too large.  Max is %d.\n", MAX);
    return;
  }
  for(i = 0; i < max; i++) {
    arr[i] = 0.5 * i;
    printf("The value of i is %d and %d/2 is %f.\n", i, i, arr[i]);
  }
}
```

(a) `while` loops and `for` loops are essentially the same thing. How would you write an equivalent `while` loop that replicates the behavior of the `for` loop?

(b) How would you modify the `main` function so that it reads in an integer value from the keyboard and then passes the result to `MyFcn`? (This replaces the statement `MyFcn(5);`.) If you need to use extra variables, make sure to define them before you use them in your snippet of code.

(c) Change `main` so that if the input value from the keyboard is between $-$`MAX` and `MAX`, you call `MyFcn` with the absolute value of the input. If the input is outside this range, then you simply call `MyFcn` with the value `MAX`. How would you make these changes using conditional statements?

(d) In C, you will often find yourself writing nested loops (a loop inside a loop) to accomplish a task. Modify the `for` loop to use nested loops to set the i^{th} element in the array `arr` to half the sum of the first $i-1$ integers, i.e., `arr[i]` $= \frac{1}{2}\sum_{j=0}^{i-1} j$. (You can easily find a formula for this that doesn't require the inner loop, but you should use nested loops for this problem.) The same loops should print the value of each `arr[i]` to 2 decimal places using the `%f` formatting directive.

38. If there are n people in a room, what is the chance that two of them have the same birthday? If $n = 1$, the chance is zero, of course. If $n > 366$, the chance is 100%. Under the assumption that births are distributed uniformly over the days of the year, write a program that calculates the chances for values of $n = 2$ to 100. What is the lowest value n^* such that the chance is greater than 50%? (The surprising result is sometimes called the "birthday paradox.") If the distribution of births on days of the year is not uniform, will n^* increase or decrease? Turn in your answer to the questions as well as your C code and the output.

39. In this problem you will write a C program that solves a "puzzler" that was presented on NPR's CarTalk radio program. In a direct quote of their radio transcript, found here http://www.cartalk.com/content/hall-lights?question, the problem is described as follows:

> **RAY**: This puzzler is from my "ceiling light" series. Imagine, if you will, that you have a long, long corridor that stretches out as far as the eye can see. In that corridor, attached to the ceiling are lights that are operated with a pull cord.
>
> There are gazillions of them, as far as the eye can see. Let's say there are 20,000 lights in a row.

> They're all off. Somebody comes along and pulls on each of the chains, turning on each one of the lights. Another person comes right behind, and pulls the chain on every second light.
>
> **TOM**: Thereby turning off lights 2, 4, 6, 8 and so on.
>
> **RAY**: Right. Now, a third person comes along and pulls the cord on every third light. That is, lights number 3, 6, 9, 12, 15, etc. Another person comes along and pulls the cord on lights number 4, 8, 12, 16 and so on. Of course, each person is turning on some lights and turning other lights off.
>
> If there are 20,000 lights, at some point someone is going to come skipping along and pull every 20,000th chain.
>
> When that happens, some lights will be on, and some will be off. Can you predict which lights will be on?

You will write a C program that asks the user the number of lights n and then prints out which of the lights are on, and the total number of lights on, after the last (nth) person goes by. Here's an example of what the output might look like if the user enters 200:

```
How many lights are there? 200

You said 200 lights.
Here are the results:
  Light number   1 is on.
  Light number   4 is on.
  ...
  Light number 196 is on.
  There are 14 total lights on!
```

Your program `lights.c` should follow the template outlined below. Turn in your code and example output.

```
/*************************************************************************
 * lights.c
 *
 * This program solves the light puzzler.  It uses one main function
 * and two helper functions:  one that calculates which lights are on,
 * and one that prints the results.
 *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>         // allows the use of the "exit()" function
#define MAX_LIGHTS 1000000  // maximum number of lights allowed

// here's a prototype for the light toggling function
// here's a prototype for the results printing function

int main(void) {

  // Define any variables you need, including for the lights' states

  // Get the user's input.
  // If it is not valid, say so and use "exit()" (stdlib.h, Sec 1.2.16).
  // If it is valid, echo the entry to the user.

  // Call the function that toggles the lights.
  // Call the function that prints the results.
```

```
  return(0);
}

// definition of the light toggling function
// definition of the results printing function
```

40. We have been preprocessing, compiling, assembling, and linking programs with commands like

```
gcc HelloWorld.c -o HelloWorld
```

The `gcc` command recognizes the first argument, `HelloWorld.c`, is a C file based on its `.c` extension. It knows you want to create an output file called `HelloWorld` because of the `-o` option. And since you didn't specify any other options, it knows you want that output to be an executable. So it performs all four of the steps to take the C file to an executable.

We could have used options to stop after each step if we wanted to see the intermediate files produced. Below is a sequence of commands you could try, starting with your `HelloWorld.c` code. Don't type the "comments" to the right of the commands!

```
> gcc HelloWorld.c -E > HW.i  // stop after preprocessing, dump into file HW.i
> gcc HW.i -S -o HW.s         // compile HW.i to assembly file HW.s and stop
> gcc HW.s -c -o HW.o         // assemble HW.s to object code HW.o and stop
> gcc HW.o -o HW              // link with stdio printf code, make executable HW
```

At the end of this process you have `HW.i`, the C code after preprocessing (`.i` is a standard extension for C code that should not be preprocessed); `HW.s`, the assembly code corresponding to `HelloWorld.c`; `HW.o`, the unreadable object code; and finally the executable code `HW`. The executable is created from linking your `HW.o` object code with object code from the `stdio` (standard input and output) library, specifically object code for `printf`.

Try this and verify that you see all the intermediate files, and that the final executable works as expected.

If our program used any math functions, the final linker command would be

```
> gcc HW.o -o HW -lm          // link with stdio and math libraries, make executable HW
```

Most libraries, like `stdio`, are linked automatically, but often the math library is not, requiring the extra `-lm` option.

The `HW.i` and `HW.s` files can be inspected with a text editor, but the object code `HW.o` and executable `HW` cannot. We can try the following commands to make viewable versions:

```
> xxd HW.o v1.txt             // can't read obj code; this makes viewable v1.txt
> xxd HW v2.txt               // can't read executable; make viewable v2.txt
```

The utility `xxd` just turns the first file's string of 0's and 1's into a string of hex characters, represented as text-editor-readable ASCII characters 0..9, A..F. It also has an ASCII sidebar: when a byte (two consecutive hex characters) has a value corresponding to a printable ASCII character, that character is printed. You can even see your message "Hello world!" buried there!

Take a quick look at the `HW.i`, `HW.s`, and `v1.txt` and `v2.txt` files. No need to understand these intermediate files any further. If you don't have the `xxd` utility, you could create your own program `hexdump.c` instead:

```
#include <stdio.h>
#define BYTES_PER_LINE 16

int main(void) {
  FILE *inputp, *outputp;                          // ptrs to in and out files
  int c, count = 0;
  char asc[BYTES_PER_LINE+1], infile[100];

  printf("What binary file do you want the hex rep of? ");
  scanf("%s",infile);                              // get name of input file
  inputp = fopen(infile,"r");                      // open file as "read"
  outputp = fopen("hexdump.txt","w");              // output file is "write"

  asc[BYTES_PER_LINE] = 0;                         // last char is end-string
  while ((c=fgetc(inputp)) != EOF) {               // get byte; end of file?
    fprintf(outputp,"%x%x ",(c >> 4),(c & 0xf));   // print hex rep of byte
    if ((c>=32) && (c<=126)) asc[count] = c;       // put printable chars in asc
    else asc[count] = '.';                         // otherwise put a dot
    count++;
    if (count==BYTES_PER_LINE) {                   // if BYTES_PER_LINE reached
      fprintf(outputp,"  %s\n",asc);               // print ASCII rep, newline
      count = 0;
    }
  }
  if (count!=0) {                                  // print last (short) line
    for (c=0; c<BYTES_PER_LINE-count; c++)         // print extra spaces
      fprintf(outputp,"   ");
    asc[count]=0;                                  // add end-string char to asc
    fprintf(outputp,"  %s\n",asc);                 // print ASCII rep, newline
  }
  fclose(inputp);                                  // close files
  fclose(outputp);
  printf("Printed hexdump.txt.\n");
  return(0);
}
```