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.

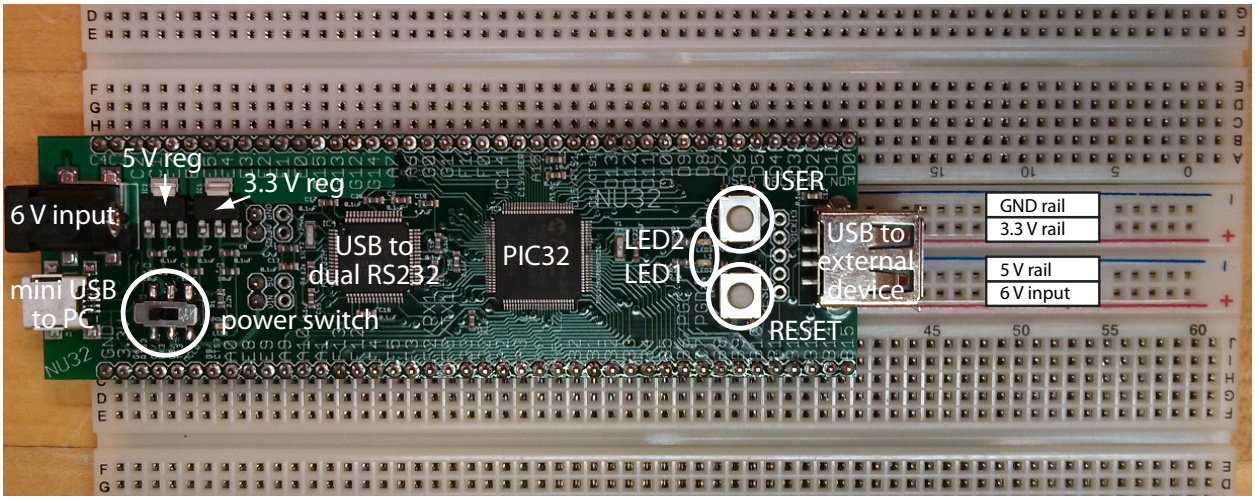


Figure 1.1: A photo of the NU32 development board.

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
  while (1) {
    NU32_ReadUART1(message, MAX_MESSAGE_LENGTH);  // get message from computer
    NU32_WriteUART1(message);                     // send message back
    NU32_WriteUART1("\r\n");                      // carriage return and newline
    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. 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, if you type `Hello! ENTER`, the PIC32 will receive `Hello!\r`, write `Hello!\r\n` to the terminal emulator, and wait 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.

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.

[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`.

- 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.

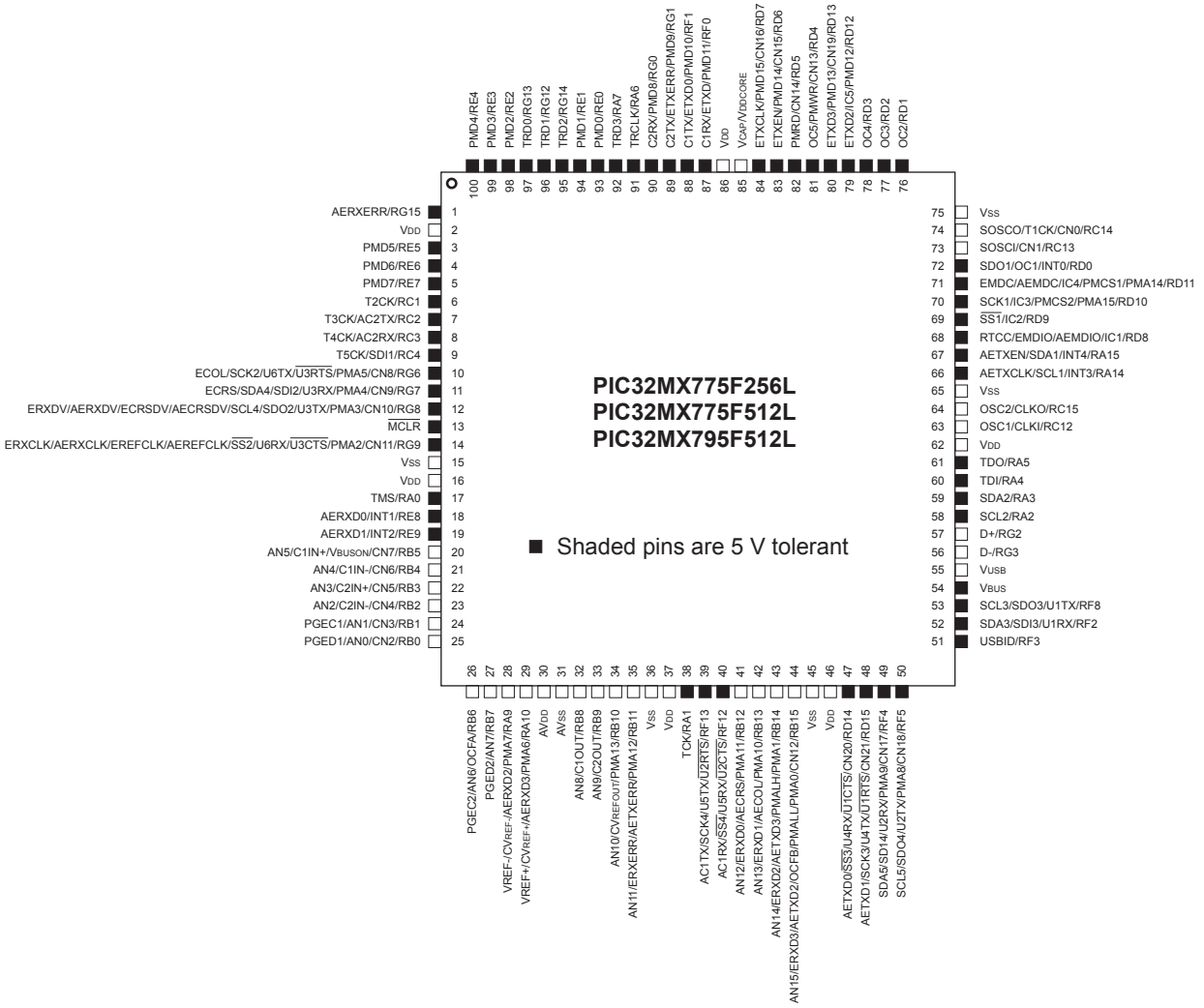


Figure 2.1: The pinout of the PIC32MX795F512L used on the NU32.

(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.

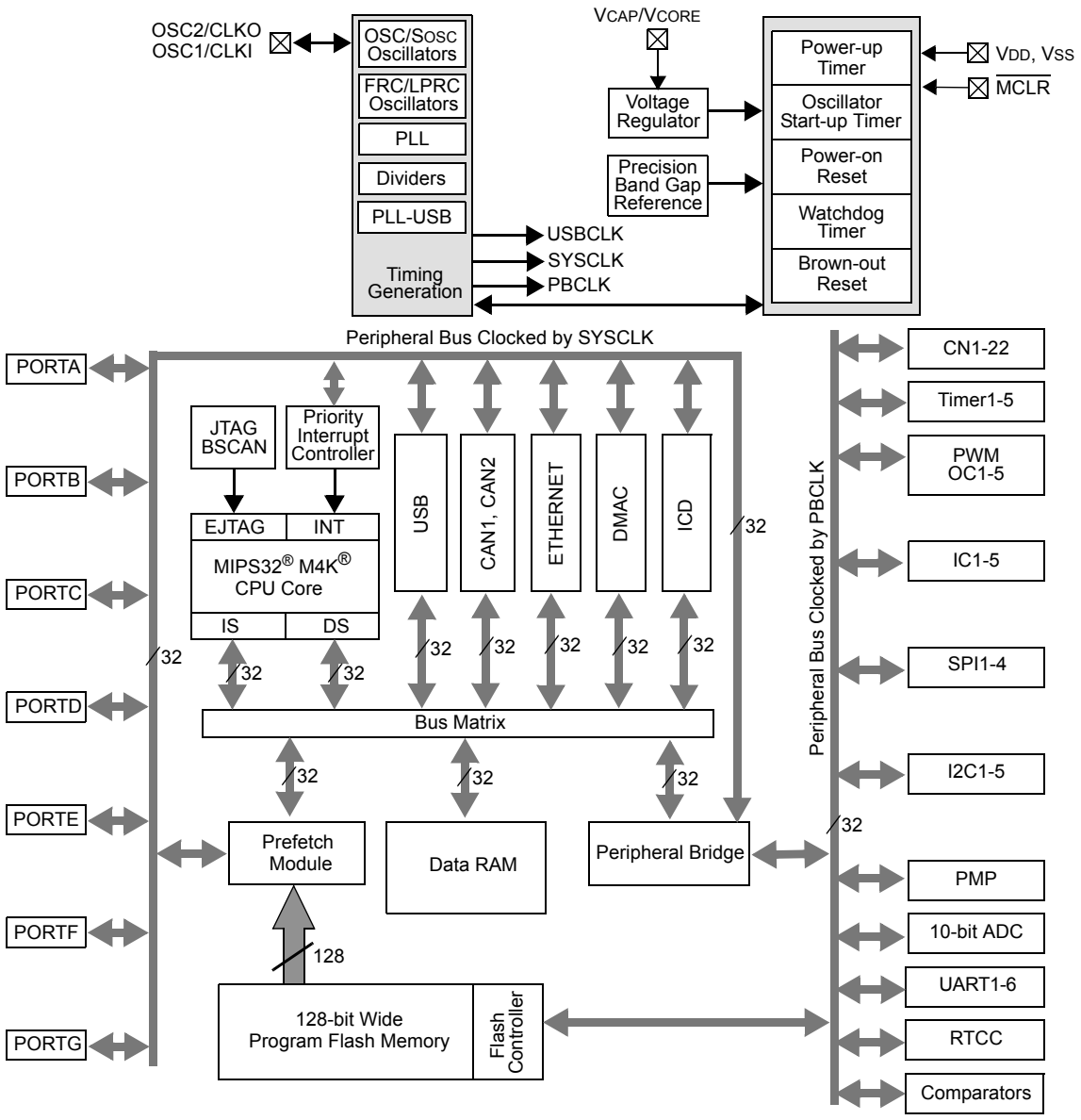


Figure 2.2: The PIC32MX5XX/6XX/7XX architecture.

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.

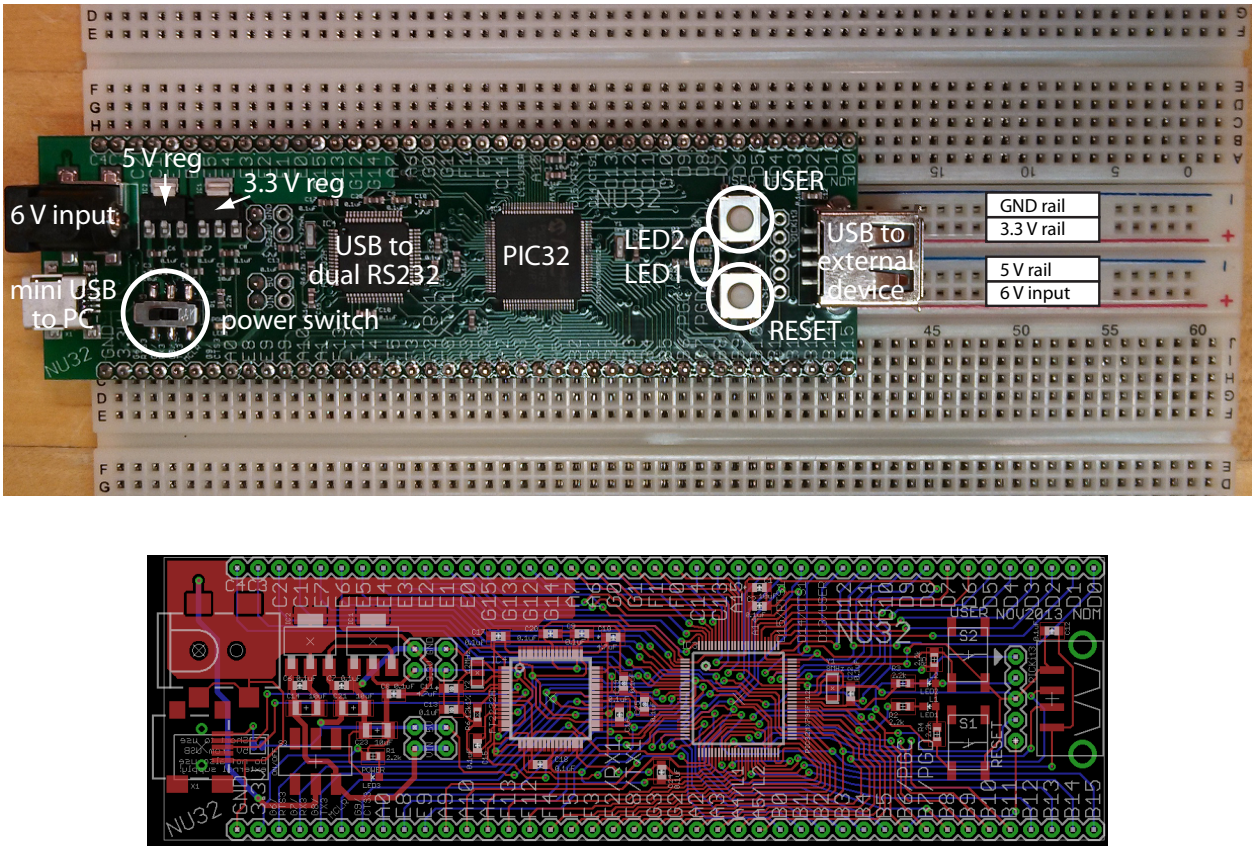


Figure 2.3: The NU32 development board: photo and PCB silkscreen.

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.

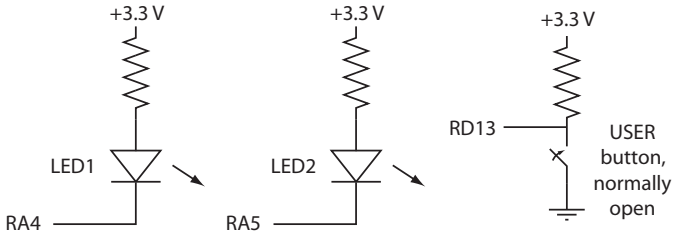


Figure 2.4: The NU32 connection of pins RA4, RA5, and RD13 to LED1, LED2, and the USER button, respectively.

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.

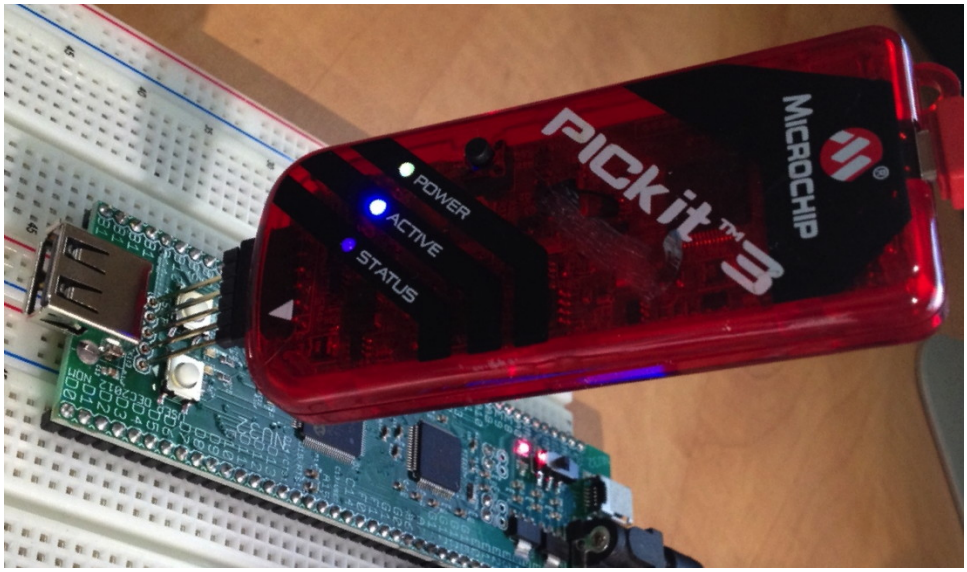


Figure 2.5: Attaching the PICkit 3 programmer to the NU32 board.

- 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.

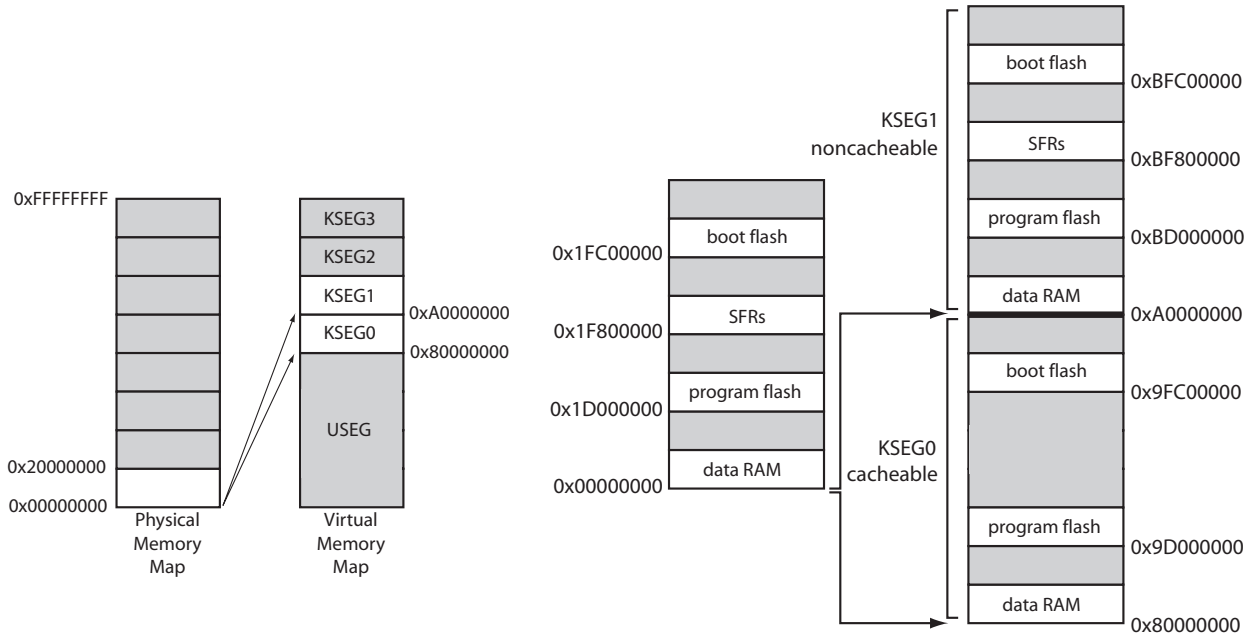


Figure 3.1: (Left) The 4 GB physical and virtual memory maps are divided into 512 MB segments. The mapping of the valid physical memory addresses to the virtual memory regions KSEG0 and KSEG1 is illustrated. The PIC32 does not use the virtual memory segments KSEG2 and KSEG3, which are allowed by the 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

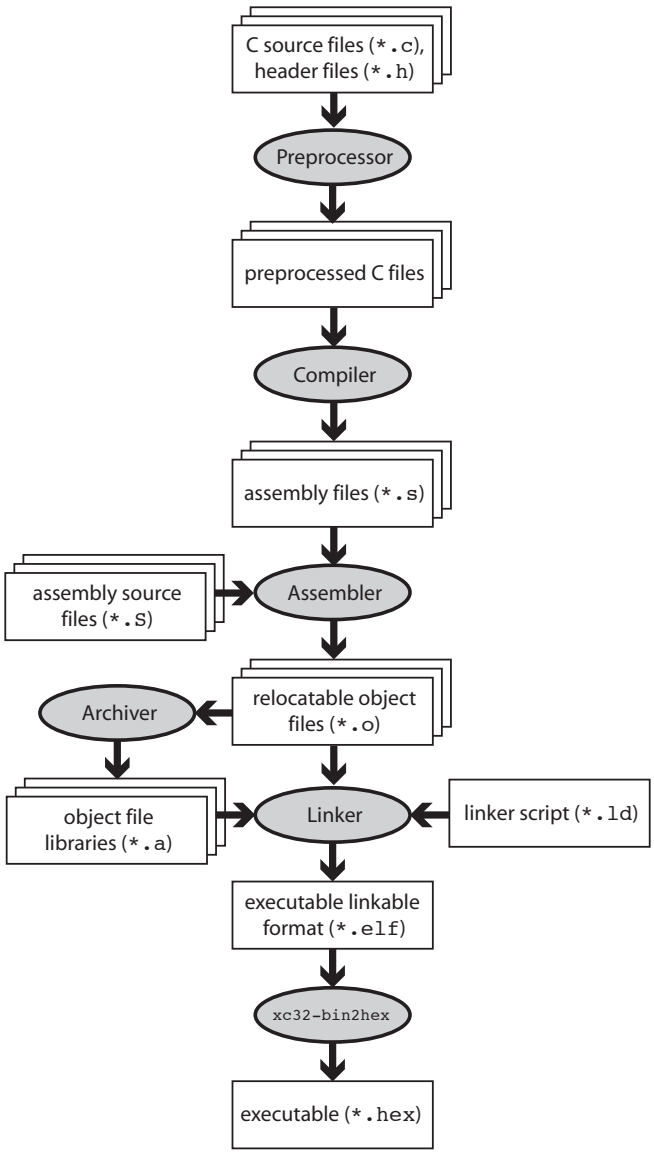


Figure 3.2: The "compilation" process.

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

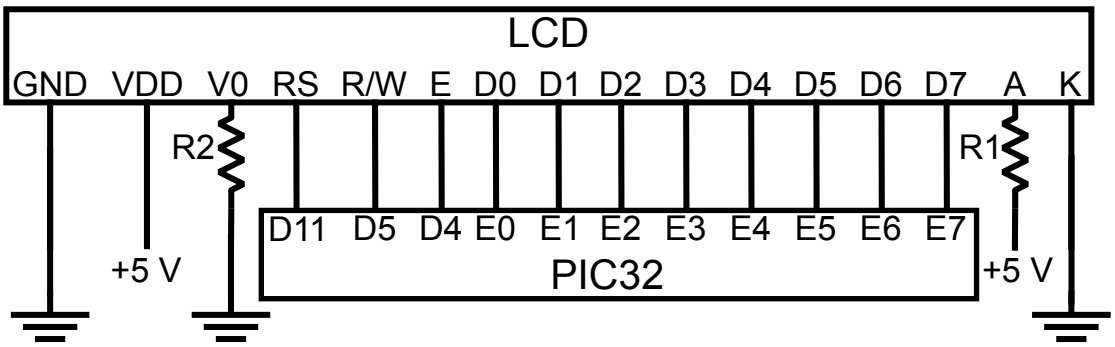


Figure 4.1: Circuit diagram for the LCD.

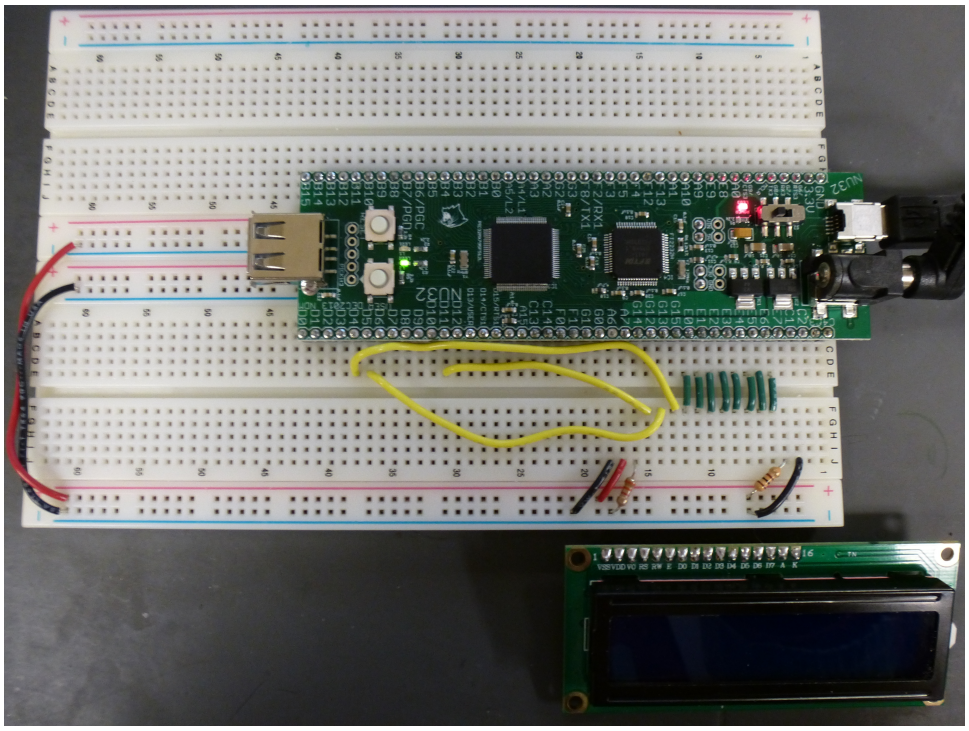


Figure 4.2: The NU32 board and the LCD screen.

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 8). 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.

If the core timer is being used to time different things, do not reset the counter to zero. Instead, read the value `initial` at the start of the timing, then the value `final` at the end, and subtract. If `final` is less than `initial`, then a core timer rollover occurred, and the actual number of elapsed ticks (assuming only one rollover) is $2^{32} + \texttt{final} - \texttt{initial}$.

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.

[2] The output from the `xc32-objdump` disassembler is not perfect. While the assembly 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.

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:

```
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

[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.

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.

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 filename.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 Exercise 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. (Hint: Any real button press should last much more than 10 milliseconds, while the mechanical bouncing period of any decent switch should be much less than 10 milliseconds.)

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. Write a program identical to the one in Problem 18, but using a 16×2 LCD screen for output 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.

Part III

Peripheral Reference

Chapter 7

Digital Input and Output

Digital inputs and outputs (DIO) are the simplest of interfaces between the PIC and other electronics, sensors, and actuators. The PIC32 has many DIO pins, each of which normally takes one of two states: high or low. The interrupt associated with DIO is change notification—an interrupt can optionally be generated when the input changes on at least one of up to 22 digital inputs.

7.1 Overview

The PIC32 offers many DIO pins, arranged into "ports" A through G. The pins are labeled Rxy, where x is the port letter and y is the pin number. For example, pin 5 of port B is named RB5. Ports B and D are full 16-bit ports, with pins 0-15. (Port B can also be used for analog input.) Other ports have a smaller number of pins, not necessarily sequentially numbered; for example, port C has pins 1-4 and 12-15. All pins labeled Rxy can be used for input or output, except for RG2 and RG3, which are input only. For more details on the available pin numbers, see the Data Sheet.

Each pin configured as an output can output 0 or 3.3 V (assuming the PIC32 is powered by 3.3 V). Some DIO pins can alternatively be configured as "open-drain" outputs. An open-drain output can take one of two states: 0 V or "floating" (disconnected). This is in contrast to the usual "buffered" output that drives the pin either low (0 V) or high (3.3 V). An open-drain output allows you to attach an external "pull-up" resistor from the output to a positive voltage you choose, up to 5 V. Then when the output is left floating by the PIC, the external pull-up resistor pulls the output up to this voltage (Figure 7.1). Voltages between 3.3 V and 5 V should only be used on 5 V tolerant pins (see, e.g., Figure 2.1 and Table 2.2).

A pin configured as an output should not be expected to source or sink more than about 10 mA. For example, it would be a mistake to connect a digital output to a 10 ohm resistor to ground. In this case, creating a digital high output of 3.3 V would require 3.3 V/10 ohms = 330 mA, which a digital output is not capable of. Instead, the actual output voltage would be much lower, dragged down by the low resistance.

An input pin will read low, or 0, if the input voltage is close to zero, and it will read high, or 1, if it is close to 3.3 V. Some pins tolerate inputs up to 5 V. Some input pins, those that can also be used for "change notification" (labeled CNy, y = 0 ... 21, spread out over several of the ports), can be configured with an internal pull-up resistor to 3.3 V. If configured this way, the input will read "high" if it is disconnected (Figure 7.1)). Otherwise, if an input pin is not connected to anything, we cannot be certain what the input will read.

Input pins have fairly high input impedance—very little current will flow in or out of an input pin, and therefore connecting an external circuit to an input pin should have little effect on the behavior of the external circuit.

The interrupt associated with digital I/O is change notification. If any of the 22 CN pins is configured as a change notification input and the change notification interrupt is enabled, then an interrupt will be generated when any of those inputs changes value. The ISR must then read the ports of those pins, or else future input changes will not result in an interrupt. The ISR can compare the new port values to the previous values to determine which input has changed.

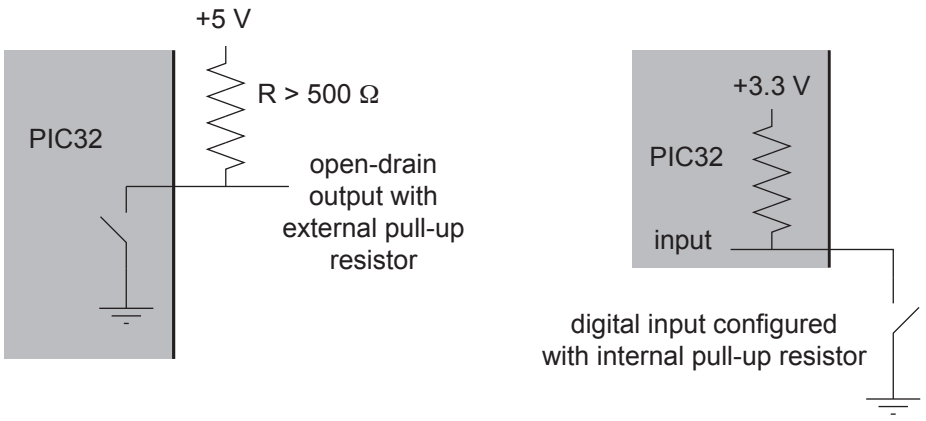


Figure 7.1: Left: A digital output configured as an open-drain output with external pull-up resistor to 5 V. (This should only be done for 5 V tolerant pins.) The resistor should allow no more than 10 mA to flow into the PIC32 when the PIC32 holds the output low. If the LAT bit controlling the pin has the value 1, the internal switch is open, and the output reads 5 V. If the bit is a 0, the internal switch is closed and the output reads 0 V. Right: A digital input configured with an internal pull-up resistor allows a simple open-close switch to yield digital high when the switch is open and digital low when the switch is closed.

Microchip recommends that unused digital I/O pins be configured as outputs and driven to a logic low state, though this is not required.

7.2 Details

AD1PCFG The PIC32MX795F512L has one analog-to-digital converter named AD1, and this AD1 pin configuration SFR determines which of the 16 pins with an analog input function (port B) are treated as analog inputs and which are treated as DIO pins. The 16 most significant bits (high bits) AD1PCFG⟨31:16⟩ are ignored. AD1PCFG⟨x⟩, or AD1PCFGbits.PCFGx, x = 0 to 15, controls whether pin ANx is an analog input or not: 0 = analog input, 1 = digital pin. The default on reset is 0, so all pins on port B are analog inputs by default.

TRISx, x = A to G These tri-state SFRs determine which of the DIO pins of port x are inputs and which are outputs. For example, TRISAbits.TRISA5 = 0 makes RA5 an output (0 = Output), and TRISAbits.TRISA5 = 1 makes RA5 an input (1 = Input). Bits of TRISx default to 1 on reset.

LATx, x = A to G A write to the latch chooses the output for pins configured as digital outputs. (Pins configured as inputs will ignore the write.) For example, if TRISD = 0x0000, making all 16 pins on port D ouptuts, then LATD = 0x00FF makes pins RD0-7 high and pins RD8-15 low. An individual pin can be referenced using LATDbits.LATD13. A write of 1 to an open-drain output sets the output to floating, while a write of 0 sets the output to low.

PORTx, x = A to G PORTD returns the current digital inputs for DIO pins on port D configured as inputs. PORTDbits.RD6 returns the digital input for RD6.

ODCx, x = A to G The open-drain configuration SFR determines whether outputs are open drain or not. If TRISDbits.TRISD8 = 0, making RD8 an output, then ODCDbits.ODCD8 = 1 configures RD8 as an open-drain output, and ODCDbits.ODCD8 = 0 configures RD8 as a typical buffered output. The reset default for all bits is 0.

CNPUE The relevant bits of the change notification pull-up enable SFR are CNPUE⟨21:0⟩, and they apply only to the 22 pins that have a change notification function, CN0 . . . CN21. If CNPUEbits.CNPUE2 =

1, then CN2/RB0 has the internal pull-up resistor enabled, and if CNPUEbits.CNPUE2 = 0, then it is disabled. The reset default for all bits is 0. An internal pull-up resistor can be convenient

Other SFRs specific to the interrupt function of the digital I/O peripheral are:

CNCON The change notification control SFR enables CN interrupts if CNCON⟨15⟩ (or CNCONbits.ON) equals 1. The default is 0.

CNEN A particular pin CNx is included in the change notification scan if CNEN⟨x⟩ (or CNENbits.CNENx) is 1. Otherwise it is not included in the change notification.

A recommended procedure for enabling the CN interrupt is to (1) write an ISR using the vector 26 (also defined as the `_CHANGE_NOTICE_VECTOR` in `p32mx795f512l.h`) that both clears the IRQ flag status IFS1bits.CNIF and reads the pins involved in the CN scan to reenable the interrupt; (2) disable interrupts at the CPU; (3) set CNCONbits.ON to 1 and CNENbits.CNENx to 1, for each pin x included in the change notification; (4) choose the vector priority IPC6bits.CNIP (or IPC6⟨28:26⟩) and subpriority IPC6bits.CNIS (or IPC6⟨25:24⟩); (5) clear the IRQ flag status IFS1bits.CNIF (or IFS1⟨0⟩) to 0; (6) set the IRQ enable bit IEC1bits.CNIE (or IEC1⟨0⟩) to 1; and (7) reenable interrupts at the CPU.

7.3 Sample Code

Our first program, `simplePIC.c`, demonstrated the use of two digital outputs (to control two LEDs) and one digital input (the USER button).

The example below sets up the following inputs and outputs:

- Pins RB0 and RB1 as digital inputs with internal pull-up resistors enabled.
- Pins RB2 and RB3 as digital inputs without pull-ups.
- Pins RB4 and RB5 as buffered digital outputs.
- Pins RB6 and RB7 as open-drain digital outputs.
- Pins AN8–AN15 as analog inputs.
- Port F as digital input, with RF4 with an internal pull-up.
- Change notification based on pins RB0, RF4, and RF5. Since both ports B and F are involved in the change notification, both ports must be read inside the ISR to allow the interrupt to be re-enabled. The ISR toggles one of the NU32 LEDs to indicate that a change has been noticed on pin RB0, RF4, or RF5.

Code Sample 7.1. `DIO_sample.c`. Digital input, output, and change notification.

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

unsigned int oldB = 0, oldF = 0, newB = 0, newF = 0;   // to hold the port B and F reads

void __ISR(_CHANGE_NOTICE_VECTOR, IPL3SOFT) CNISR(void) { // INT step 1
  newB = PORTB;           // since pins on port B and F are being monitored
  newF = PORTF;           // by CN, must read both to allow continued functioning
  // ... do something here with oldB, oldF, newB, newF ...
  oldB = newB;            // save the current values for future use
  oldF = newF;
  LATBINV = 0xF0;         // toggle buffered RB4, RB5 and open-drain RB6, RB7
  NU32_LED1 = !NU32_LED1; // toggle LED1
```

```
  IFS1CLR = 1;            // clear the interrupt flag
}

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

  AD1PCFG = 0x00FF;       // set port B pins 8-15 as analog in, 0-7 as digital pins
  TRISB = 0xFF0F;         // set port B pins 4-7 as digital outputs, 0-3 as digital inputs
  ODCBSET = 0x00C0;       // set ODCB bits 6 and 7, so RB6, RB7 are open drain outputs
  CNPUEbits.CNPUE2 = 1;   // CN2/RB0 input has internal pull-up
  CNPUEbits.CNPUE3 = 1;   // CN3/RB1 input has internal pull-up
  CNPUEbits.CNPUE17 = 1;  // CN17/RF4 input has internal pull-up

  oldB = PORTB;           // bits 0-3 are relevant input
  oldF = PORTF;           // all pins of port F are inputs, by default
  LATB = 0x0050;          // RB4 is buffered high, RB5 is buffered low,
                          // RB6 is floating open drain (could be pulled to 3.3 V by
                          // external pull-up resistor), RB7 is low

  __builtin_disable_interrupts(); // step 1: disable interrupts
  CNCONbits.ON = 1;               // step 2: configure peripheral: turn on CN
  CNEN = (1<<2)|(1<<17)|(1<<18);  //         listen to CN2/RB0, CN17/RF4, CN18/RF5
  IPC6bits.CNIP = 3;              // step 3: set interrupt priority
  IPC6bits.CNIS = 2;              // step 4: set interrupt subpriority
  IFS1bits.CNIF = 0;              // step 5: clear the interrupt flag
  IEC1bits.CNIE = 1;              // step 6: enable the CN interrupt
  __builtin_enable_interrupts();  // step 7: CPU enabled for mvec interrupts

  while(1) {
      ;                           // infinite loop
  }
  return 0;
}
```

7.4 Chapter Summary

- The PIC32 has DIO ports A through G. Only ports B and D have all 16 bits. Nearly all of the pins can be configured to be digital input or digital output. Some of the outputs can be configured to be open drain. Only port B inputs can be used for analog input.
- Twenty-two pins can be configured as change notification pins. For those pins that are enabled as change notification pins, any change on their inputs will generate an IRQ. To re-enable the interrupt, the ISR should both clear the IRQ flag status as well as read the pins involved in the change notification.
- The change notification pins offer an optional internal pull-up resistor so that the input registers as high when it is left floating. These pull-up resistors can be used regardless of whether change notification is used on the pins. The internal pull-up resistor allows simple interfacing of pushbuttons, for example.

7.5 Exercises

1. True or false? If an input pin is not connected to anything, it will always read digital low.
2. You are setting up port B to receive analog input and digital input, and to write digital output. Here is how you would like to configure the port. (Pin x corresponds to RBx.)
    - Pin 0 is an analog input.

- Pin 1 is a "typical" buffered digital output.
- Pin 2 is an open-drain digital output.
- Pin 3 is a "typical" digital input.
- Pin 4 is a digital input with an internal pull-up resistor.
- Pins 5-15 are analog inputs.
- Pin 3 is monitored for change notification, and the change notification interrupt is enabled.

Questions:

(a) Which digital pin would most likely have an external pull-up resistor? What would be a reasonable range of resistances to use? Explain what factors set the lower bound on the resistance and what factors set the upper bound on the resistance.

(b) To achieve the configuration described above, give the eight-digit hex values you should write to AD1PCFG, TRISB, ODCB, CNPUE, CNCON, and CNEN. (Some of these SFRs have unimplemented bits 16-31; write 0 for those bits.)

Chapter 8

Counter/Timers

The basic function of counters is quite simple: they count pulse rising edges. The pulses may come from the internal peripheral bus clock or external sources. If the pulses come from a fixed frequency clock, they can be thought of as timers, hence the words counter and timer are often used interchangeably. Since Microchip's documentation mostly refers to them as timers, we'll adopt that terminology. Timers can generate an interrupt (1) when a preset number of pulses has been counted or (2) on the falling edge of an external pulse whose duration is being timed.

8.1 Overview

The PIC32 is equipped with five 16-bit peripheral timers, Timer1-5. A timer increments on the rising edge of a clock signal, which may come from the PBCLK or from an external source of pulses, such as an encoder. If an external source is used, the input for Timerx is on the pin TxCK. A prescaler $N \geq 1$ determines how many clock pulses must be received before the counter increments. If the prescaler is set to $N = 1$, the counter increments on each clock rising edge; if it is set to $N = 8$, it increments every eighth rising edge. The clock source type (internal or external) and the prescaler value is chosen by setting the value of the SFR TxCON.

Each 16-bit timer can count from 0 up to a period $P \leq 2^{16} - 1 = 65535 = 0\text{xFFFF}$. The current count is stored in the SFR TMRx and the value of P can be chosen by writing to the period register SFR PRx. When the timer reaches the value P, a *period match* occurs, and after N more pulses are received, the counter "rolls over" to 0. If the input to the timer is the 80 MHz PBCLK, with 12.5 ns between rising edges, then the time between rollovers is $T = (P + 1) \times N \times 12.5$ ns. Choosing $N = 1$ and $P = 79,999$, we get $T = 1$ ms, and changing N to 8 gives $T = 8$ ms. By configuring the timer to use the PBCLK and to generate an interrupt when a period match occurs, the timer can implement a function to run at a fixed frequency (a controller, for example). See Figure 8.1.

If the period P is set to 0, then once the count reaches zero, the timer will never increment again (it keeps rolling over). No interrupt can be generated by a period match if $P = 0$.

There are two types of timer on the PIC32, Type A and Type B, with slightly different features explained shortly. Only Timer1 is type A; Timers 2-5 are type B. The timers can be used in the following modes, chosen by the SFR TxCON:

Counting PBCLK pulses. This is the mode described above, and it is often used to generate interrupts at a desired frequency by appropriate setting of N and P. It could also be used to time the duration of some code, much like the core timer is used in Chapter 5, except that peripheral timers can be set to increment once every N PBCLK cycles, including $N = 1$, not just every 2 SYSCLK cycles.

Synchronous counting of external pulses. For the timer Timerx, an external pulse source is connected to the pin TxCK. The timer count increments after each rising edge of the external source. This mode is called "synchronous" because timer increments are synchronized to the PBCLK; the timer does not actually increment until the rising edge of PBCLK after the rising edge of the external source. If the external pulses are too fast, the timer will not accurately count them. According to the Electrical

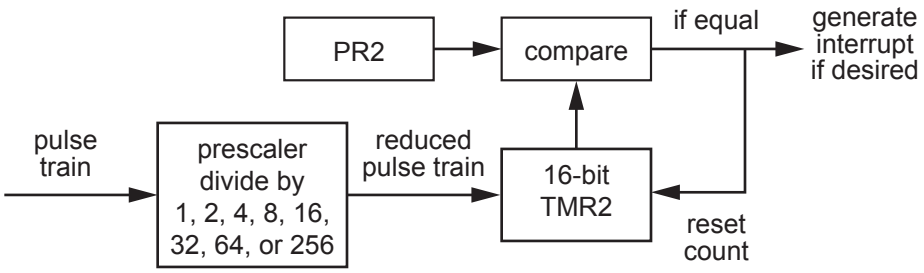


Figure 8.1: A simplified block diagram for a typical use of the 16-bit Timer2. The input pulse train, which can be from the PBCLK or an external source, goes into a prescaler, which produces one output pulse for every N input pulses, where N could be 1, 2, 4, 8, 16, 32, 64, or 256. These pulses are counted and the count is stored in the TMR2 SFR. TMR2 resets to zero on the next pulse after its value matches the value in the period register PR2. By default, the PR2 value is 0xFFFF, so TMR2 counts up to $2^{16} - 1$ before rolling over to zero. Timers 3, 4, and 5 are similar to Timer2, but the prescaler for Timer1 can only take the values $N = 1, 8, 64, 256$.

Characteristics section of the Data Sheet, the duration of the high and low portions of a pulse should be at least 37.5 ns.

Asynchronous counting of external pulses (Type A Timer1 only). The Type A pulse counting circuit can be configured to increment independently of the PBCLK. Because of this, Timer1 can count even when the PBCLK is not operating, such as in the power-saving Sleep mode. If there is a period match, Timer1 can generate an interrupt and wake up the PIC32.

When Timer1 is set up to count external pulses and the Secondary Oscillator is enabled by the configuration bit FSOSCEN of DEVCFG1, then the secondary oscillator on the pins SOSCO and SOSCI becomes the timer input. Typically a precise 32.768 kHz oscillator (2^{15} pulses every second) is used on SOSCO/SOSCI for the Real-Time Clock and Calendar function (Chapter **??**).

The Timer1 asynchronous counting mode can count shorter pulses than the synchronous mode, down to 10 ns high and low durations.

Timing the duration of an external pulse. This is also called "gated accumulation mode." For Timerx, when the input on external pin TxCK goes high, the counter starts incrementing according to the PBCLK and the prescaler N. When the input drops low, the count stops. The timer can also generate an interrupt when the input drops low, for example signaling an ISR to examine the duration of the pulse.

Important differences between Timer1 (Type A) and Timer2-5 (Type B) are:

- Only Timer1 can count external pulses asynchronously, as described above.
- Timer1 can have prescalers N = 1, 8, 64, and 256, while Timer2-5 can have prescalers N = 1, 2, 4, 8, 16, 32, 64, and 256.
- Timer2 and Timer3 can be chained to make a single 32-bit timer, called Timer23. Timer4 and Timer5 can similarly be used to make a single 32-bit timer, called Timer45. These combined timers allow counts up to $2^{32} - 1$, or over 4 billion. When two timers are used to make Timerxy (x < y), Timerx is called the "master" and Timery is the "slave"—only the prescaler and mode information in TxCON are relevant, while that in TyCON is ignored. When Timerx rolls over from $2^{16} - 1$ to 0, it sends a clock pulse to increment Timery. In Timerxy mode, the 16 bits of TMRy are copied to the most significant 16 bits of the SFR TMRx, so the 32 bits of TMRx contain the full count of Timerxy. Similarly, the 32 bits of PRx contain the 32-bit period match value Pxy. The interrupt associated with a period match (or a falling input in gated accumulation mode) is actually generated by Timery, so interrupt settings should be chosen for Timery's IRQ and vector.

Timers are used in conjunction with digital waveform generation by the Output Compare peripheral (Chapter 9) and in timing digital input waveforms by the Input Capture peripheral (Chapter **??**). A timer can also be used to repeatedly trigger analog to digital conversions (Chapter 10).

8.2 Details

The following SFRs are associated with the timers. All SFRs default to 0x0000, except the PRx SFRs, which default to 0xFFFF.

T1CON The Timer1 control SFR configures the behavior of the Type A timer Timer1. Important bits include

T1CON⟨15⟩, or T1CONbits.ON: 1 = timer is enabled, 0 = timer is off.

T1CON⟨7⟩, or T1CONbits.TGATE: 1 = gated time accumulation is enabled, 0 = gated time accumulation is disabled. The gate input is T1CK. (Gated time acculumation mode requires T1CON⟨1⟩ = 0.)

T1CON⟨5:4⟩, or T1CONbits.TCKPS: 0b11 = 1:256 prescaler, 0b10 = 1:64 prescaler, 0b01 = 1:8 prescaler, 0b00 = 1:1 prescaler.

T1CON⟨2⟩, or T1CONbits.TSYNC: 1 = external clock inputs are synchronized with the PBCLK, 0 = unsynchronized.

T1CON⟨1⟩, or T1CONbits.TCS: 1 = external clock from T1CK pin, 0 = from PBCLK.

TxCON, x = 2 to 5 These SFRs configure the behavior of the Type B timers Timer2-5.

TxCON⟨15⟩, or TxCONbits.ON: 1 = timer is enabled, 0 = timer is off.

TxCON⟨7⟩, or TxCONbits.TGATE: 1 = gated time accumulation is enabled, 0 = gated time accumulation is disabled. The gate input is TxCK. (Gated time acculumation mode requires TxCON⟨1⟩ = 0.)

TxCON⟨6:4⟩, or TxCONbits.TCKPS: 0b111 = 1:256 prescaler, 0b110 = 1:64 prescaler, 0b101 = 1:32 prescaler, 0b100 = 1:16 prescaler, 0b011 = 1:8 prescaler, 0b010 = 1:4 prescaler, 0b001 = 1:2 prescaler, 0b000 = 1:1 prescaler.

TxCON⟨3⟩, or TxCONbits.T32: This bit is only relevant for x = 2 and 4 (Timer2 and Timer4), and it determines whether Timer3 and Timer5 are chained to make a 32-bit timer, respectively. 1 = chain Timerx with Timery to make a 32-bit timer Timerxy, 0 = Timerx and Timery are separate 16-bit timers. For a 32-bit timer, TyCON settings are ignored; TMRy is enabled and its clock comes from the rollover of TMRx after hitting 0xFFFF.

TxCON⟨1⟩, or TxCONbits.TCS: 1 = external clock from TxCK pin, 0 = from PBCLK.

TMRx, x = 1 to 5 TMRx⟨15:0⟩ stores the 16-bit count of Timerx. TMRx resets to 0 on the next clock input (after the prescaler) after TMRx reaches the number stored in PRx. This is called a period match. In Timerxy 32-bit mode, TMRx contains the 32-bit value of the chained timer, and period match occurs when TMRx = PRx.

PRx, x = 1 to 5 PRx⟨15:0⟩ of the period register contains the maximum value of the count of TMRx before it resets to zero on the next count. An interrupt can be generated on this period match. In Timerxy 32-bit timer mode, PRx contains the 32-bit value of the period *P*xy.

If the timer is enabled (TxCON.ON = 1), the timer can generate an interrupt on the falling edge of the gate input when it is in gated mode (TxCONbits.TCS = 0 and TxCONbits.TGATE = 1) or a period match otherwise. To enable the interrupt for Timerx, the interrupt enable bit IEC0bits.TxIE must be set. The interrupt flag bit IFS0bits.TxIF should be cleared and the priority and subpriority bits IPCxbits.TxIP and IPCxbits.TxIS must be written. See the the table in Chapter 6 or in the Interrupt Controller section of the Data Sheet. In 32-bit Timerxy mode, interrupts are generated by Timery; interrupt settings for Timerx are ignored.

8.3 Sample Code

8.3.1 A Fixed Frequency ISR

To create a 5 Hz ISR with an 80 MHz PBCLK, the interrupt must be triggered every 16 million PBCLK cycles. The highest a 16-bit timer can count is $2^{16} - 1$. Insteaad of wasting two timers to make a 32-bit timer with a prescaler $N = 1$, let's use a single 16-bit timer with a prescaler $N = 256$. We'll use Timer1. We should choose PR1 to satisfy

```
16000000 = (PR1 + 1) * 256
```

that is, PR1 = 62499. The ISR in the following code toggles a digital output at 5 Hz, creating a 2.5 Hz square wave (a flashing LED on the NU32).

Code Sample 8.1. `TMR_5Hz.c`. Timer1 toggles RA5 five times a second (LED2 on the NU32 flashes).

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

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1ISR(void) {  // INT step 1: the ISR
  LATAINV = 0x20;                 // toggle RA5
  IFS0bits.T1IF = 0;              // clear interrupt flag
}

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

  __builtin_disable_interrupts(); // INT step 2: disable interrupts at CPU
                                  // INT step 3: setup peripheral
  PR1 = 62499;                    //             set period register
  TMR1 = 0;                       //             initialize count to 0
  T1CONbits.TCKPS = 3;            //             set prescaler to 256
  T1CONbits.TGATE = 0;            //             not gated input (the default)
  T1CONbits.TCS = 0;              //             PCBLK input (the default)
  T1CONbits.ON = 1;               //             turn on Timer1
  IPC1bits.T1IP = 5;              // INT step 4: priority
  IPC1bits.T1IS = 0;              //             subpriority
  IFS0bits.T1IF = 0;              // INT step 5: clear interrupt flag
  IEC0bits.T1IE = 1;              // INT step 6: enable interrupt
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU
  while (1) {
      ;                           // infinite loop
  }
}
```

8.3.2 Counting External Pulses

The following code uses the 16-bit timer Timer2 to count the rising edges on the input T2CK. The 32-bit Timer45 creates an interrupt at 2 kHz to toggle a digital output, generating a 1 kHz pulse train on RD1 that acts as input to T2CK. Although a 16-bit timer can certainly generate a 1 kHz pulse train, we use a 32-bit timer just to show the configuration. In Chapter 9 we will learn about the Output Compare peripheral, a better way to use a timer to create more flexible waveforms.

To create an IRQ every 0.5 ms (2 kHz), we use a prescaler $N = 1$ and a period match PR4 = 39,999, so

```
(PR4 + 1) * N * 12.5 ns = 0.5 ms
```

The code below displays to your computer's screen the amount of time that has elapsed since the PIC32 was reset, in milliseconds. If you wait 65 seconds, you'll see Timer2 roll over.

Code Sample 8.2. TMR_external_count.c. Timer45 creates a 1 kHz pulse train on RD1, and these external pulses are counted by Timer2. The elapsed time is periodically reported back to the host computer screen.

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

void __ISR(_TIMER_5_VECTOR, IPL4SOFT) Timer5ISR(void) {  // INT step 1: the ISR
  LATDINV = 0b10;                                        // toggle RD1
  IFS0bits.T5IF = 0;                                     // clear interrupt flag
}

int main(void) {
  char message[200] = {};
  int i = 0;

  NU32_Startup();                 // cache on, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts(); // INT step 2: disable interrupts

  TRISDbits.TRISD1 = 0;           // make D1 an output. connect D1 to C1/T2CK!

                                  // configure Timer2 to count external pulses.
                                  // The remaining settings are left at their defaults
  T2CONbits.TCS = 1;              // count external pulses
  PR2 = 0xFFFF;                   // enable counting to max value of 2^16 - 1
  TMR2 = 0;                       // set the timer count to zero
  T2CONbits.ON = 1;               // turn Timer2 on and start counting

                                  // 1 kHz pulses with 2 kHz interrupt from Timer45
  T4CONbits.T32 = 1;              // INT step 3: set up Timers 4 and 5 as 32-bit Timer45
  PR4 = 39999;                    //             rollover at 40,000; 80MHz/40k = 2 kHz
  TMR4 = 0;                       //             set the timer count to zero
  T4CONbits.ON = 1;               //             turn the timer on
  IPC5bits.T5IP = 4;              // INT step 4: priority for Timer5 (int goes with T5)
  IFS0bits.T5IF = 0;              // INT step 5: clear interrupt flag
  IEC0bits.T5IE = 1;              // INT step 6: enable interrupt
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU

  while (1) {
                                  // display the elapsed time in ms
    sprintf(message,"Elapsed time:  %u ms\r\n", TMR2);
    NU32_WriteUART1(message);

    for(i = 0; i < 10000000; ++i){// loop to delay printing
      _nop();                     // include nop so loop is not optimized away
    }
  }
  return 0;
}
```

8.3.3 Timing the Duration of an External Pulse

In this last example we modify our previous code so that Timer45 creates a train of pulses on RD1 that are high for one second and low for one second (a 0.5 Hz square wave). These pulses are timed by the 32-bit Timer23 in gated accumulation mode. The accumulated count begins when the T2CK input from RD1 goes

high and stops when the T2CK input drops low. The falling edge calls an ISR that resets the Timer23 count and displays the count, and the duration in seconds, to the screen. You should find that the count is very close to 80 million, as expected for the one second pulses generated by Timer45.

Code Sample 8.3. TMR_pulse_duration.c. Timer45 creates a series of 1 second pulses on RD1. These pulses are input to T2CK and Timer23 measures their duration.

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

void __ISR(_TIMER_5_VECTOR, IPL4SOFT) Timer5ISR(void) {  // INT step 1: the ISR
  LATDINV = 0b10;                 // toggle RD1
  IFS0bits.T5IF = 0;              // clear interrupt flag
}

void __ISR(_TIMER_3_VECTOR, IPL3SOFT) Timer23ISR(void) { // INT step 1: the ISR
  char msg[100] = {};

  sprintf(msg,"The count was %u, or %10.8f seconds.\r\n", TMR2, TMR2/80000000.0);
  NU32_WriteUART1(msg);
  TMR2 = 0;                       // reset Timer23
  IFS0bits.T3IF = 0;              // clear interrupt flag
}

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

  __builtin_disable_interrupts(); // INT step 2: disable interrupts

  TRISDbits.TRISD1 = 0;           // make D1 an output. connect D1 to C1/T2CK!

                                  // INT step 3
  T2CONbits.T32 = 1;              // for T23:    combine Timers 2 and 3 to make Timer23
  T2CONbits.TGATE = 1;            //             Timer23 in gated accumulation mode
  PR2 = 0xFFFFFFFF;               //             use the full period of Timer23
  T2CONbits.TON = 1;              //             turn Timer23 on
  T4CONbits.T32 = 1;              // for T45:    enable 32 bit mode Timer45
  PR4 = 79999999;                 //             set PR so timer rolls over at 1 Hz
  TMR4 = 0;                       //             initialize count to 0
  T4CONbits.TON = 1;              //             turn Timer45 on
  IPC5bits.T5IP = 4;              // INT step 4: priority for Timer5 (int for Timer45)
  IPC3bits.T3IP = 3;              //             priority for Timer3 (int for Timer23)
  IFS0bits.T5IF = 0;              // INT step 5: clear interrupt flag for Timer45
  IFS0bits.T3IF = 0;              //             clear interrupt flag for Timer23
  IEC0bits.T5IE = 1;              // INT step 6: enable interrupt for Timer45
  IEC0bits.T3IE = 1;              //             enable interrupt for Timer23
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at the CPU
  while (1) {
    ;
  }
}
```

8.4 Chapter Summary

- All five of the 16-bit PIC32 timers can be used to generate fixed-frequency interrupts, count external pulses, and time the duration of external pulses. Additionally, the Type A Timer1 can asynchronously

count external pulses even when the PIC32 is in Sleep mode, while the Type B timers Timer2 and Timer3 can be chained to make the 32-bit timer Timer23. Similarly, Timer4 and Timer5 can be chained to make the 32-bit timer Timer45.

- For a 32-bit timer Timerxy, the timer configuration information in TxCON is used (TyCON is ignored), and the interrupt enable, flag status, and priority bits are configured for Timery (this information for Timerx is ignored). The 32-bit Timerxy count is held in TMRx and the 32-bit period match value is held in PRx.
- A timer can generate an interrupt when (1) the external pulse being timed falls low (gated accumulation mode) or (2) the count reaches a value stored in a period register (period match).

8.5 Exercises

1. Assume PBCLK is running at 80 MHz. Give the four-digit hex values for T3CON and PR3 so that Timer3 is enabled, accepts PBCLK as input, has a 1:64 prescaler, and rolls over (generates an interrupt) every 16 ms.
2. Using a 32-bit timer (Timer23 or Timer45), what is the longest duration you can time, in seconds, before the timer rolls over? (Use the prescaler that maximizes this time.)

Chapter 9

Output Compare

"Output compare" refers to the fact that the count of a timer is compared against a fixed value, and when they are equal, a digital output is set high or low. This can be used to generate a single pulse of specified duration or a continuous train of pulses. Each mode of operation can generate an interrupt.

Like most microcontrollers, the PIC32 does not come with analog output (digital-to-analog converter, or DAC). Instead, one way to transmit an analog value is by using the timing of a fixed period pulse train from a single digital output: the analog value is proportional to the duty cycle of the pulse train, where the duty cycle is the percentage of the period that the signal is high. This is often called "pulse width modulation" (PWM). PWM signals are commonly used as input to H-bridge amplifiers that drive motors. High-frequency PWM can also be low-pass filtered to create a true analog output.

9.1 Overview

The five output compare peripherals can be configured to operate in seven different modes. In all modes, the module uses either the count of the 16-bit timer Timer2 or Timer3, or the count of the 32-bit timer Timer23, depending on the Output Compare Control SFR OCxCON (where x = 1...5 refers to the particular output compare module). We will call the timer Timery, where y = 2, 3, or 23. Timery must be set up with its own prescaler and period register, which will influence the behavior of the OC peripheral.

The OC peripheral's operating modes consist of three "single compare" modes, two "dual compare" modes, and two PWM modes, as chosen by three bits in OCxCON. In the single compare modes, the Timery count TMRy is compared to the value in OCxR. When they match, the OC output is either set high, cleared low, or toggled, depending on the mode. The last mode generates a continuous pulse train, with the period determined by the period register PRy of Timery and the pulse duration determined by OCxR.

In the dual compare modes, TMRy is compared to two values, OCxR and OCxRS. On the first match, the output is driven high, and on the second the output is driven low. Depending on a bit in OCxCON, either a single pulse is generated or a continuous pulse train is produced.

The two PWM modes create continuous pulse trains. Each pulse begins (is set high) at rollover of Timery, as set by the period register PRy. The output is then set low when the timer count reaches OCxR. To change the value of OCxR, the user's program may alter the value in OCxRS at any time. This value will then be transferred to OCxR at the beginning of the new time period. The duty cycle of the pulse train, as a percentage, is

```
duty cycle = OCxR/(PRy + 1) x 100%.
```

One of the two PWM modes offers the use of a fault protection input. If this is chosen, then the OCFA input pin (corresponding to OC1 through OC4) or the OCFB input pin (corresponding to OC5) must be high for PWM to operate. If the pin drops to logic low, corresponding to some external fault condition, then the the PWM output will be high impedance (effectively, the output is turned off, and is neither logic high nor low) until the fault condition is removed and the PWM mode is reset by a write to OCxCON.

9.2 Details

The output compare modules are controlled by the following SFRs. The OCxCON SFRs default to 0x0000 on reset; the OCxR and OCxRS SFR values are unknown after reset.

OCxCON, x = 1 to 5 This output compare control SFR determines the operating mode of OCx.

OCxCON⟨15⟩, or OCxCONbits.ON: 1 = output compare is enabled, 0 = disabled.

OCxCON⟨5⟩, or OCxCONbits.OC32: 1 = use the 32-bit timer source Timer23, 0 = use a 16-bit timer source Timer2 or Timer3.

OCxCON⟨4⟩, or OCxCONbits.OCFLT: The read-only PWM fault condition status bit. 1 = PWM fault has occurred, 0 = no fault has occurred.

OCxCON⟨3⟩, or OCxCONbits.OCTSEL: This timer select bit chooses the timer used for comparison. 1 = Timer3, 0 = Timer2. If the 32-bit timer Timer23 is selected using OCxCONbits.OC32 = 1, then the OCTSEL bit is ignored.

OCxCON⟨2:0⟩, or OCxCONbits.OCM: These three bits determine the operating mode:

- 0b111 PWM mode with fault pin enabled. OCx is set high on the timer rollover, then set low when the timer value matches OCxR. The SFR OCxRS can be altered at any time, and it is copied to OCxR at the beginning of the next timer period. (OCxR should be initialized before the OC module is enabled, to handle the first PWM cycle. After the OC module is enabled, OCxR is read-only.) The duty cycle of the PWM signal is OCxR/(PRy + 1) × 100%, where PRy is the period register of the timer. If the fault pin, OCFA for OC1-4 and OCFB for OC5, drops low, the read-only fault status bit OCxCONbits.OCFLT is set to 1, the OCx output is set to high impedance, and an interrupt is generated. The fault condition is cleared and PWM resumes once the fault pin goes high and the OCxCONbits.OCM bits are rewritten. One possible use of the fault pin is in conjunction with an Emergency Stop button that is normally high but drops low when the user presses it. If the OCx output is driving an H-bridge that powers a motor, setting the OCx output to high impedance will signal the H-bridge to stop sending current to the motor.
- 0b110 PWM mode with fault pin disabled. Identical to above, without the fault pin.
- 0b101 Dual compare mode, continuous output pulses. When the module is enabled, OCx is driven low. OCx is driven high on a match with OCxR and driven low on a match with OCxRS. The process repeats, creating an output pulse train. An interrupt can be generated when OCx is driven low.
- 0b100 Dual compare mode, single output pulse. Same as above, except the OCx pin will remain low after the match with OCxRS until the OC mode is changed or the module is disabled.
- 0b011 Single compare mode, continuous pulse train. When the module is enabled, OCx is driven low. The output is toggled on all future matches with OCxR until a mode change has been made or the module is disabled. An interrupt can be generated on each toggle.
- 0b010 Single compare mode, single high pulse. When the module is enabled, OCx is driven high. OCx will be driven low and an interrupt can be generated on a match with OCxR. OCx will remain low until the mode is changed or the module disabled.
- 0b001 Single compare mode, single low pulse. When the module is enabled, OCx is driven low. OCx will be driven high and an interrupt can be generated on a match with OCxR. OCx will remain high until the mode is changed or the module disabled.
- 0b000 The output compare module is disabled but still drawing current, unless OCxCONbits.ON = 0.

OCxR, x = 1 to 5 If OCxCONbits.OC32 = 1, then all 32-bits of OCxR are used for the compare to the 32-bit counter TMR23. Otherwise, only OCxR⟨15:0⟩ is compared to the 16-bit counter Timer2 or Timer3, depending on OCxCONbits.OCTSEL.

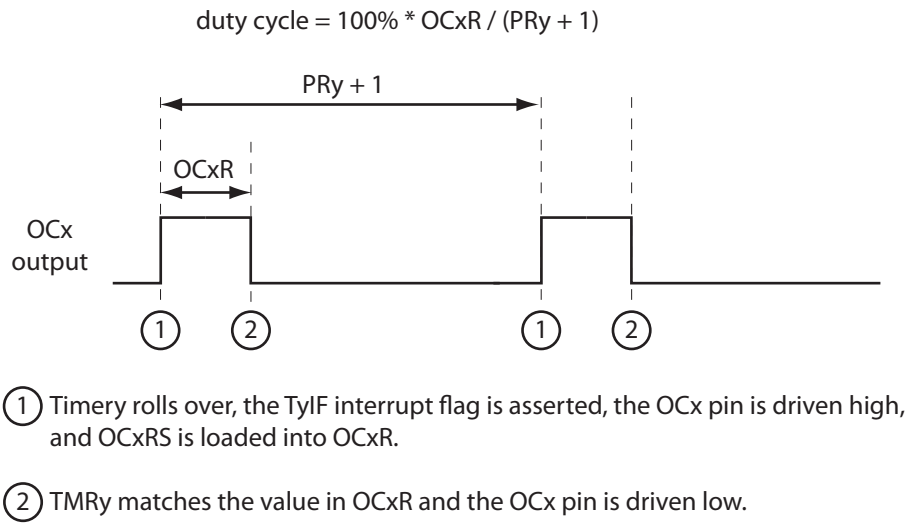


Figure 9.1: A PWM waveform from OCx using Timery as the time base.

OCxRS, x = 1 to 5 If OCxCONbits.OC32 = 1, then all 32-bits of OCxRS are used for the compare to the 32-bit counter TMR23. Otherwise, only OCxRS⟨15:0⟩ is compared to the 16-bit counter Timer2 or Timer3, depending on OCxCONbits.OCTSEL. This SFR is not used in the single compare modes.

Timer2, Timer3, or Timer23 (depending on OCxCONbits.OC32 and OCxCONbits.OCTSEL) must be separately configured. Output compare modules do not affect the behavior of the timers; they simply compare values in OCxR and OCxRS and alter the digital output OCx on match events.

The interrupt flag status and enable bits for OCx are IFS0bits.OCxIF and IEC0bits.OCxIE, and the priority and subpriority bits are IPCxbits.OCxIP and IPCxbits.OCxIS.

PWM Modes The most common modes for Output Compare are the PWM modes. They can be used to drive H-bridges powering motors or to continuously transmit analog values represented by the duty cycle. Microchip often refers to the duty cycle as the duration OCxR of the high portion of the PWM waveform, but it is more standard to refer to the duty cycle in the range 0 to 100%. A plot of a PWM waveform is shown in Figure 9.1.

9.3 Sample Code

9.3.1 Generating a Pulse Train with PWM

Below is sample code using OC1 with Timer2 to generate a 10 kHz PWM signal, initially at 25% duty cycle and then changed to 50% duty cycle. The fault pin is not used.

Code Sample 9.1. `OC_PWM.c` Generating 10 kHz PWM with 50% duty cycle.

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

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

  T2CONbits.TCKPS = 2;     // Timer2 prescaler N=4 (1:4)
  PR2 = 1999;              // period = (PR2+1) * N * 12.5 ns = 100 us, 10 kHz
  TMR2 = 0;                // initial TMR2 count is 0
```

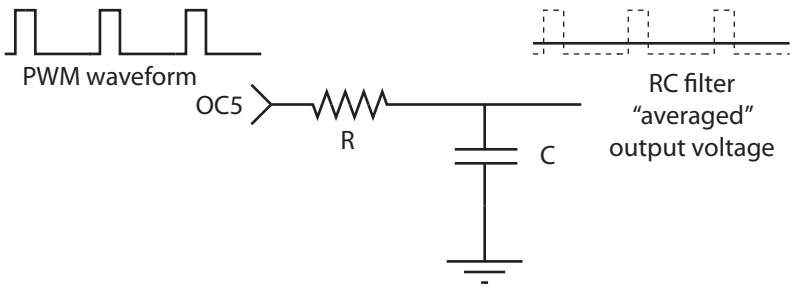


Figure 9.2: An RC low-pass filter "averaging" the PWM output from OC3.

```
  OC1CONbits.OCM = 0b110;  // PWM mode without fault pin; other OC1CON bits are defaults
  OC1RS = 500;             // duty cycle = OC1RS/(PR2+1) = 25%
  OC1R = 500;              // initialize before turning OC1 on; afterward it is read-only
  T2CONbits.ON = 1;        // turn on Timer2
  OC1CONbits.ON = 1;       // turn on OC1

  _CP0_SET_COUNT(0);       // delay 4 seconds so you can see the 25% duty cycle on a 'scope
  while(_CP0_GET_COUNT() < 4 * 40000000) {
    ;
  }
  OC1RS = 1000;            // set duty cycle to 50%
  while(1) {
    ;                      // infinite loop
  }
  return 0;
}
```

9.3.2 Analog Output

DC Analog Output

We can use a PWM signal of constant duty cycle to generate a DC (constant) analog output by low-pass filtering the PWM. The low-pass filter essentially time-averages the high and low voltages of the waveform,

```
average voltage = duty cycle * 3.3 V
```

assuming that the output compare module swings between 0 and 3.3 V (the range may actually be a bit less).

There are many ways to build circuits to low-pass filter a signal, including *active* filter circuits using op amps. Here we focus on a simple *passive* RC filter, as shown in Figure 9.2. The voltage V_C across the capacitor C is the output of the filter. In the limit where R is zero, then the output compare module attempts to source or sink enough current to allow the capacitor voltage to exactly track the nominal PWM square wave, and there is no "averaging" effect. As the resistance R is increased, however, the resistor increasingly limits the current I available to charge or discharge the capacitor, meaning that the capacitor's voltage changes more and more slowly, according to the relationship $dV_C/dt = I/C$.

The charging and discharging of the capacitor, and its relationship to the product RC, is shown in Figure 9.3. During the low portion of the PWM, the voltage across the capacitor decays toward zero according to the first-order exponential

$$V_C(t) = V_C(0)\, e^{-t/RC},$$

where time $t = 0$ corresponds to the beginning of the low portion of the PWM. The time constant RC, which has units of seconds, can be visualized by extending the slope dV_C/dt from $V_C(0)$ until it crosses $V_C = 0$. The time between $t = 0$ and the crossover is equal to RC.

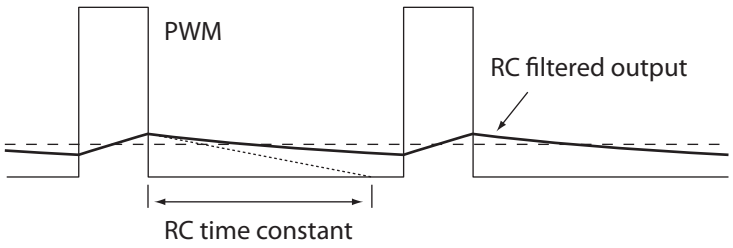


Figure 9.3: A close-up of the PWM, the RC filter output (with RC charging/discharging time constant illustrated), and the true time-averaged output (dashed). If the variation in the RC filtered output is unacceptably large, a larger value of RC should be chosen.

During the high portion of the PWM, the voltage follows a first-order exponential rise toward 3.3 V.

In Figure 9.3, the RC filter voltage variation during one PWM cycle is rather large. To reduce this variation, we would choose a larger product RC by increasing the resistance and/or capacitance. The drawback of a large RC is that the filter's output voltage changes slowly in response to a change in the PWM duty cycle. We address this in more detail below.

The PWM OCxR value can range from 0 to PRy + 1, where PRy is the period register of the Timery base for the OCx module. This means that PRy + 2 different average voltage levels are achievable.

Time-Varying Analog Output

If the PWM duty cycle is changing over time, it is more convenient to think of the RC filter's action on its input in terms of the filter's *frequency response*. Since the RC filter is linear, any sinusoidal input voltage $v_{\text{in}}(t) = A\sin ft$ yields a scaled, phase-shifted sinusoidal output of the same frequency f, $v_{\text{out}} = G(f)A\sin(ft + \phi(f))$, where the filter gain $G(f)$ and phase $\phi(f)$ are a function of the frequency f of the input. The gain and phase response of an RC filter are shown in Figure 9.4. At low input frequencies, the gain is approximately 1 and the phase is zero. At high frequencies, the gain approaches zero and the phase approaches -90 degrees. At the *cutoff frequency* $f_{\text{c}} = 1/(2\pi RC)$, the gain is $G(f_{\text{c}}) = 1/\sqrt{2}$ and the phase is $\phi(f_{\text{c}}) = -45°$, as shown in Figure 9.4. An input signal at the cutoff frequency is reduced to about 71% of its magnitude and is phase delayed by 45 degrees.

Suppose we want to create a sinusoidal analog output voltage by changing the duty cycle of the PWM. Let's call the frequency of this desired analog output f_{a}. Now we have three relevant frequencies: the PWM frequency f_{PWM}, the RC filter cutoff frequency f_{c}, and the desired analog voltage frequency f_{a}. Examining the frequency response of the RC filter in Figure 9.4, we could adopt the following rules of thumb for choosing these three frequencies:

- $f_{\text{PWM}} \geq 100 f_{\text{c}}$: The PWM waveform consists of a base frequency at f_{PWM} plus higher harmonics to create the square wave output. According to the gain response of the filter, only about 1% of the magnitude of the PWM frequency component at $100 f_{\text{c}}$ makes it through the RC filter. This is probably acceptable.

- $f_{\text{c}} \geq 10 f_{\text{a}}$: Again consulting the RC filter frequency response, we see that signals at 10 times less than f_{c} are relatively unaffected by the RC filter: the phase delay is only a few degrees and the gain is nearly 1.

As an example, if the PWM is at 100 kHz, then we might choose an RC filter cutoff frequency of 1 kHz, and the highest frequency analog output we should expect to be able to create would be 100 Hz. In other words, we can vary the PWM duty cycle through a full sinusoid (e.g., from 50% duty cycle to 100% duty cycle to 0% duty cycle and back to 50% duty cycle) 100 times per second.[1] If the desired analog output is not a sinusoid, then it should be a sum of signals at frequencies less than 100 Hz.

[1] Note that this creates a signal that is the sum of a 100 Hz sinusoid with a duty cycle amplitude equal to 50% plus a DC (zero frequency) component of amplitude equal to 50% duty cycle.

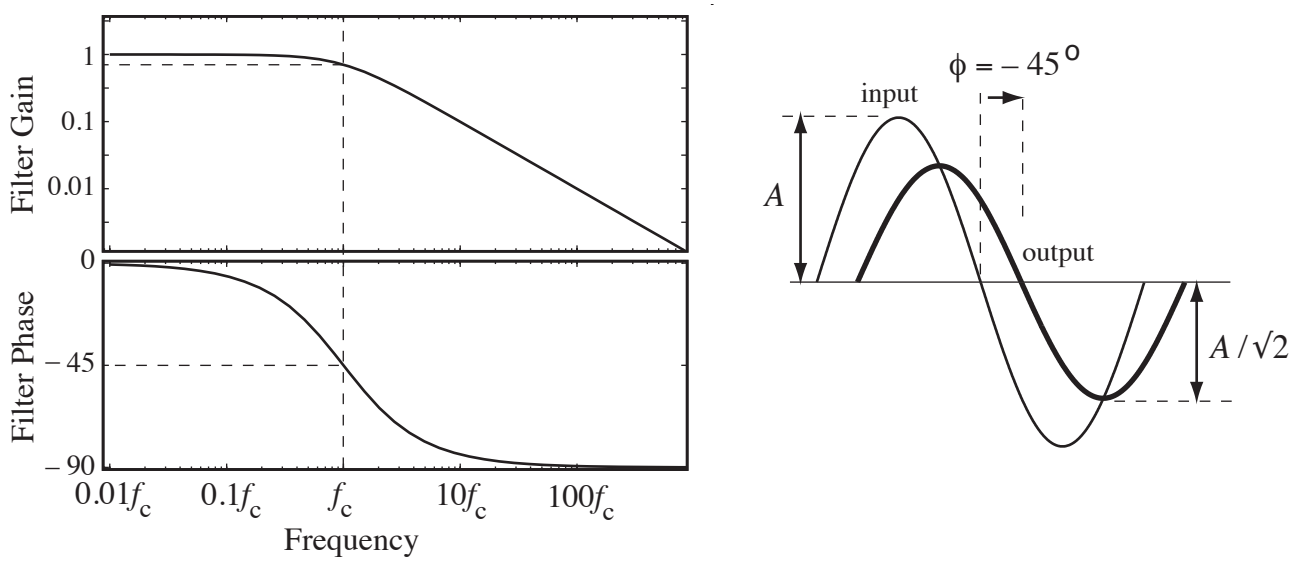


Figure 9.4: (Left) The frequency-dependent gain and phase response of an RC low-pass filter, with the gain and phase indicated at the cutoff frequency f_c. (Right) A sine wave input of amplitude A and frequency f_c is phase shifted by $-45°$ and reduced in amplitude by a factor of $1/\sqrt{2}$.

The maximum possible PWM frequency is determined by the 80 MHz PBCLK and the number of bits of resolution we require of the analog output. For example, if we want 8 bits of resolution in the analog output levels, this means we need $2^8 = 256$ different PWM duty cycles. Therefore the maximum PWM frequency is 80 MHz / 256 = 312.5 kHz.[2] On the other hand, if we require $2^{10} = 1024$ voltage levels, the maximum PWM frequency is 78.125 kHz. Thus there is a fundamental tradeoff between the voltage resolution and the maximum PWM frequency (and therefore the maximum analog output frequency f_a). While higher resolution analog output is generally desirable, there are limits to the value of increasing resolution beyond a certain point, because (1) the device receiving the analog input may have a limit to its analog input sensing resolution, and (2) the transmission lines for an analog signal may be subject to electromagnetic noise that create voltage variations larger than the analog output resolution.

Below is code to generate PWM at 78.125 kHz with a duty cycle determined by OC3R in the range 0 to 1024. The timer base is Timer2. With a suitable resistor and capacitor attached to OC3, the voltage across the capacitor reflects the analog voltage requested by the user. Because the voltage does not change quickly in this example, it is fine to choose f_c significantly lower than 781.25 Hz.

Code Sample 9.2. `OC_analog_out.c` Using Timer2, OC3, and an RC low-pass filter to create analog output.

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

#define PERIOD 1024        // this is PR2 + 1
#define MAXVOLTAGE 3.3     // corresponds to max high voltage output of PIC32

int getUserPulseWidth(void) {
  char msg[100] = {};
  float f = 0.0;

  sprintf(msg, "Enter the desired voltage, from 0 to %3.1f (volts): ", MAXVOLTAGE);
  NU32_WriteUART1(msg);
```

[2]Technically this yields 257 possible duty cycle levels, since OCxR = 0 corresponds to 0% duty cycle and OCxR = 256 corresponds to 100% duty cycle.

```
  NU32_ReadUART1(msg,10);
  sscanf(msg, "%f", &f);

                                  // clamp the input voltage to the appropriate range
  if (f > MAXVOLTAGE) {
    f = MAXVOLTAGE;
  } else if (f < 0.0) {
    f = 0.0;
  }

  sprintf(msg, "\r\nCreating %5.3f volts.\r\n", f);
  NU32_WriteUART1(msg);
  return PERIOD * (f / MAXVOLTAGE);  // convert volts to counts
}

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

  PR2 = PERIOD - 1;       // Timer2 is the base for OC3, PR2 defines PWM frequency, 78.125 kHz
  TMR2 = 0;               // initialize value of Timer2
  T2CONbits.ON = 1;       // turn Timer2 on, all defaults are fine (1:1 divider, etc.)
  OC3CONbits.OCTSEL = 0;  // use Timer2 for OC3
  OC3CONbits.OCM = 0b110; // PWM mode with fault pin disabled
  OC3CONbits.ON = 1;      // Turn OC3 on
  while (1) {
    OC3RS = getUserPulseWidth();
  }
  return 0;
}
```

9.4 Chapter Summary

- Output Compare modules pair with Timer2, Timer3, or the 32-bit Timer23 to generate a single timed pulse or a continuous pulse train with controllable duty cycle. Microcontrollers commonly control motors using pulse-width modulation (PWM) to drive H-bridge amplifiers that power the motors.

- Low-pass filtering of PWM signals, perhaps using an RC filter with a cutoff frequency f_c, allows the generation of analog outputs. There is a fundamental tradeoff between the resolution of the analog output and the maximum possible frequency component f_a of the generated analog signal. If the PWM frequency is f_{PWM}, then generally the frequencies should satisfy $f_{PWM} \gg f_c \gg f_a$.

9.5 Exercises

1. Enforce the constraints $f_{PWM} \geq 100 f_c$ and $f_c \geq 10 f_a$. Given that PBCLK is 80 MHz, provide a formula for the maximum f_a given that you require n bits of resolution in your DC analog voltage outputs. Provide a formula for RC in terms of n.

2. You will use PWM and an RC low-pass filter to create a time-varying analog output waveform that is the sum of a constant offset and two sinusoids of frequency f and kf, where k is an integer greater than 1. The PWM frequency will be 10 kHz and f satisfies 50 Hz $\geq f \geq$ 10 Hz. Use OC1 and Timer2 to create the PWM waveform, and set PR2 to 999 (so the PWM waveform is 0% duty cycle when OC1R = 0 and 100% duty cycle when OC1R = 1000). You can break this program into the following pieces:

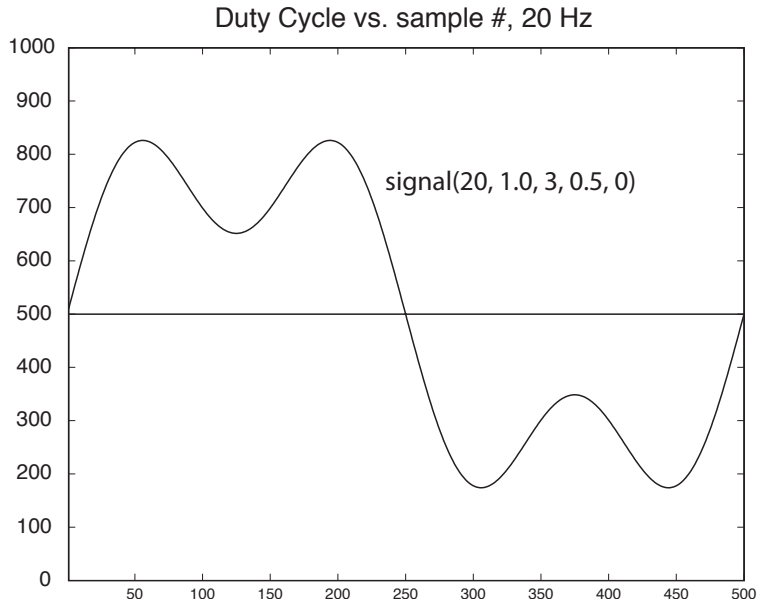


Figure 9.5: An example analog output waveform from Problem 2, plotted as the duration 0 to 1000 of the high portion of the PWM waveform, which has a period of 1000.

(a) Write a function that forms a sampled approximation of a single period of the waveform

$$V_{\text{out}}(t) = C + A_1 \sin(2\pi f t) + A_2 \sin(2\pi k f t + \phi),$$

where the constant C is 1.65 V (half of the full range 0 to 3.3 V), A_1 is the amplitude of the sinusoid at frequency f, A_2 is the amplitude of the sinusoid at frequency kf, and ϕ is the phase offset of the higher frequency component. Typically values of A_1 and A_2 would be 1 V or less so the analog output is not saturated at 0 or 3.3 V. The function takes A_1, A_2, k, f, and ϕ as input and creates an array `dutyvec`, of appropriate length, where each entry is a value 0 to 1000 corresponding to the voltage range 0 to 3.3 V. Each entry of `dutyvec` corresponds to a time increment of 1/10 kHz = 0.1 ms, and `dutyvec` holds exactly one cycle of the analog waveform, meaning that it has $n = 10$ kHz$/f$ elements. A Matlab implementation is given below. You can experiment plotting waveforms or just use the function for reference. A reasonable call of the function is `signal(20, 0.5, 2, 0.25, 45)`, where the phase 45 is in degrees. An example waveform is shown in Figure 9.5.

```
function signal(BASEFREQ,BASEAMP,HARMONIC,HARMAMP,PHASE)

% This function calculates the sum of two sinusoids of different
% frequencies and populates an array with the values.  The function
% takes the arguments
%
% * BASEFREQ:  the frequency of the low frequency component (Hz)
% * BASEAMP:   the amplitude of the low frequency component (volts)
% * HARMONIC:  the other sinusoid is at HARMONIC*BASEFREQ Hz; must be
%              an integer value > 1
% * HARMAMP:   the amplitude of the other sinusoid (volts)
% * PHASE:     the phase of the second sinusoid relative to
%              base sinusoid (degrees)
%
% Example matlab call:  signal(20,1,2,0.5,45);
```

```
% some constants:

MAXSAMPS = 1000;     % no more than MAXSAMPS samples of the signal
ISRFREQ = 10000;     % frequency of the ISR setting the duty cycle; 10kHz

% Now calculate the number of samples in your representation of the
% signal; better be less than MAXSAMPS!

numsamps = ISRFREQ/BASEFREQ;
if (numsamps>MAXSAMPS)
  disp('Warning: too many samples needed; choose a higher base freq.');
  disp('Continuing anyway.');
end
numsamps = min(MAXSAMPS,numsamps);  % continue anyway

ct_to_samp = 2*pi/numsamps;         % convert counter to time
offset = 2*pi*(PHASE/360);          % convert phase offset to signal counts

for i=1:numsamps  % in C, we should go from 0 to NUMSAMPS-1
  ampvec(i) = BASEAMP*sin(i*ct_to_samp) + ...
              HARMAMP*sin(HARMONIC*i*ct_to_samp + offset);
  dutyvec(i) = 500 + 500*ampvec(i)/1.65; % duty cycle values,
                                         % 500 = 1.65 V is middle of 3.3V
                                         % output range
  if (dutyvec(i)>1000) dutyvec(i)=1000;
  end
  if (dutyvec(i)<0) dutyvec(i)=0;
  end
end

% ampvec is in volts; dutyvec values are in range 0...1000

plot(dutyvec);
hold on;
plot([1 1000],[500 500]);
axis([1 numsamps 0 1000]);
title(['Duty Cycle vs. sample #, ',int2str(BASEFREQ),' Hz']);
hold off;
```

(b) Write a function using the NU32 library that prompts the user for A_1, A_2, k, f, and ϕ. The array `dutyvec` is then updated based on the input.

(c) Use Timer2 and OC1 to create a PWM signal at 10 kHz. Enable the Timer2 interrupt, which generates an IRQ at every Timer2 rollover (10 kHz). The ISR for Timer2 should update the PWM duty cycle with the next entry in the `dutyvec` array. When the last element of the `dutyvec` array is reached, wrap around to the beginning of `dutyvec`. Use the shadow register set for the ISR.

(d) Choose reasonable values for RC for your RC filter. Justify your choice.

(e) The main function of your program should sit in an infinite loop, asking the user for new parameters. In the meantime, the old waveform continues to be "played" by the PWM. For the values given in Figure 9.5, use your oscilloscope to confirm that your analog waveform looks correct.

Your code will be graded on organization, comments, simplicity/elegance, and correctness. Turn in your C file for testing.

Chapter 10

Analog Input

The PIC32 has a single analog-to-digital converter (ADC) that, through the use of multiplexers, can be used to sample the analog voltage at up to 16 different pins (Port B). The ADC has 10-bit resolution, which means it can distinguish $2^{10} = 1024$ different voltage values, usually in the range 0 to 3.3 V, the voltage used to power the PIC32. This yields 3.3 V/1024 = 3 mV resolution. The ADC can take up to one million analog readings per second. The ADC is typically used in conjunction with sensors that produce analog voltage values.

10.1 Overview

A block diagram of the ADC peripheral, adapted from the Reference Manual, is shown in Figure 10.1. There is a lot going on in this figure, but let's start at the differencing amp. Some control logic (determined by SFRs) selects the + input of the differencing amp from the analog input pins AN0 to AN15. Other control logic selects the − input to be either AN1 or V_{REFL}, the low reference voltage to the ADC selected by the bits VCFG. (This reference V_{REFL} can be chosen to be either V_{REF-}, a voltage provided on an external pin, or AV_{SS}, the PIC32's GND line, also known as V_{SS}.) For proper operation, the − input voltage V_{INL} should be less than or equal to the + input voltage V_{INH}.

The difference between the two input voltages, $V_{SHA} = V_{INH} - V_{INL}$, is sent to the Sample and Hold Amplifier (SHA). During the *sampling* (or *acquisition*) stage, a 4.4 pF internal holding capacitor is charged or discharged to hold the voltage difference V_{SHA}. Once the sampling period has ended, the SHA is disconnected from the inputs. This allows V_{SHA} to be constant during the *conversion* stage, even if the voltages on the inputs are changing. The Successive Approximation Register (SAR) ADC converts V_{SHA} to a 10-bit result depending on its relationship to the low and high reference voltages V_{REFL} and V_{REFH}. V_{REFL} was mentioned earlier, and V_{REFH} can be chosen from a voltage reference on the external pin V_{REF+} or from the PIC32's power supply rail AV_{DD} (also called V_{DD}). If $V_{SHA} = V_{REFL}$, the 10-bit result is 0. If $V_{SHA} = V_{REFH}$, the 10-bit result is $2^{10} - 1 = 1023$. For a voltage V_{SHA} that is x% of the way from V_{REFL} to V_{REFH}, the 10-bit result is 1023 × x/100. (See the Reference Manual for more details on the ADC transfer function.) The 10-bit conversion result is written to the buffer ADC1BUF which is read by your program. If you don't read the result right away, ADC1BUF can store up to 16 results (in the SFRs ADC1BUF0, ADC1BUF1, ..., ADC1BUFF) before the ADC begins overwriting old results.

Sampling and Conversion Timing The two main stages of an ADC read are sampling/acquisition and conversion. During the sampling stage, we must allow sufficient time for the internal holding capacitor to converge to the difference $V_{INH} - V_{INL}$. According to the Electrical Characteristics section of the Data Sheet, this is 132 ns when the SAR ADC uses the external voltage references V_{REF-} and V_{REF+} as its low and high references. The minimum sampling time is 200 ns when using AV_{SS} and AV_{DD} as the low and high references.

Once the sampling stage has concluded, the SAR ADC requires 12 ADC clock cycles to accomplish the conversion: one cycle for each of the 10 bits, plus two more. This can be understood by the fact that the

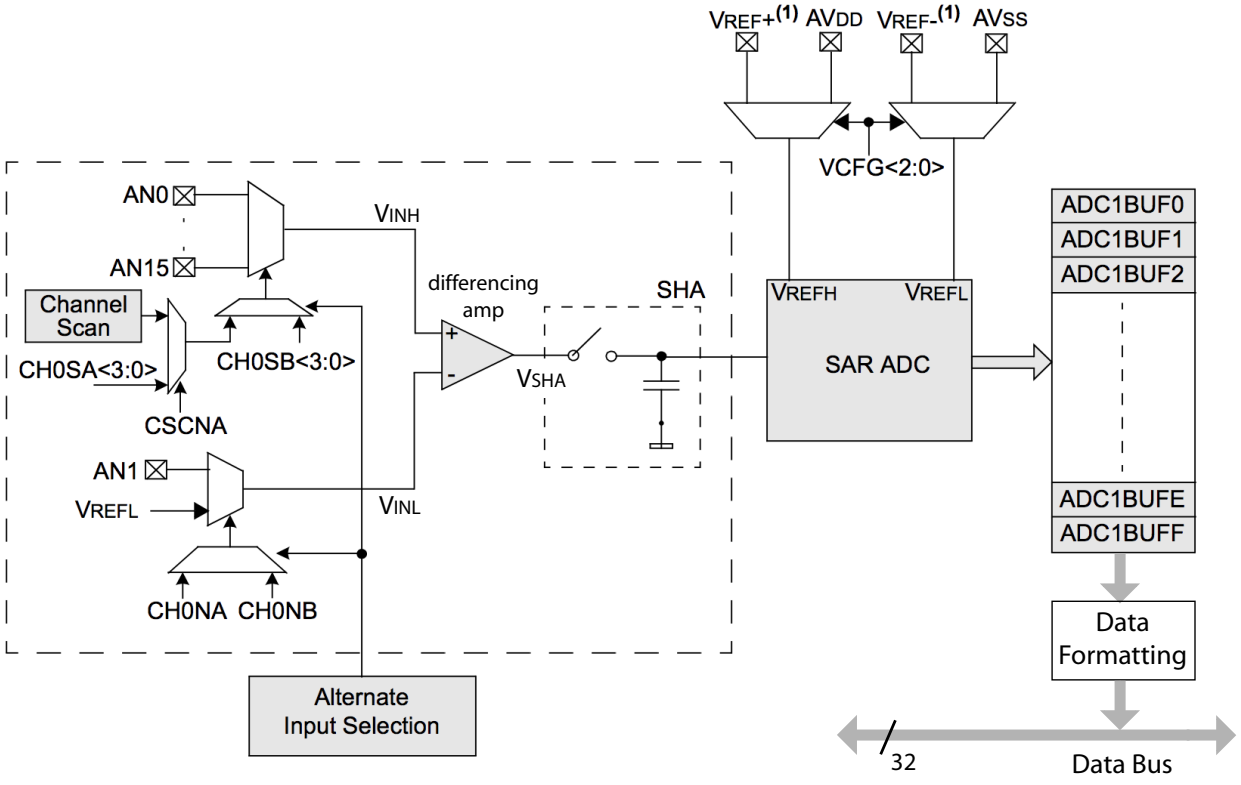


Figure 10.1: A simplified schematic of the ADC module.

ADC uses successive approximation to find the digital representation of the voltage. In this method, V_{SHA} is iteratively compared to a test voltage produced by an internal digital-to-analog converter (DAC). The DAC takes a 10-bit number and produces a test voltage in the range [V_{REFL}, V_{REFH}], where 0x000 produces V_{REFL} and 0x3FF produces V_{REFH}. In the first cycle of the conversion process, the test value to the DAC is 0x200 = 0b1000000000, which produces a voltage in the middle of the reference voltage range. If V_{SHA} is greater than this DAC voltage, the first bit of the conversion result is 1, otherwise it is zero. On the next cycle, the DAC's most significant bit is set to the result from the first test, and the second most significant bit is set to 1 for the next test. The process continues until all 10 bits of the result are determined. This process is a binary search. The entire process takes 10 cycles, plus 2 more, or 12 ADC clock cycles.

The ADC clock is derived from PBCLK. According to the Electrical Characteristics section of the Data Sheet, the ADC clock period (TAD, or Tad) must be at least 65 ns to allow time for the conversion of a single bit. The ADC SFR AD1CON3 allows us to choose the ADC clock period as 2 × k × Tpb, where Tpb is the PBCLK period and k is any integer from 1 to 256. Since Tpb is 12.5 ns for the NU32, to meet this specification, we can choose k = 3, or Tad = 75 ns.

The minimum time between samples is the sum of the sampling time and the conversion time. If the ADC is set up to take samples automatically, we have to choose the sampling time to be an integer multiple of Tad. The shortest time we can choose is 2 × Tad = 150 ns to satisfy the 132 ns minimum sampling time. Thus the fastest we can read from an analog input is

```
minimum read time = 150 ns + 12 * 75 ns = 1050 ns
```

or just over 1 microsecond. We can take almost a million samples per second, theoretically.

Multiplexers Two multiplexers determine which of the analog input pins to connect to the differencing amp. These two multiplexers are called MUX A and MUX B. MUX A is the default active multiplexer, and the SFR AD1CON3 contains CH0SA bits that determine which of AN0-AN15 is connected to the + input and CH0NA bits that determine which of AN1 and V_{REF-} is connected to the − input. It is possible to alternate between MUX A and MUX B, but you are unlikely to need this.

Options The ADC peripheral provides a bewildering array of options, some of which are described here. No need to remember them all! The sample code should provide a good starting point.

- Data format: The result of the conversion is stored in a 32-bit word, and it can be represented as a signed integer, unsigned integer, fractional value, etc. Typically we would use a 16-bit or 32-bit unsigned integer.
- Sampling and conversion initiation events: Sampling can be initiated by a software command, or immediately after the previous conversion has completed (auto sample). Conversion can be initiated by (1) a software command, (2) the expiration of a specified sampling period (auto convert), (3) a period match with Timer3, or (4) a signal change on the INT0 pin. If sampling and conversion are being done automatically (not through software commands), the conversion results will be placed in the ADC1BUF in successively higher addresses, before returning to the first address in ADC1BUF after a specified number of conversions.
- Input scan and alternating modes: You can read in a single analog input at a time, you can scan through a list of analog inputs using MUX A, or you can alternate between two inputs, one from MUX A and one from MUX B.
- Voltage reference: The ADC can be configured to use reference voltages 0 and 3.3 V (the power rails of the PIC32). If you are interested in voltages in a different range, say 1.0 V to 2.0 V, for example, you can instead set up the ADC so 0x000 corresponds to 1.0 V and 0x3FF corresponds to 2.0 V, to get better resolution in this smaller range: $(2\text{ V} - 1\text{ V})/1024 = 1$ mV resolution. These reference voltage limits are V_{REF-} and V_{REF+} and must be provided to the PIC32 externally. (These must be limited to the range 0 to 3.3 V.)
- Unipolar differential mode: Any of the analog inputs (say AN5) can be compared to AN1, so you read the difference between the voltage on AN5 and AN1 (where the voltage on AN5 should not be less than the voltage on AN1).
- Interrupts: An interrupt may be generated after a specified number of conversions.
- ADC clock period: The ADC clock period Tad can range from 2 times the PB clock period up to 512 times the PB clock period, in integer multiples of two. You may also choose Tad to be the period of the ADC internal RC clock.
- Dual buffer mode for reading and writing conversion results: When an ADC conversion is complete, it is written into the output buffer ADC1BUF. After a series of one or more conversions is complete, an interrupt flag is set, indicating that the results are available for the program to read. If the program is too slow to respond, however, the next set of conversions may begin to overwrite the previous results. To make this less likely, we can divide the 16-word ADC1BUF into two buffers, each consisting of 8 words: one in which the current conversions are being written, and one from which the program should read the previous results.

10.2 Details

The operation of the ADC peripheral is determined by the following SFRs:

AD1PCFG Only the least significant 16 bits are relevant. If a bit is 0, the associated pin on port B is configured as an analog input. If a bit is 1, it is digital I/O. The default on reset is all zeroes (analog inputs).

AD1CON1 Determines whether the ADC is on or off; the output format of the conversion; the "start conversion" signal generator; whether the ADC continually does conversions or just does one sequence of samples; and whether sampling begins again immediately after the previous conversion ends, or waits for a signal from the user. Also indicates if the most recent conversion is finished.

AD1CON2 Determines the voltage references for the ADC (positive reference could be 3.3 V or V_{REF+}, negative reference could be GND or V_{REF-}); whether or not inputs will be scanned; whether MUX A and MUX B will be used in alternating mode; whether dual buffer mode is selected; and the number of conversions to be done before generating an interrupt.

AD1CON3 Determines whether Tad is generated from the ADC internal RC clock or the PB clock; the number of Tad cycles to sample the signal; and the number of Tad cycles allowed for conversion of each bit (must be at least 65 ns).

AD1CHS This SFR determines which pins will be sampled (the "positive" inputs) and what they will be compared to (i.e., V_{REF-} or AN1). When in scan mode, the sample pins specified in this SFR are ignored.

AD1CSSL Bits set to 1 in this SFR indicate which analog inputs will be sampled in scan mode (if AD1CON2 has configured the ADC for scan mode). Inputs will be scanned from lower number inputs to higher numbers.

Apart from these SFRs, the ADC module has bits associated with the ADC interrupt in IFS1bits.AD1IF, IEC1bits.AD1IE, IPC6bits.AD1IP, and IPC6bits.AD1IS.

For more details, see the Reference Manual.

10.3 Sample Code

10.3.1 Manual Sampling and Conversion

There are many ways to read the analog inputs, but the sample code below is perhaps the simplest. This code reads in analog inputs AN14 and AN15 every half second and sends their values to the user's terminal. It also logs the time it takes to do the two samples and conversions, which is a bit under 5 microseconds total. In this program we set the ADC clock period Tad to be 6 × Tpb = 75 ns, and the acquisition time to be at least 250 ns. There are two places in this program where we are just sitting around waiting: during the sampling and during the conversion. If speed were an issue, we could use more advanced settings to let the ADC do the work in the background and let us know via an interrupt when the samples are ready.

Code Sample 10.1. `ADC_Read2.c` Reading two analog inputs with manual initialization of sampling initialization and conversion.

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

#define VOLTS_PER_COUNT (3.3/1024)
#define CORE_TICK_TIME 25    // nanoseconds between core ticks
#define SAMPLE_TIME 10       // 10 core timer ticks = 250 ns
#define DELAY_TICKS 20000000 // delay 1/2 sec, 20 M core ticks, between messages

unsigned int adc_sample_convert(int pin) { // sample and convert the value on the given adc pin
                                           // the pin should be configured as an analog input in
                                           // AD1PCFG
    unsigned int elapsed = 0, finish_time = 0;
    AD1CHSbits.CH0SA = pin;                // connect pin AN14 to MUXA for sampling
    AD1CON1bits.SAMP = 1;                  // start sampling
    elapsed = _CP0_GET_COUNT();
```

```
    finish_time = elapsed + SAMPLE_TIME;
    while (_CP0_GET_COUNT() < finish_time) {
      ;                                 // sample for more than 250 ns
    }
    AD1CON1bits.SAMP = 0;               // stop sampling and start converting
    while (!AD1CON1bits.DONE) {
      ;                                 // wait for the conversion process to finish
    }
    return ADC1BUF0;                    // read the buffer with the result
}

int main(void) {
  unsigned int sample14 = 0, sample15 = 0, elapsed = 0;
  char msg[100] = {};

  NU32_Startup();    // cache on, min flash wait, interrupts on, LED/button init, UART init
  AD1PCFGbits.PCFG14 = 0;               // AN14 is an adc pin
  AD1PCFGbits.PCFG15 = 0;               // AN15 is an adc pin
  AD1CON3bits.ADCS = 2;                 // ADC clock period is Tad = 2*(ADCS+1)*Tpb =
                                        //                           2*3*12.5ns = 75ns
  AD1CON1bits.ADON = 1;                 // turn on A/D converter
  while (1) {
    _CP0_SET_COUNT(0);                  // set the core timer count to zero
    sample14 = adc_sample_convert(14);  // sample and convert pin 14
    sample15 = adc_sample_convert(15);  // sample and convert pin 15
    elapsed = _CP0_GET_COUNT();         // how long it took to do two samples
                                        // send the results over serial
    sprintf(msg, "Time elapsed: %5u ns  AN14: %4u (%5.3f volts)"
                 "AN15: %4u (%5.3f volts) \r\n",
                 elapsed * CORE_TICK_TIME,
                 sample14, sample14 * VOLTS_PER_COUNT,
                 sample15, sample15 * VOLTS_PER_COUNT);
    NU32_WriteUART1(msg);
    _CP0_SET_COUNT(0);                  // delay to prevent a flood of messages
    while(_CP0_GET_COUNT() < DELAY_TICKS) {
      ;
    }
  }
  return 0;
}
```

10.3.2 Maximum Possible Sample Rate

The program `ADC_max_rate.c` reads from a single analog input, AN0, at the maximum speed that fits the PIC32 Electrical Characteristics and the 80 MHz PBCLK (Tpb = 12.5 ns). We choose

```
Tad = 6 * Tpb = 75 ns
```

as the smallest time that is an even integer multiple of Tpb and greater than the 65 ns required in the Electrical Characteristics section of the Data Sheet. We choose the sample time to be

```
Tsamp = 2 * Tad = 150 ns,
```

the smallest integer multiple of Tad that meets the minimum spec of 132 ns in the Data Sheet. The ADC is set up to auto-sample and auto-convert 8 samples, then generate an interrupt. The ISR reads in 8 samples from ADCBUF0-7 or ADCBUF8-F while the ADC is busily filling the other 8-word section. The ISR must

finish reading in one 8-word section before the other 8-word section is filled. Otherwise, the ADC results will start overwriting the unread results.

After reading in 1000 samples, the ADC interrupt is disabled to free up the CPU from servicing the ISR, and the data points are written to the user's terminal, along with the average time it took to get each sample. Theoretically the time is 150 ns + (12 * 75 ns) = 1050 ns, but on average we get an extra 75 ns (6 Tpb) for 1125 ns. This is 888.89 kHz sampling.[1]

High-speed sampling requires that pin RA9/V_{REF-} be connected to ground and pin RA10/V_{REF+} be connected to 3.3 V. These are the external low and high voltage references for analog input in high speed mode. Actually, in the Reference Manual, Microchip says that V_{REF-} should be attached to ground through a 10 ohm resistor and V_{REF+} should be attached to two capacitors in parallel to ground (0.1 μF and 0.01 μF) as well as a 10 ohm resistor to 3.3 V.

To provide input to the ADC, this code sets up OC1 using Timer2 as a base to output a square wave at 5 kHz and 25% duty cycle. The program also uses Timer45 to time the duration between ISR entries. The first 1000 analog input samples are written to the screen, as well as the time they were taken, confirming that the samples correspond to 889 kHz sampling of a 5 kHz 25% duty cycle waveform. The ISR that reads eight samples from ADCBUF also toggles an LED once every million times it is entered, allowing the time to take eight million samples to be measured with a stopwatch (about nine seconds).

The output to your computer's screen should look something like

```
... (earlier output snipped)
928       1019     3.284
929       1015     3.271
930       1015     3.271
931       1015     3.271
932       1015     3.271
933          4     0.013
934          4     0.013
935          4     0.013    9.0000 us; 1.1250 us/read for last 8 reads
... (later output snipped)
```

showing the sample number, the ADC counts, and the corresponding actual voltage for samples 0 to 999. The high output voltage from OC1 is measured as approximately 3.27 V, and the low output voltage is measured as approximately 0.01 V. You can also see that the output is high for 45 consecutive samples and low for 135 consecutive samples, corresponding to the 25% duty cycle of OC1. In the snippet above, OC1's switch from high to low is measured at sample 933.

Code Sample 10.2. `ADC_max_rate.c` Reading a single analog input at the maximum possible rate to meet the Electrical Characteristics section of the Data Sheet, given that the PBCLK is 80 MHz.

```
// ADC_max_rate.c
//
// This program reads from a single analog input, AN0, at the maximum speed
// that fits the PIC32 Electrical Characteristics and the 80 MHz PBCLK
// (Tpb = 12.5 ns).  The input to AN0 is a 5 kHz 25% duty cycle PWM from
// OC1.  The results of 1000 analog input reads is sent to the user's
// terminal.  An LED on the NU32 also toggles every 8 million samples.
//
// RA9/VREF- must be connected to ground and RA10/VREF+ connected to 3.3 V.
//

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

[1]The Electrical Characteristics section of the Data Sheet lists 132 ns as the minimum sampling time for an analog input coming from a source with 500 ohm output impedance. If the source is a much lower output impedance, it may be possible to reduce the sampling time to, say, 1 * Tad = 75 ns. Accounting for the extra 75 ns, we get 1050 ns sample time or 952.38 kHz sampling. This seemed to work fine in tests with low impedance outputs.

```
#define NUM_ISRS 125                // the number of 8-sample ISR results to be printed
#define NUM_SAMPS (NUM_ISRS*8)      // the number of samples stored
#define LED_TOGGLE 1000000          // toggle the LED every 1M ISRs (8M samples)

// these variables are static because they are not needed outside this C file
// volatile because they are written to by ISR, read in main

static volatile int storing = 1;   // if 1, currently storing data to print; if 0, done
static volatile unsigned int trace[NUM_SAMPS];   // array of stored analog inputs
static volatile unsigned int isr_time[NUM_ISRS]; // time of ISRs from Timer45


void __ISR(_ADC_VECTOR, IPL7SRS) ADCHandler(void) { // interrupt every 8 samples
  static unsigned int isr_counter = 0; // the number of times the isr has been called
                                       // "static" means the variable maintains its value
                                       // in between function (ISR) calls
  static unsigned int sample_num = 0;  // current analog input sample number

  if (isr_counter <= NUM_ISRS) {
    isr_time[isr_counter] = TMR4;      // keep track of Timer45 time the ISR is entered
  }

  if (AD1CON2bits.BUFS) {              // 1=ADC now filling BUF8-BUFF, 0=filling BUF0-BUF7
    trace[sample_num++] = ADC1BUF0;    // all ADC samples must be read in, even
    trace[sample_num++] = ADC1BUF1;    // if we don't want to store them, so that
    trace[sample_num++] = ADC1BUF2;    // the interrupt can be cleared
    trace[sample_num++] = ADC1BUF3;
    trace[sample_num++] = ADC1BUF4;
    trace[sample_num++] = ADC1BUF5;
    trace[sample_num++] = ADC1BUF6;
    trace[sample_num++] = ADC1BUF7;
  }
  else {
    trace[sample_num++] = ADC1BUF8;
    trace[sample_num++] = ADC1BUF9;
    trace[sample_num++] = ADC1BUFA;
    trace[sample_num++] = ADC1BUFB;
    trace[sample_num++] = ADC1BUFC;
    trace[sample_num++] = ADC1BUFD;
    trace[sample_num++] = ADC1BUFE;
    trace[sample_num++] = ADC1BUFF;
  }
  if (sample_num >= NUM_SAMPS) {
    storing = 0;                       // done storing data
    sample_num = 0;                    // reset sample number
  }
  ++isr_counter;                       // increment ISR count
  if (isr_counter == LED_TOGGLE) {     // toggle LED every 1M ISRs (8M samples)
    LATAINV = 0x20;
    isr_counter = 0;                   // reset ISR counter
  }

  IFS1bits.AD1IF = 0;                  // clear interrupt flag
}

int main(void) {
  int i = 0, j = 0, ind = 0;      // variables used for indexing
  float tot_time = 0.0;           // time between 8 samples
```

```
  char msg[100] ={};              // buffer for writing messages to uart
  unsigned int prev_time = 0;     // used for calculating time differences

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

  __builtin_disable_interrupts(); // INT step 2: disable interrupts

                                  // configure OC1 to use T2 to make 5 kHz 25% DC
  PR2 = 15999;                    // (15999+1)*12.5ns = 200us period = 5kHz
  T2CONbits.ON = 1;               // turn on Timer2
  OC1CONbits.OCM = 0b110;         // OC1 is PWM with fault pin disabled
  OC1R = 4000;                    // hi for 4000 counts, lo for rest (25% DC)
  OC1RS = 4000;
  OC1CONbits.ON = 1;              // turn on OC1

                                  // set up Timer45 to count every pbclk cycle
  T4CONbits.T32 = 1;              // configure 32-bit mode
  PR4 = 0xFFFFFFFF;               // rollover at the maximum possible period, the default
  T4CONbits.TON = 1;              // turn on Timer45

                                  // INT step 3: configure ADC generating interrupts
  AD1PCFGbits.PCFG0 = 1;          //             make RB0/AN0 an analog input
  AD1CON3bits.SAMC = 2;           //             sample for 2 Tad
  AD1CON3bits.ADCS = 2;           //             Tad = 6*Tpb
  AD1CON2bits.VCFG = 3;           //             external Vref+ and Vref- for VREFH and VREFL
  AD1CON2bits.SMPI = 7;           //             interrupt after every 8th conversion
  AD1CON2bits.BUFM = 1;           //             adc buffer is two 8-word buffers
  AD1CON1bits.FORM = 0b100;       //             unsigned 32 bit integer output
  AD1CON1bits.ASAM = 1;           //             autosampling begins after conversion
  AD1CON1bits.SSRC = 0b111;       //             conversion starts when sampling ends
  AD1CON1bits.ON = 1;             //             turn on the ADC
  IPC6bits.AD1IP = 7;             // INT step 4: IPL7, to use shadow register set
  IFS1bits.AD1IF = 0;             // INT step 5: clear ADC interrupt flag
  IEC1bits.AD1IE = 1;             // INT step 6: enable ADC interrupt
  __builtin_enable_interrupts();  // INT step 7: enable interrupts at CPU

  TMR4 = 0;                       // start timer 4 from zero
  while(storing) {
    ;                             // wait until first NUM_SAMPS samples taken
  }
  IEC1bits.AD1IE = 0;             // disable ADC interrupt

  sprintf(msg,"Values of %d analog reads\r\n",NUM_SAMPS);
  NU32_WriteUART1(msg);
  NU32_WriteUART1("Sample #   Value   Voltage   Time");

  for (i = 0; i < NUM_ISRS; ++i) {// write out NUM_SAMPS analog samples
    for (j = 0; j < 8; ++j) {
      ind = i * 8 + j;            // compute the index of the current sample
      sprintf(msg,"\r\n%5d %10d %9.3f ", ind, trace[ind], trace[ind]*3.3/1024);
      NU32_WriteUART1(msg);
    }
    tot_time = (isr_time[i] - prev_time) *0.0125; // total time elapsed, in microseconds
    sprintf(msg,"%9.4f us; %6.4f us/read for last 8 reads",tot_time, tot_time/8.0);
    NU32_WriteUART1(msg);
    prev_time = isr_time[i];
  }
  NU32_WriteUART1("\r\n");
  IEC1bits.AD1IE = 1;             // enable ADC interrupt. won't print the information again,
```

```
                                    // but you can see the light blinking
  while(1) {
    ;
  }
  return 0;
}
```

10.4 Chapter Summary

- The ADC peripheral converts an analog voltage to a 10-bit digital value, where 0x000 corresponds to an input voltage at V_{REFL} (typically GND) and 0x3FF corresponds to an input voltage at V_{REFH} (typically 3.3 V). There is a single ADC on the PIC32, AD1, but it can be multiplexed to sample from any or all of the 16 pins on Port B.
- Getting an analog input is a two-step process: sampling and conversion. Sampling requires a minimum time to allow the sampling capacitor to stabilize its voltage. Once the sampling terminates, the capacitor is isolated from the input so its voltage does not change during conversion. The conversion process is performed by a Successive Approximation Register (SAR) ADC which carries out a 10-step binary search, comparing the capacitor voltage to a new reference voltage at each step.
- The ADC provides a huge array of options which are only touched on in this chapter. The sample code in this chapter provides a manual method for taking a single ADC reading in the range 0 to 3.3 V in just over 2 microseconds. For details on how to use other reference value ranges, sample and convert in the background and use interrupts to announce the end of a sequence of conversions, etc., consult the long section in the Reference Manual.

10.5 Exercises

Part IV

Mechatronics

Chapter 11

PID Feedback Control

The cruise control system on a car is an example of a feedback control system. The actual speed v of the car is measured by a sensor (e.g., the speedometer) to yield a sensed value s; the sensed value is compared to a reference speed r set by the driver; and the error $e = r - s$ is fed to a control algorithm that calculates a control for the motor that drives the throttle angle. This control therefore changes the car's speed v and the sensed speed s. The controller tries to drive the error $e = r - s$ to zero.

Figure 11.1 shows a block diagram for a typical feedback control system, like the cruise control system. The calculation of the error between the reference and the sensor value, as well as the control algorithm itself, is typically implemented by a computer (the PIC32 in our case). The controller produces a control signal that goes to the *plant*, which describes the dynamics of the system being controlled. The plant produces an output which is measured by a sensor. This is called a *closed-loop* control system because the sensor feedback causes the block diagram to form a closed loop.

In this chapter we will make the simplifying assumption that the sensor is perfect. Therefore, the sensed output s and the plant's actual output are the same thing.

A common test of a controller's performance is to start with the reference r equal to zero, then suddenly switch r to 1 and keep it constant for all time. (Equivalently, for the cruise control case, r could start at 50 mph then change suddenly to 51 mph.) This is called a *step input* and the resulting error $e(t)$ is called the *step error response*. Figure 11.2 shows a typical step error response, and it illustrates three key metrics by which controller performance is measured: overshoot, 2% settling time t_s, and steady-state error e_{ss}. The settling time t_s is the amount of time it takes for the error to settle to within 0.02 of its final value, and the steady-state error e_{ss} is the final error. A controller performs well if the system has a short settling time and zero (or small) overshoot, oscillation, and steady-state error.

Perhaps the most popular feedback control algorithm is the *proportional-integral-derivative* (PID) controller. Entire books are devoted to the analysis and design of PID controllers, but PID control can also be used effectively with a little intuition and experimentation. This chapter provides a brief introduction to help with that intuition.

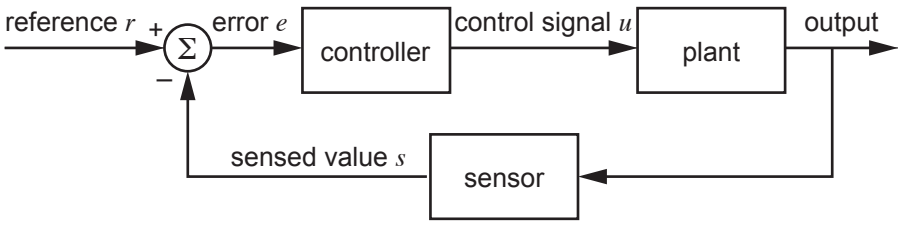


Figure 11.1: A block diagram of a typical control system.

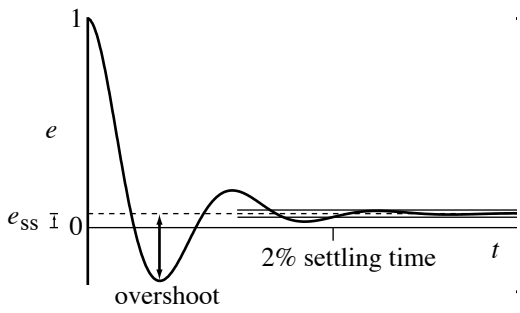


Figure 11.2: A typical error response to a step change in the reference, showing overshoot, 2% settling time, and steady-state error.

11.1 The PID Controller

Assume the reference signal is $r(t)$, the sensed output is $s(t)$, the error is $e(t) = r(t) - s(t)$, and the control signal is $u(t)$. Then the PID controller can be written

$$u(t) = K_p e(t) + K_i \int_0^t e(z)dz + K_d \dot{e}(t), \tag{11.1}$$

where K_p, K_i, and K_d are called the proportional, integral, and derivative gains, respectively. To make the discussion of the control law (11.1) concrete, let's assume r is the desired position of a mass moving on a line, s is the sensed position of the mass, and u is the linear force applied to the mass by a motor. Let's look at each of the proportional, derivative, and integral terms individually.

Proportional. The term $K_p e(t)$ creates a force proportional to the distance between the desired and measured position of the mass. This is exactly what a mechanical spring does: it creates a force that pulls or pushes the mass proportional to a position displacement. Thus the proportional term $K_p e$ acts like a spring with a rest length of zero, with one end attached to the mass at s and the other end attached to the desired position r. The larger K_p is, the stiffer the virtual spring is.

Because we define the error as $e = r - s$, K_p should be positive. If K_p were negative, then in the case $s > r$ (the mass's position s is "ahead" of the reference r), the force $K_p(r - s) > 0$ would try to push the mass even further ahead of the reference. Such a controller is called *unstable*, as the actual error tends toward infinity, not zero.

Derivative. The term $K_d \dot{e}(t)$ creates a force proportional to $\dot{e}(t) = \dot{r}(t) - \dot{s}(t)$, the difference between the desired velocity $\dot{r}(t)$ and the measured velocity $\dot{s}(t)$. This is exactly what a mechanical damper does: it creates a force that tries to zero the relative velocity between its two ends. Thus the derivative term $K_d \dot{e}$ acts like a damper. An example of a spring and a damper working together is an automatic door closing mechanism: the spring pulls the door shut, but the damper acts against large velocities so the door does not slam. Derivative terms are used similarly in PID controllers, to damp overshoot and oscillation typical of mass-spring systems.

As with K_p, K_d should be nonnegative.

Integral. The term $K_i \int_0^t e(z)dz$ creates a force proportional to the time integral of the error. This term is less easily explained in terms of a mechanical analog, but we can still use a mechanical example. Assume the mass moves vertically in gravity with a gravitational force $-mg$ acting downward. If the goal is to hold the mass at a constant height r, then a controller using only proportional and derivative terms would bring the mass to rest with a nonzero error e satisfying $K_p e = mg$, the upward force needed to balance the gravitational force. (Note that the derivative term $K_d \dot{e}$ is zero when the mass is at rest.) By increasing the stiffness of K_p, the error e can be made small, but it can never be made zero—nonzero error is always needed for the motor to produce nonzero force.

Using an integral term allows the controller to produce a nonzero force even when the error is zero. Starting from rest with error $e = mg/K_p$, the time-integral of the error builds up. As a result, the integral term $K_i \int_0^t e(z)dz$ grows, pushing the mass upward toward r, and the proportional term $K_p e$ shrinks, due to the shrinking error e. Eventually, the term $K_i \int_0^t e(z)dz$ equals mg, and the mass comes to rest at r ($e = 0$). Thus the integral term can drive the steady-state error to zero in systems where proportional and derivative terms alone cannot.

As with K_p and K_d, K_i should be nonnegative.

It is possible to implement a PID controller purely in electronics using op amps. However, nearly all modern PID controllers are implemented digitally on computers.[1] Every `dt` seconds, the computer reads the sensor value and calculates a new control signal. The error derivative $\dot{e}$ becomes an error difference, and the error integral $\int_0^t e(z)dz$ becomes an error sum. Pseudocode for a digital implementation of PID control is given below.

```
eprev = 0;                        // initial "previous error" is zero
eint = 0;                         // initial error integral is zero
now = 0;                          // "now" tracks the elapsed time

every dt seconds do {
  s = readSensor();               // read sensor value
  r = referenceValue(now);        // get reference signal for time "now"
  e = r - s;                      // calculate the error
  edot = e - eprev;               // error difference
  eint = eint + e;                // error sum
  u = Kp*e + Ki*eint + Kd*edot;   // calculate the control signal
  sendControl(u);                 // send control signal to the plant
  eprev = e;                      // current error is now previous error
  now = now + dt;                 // update the "now" time
}
```

A few notes about the algorithm:

- **The timestep `dt` and delays.** In general, the shorter `dt` is, the better. If computing resources are limited, however, it is enough to know that the timestep `dt` should be significantly shorter than time constants associated with the dynamics of the plant. So if the plant is "slow," you can afford a longer `dt`, but if the system can go unstable quickly, a short `dt` is needed. The primary reason is that near the end of a control cycle, the control applied by the controller is in response to old sensor data. Control based on old measurements can cause the system to become unstable.

  For many robot control systems, `dt` is 1 ms.

- **Error difference and sum.** The pseudocode uses an error difference and an error sum. Instead, the error derivative could be approximated as `edot = (e - eprev)/dt` and the error integral could be approximated using `eint = eint + e*dt`. There is no need to do these extra divisions and multiplications, however, which simply scale the results. This scaling can be incorporated in the gains K_d and K_i.

- **Integer math vs. floating point math.** As we have seen, addition, subtraction, multiplication, and division of integer data types is much faster than with floating point types. If we wish to ensure that the control loop runs as quickly as possible, we should use integers where possible. If necessary, this can be done by scaling up gains and error signals to maintain good resolution while only using integer values, while also making sure that integer overflow does not occur during mathematical operations. After calculations, the control signal can be scaled back down to an appropriate range. For many applications, since the PID controller involves only a few adds, subtracts, and multiplies, integer math is not necessary.

[1] A digital PID controller can be viewed as a type of digital filter, as seen in Chapter ??.

control	output of plant	order	recommended controller
force	position of mass	2	PD, PID
force	velocity of mass	1	P, PI
current	voltage across capacitor		
current	brightness of LED	0	P, PI

Table 11.1: Recommended PID variants based on the order of the plant.

- **Control saturation.** There are practical limits on the control signal `u`. The function `sendControl(u)` enforces these limits. If the control calculation yields `u=100`, for example, but the maximum control effort available is 50, the value sent by `sendControl(u)` is 50. If large controller gains `Kp`, `Ki`, and `Kd` are used, the control signal may often be saturated at the limits.

- **Integrator anti-windup.** Imagine that the integrator error `eint` is allowed to build up to a large value. This creates a large control signal that tries to create error of the opposite sign, to try to dissipate the integrated error. To limit the oscillation caused by this effect, `eint` can be upper bounded. This can be implemented by adding two lines to the code above:

```
eint = eint + e;                        // error sum
if (eint > EINTMAX) eint = EINTMAX;     // ADDED:  integrator anti-windup
if (eint < -EINTMAX) eint = -EINTMAX;   // ADDED:  integrator anti-windup
```

  This is called *integrator anti-windup*. How to choose `EINTMAX` is a bit of an art, but a good rule of thumb is that `Ki*EINTMAX` should be no more than the maximum control effort available from the actuator.

- **Sensor noise, quantization, and filtering.** The sensor data take discrete or *quantized* values. If this quantization is coarse, or if the time interval `dt` is short, the error `e` is unlikely to change much from one cycle to the next, and therefore `edot = e - eprev` is likely to take only a small set of different values. This means `edot` is likely to be a jumpy, low-resolution signal. The sensor may also be noisy, adding to the jumpiness of the `edot` signal. Digital low-pass filtering, or averaging `edot` over several cycles, yields a smoother signal, at the expense of added delay from considering older `edot` values.

While the PID control algorithm is quite simple, the challenge is finding control gains that yield good performance. This is the topic of Section 11.3.

11.2 Variants of the PID Controller

Common variants of the PID controller are P, PI, and PD controllers. These are obtained by setting K_i and/or K_d equal to zero. Which variant to use depends on the performance specifications and the dynamics of the plant, particularly its *order*. The order of a plant is the number of integrations from the control signal to the output. For example, consider the case where the control is a force to drive a mass, and the objective is to control the position of the mass. The force directly creates an acceleration, and the position is obtained from two integrations of the acceleration. Hence this is a second-order system. For such a system, derivative control is helpful, to slow down the output as it approaches the desired value. For zeroth-order or first-order systems, derivative control is generally not needed. Integral control should always be considered if it is important to eliminate steady-state error.

A rough guide to choosing the PID variant is given in Table 11.1. While PID control can be effective for plants of order higher than two, we will not consider such systems, which can have unintuitive behavior. An example is controlling the endpoint location of a flexible robot link by controlling the torque at the joint. The bending modes of the link introduce more state variables, increasing the order of the system.

11.3 Empirical Gain Tuning

Although there is no substitute for analytic design techniques using a good model of the system, useful controllers can also be designed using empirical methods.

Empirical gain tuning is the art of experimenting with different control gains on the actual system and choosing gains that give a good step error response. Searching for good gains in the three-dimensional K_p-K_i-K_d space can be tricky, so it is best to be systematic and to obey a few rules of thumb:

- **Steady-state error.** If the steady-state error is too big, consider increasing K_p. If the steady-state error is still unacceptable, consider introducing a nonzero K_i. Be careful with K_i, though, as large K_i can destabilize the system.
- **Overshoot and oscillation.** If there is too much overshoot and oscillation, consider increasing the damping K_d.
- **Settling time.** If the settling time is too long, consider simultaneously increasing K_p and K_d.

It is a good idea to first get the best possible performance with simple P control ($K_i = K_d = 0$). Then, starting from your best K_p, if you are using PD or PID control, experiment with K_d and K_p simultaneously. Experimenting with K_i should be saved until last, when you have your best P or PD controller, as nonzero K_i can lead to unintuitive behavior and instability.

Assuming you won't break your system, you should experiment with a wide range of control gains. Figure 11.3 shows an example exploration of a PD gain space for a plant with unknown dynamics. The control gains are varied over a few orders or magnitude. For larger control gains, the actuator effort $u(t)$, indicated by dotted lines, is often saturated. As expected, as K_p increases, oscillation and overshoot increases while the steady-state error decreases. As K_d increases, oscillation and overshoot is damped.

There are practical limits to how large controller gains can be. Large controller gains, combined with noisy sensor measurements or long cycle times `dt`, can lead to instability. At the least they can result in controls that chatter between the actuator limits.

11.4 Model-Based Control

With a feedback controller, like the PID controller, no control signal is generated unless there is error. If you have a reasonable model of the system's dynamics, why wait for error before applying a control? Using a model to anticipate the control effort needed is called *feedforward control*, because it depends only on the reference trajectory $r(t)$, not sensor feedback. A model-based feedforward controller can be written as

$$u_{\text{ff}}(t) = f(r(t), \dot{r}(t), \ddot{r}(t), \ldots),$$

where $f(\cdot)$ is a model of the inverse plant dynamics that computes the control effort needed as a function of $r(t)$ and its derivatives (Figure 11.4).

Since feedforward control is not robust to inevitable model errors, it can be combined with PID feedback control to get the control law

$$u(t) = u_{\text{ff}}(t) + K_p e(t) + K_i \int_0^t e(z)dz + K_d \dot{e}(t). \tag{11.2}$$

In control of a robot arm, this control law is called *computed torque control*, where u is the set of joint torques and the model $f(\cdot)$ computes the joint torques needed given the desired joint angles, velocities, and accelerations.

A related control strategy is to use the reference trajectory $r(t)$ and the error $e(t)$ to calculate a desired change of state. For example, if the plant is a second-order system (the control u directly controls $\ddot{s}$), then the desired acceleration $\ddot{s}_d(t)$ of the plant output can be written as the sum of the planned acceleration $\ddot{r}(t)$ and a PID feedback term to correct for errors:

$$\ddot{s}_d(t) = \ddot{r}(t) + K_p e(t) + K_i \int_0^t e(z)dz + K_d \dot{e}(t).$$

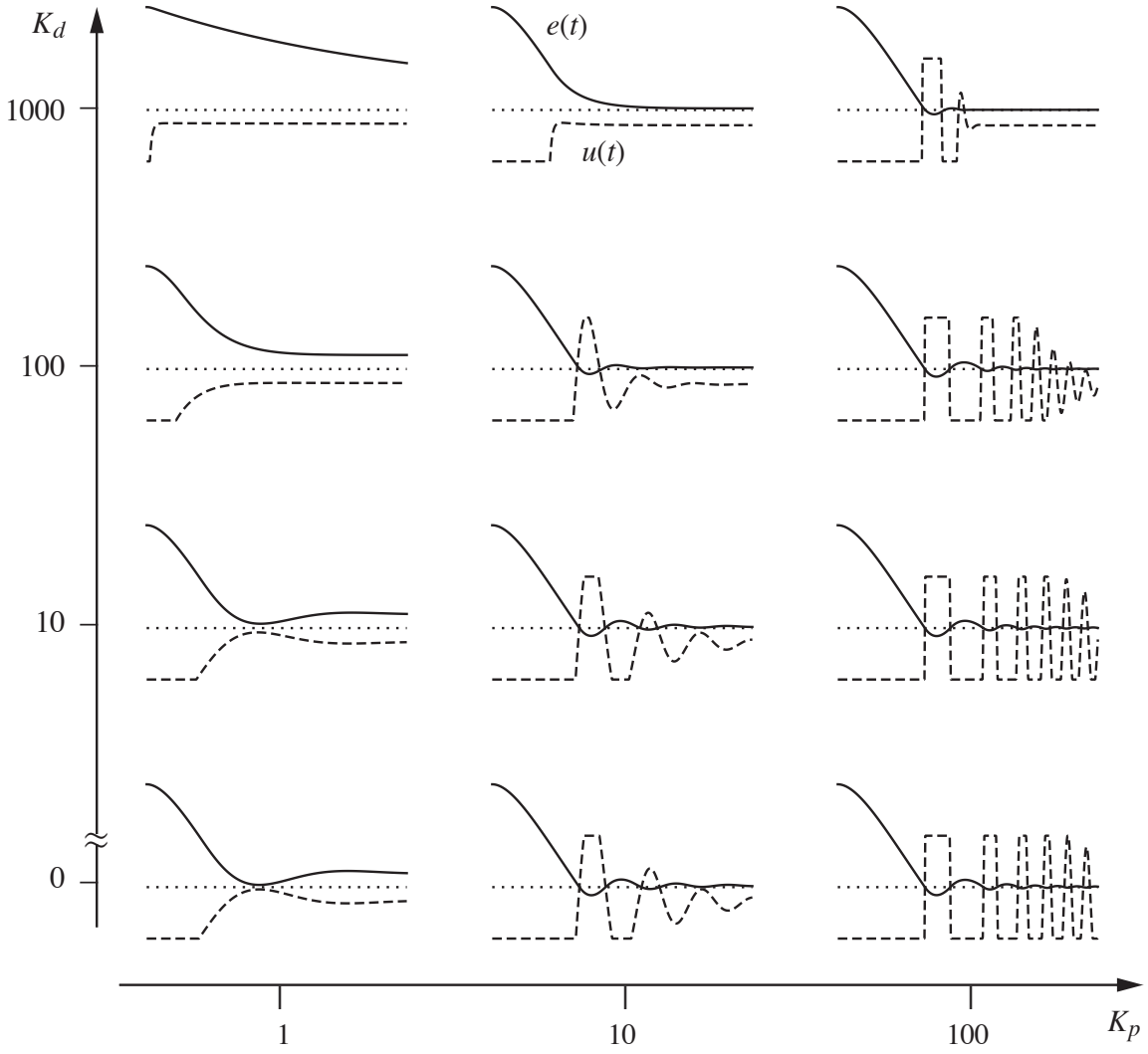


Figure 11.3: The step error response for a mystery system controlled by different PD controllers. The solid lines indicate the error response $e(t)$ and the dotted lines indicate the control effort $u(t)$. Which controller is "best?"

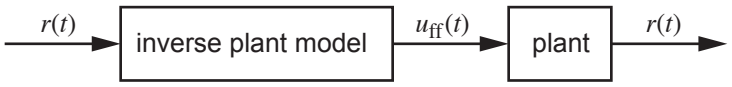


Figure 11.4: An ideal feedforward controller. If the inverse plant model is perfect, the output of the plant exactly tracks the reference $r(t)$.

Then the actual control $u(t)$ is calculated using the inverse model,

$$u(t) = f(s(t), \dot{s}(t), \ddot{s}_d(t)). \tag{11.3}$$

If the inverse model is good, an advantage of the control law (11.3) over (11.2) is that the effect of the constant PID gains is the same at different states of the plant. This can be important for systems like robot arms, where the inertia about a joint can change depending on the angle of outboard joints. For example, the inertia about your shoulder is large when your elbow is fully extended and smaller when your elbow is bent.

A shoulder PID controller designed for a bent elbow may not work so well when the elbow is extended if the output of the PID controller is a joint torque. By treating the PID terms as accelerations instead of joint torques, and by passing these accelerations through the inverse model, the shoulder PID controller should have the same performance regardless of the configuration of the elbow.[2]

Feedforward plus feedback control laws like (11.2) and (11.3) provide the advantage of smaller errors with less control effort as compared to feedback control alone. The cost is in developing a good model of the plant dynamics and in increased computation time for the controller.

11.5 Chapter Summary

- Performance of a control system is often evaluated by the overshoot, 2% settling time, and steady-state error of the step error response.
- The PID control law is $u(t) = K_p e(t) + K_i \int_0^t e(z)dz + K_d \dot{e}(t)$.
- The proportional gain K_p acts like a virtual spring and the derivative gain K_d acts like a virtual damper. The integral gain K_i can be useful for eliminating steady-state error, but large values of K_i may cause the system to become unstable.
- Common variants of PID control are P, PI, and PD control.
- To reduce steady-state error, K_p and K_i can be increased. To reduce overshoot and oscillation, K_d can be increased. To reduce settling time, K_p and K_d can be increased simultaneously. Stability considerations place practical limits on controller gains.
- Feedback control requires error to produce a control signal. Model-based feedforward control can be used in conjunction with feedback control to anticipate the controls needed, thereby reducing errors.

11.6 Exercises

1. Provided with this chapter is a simple Matlab model of a one-joint revolute robot arm moving in gravity. Perform empirical PID gain tuning by doing tests of the error response to a step input, where the step input asks the joint to move from $\theta = 0$ (hanging down in gravity) to $\theta = 1$ radian. Tests can be performed using

   ```
   pidtest(Kp, Ki, Kd)
   ```

   Find good gains K_p, K_i, and K_d, and turn in a plot of the resulting step error response. An example output of `pidtest(50, 0, 3000)` is given in Figure 11.5.

Code Sample 11.1. `pidtest.m`. Empirical gain tuning in Matlab for a simulated one-joint revolute robot in gravity.

```
function pidtest(Kp, Ki, Kd)

INERTIA = 0.5;        % The plant is a link attached to a revolute joint
MASS = 1;             % hanging in GRAVITY, and the output is the angle of the joint.
CMDIST = 0.1;         % The link has INERTIA about the joint, MASS center at CMDIST
DAMPING = 0.1;        % from the joint, and there is frictional DAMPING.
GRAVITY = 9.81;
DT = 0.001;           % timestep of control law
NUMSAMPS = 1001;      % number of control law iterations
UMAX = 20;            % maximum joint torque by the motor
```

[2] Provided the control effort does not saturate.

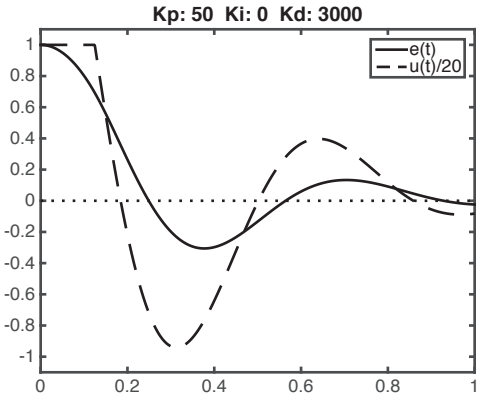


Figure 11.5: The step error and control response of the one-joint robot for $K_p = 50, K_i = 0, K_d = 3000$.

```
eprev = 0;
eint = 0;
r = 1;                % reference is constant at one radian
vel = 0;              % velocity of the joint is initially zero
s(1) = 0.0; t(1) = 0; % initial joint angle and time
for i=1:NUMSAMPS
  e = r - s(i);
  edot = e - eprev;
  eint = eint + e;
  u(i) = Kp*e + Ki*eint + Kd*edot;
  if (u(i) > UMAX)
    u(i) = UMAX;
  elseif (u(i) < -UMAX)
    u(i) = -UMAX;
  end
  eprev = e;
  t(i+1) = t(i) + DT;

  % acceleration due to control torque and dynamics
  acc = (u(i) - MASS*GRAVITY*CMDIST*sin(s(i)) - DAMPING*vel)/INERTIA;

  % a simple numerical integration scheme
  s(i+1) = s(i) + vel*DT + 0.5*acc*DT*DT;
  vel = vel + acc*DT;
end

plot(t(1:NUMSAMPS),r-s(1:NUMSAMPS),'Color','black');
hold on;
plot(t(1:NUMSAMPS),u/20,'--','Color','black');
set(gca,'FontSize',18);
legend({'e(t)','u(t)/20'},'FontSize',18);
plot([t(1),t(length(t)-1)],[0,0],':','Color','black');
axis([0 1 -1.1 1.1])
title(['Kp: ',num2str(Kp),'  Ki: ',num2str(Ki),'  Kd: ',num2str(Kd)]);
hold off
```

Chapter 12

Feedback Control of LED Brightness

In this project you will use feedback control to control the brightness of an LED. This project makes use of counter/timer, output compare, and analog input peripherals.

Figure 12.1 shows the LED and sensor circuits and their connection to the OC1 output and the AN0 analog input. A PWM waveform from OC1 turns the LED on and off at 20 kHz, too fast for the eye to see, yielding an apparent averaged brightness between off and full on. The phototransistor is activated by the LED's light, creating an emitter current proportional to the incident light. The resistor R turns this current into a sensed voltage. The 1 μF capacitor in parallel with R creates a low-pass filter with a time constant $\tau = RC$, filtering out high-frequency components due to the rapidly switching PWM signal and instead giving a time-averaged voltage. This is similar to the low-pass filtering of your visual perception, which does not allow you to see the LED turning on and off rapidly.

A block diagram of the control system is shown in Figure 12.2. The PIC32 reads the analog voltage from the phototransistor circuit, calculates the error as the desired brightness (in ADC counts) minus the measured voltage in ADC counts, and uses a proportional-integral (PI) controller to generate a new PWM duty cycle on OC1. This in turn changes the average brightness of the LED, which is sensed by the phototransistor circuit.

Your PIC32 program will generate a reference waveform, the desired light brightness (measured in ADC counts) as a function of time. Then a 1 kHz control loop will read the sensor voltage (in ADC counts) and update the duty cycle of the 20 kHz OC1 PWM signal in an effort to make the measured ADC counts track the reference waveform.

This is a significant project, so it is useful to break it down into smaller pieces and make sure each of those pieces functions properly.

12.1 Wiring and Testing the Circuit

1. **LED diode drop.** Connect the LED anode to 3.3 V, the cathode to a 330 Ω resistor, and the other end of the resistor to ground. This is the LED at its max brightness. Use your multimeter to record the forward bias voltage drop across the LED. Calculate or measure the current through the LED. Is this current safe for the PIC32 to provide?

2. **Choose R.** Wire up the circuit as shown in Figure 12.1, except for the connection from the LED to OC1. The LED and phototransistor should be pointing toward each other, with approximately one inch separation, as shown in Figure 12.3. Now choose R to be as small as possible while ensuring that the voltage V_{out} at the phototransistor emitter is close to 3 V when the LED anode is connected to 3.3 V (maximum LED brightness) and close to 0 V when the LED anode is disconnected (the LED is off). (Something in the 10 kΩ range may work, but use a smaller resistance if you can still get the same voltage swing.) Record your value of R. Now connect the anode of the LED to OC1 for the rest of the project.

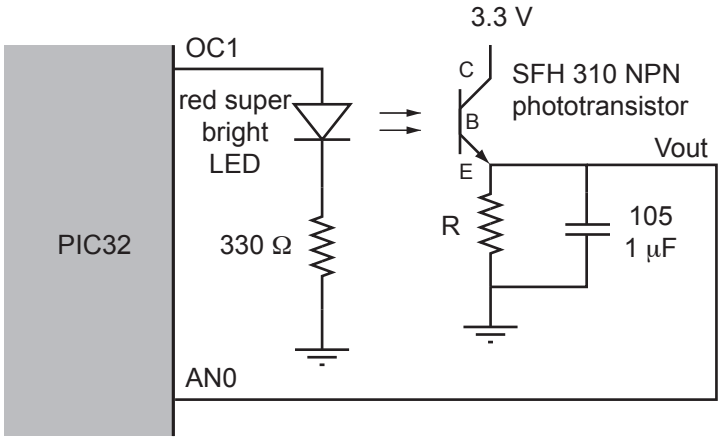


Figure 12.1: The LED control circuit. The long leg of the LED (anode) is connected to OC1 and the short leg (cathode) is connected to the 330 Ω resistor. The short leg of the phototransistor (collector) is attached to 3.3 V and the long leg (emitter) is attached to AN0, the resistor R, and the 1 μF capacitor.

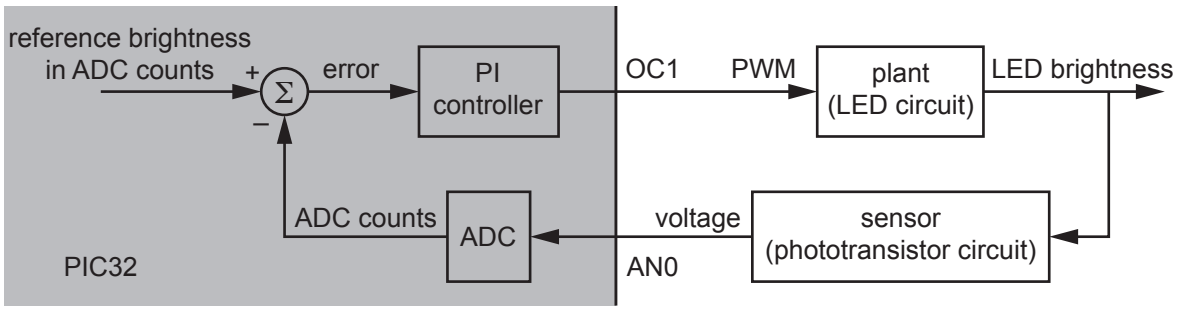


Figure 12.2: A block diagram of the LED brightness control system.

12.2 Powering the LED with OC1

1. **PWM calculation.** You will use Timer3 as the timer base for OC1. You want a 20 kHz PWM on OC1. Timer3 takes the PBCLK as input and uses a prescaler of 1. What should PR3 be?

2. **PWM program.** Write a program that uses your previous result to create a 20 kHz PWM output on OC1 (with no fault pin) using Timer3. Set the duty cycle to 75%. Get the following screenshots from your scope:

   (a) The OC1 waveform. Verify that this looks like what you expect.

   (b) The sensor voltage V_{out}.

   (c) Now remove the 1 μF capacitor and get another screenshot of V_{out}. Explain the difference from the previous waveform.

   Insert the 1 μF capacitor back into the circuit for the rest of the project.

12.3 Playing an Open-loop PWM Waveform

Now you will modify your program to generate a waveform stored in an `int` array. This will eventually be the reference brightness waveform for your feedback controller, but not yet. Modify your program to define a constant `NUMSAMPS` and the global `volatile int` array `Waveform` by putting the following code near the top of the C file (outside of `main`):

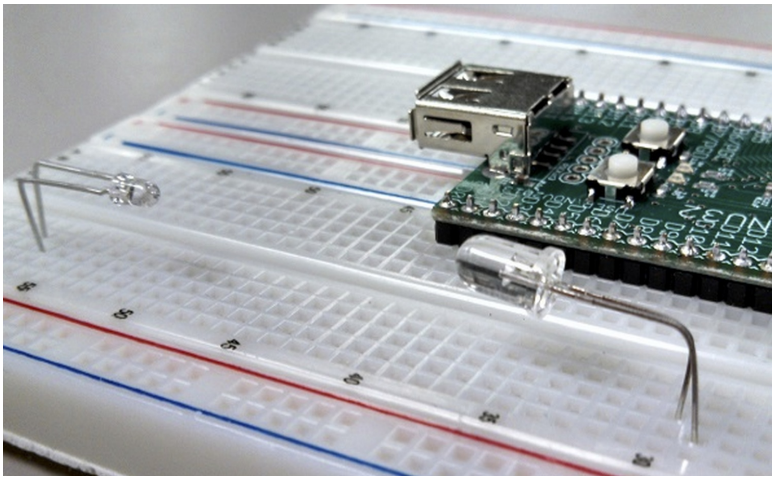


Figure 12.3: The phototransistor (left) and LED (right) pointed toward each other.

```
#define NUMSAMPS   1000             // number of points in waveform
volatile int Waveform[NUMSAMPS];    // waveform
```

Then create a function `makeWaveform()` to store a square wave in `Waveform[]` and call it near the beginning of `main`. The square wave has amplitude `A` centered about the value `center`. Initialize `center` as (PR3+1)/2 and `A` as (PR3+1)/4, for the PR3 you calculated in the previous section.

```
void makeWaveform() {
  int i, center=???, A=???; // square wave, fill in the center value and amplitude
  for (i=0; i<NUMSAMPS; i++) {
    if (i<NUMSAMPS/2) Waveform[i] = center + A;
    else Waveform[i] = center - A;
  }
}
```

Now set up Timer2 to call an ISR at a fixed frequency of 1 kHz. This ISR will eventually implement the controller that reads the ADC and calculates the new duty cycle of the PWM. For now we will use it to modify the duty cycle according to the waveform in `Waveform[]`. Call the ISR `Controller` and make it interrupt priority level 5. It will make use of a `static` local `int` that counts up the number of entries into the ISR and resets after 1000 entries.[1] In other words, the ISR should be of the form

```
void __ISR(_TIMER_2_VECTOR, IPL5SOFT) Controller(void) {  // _TIMER_2_VECTOR = 8 (p32mx795f512l.h)
  static int counter = 0;               // initialize counter once

  // insert line(s) to set OC1RS

  counter++;                            // add one to counter every time ISR is entered
  if (counter==NUMSAMPS) counter = 0;   // roll the counter over when needed

  // insert line to clear interrupt flag
}
```

In addition to clearing the interrupt flag, your `Controller` ISR should simply set OC1RS to be equal to `Waveform[counter]`. Since your ISR is called every 1 ms, and the period of the square wave in `Waveform[]` is 1000 cycles, your PWM duty cycle will undergo a square wave period every 1 s. You should see your LED become bright and dim once each second.

[1] Recall that a `static` local variable is only initialized once, not upon every entry to the function, and the value of the variable is retained between calls of the function.

1. Get a screenshot of your scope trace of V_{out} showing 2-4 periods of what should be an approximately square-wave sensor reading.

2. Turn in your code.

12.4 Establishing Communication with Matlab

By establishing communication between your PIC32 and Matlab, the PIC32 gains access to Matlab's extensive scientific computing and graphics capabilities, and Matlab can use the PIC32 as a data acquisition and control device.

Let's start with a very simple example: using Matlab to communicate with the `talkingPIC.c` program from Chapter 1. You will use the Matlab code `talkingPIC.m` to talk to `talkingPIC` on your PIC32. The Matlab code `talkingPIC.m` is listed below; you only need to edit the first line with the name of the `PORTA` COM port from your `Makefile`.

Code Sample 12.1. `talkingPIC.m` Simple Matlab code to talk to `talkingPIC` on the PIC32.

```
port='COM3'; % Edit this with the correct name of your PORTA.

% Makes sure port is closed
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
fprintf('Opening port %s....\n',port);

% Defining serial variable
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware');

% Opening serial connection
fopen(mySerial);

% Writing some data to the serial port
fprintf(mySerial,'%f %d %d\n',[1.0,1,2])

% Reading the echo from the PIC32 to verify correct communication
data_read = fscanf(mySerial,'%f %d %d')

% Closing serial connection
fclose(mySerial)
```

All `talkingPIC.m` does is open a serial port, send three numerical values to the PIC32, receive the values back, and close the serial port. Run `talkingPIC` on your PIC32, then execute `talkingPIC.m` in Matlab.

1. Make sure you are able to communicate between `talkingPIC` on the PIC32 and `talkingPIC.m` in Matlab. Do not proceed further until you have verified correct communication.

12.5 Plotting Data in Matlab

Now that you have Matlab communication working, you will build on your code from Section 12.3 by sending your controller's reference and sensed ADC data to Matlab for plotting. This will be helpful for seeing how well your controller is working, for empirical PI gain tuning.

First let's add to your constants and global variables at the top of your program. The PIC32 program will send to Matlab `PLOTPTS` data points upon request from Matlab, where the constant `PLOTPTS` is set to 200. It is not necessary to record data from every control loop, so the program will record the data once every

DECIMATION times, where DECIMATION is 10. Since the control loop is running at 1000 Hz, data is collected at 1000 Hz/DECIMATION = 100 Hz.

We will also define the global int arrays ADCarray and REFarray to hold the values of the sensor signal and the reference signal. The int StoringData is a flag that indicates whether data is currently being collected. When it transitions from TRUE (1) to FALSE (0), this indicates that PLOTPTS data points have been collected and it is time to send ADCarray and REFarray to Matlab. Finally, Kp and Ki are global floats with the PI gains. All of the variables have the type specifier volatile because they are shared between the ISR and main.

So you should have the following constants and variables near the beginning of your program:

```
#define NUMSAMPS   1000             // number of points in waveform
#define PLOTPTS     200             // number of data points to plot
#define DECIMATION   10             // plot every 10th point

volatile int Waveform[NUMSAMPS];    // waveform
volatile int ADCarray[PLOTPTS];     // measured values to plot
volatile int REFarray[PLOTPTS];     // reference values to plot
volatile int StoringData = 0;       // if this flag = 1, currently storing plot data
volatile float Kp = 0, Ki = 0;      // control gains
```

You should also modify your main function to define these local variables near the beginning:

```
  char message[100];                // message to and from Matlab
  float kptemp = 0, kitemp = 0;     // temporary local gains
  int i = 0;                        // plot data loop counter
```

These local variables are used in the infinite loop in main, below. This loop is interrupted by the ISR at 1 kHz. The loop waits for a message from Matlab, which contains the new PI gains requested by the user. When a message is received, the gains from Matlab are scanned into the local variables kptemp and kitemp. Then interrupts are disabled, these local values are copied into the global gains Kp and Ki, and interrupts are reenabled. Interrupts are disabled while Kp and Ki are assigned to ensure that the ISR does not interrupt in the middle of these assignments, causing it to use the new value of Kp but the old value of Ki. In addition, local variables are used for the sscanf to avoid having the sscanf command during the interrupt disabled period, since sscanf can take a relatively longer time to execute than simple variable assignments. We want to keep the time that interrupts are disabled as brief as possible, to avoid interfering with the timing of our 1 kHz control loop.

The next thing that happens is that the flag StoringData is set to TRUE (1), to tell the ISR to begin storing data. The ISR sets StoringData to FALSE (0) when PLOTPTS data points have been collected, indicating that it is time to send the stored data to Matlab for plotting. Your infinite loop in main should be the following:

```
  while (1) {
    NU32_ReadUART1(message,99);     // wait for a message from Matlab
    sscanf(message, "%f %f", &kptemp, &kitemp);
    __builtin_disable_interrupts(); // keep ISR disabled as briefly as possible
    Kp = kptemp;                    // copy local variables to globals used by ISR
    Ki = kitemp;
   __builtin_enable_interrupts();   // only 2 simple C commands while ISRs disabled
    StoringData = 1;                // message to ISR to start storing data
    while (StoringData) {           // wait until ISR says data storing is done
      ; // do nothing
    }
    for (i=0; i<PLOTPTS; i++) {     // send plot data to Matlab
                                    // when first number sent = 1, Matlab knows we're done
      sprintf(message, "%d %d %d\r\n", PLOTPTS-i, ADCarray[i], REFarray[i]);
      NU32_WriteUART1(message);
    }
  }
```

Finally, you will need to write code in your ISR to record data when `StoringData` is TRUE. This code will make use of the new local `static int` variables `plotind`, `decctr`, and `adcval`. `plotind` is the index, 0 to `PLOTPTS-1`, of the next set of data to collect. `decctr` counts from 1 up to `DECIMATION` to implement the once-every-`DECIMATION` data storing. `adcval` is set to zero for now, until you start reading the ADC.

Your code should look like the following. You only need to insert lines to set OC1RS and to clear the interrupt flag.

```
void __ISR(_TIMER_2_VECTOR, IPL5SOFT) Controller(void) {
  static int counter = 0;               // initialize counter once
  static int plotind = 0;               // index for data arrays; counts up to PLOTPTS
  static int decctr = 0;                // counts to store data one every DECIMATION
  static int adcval = 0;                //

 // insert line(s) to set OC1RS

  if (StoringData) {
    decctr++;
    if (decctr == DECIMATION) {         // after DECIMATION control loops,
      decctr = 0;                       // reset decimation counter
      ADCarray[plotind] = adcval;       // store data in global arrays
      REFarray[plotind] = Waveform[counter];
      plotind++;                        // increment plot data index
    }
    if (plotind == PLOTPTS) {           // if max number of plot points plot is reached,
      plotind = 0;                      // reset the plot index
      StoringData = 0;                  // tell main data is ready to be sent to Matlab
    }
  }
  counter++;                            // add one to counter every time ISR is entered
  if (counter==NUMSAMPS) counter = 0;   // rollover counter over when end of waveform reached

  // insert line to clear interrupt flag
}
```

The Matlab code to communicate with the PIC32 is below. Load your new PIC32 code and, in Matlab, use a command like

```
data = pid_plot('COM3', 2.0, 1.0)
```

where you should replace `'COM3'` with your appropriate `PORTA` from your `Makefile`. The `2.0` is your K_p and the `1.0` is your K_i. Since your program doesn't do anything with the gains yet, it doesn't matter what gains you type. If all is working properly, Matlab should plot two cycles of your square wave duty cycle waveform and zero for your measured ADC value (which you haven't implemented yet).

Code Sample 12.2. `pid_plot.m` Matlab code to plot data from your PIC32 LED control program.

```
function data = pid_plot(port,Kp,Ki)
%   pid_plot plot the data from the pwm controller to the current figure
%
%   data = pid_plot(port,Kp,Ki)
%
%   Input Arguments:
%       port - the name of the com port.  This should be the same as what
%              you use in screen or putty in quotes ' '
%       Kp - proportional gain for controller
%       Ki - integral gain for controller
%   Output Arguments:
%       data - The collected data.  Each column is a time slice
```

```
%
%   Example:
%       data = pid_plot('/dev/ttyUSB0',1.0,1.0) (Linux)
%       data = pid_plot('/dev/tty.usbserial-00001014A',1.0,1.0) (Mac)
%       data = pid_plot('COM3',1.0,1.0) (PC)
%

%% Opening COM connection
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
fprintf('Opening port %s....\n',port);
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware');
fopen(mySerial); % opens serial connection
clean = onCleanup(@()fclose(mySerial)); % closes serial port when function exits

%% Sending Data
% Printing to matlab Command window
fprintf('Setting Kp = %f, Ki = %f\n', Kp, Ki);

% Writing to serial port
fprintf(mySerial,'%f %f\n',[Kp,Ki]);

%% Reading data
fprintf('Waiting for samples ...\n');

sampnum = 1; % index for number of samples read
read_samples = 10; % When this value from PIC32 equals 1, it is done sending data
while read_samples > 1
    data_read = fscanf(mySerial,'%d %d %d'); % reading data from serial port

    % Extracting variables from data_read
    read_samples=data_read(1);
    ADCval(sampnum)=data_read(2);
    ref(sampnum)=data_read(3);

    sampnum=sampnum+1; % incrementing loop number
end
data = [ref;ADCval]; % setting data variable

%% Plotting data
clf;
hold on;
t = 1:sampnum-1;
plot(t,ref);
plot(t,ADCval);
legend('Reference', 'ADC Value')
title(['Kp: ',num2str(Kp),'  Ki: ',num2str(Ki)]);
hold off;
end
```

---

1. Turn in a Matlab plot showing `pid_plot.m` is communicating with your PIC32 code.

12.6 Writing to the LCD Screen

1. Write the function `printGainsToLCD()`, and its function prototype `void printGainsToLCD(void);`. This function writes the gains `Kp` and `Ki` on your LCD screen, one per row, like

   ```
   Kp:  12.30
   Ki:   1.00
   ```

   This function should be called by `main` just after `StoringData` is set to 1. Verify that it works before continuing to the next section.

12.7 Reading the ADC

1. Read in the ADC value in your ISR, just before the `if (StoringData)` line of code. The value should be called `adcval`, so it will be stored in `ADCarray`. Turn in a Matlab plot showing the measured `ADCarray` and the `REFArray`.

12.8 PI Control

Now you will implement the PI controller. Change `makeWaveform` so that `center` is 500 and the amplitude `A` is 300, making a square wave swinging between `200` and `800`. This waveform is now the desired brightness of the LED, in ADC counts. Use the `adcval` read in from the ADC and the reference waveform as inputs to the PI controller. Call `u` the output of the PI controller.

The output `u` may take positive or negative values, but we know that the PWM duty cycle can only be between 0 and 100%. If we treat the value `u` as a percentage, we can make it centered at 50% by adding 0.5 to the value, then saturate it at 0% and 100%, by the following code:

```
unew = u + 0.5;
if (unew > 1.0) unew = 1.0;
if (unew < 0.0) unew = 0.0;
```

Finally we have to convert this to an OC1RS in the range 0 to PR3:

```
OC1RS = (unsigned int) (unew * PR3);
```

It is highly recommended that you define the integral of the control error, `Eint`, as a global `volatile int`. Then reset `Eint` to zero in `main` every time a new `Kp` and `Ki` are received from Matlab. This ensures that this new controller starts fresh, without a potentially large error integral from the previous controller.

1. Using your Matlab interface, tune your gains K_p and K_i until you get good tracking of the square wave reference. Turn in a plot of the performance.

12.9 Going Further

Some other things you can try:

1. In addition to plotting the reference waveform and the actual measured signal, plot the OC1RS value, so you can see the control effort.

2. Create a new reference waveform shape. For example, make the LED brightness follow a sinusoidal waveform. You can calculate this reference waveform on the PIC32. You should be able to choose which waveform to use by an input argument in your Matlab interface. Perhaps even allow the user to specify parameters of the waveform (like `center` and `A`). Or, change the PIC32 and Matlab code so that Matlab sends over the 1000 samples of an arbitrary reference trajectory. Then you can use Matlab code to flexibly create a wide variety of reference trajectories.

12.10 Chapter Summary

- Control of the brightness of an LED can be achieved by a PWM signal at a frequency beyond that perceptible by the eye. The brightness can be sensed by a phototransistor and resistor circuit. A capacitor in parallel with the resistor low-pass filters the sensor signal with a cutoff frequency $f_c = 1/(2\pi RC)$, rejecting the high-frequency components due to the PWM frequency and its harmonics while keeping the low-frequency components (those perceptible by the eye).
- A reference brightness, as a function of time, can be stored in an array with N samples. By cyclically indexing through this array in an ISR invoked at a fixed frequency of f_{ISR}, the reference brightness waveform is periodic with frequency f_{ISR}/N.
- Since LED brightness control by a PWM signal is a zeroth-order system (the PWM voltage directly changes the LED current and therefore brightness, without any integrations), a good choice for a feedback controller is a PI controller.
- When accepting new gains `Kp` and `Ki` from the user, interrupts should be disabled to ensure that the ISR is not called in the middle of updating `Kp` and `Ki`. Interrupts should be disabled as briefly as possible, however, to avoid interfering with expected ISR timing. This can be achieved by keeping the relatively slow `sscanf` outside the period that interrupts are disabled. Interrupts are only disabled during the short period that the values read by `scanf` are copied to `Kp` and `Ki`.

12.11 Exercises

Complete the LED brightness control project as outlined in the chapter.

Chapter 13

Brushed Permanent Magnet DC Motors

Essentially all electric motors operate on the same principle: current flowing through a magnetic field creates a force. Because of this relationship between current and force, electric motors can be used to convert electrical power to mechanical power. They can also be used to convert mechanical power to electrical power; generators in hydropower dams and regenerative braking in electric and hybrid cars are examples of this.

In this chapter we study perhaps the simplest, cheapest, most common, and arguably most useful electrical motor: the brushed permanent magnet direct current (DC) motor. For brevity, we refer to these simply as DC motors. A DC motor has two input terminals, and a voltage applied across those terminals causes the motor shaft to spin. For a constant load or resistance at the motor shaft, the motor shaft achieves a speed proportional to the input voltage. Positive voltage causes spinning in one direction, and negative voltage causes spinning in the other.

Depending on the specifications, you can buy DC motors for tens of cents up to thousands of dollars. For most small-scale or hobby applications, appropriate DC motors typically cost a few dollars. DC motors are often outfitted with a sensing device, most commonly an encoder, to track the position and speed of the motor, and a gearhead to reduce the output speed and increase the output torque.

13.1 Motor Physics

DC motors exploit the *Lorentz force law*,

$$\mathbf{F} = \ell \mathbf{I} \times \mathbf{B}, \tag{13.1}$$

where $\mathbf{F}, \mathbf{I}$, and $\mathbf{B}$ are 3-vectors, $\mathbf{B}$ describes the magnetic field created by permanent magnets, $\mathbf{I}$ is the current vector (including the magnitude and direction of the current flow through the conductor), ℓ is the length of the conductor in the magnetic field, and $\mathbf{F}$ is the force on the conductor. For the case of a current perpendicular to the magnetic field, the force is easily understood using the right-hand rule for cross-product: with your right hand, point your index finger along the current direction and your middle finger along the magnetic field flux lines. Your thumb will then point in the direction of the force. See Figure 13.1.

Now let's replace the conductor by a loop of wire, and constrain that loop to rotate about its center. See Figures 13.2 and 13.3. In one half of the loop, the current flows into the page, and in the other half of the loop the current flows out of the page. This creates forces of opposite directions on the loop. Referring to Figure 13.3, let the magnitude of the force acting on each half of the loop be f, and let d be the distance from the halves of the loop to the center of the loop. Then the total torque acting on the loop about its center can be written

$$\tau = 2df \cos\theta,$$

where θ is the angle of the loop. The torque changes as a function of θ. For $-90° < \theta < 90°$, the torque is positive, and it is maximum at $\theta = 0$. A plot of the torque on the loop as a function of θ is shown in

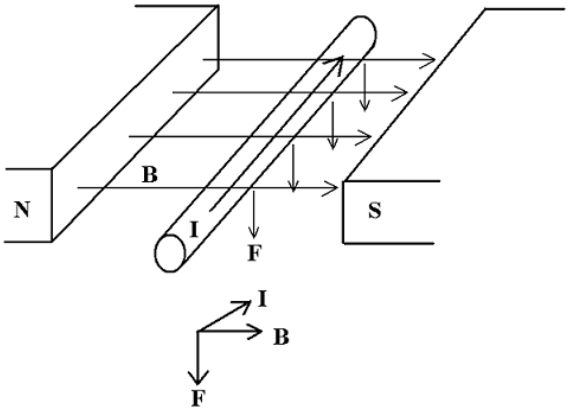


Figure 13.1: Two magnets create a magnetic field **B**, and a current **I** along the conductor causes a force **F** on the conductor.

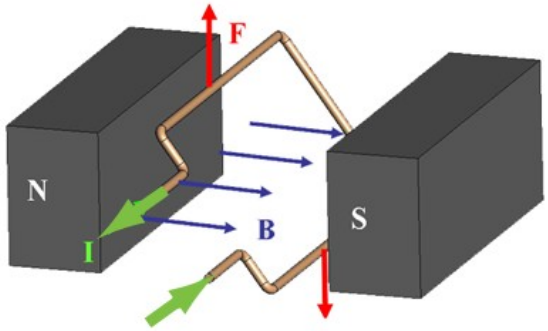


Figure 13.2: A current-carrying loop of wire in a magnetic field.

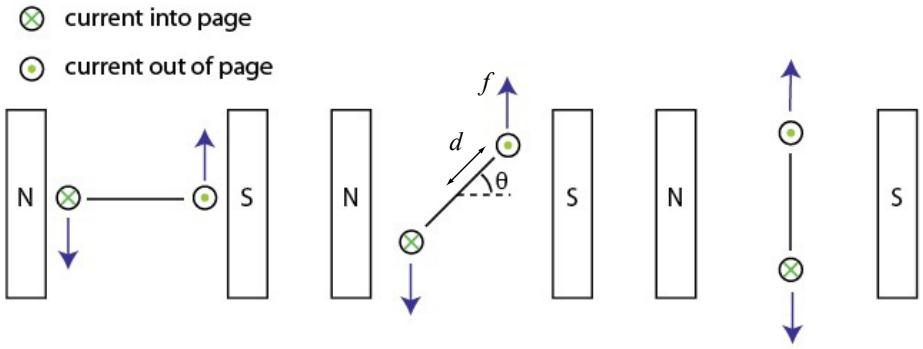


Figure 13.3: A loop of wire in a magnetic field, viewed end-on. Current flows into the page on one side of the loop and out of the page on the other, creating forces of opposite directions on the two halves of the loop. These opposite forces create torque on the loop about its center at most angles θ of the loop.

Figure 13.4(a). The torque is zero at $\theta = -90°$ and $90°$, and of these two, $\theta = 90°$ is stable while $\theta = -90°$ is unstable. Therefore, if we send a constant current through the loop, it will eventually come to rest at $\theta = 90°$.

To make a more useful motor, we can reverse the direction of current at $\theta = -90°$ and $\theta = 90°$. This has the effect of making the torque nonnegative at all angles (Figure 13.4(b)). The torque is still zero at

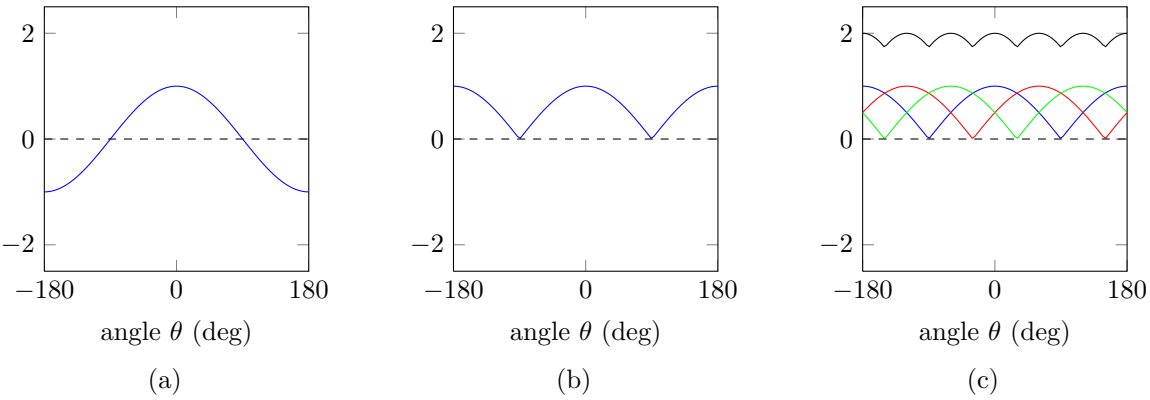


Figure 13.4: (a) The torque on the loop of Figure 13.3 as a function of its angle for a constant current. (b) If we reverse the current direction at the angles $\theta = -90°$ and $\theta = 90°$, we can make the torque nonnegative at all θ. (c) If we use several loops offset from each other, the sum of their torques becomes more constant as a function of angle. The remaining variation contributes to torque ripple.

$\theta = -90°$ and $\theta = 90°$, however, and it undergoes a large variation as a function of θ. To make the torque more constant as a function of θ, we can introduce more loops of wire, each offset from the others in angle, and each reversing their current direction at appropriate angles. Figure 13.4(c) shows an example with three loops of wire offset from each other by 120°. Their component torques sum to give a more constant torque as a function of angle. The remaining variation in torque contributes to angle-dependent *torque ripple*.

Finally, to increase the torque generated, each loop of wire is replaced by a coil of wire that loops back and forth through the magnetic field many times. If the coil consists of 100 loops, it generates 100 times the torque of the single loop for the same current. Wire used to create coils in motors, like magnet wire, is very thin, so there is resistance from one end of a coil to the other, typically from fractions of an ohm up to hundreds of ohms.

The only thing missing is the method to switch the current direction. Figure 13.5 shows the idea behind the solution for brushed DC motors. The two input terminals are connected to *brushes*, typically made of a soft conducting metal like graphite, which are spring-loaded to press against the *commutator*, which is connected to the motor coils. As the motor rotates, the brushes slide over the commutator and switch between commutator *segments*, each of which is electrically connected to the end of one or more coils. This switching changes the direction of current through the coils. (Unlike the simplified example in Figure 13.4, however, not all coils are energized at the same time.) Figure 13.5 shows a schematic of a minimal motor design with three commutator segments and a coil between each pair of segments. Most high quality motors have more commutator segments and coils.

And that is how brushed DC motors work. The basic concept hasn't changed much since the late 1800's. Figure 13.6 shows a DC motor that has been opened up, exposing the brushes, commutator, and coils, as well as one of the two permanent magnets on the interior of the housing. The rotating portion of the motor is called the *rotor* and the housing is called the *stator*.

Brushless motors are a variant that use electronic commutation as opposed to brushed commutation. For a brushless motor, the permanent magnets are on the rotor and the coils (armature) are attached to the interior of the motor housing. External commutation circuitry switches the direction of the current through the armature based on the motor angle sensed by Hall effect sensors. Brushless motors are common, but brushed DC motors still dominate inexpensive applications. Drawbacks of brushed motors relative to brushless motors include

- brush wear: the brushes will eventually wear down, limiting lifetime compared to brushless motors;
- particles due to the wearing brushes;
- friction and noise due to the brushes;

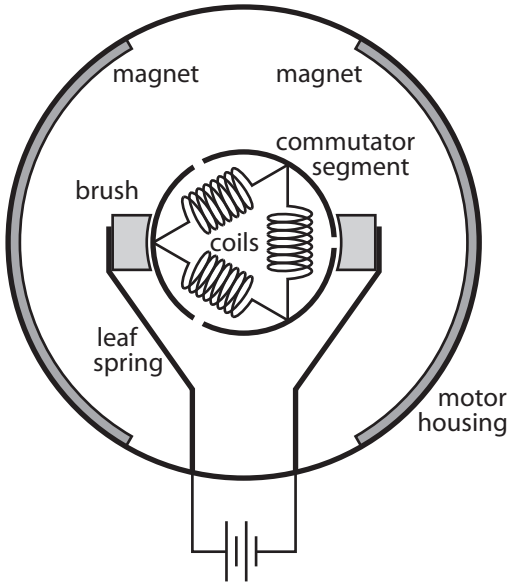


Figure 13.5: A schematic end-on view of a simple DC motor. The two brushes are held against the commutator by leaf springs which are electrically connected to the external motor terminals. This commutator has three segments and there are coils between each segment pair. The stator magnets are epoxied to the inside of the motor housing.

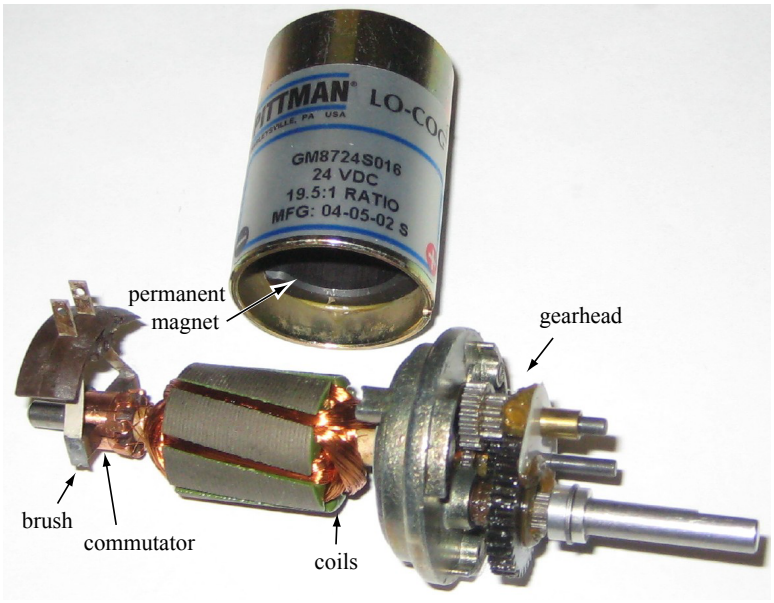


Figure 13.6: The two brushes of this disassembled DC motor are attached to the motor housing, which has otherwise been removed. One of the two permanent magnets is visible inside the housing. Coils are often wrapped around iron or other ferromagnetic material to increase magnetic permeability. This motor has a gearhead on the output.

- greater electrical noise due to the abrupt brush-commutation switching events; and
- lower continuous current ratings, as brushless motors dissipate heat more effectively (and therefore can

sustain higher continuous currents) due to the attachment of the coils to the motor housing.

Brushless motors have the disadvantage of more complex drive circuitry, however, and the drawbacks of brushed motors are not critical for many applications.

13.2 Governing Equations

To derive an equation to approximately model the motor's behavior, let's ignore the details of the commutation. Let's focus instead on electrical and mechanical power. The electrical power put into the motor is IV, where I is the current through the motor and V is the voltage across the motor. We know that the motor converts some of this input power to mechanical power $\tau\omega$, where τ and ω are the torque and velocity of the output shaft, respectively. Electrically, the motor is described by a resistance R between the two terminals as well as an inductance L due to the coils. The resistance of the motor coils dissipates power I^2R as heat. The motor also stores energy $\frac{1}{2}LI^2$ in the inductor's magnetic field, and the time rate of change of this is $LI(dI/dt)$, the power into (charging) or out of (discharging) the inductor. Finally, power is dissipated as sound, heat due to friction at the brush-commutator interface and at the bearings between the motor shaft and the housing, etc. In SI units, all these power components are expressed in watts. Putting these all together, we have a full accounting for the electrical power put into the motor:

$$IV = \tau\omega + I^2R + LI\frac{dI}{dt} + \text{ power dissipated due to friction, sound, etc.}$$

Ignoring the last term, we have our simple motor model, written in terms of power:

$$IV = \tau\omega + I^2R + LI\frac{dI}{dt}. \tag{13.2}$$

From Equation (13.2) we can derive all other relationships of interest. For example, dividing both sides of (13.2) by I, we get

$$V = \frac{\tau}{I}\omega + IR + L\frac{dI}{dt}. \tag{13.3}$$

The ratio τ/I is a constant, an expression of the Lorentz force law for the particular motor design. This constant is called the *torque constant* k_t, and this constant relating current to torque is one of the most important properties of the motor:

$$k_t = \frac{\tau}{I} \quad \text{or} \quad \tau = k_tI. \tag{13.4}$$

The SI units of k_t are Nm/A. (In this chapter, we only use SI units, but you should be aware that many different units are used by different manufacturers, as on the speed-torque curve and data sheet in Figure 13.15 in the Exercises.) Equation (13.3) also shows that the SI units for k_t can be written equivalently as Vs/rad, or simply Vs. When using these units, we sometimes call the motor constant the *electrical constant* k_e. The inverse is sometimes called the *speed constant*. You should recognize that these terms all refer to the same property of the motor. For consistency, we usually refer to the torque constant k_t.

With this, we write the motor model in terms of voltage as

$$V = k_t\omega + IR + L\frac{dI}{dt}. \tag{13.5}$$

You should remember, or be able to quickly rederive, the power equation (13.2), the torque constant (13.4), and the voltage equation (13.5).

The term $k_t\omega$, with units of voltage, is called the *back-emf*, where emf is short for *electromotive force*. We could also call this "back-voltage." Back-emf is the voltage generated by a spinning motor to "oppose" the input voltage generating the motion. As an example, say that the motor's terminals are not connected to anything (open circuit). Then clearly $I = 0$, and (13.5) reduces to

$$V = k_t\omega.$$

This indicates that back-driving the motor (e.g., spinning it by hand) will generate a voltage at the terminals. If we were to connect a capacitor across the motor terminals, then spinning the motor by hand would cause the capacitor to charge up, storing some of the mechanical energy we are putting in as electrical energy in the capacitor. This is basically how hydropower dams and regenerative braking in hybrid cars works.

The existence of this back-emf term also means that if we put a constant voltage V across a free-spinning frictionless motor, after some time it will reach a constant speed V/k_t. At this speed, by (13.5), the current I drops to zero, meaning there is no more torque τ to accelerate the motor.

13.3 The Speed-Torque Curve

If we assume the motor is at steady state, i.e., $dI/dt = 0$, Equation (13.5) reduces to

$$V = k_t\omega + IR. \tag{13.6}$$

Using the definition of the torque constant, we get the equivalent form

$$\omega = \frac{1}{k_t}V - \frac{R}{k_t^2}\tau. \tag{13.7}$$

Equation (13.7) gives ω as a linear function of τ for a given constant V. This line, of slope $-R/k_t^2$, is called the *speed-torque curve* for the voltage V.

The speed-torque curve plots all the possible steady-state operating conditions with voltage V across the motor. Assuming friction torque is zero, the line intercepts the $\tau = 0$ axis at

$$\omega_0 = V/k_t = \textit{no load speed}.$$

The line intercepts the $\omega = 0$ axis at

$$\tau_{\text{stall}} = \frac{k_t V}{R} = \textit{stall torque}.$$

At the no-load condition, $\tau = I = 0$; the motor rotates at maximum speed with no current or torque. At the stall condition, the shaft is blocked from rotating, and the current ($I_{\text{stall}} = \tau_{\text{stall}}/k_t = V/R$) and output torque are maximized due to the lack of back-emf. Which point along the speed-torque curve the motor actually operates at is determined by the load attached to the motor shaft.

An example speed-torque curve is shown in Figure 13.7. This motor has $\omega_0 = 500$ rad/s and $\tau_{\text{stall}} = 0.1067$ Nm for a nominal voltage of $V_{\text{nom}} = 12$ V. The *operating region* is any point below the speed-torque curve, corresponding to voltages less than or equal to 12 V. If the motor is operated at a different voltage cV_{nom}, the intercepts of the speed-torque curve are linearly scaled to $c\omega_0$ and $c\tau_{\text{stall}}$.

The speed-torque curve corresponds to constant V, but not to constant input power IV. The current I is linear with τ, so the input electrical power increases linearly with τ. The output mechanical power is $P_{\text{out}} = \tau\omega$, and the *efficiency* in converting electrical to mechanical power is $\eta = \tau\omega/IV$. We come back to efficiency in Section 13.4.

To find the point on the speed-torque curve that maximizes the mechanical output power, we can write points on the curve as $(\tau, \omega) = (c\tau_{\text{stall}}, (1-c)\omega_0)$ for $0 \le c \le 1$, so the output power is expressed as

$$P_{\text{out}} = \tau\omega = (c - c^2)\tau_{\text{stall}}\omega_0,$$

and the value of c that maximizes the power output is found by solving

$$\frac{d}{dc}\big((c-c^2)\tau_{\text{stall}}\omega_0\big) = (1-2c)\tau_{\text{stall}}\omega_0 = 0 \quad \rightarrow \quad c = \frac{1}{2}.$$

Thus the mechanical output power is maximized at $\tau = \tau_{\text{stall}}/2$ and $\omega = \omega_0/2$. This maximum output power is

$$P_{\max} = \left(\frac{1}{2}\tau_{\text{stall}}\right)\left(\frac{1}{2}\omega_0\right) = \frac{1}{4}\tau_{\text{stall}}\omega_0.$$

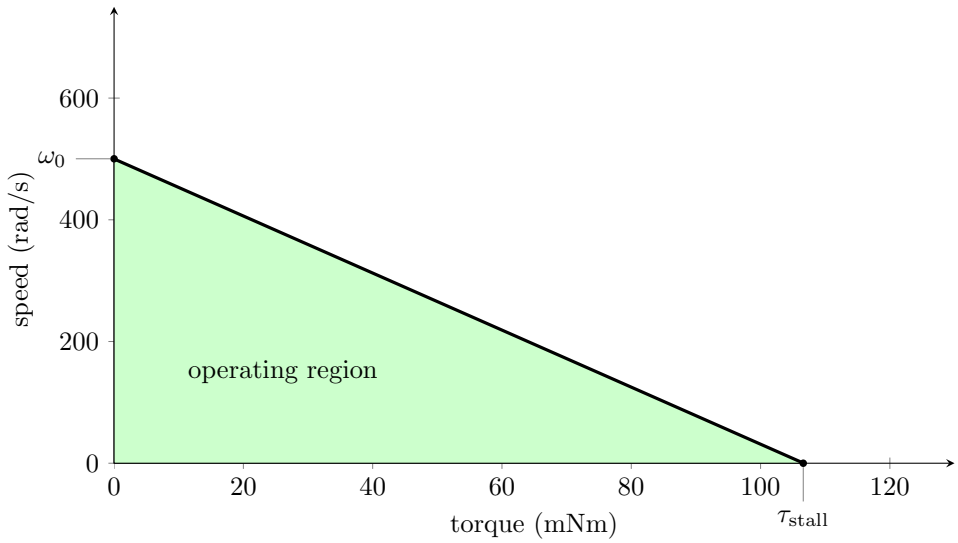


Figure 13.7: A speed-torque curve. Many speed-torque curves use rpm for speed, but we prefer SI units.

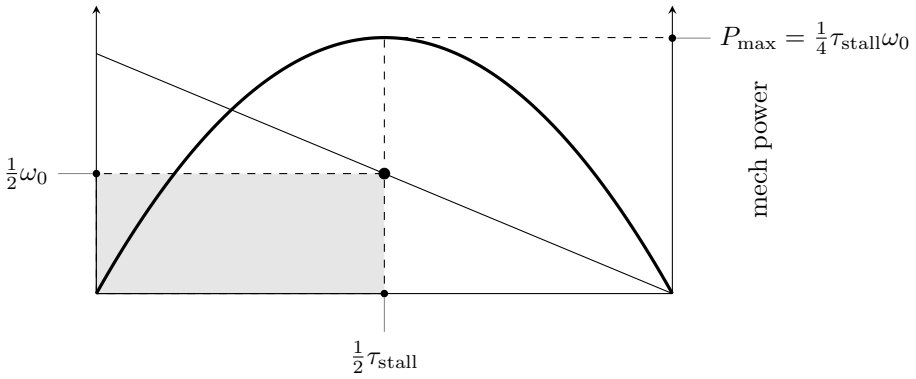


Figure 13.8: The quadratic mechanical power plot $P = \tau\omega$ plotted alongside the speed-torque curve. The area of the speed-torque rectangle below and to the left of the operating point is the mechanical power.

See Figure 13.8.

Motor current is proportional to motor torque, so operating at high torques means large coil heating losses I^2R, sometimes called *ohmic heating*. For that reason, motor manufacturers specify a *maximum continuous current* I_{cont}, the largest continuous current such that the coils' steady-state temperature remains below a critical point.[1] The maximum continuous current has a corresponding *maximum continuous torque* τ_{cont}. Points to the left of this torque and under the speed-torque curve are called the *continuous operating region*. The motor can be operated intermittently outside of the continuous operating region, in the *intermittent operating region*, provided the motor is allowed to cool sufficiently between uses in this region. Motors are commonly rated with a nominal voltage that places the maximum mechanical power operating point (at $\tau_{\text{stall}}/2$) outside the continuous operating region.

Given thermal characteristics of the motor of Figure 13.7, the speed-torque curve can be refined to Figure 13.9, showing the continuous and intermittent operating regions of the motor. The point on the speed-torque curve at τ_{cont} is the *rated* or *nominal operating point*, and the mechanical power output at this

[1] The maximum continuous current depends on thermal properties governing how fast coil heat can be transferred to the environment. This depends on the environment temperature, typically considered to be room temperature. The maximum continuous current can be increased by cooling the motor.

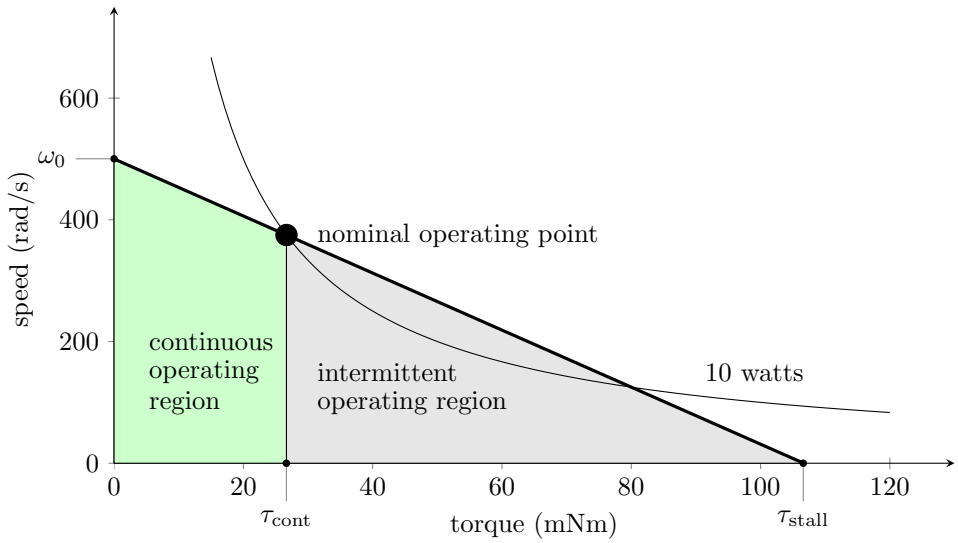


Figure 13.9: The continuous operating region (under the speed-torque curve and left of τ_{cont}) and the intermittent operating region (the rest of the area under the speed-torque curve). The 10 W mechanical power hyperbola is indicated, including the nominal operating point at τ_{cont}.

point is called the motor's *power rating*. For the motor of Figure 13.9, $\tau_{\text{cont}} = 26.67$ mNm, which occurs at $\omega = 375$ rad/s, for a power rating of

$$0.02667 \text{ Nm} \times 375 \text{ rad/s} = 10.0 \text{ W}.$$

Figure 13.9 also shows the constant output power hyperbola $\tau\omega = 10$ W passing through the nominal operating point.

The speed-torque curve for a motor is drawn based on a nominal voltage. This is a "safe" voltage that the manufacturer recommends. It is possible to overvolt the motor, however, provided it is not continuously operated beyond the maximum continuous current. A motor also may have a specified *maximum permissible speed* $\omega_{\max}$, which creates a horizontal line constraint on the permissible operating range. This speed is determined by properties of the shaft bearings or allowable brush wear, and it is typically larger than the no-load speed ω_0. The shaft and bearings may also have a maximum torque rating $\tau_{\max} > \tau_{\text{stall}}$. These limits allow the loose definition of overvolted continuous and intermittent operating regions, as shown in Figure 13.10.

13.4 Friction and Motor Efficiency

Until now we have been assuming that the full torque $\tau = k_t I$ generated by the windings is available at the output shaft. In practice, some torque is lost due to friction at the brushes and the shaft bearings. Let's use a simple model of friction: assume a torque $\tau \geq \tau_{\text{fric}} > 0$ must be generated to overcome friction and initiate motion, and any torque beyond τ_{fric} is available at the output shaft regardless of the motor speed (e.g., no friction that depends on speed magnitude). When the motor is spinning, the torque available at the output shaft is

$$\tau_{\text{out}} = \tau - \tau_{\text{fric}}.$$

Nonzero friction results in a nonzero *no-load current* $I_0 = \tau_{\text{fric}}/k_t$ and a no-load speed ω_0 less than V/k_t. The speed-torque curve of Figure 13.10 is modified to show a small friction torque in Figure 13.11. The torque actually delivered to the load is reduced by τ_{fric}.

Taking friction into account, the motor's efficiency in converting electrical to mechanical power is

$$\eta = \frac{\tau_{\text{out}}\omega}{IV}. \tag{13.8}$$

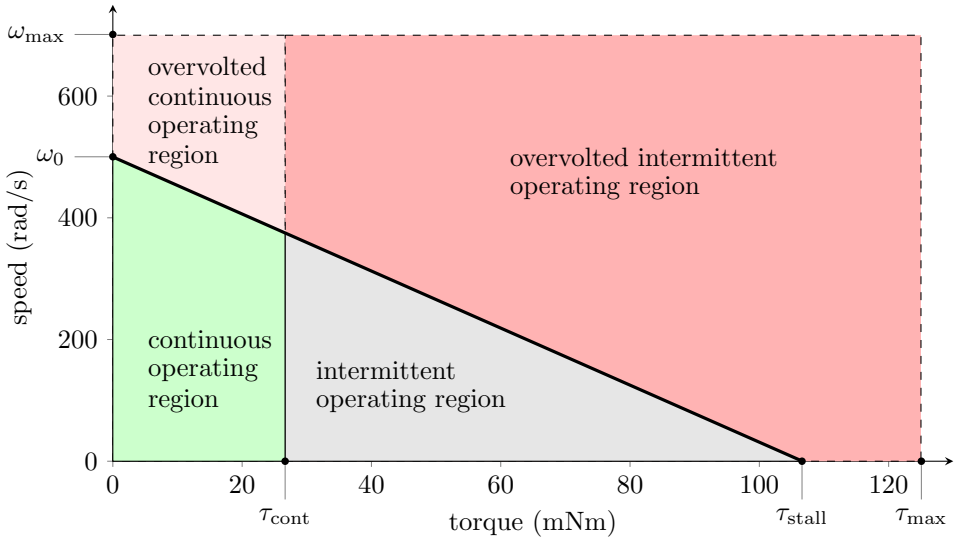


Figure 13.10: It is possible to exceed the nominal operating voltage, provided the constraints $\omega < \omega_{\max}$ and $\tau < \tau_{\max}$ are respected and τ_{cont} is only intermittently exceeded.

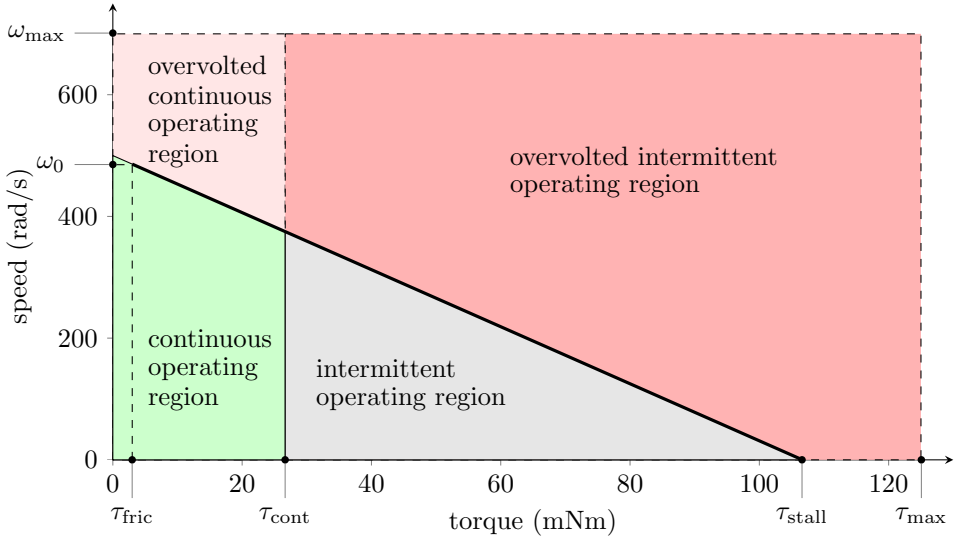


Figure 13.11: The speed-torque curve of Figure 13.10 modified to show a nonzero friction torque τ_{fric} and the resulting reduced no-load speed ω_0.

The efficiency depends on the operating point on the speed-torque curve, and it is zero when either τ_{out} or ω is zero, as there is no mechanical power output. Maximum efficiency generally occurs at high speed and low torque, approaching the limit of 100% efficiency at $\tau = \tau_{\text{out}} = 0$ and $\omega = \omega_0$ as τ_{fric} approaches zero. As an example, Figure 13.12 plots efficiency vs. torque for the same motor with two different values of τ_{fric}. Lower friction results in a higher maximum efficiency $\eta_{\max}$, occurring at a higher speed and lower torque.

To derive the maximally efficient operating point and the maximum efficiency $\eta_{\max}$ for a given motor, we can express the motor current as

$$I = I_0 + I_a,$$

where I_0 is the no-load current necessary to overcome friction and I_a is the added current to create torque to drive the load. Recognizing that $\tau_{\text{out}} = k_t I_a$, $V = I_{\text{stall}} R$, and $\omega = R(I_{\text{stall}} - I_a - I_0)/k_t$ by the linearity of

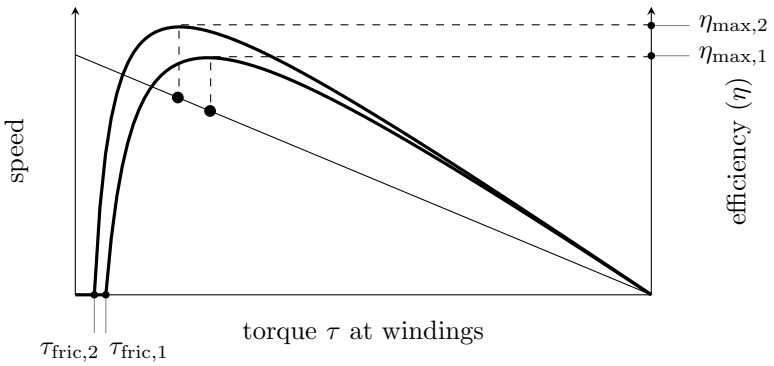


Figure 13.12: The speed-torque curve for a motor and two efficiency plots, one for high friction torque (case 1) and one for low friction torque (case 2). For each case, efficiency is zero for all τ below the level needed to overcome friction. The low friction version of the motor (case 2) achieves a higher maximum efficiency, at a higher speed and lower torque, than the high friction version (case 1).

the speed-torque curve, we can rewrite the efficiency (13.8) as

$$\eta = \frac{I_a(I_{\text{stall}} - I_0 - I_a)}{(I_0 + I_a)I_{\text{stall}}}. \tag{13.9}$$

To find the operating point I_a^* maximizing η, we solve $d\eta/dI_a = 0$ for I_a^*, and recognizing that I_0 and I_{stall} are nonnegative, the solution is

$$I_a^* = \sqrt{I_{\text{stall}} I_0} - I_0.$$

In other words, as the no-load current I_0 goes to zero, the maximally efficient current (and therefore τ) goes to zero.

Plugging I_a^* into (13.9), we find

$$\eta_{\max} = \left(1 - \sqrt{\frac{I_0}{I_{\text{stall}}}}\right)^2.$$

This answer has the form we would expect: maximum efficiency approaches 100% as the friction torque approaches zero, and maximum efficiency approaches 0% as the friction torque approaches the stall torque.

Choosing an operating point that maximizes motor efficiency can be important when trying to maximize battery life in mobile applications. For the majority of analysis and motor selection problems, however, ignoring friction is a good first approximation.

13.5 Motor Windings and the Motor Constant

It is possible to build two different versions of the same motor by simply changing the windings while keeping everything else the same. For example, imagine a coil of resistance R with N loops of wire of cross-sectional area A. The coil carries a current I and therefore has a voltage drop IR. Now we replace that coil with a new coil with N/c loops of wire with cross-sectional area cA. This preserves the volume occupied by the coil, fitting in the same form factor with similar thermal properties. Without loss of generality, let's assume that the new coil has fewer loops and uses thicker wire ($c > 1$).

The resistance of the new coil is reduced to R/c^2 (a factor of c due to the shorter coil and another factor of c due to the thicker wire). To keep the torque of the motor the same, the new coil would have to carry a larger current cI to make up for the fewer loops, so that the current times the pathlength through the magnetic field is unchanged. The voltage drop across the new coil is $(cI)(R/c^2) = IR/c$.

Replacing the coils allows us to create two versions of the motor: a many-loop, thin wire version that operates at low current and high voltage, and a fewer-loop, thick wire version that operates at high current

and low voltage. Since the two motors create the same torque with different currents, they have different torque constants. Each motor has the same *motor constant* k_m, however, where

$$k_m = \frac{\tau}{\sqrt{I^2 R}} = \frac{k_t}{\sqrt{R}}$$

with units of $\text{Nm}/\sqrt{\text{W}}$. The motor constant defines the torque generated per square root of the power dissipated by coil resistance. In the example above, the new coil dissipates $(cI)^2(R/c^2) = I^2R$ power as heat, just as the original coil does, while generating the same torque.

Figure 13.15 shows the data sheet for a motor that comes in several different versions, each identical in every way except for the winding. Each version of the motor has a similar stall torque and motor constant but different nominal voltage, resistance, and torque constant.

13.6 Other Motor Characteristics

Electrical time constant When the motor is subject to a step in the voltage across it, the *electrical time constant* T_e measures the time it takes for the current to reach 63% of its final value. The motor's voltage equation is

$$V = k_t\omega + IR + L\frac{dI}{dt}.$$

Ignoring back-emf (because the motor speed does not change significantly over one electrical time constant), assuming an initial current through the motor of I_0, and an instantaneous drop in the motor voltage to 0, we get the differential equation

$$0 = I_0 R + L\frac{dI}{dt}$$

or

$$\frac{dI}{dt} = -\frac{R}{L}I_0,$$

with solution

$$I(t) = I_0\, e^{-tR/L} = I_0\, e^{-t/T_e}.$$

The time constant of this first-order decay of current is the motor's electrical time constant, $T_e = L/R$.

Mechanical time constant When the motor is subject to a step voltage across it, the *mechanical time constant* T_m measures the time it takes for the motor speed to reach 63% of its final value. Beginning from the voltage equation

$$V = k_t\omega + IR + L\frac{dI}{dt},$$

ignoring the inductive term, and assuming an initial speed ω_0 at the moment the voltage drops to zero, we get the differential equation

$$0 \quad = \quad IR + k_t\omega_0 \quad = \quad \frac{R}{k_t}\tau + k_t\omega_0 \quad = \quad \frac{JR}{k_t}\frac{d\omega}{dt} + k_t\omega_0$$

or

$$\frac{d\omega}{dt} = -\frac{k_t^2}{JR}\omega_0$$

with a time constant of $T_m = JR/k_t^2$. If the motor is attached to a load that increases the inertia, the mechanical time constant increases.

Short-circuit damping When the terminals of the motor are shorted together, the voltage equation (ignoring inductance) becomes

$$0 \quad = \quad k_t\omega + IR \quad = \quad k_t\omega + \frac{\tau}{k_t}R$$

or

$$\tau \quad = \quad -B\omega \quad = \quad -\frac{k_t^2}{R}\omega,$$

where $B = k_t^2/R$ is the short-circuit damping. A spinning motor is slowed more quickly by shorting its terminals together, compared to leaving the terminals open circuit, due to this damping.

13.7 Motor Data Sheet

Motor manufacturers summarize motor properties described above in a speed-torque curve and in a data sheet similar to the one in Figure 13.13. When you buy a motor second-hand or surplus, you may need to measure these properties yourself. We will use all SI units, which is not the case on most motor data sheets.

Many of these properties have been introduced already. Below we describe some methods for estimating them.

Experimentally Characterizing a Brushed DC Motor

Given a mystery motor with an encoder, you can use a function generator, oscilloscope, multimeter and perhaps some resistors and capacitors to estimate most of the important properties of the motor. Below are some suggested methods; you may be able to devise others.

Terminal resistance R You can measure R with a multimeter. The resistance may change as you rotate the shaft by hand, as the brushes move to new positions on the commutator. You should record the minimum resistance you can reliably find. A better choice, however, may be to measure the current when the motor is stalled.

Torque constant k_t You can measure this by spinning the shaft of the motor, measuring the back-emf at the motor terminals, and measuring the rotation rate using the encoder. Or, if friction losses are negligible, a good approximation is V_{nom}/ω_0. This eliminates the need to spin the motor externally.

Electrical constant k_e Identical to the torque constant in SI units. The torque constant k_t is often expressed in units of Nm/A or mNm/A or English units like oz-in/A, and often k_e is given in V/rpm, but k_t and k_e have identical numerical values when expressed in Nm/A and Vs/rad, respectively.

Speed constant k_s Just the inverse of the electrical constant.

Motor constant k_m The motor constant is calculated as $k_m = k_t/\sqrt{R}$.

Max continuous current $I_{\textbf{cont}}$ This is determined by thermal considerations, which are not easy to measure. It is typically less than half the stall current.

Max continuous torque $\tau_{\textbf{cont}}$ This is determined by thermal considerations, which are not easy to measure. It is typically less than half the stall torque.

Short-circuit damping B This is most easily calculated from estimates of R and k_t: $B = k_t^2/R$.

Motor Characteristic	Symbol	Value	Units	Comments
Terminal resistance	R		Ω	Resistance of the motor windings. May change as brushes slide over commutator segments. Increases with heat.
Torque constant	k_t		Nm/A	The constant ratio of torque produced to current through the motor.
Electrical constant	k_e		Vs/rad	Same numerical value as the torque constant (in SI units). Also called voltage or back-emf constant.
Speed constant	k_s		rad/(Vs)	Inverse of electrical constant.
Motor constant	k_m		Nm/$\sqrt{\text{W}}$	Torque produced per square root of power dissipated by the coils.
Max continuous current	I_{cont}		A	Max continuous current without overheating.
Max continuous torque	τ_{cont}		Nm	Max continuous torque without overheating.
Short-circuit damping	B		Nms/rad	Not included in most data sheets, but useful for motor braking (and haptics).
Terminal inductance	L		H	Inductance due to the coils.
Electrical time constant	T_e		s	The time for the motor current to reach 63% of its final value. Equal to L/R.
Rotor inertia	J		kgm^2	Often given in units gcm^2.
Mechanical time constant	T_m		s	The time for the motor to go from rest to 63% of its final speed under constant voltage and no load. Equal to JR/k_t^2.
Friction				Not included in most data sheets. See explanation.
Values at Nominal Voltage				
Nominal voltage	V_{nom}		V	Should be chosen so the no-load speed is safe for brushes, commutator, and bearings.
Power rating	P		W	Output power at the nominal operating point (max continuous torque).
No-load speed	ω_0		rad/s	Speed when no load and powered by V_{nom}. Usually given in rpm (revs/min, sometimes m^{-1}).
No-load current	I_0		A	The current required to spin the motor at the no-load condition. Nonzero because of friction torque.
Stall current	I_{stall}		A	Same as starting current, V_{nom}/R.
Stall torque	τ_{stall}		Nm	The torque achieved at the nominal voltage when the motor is stalled.
Max mechanical power	P_{max}		W	The max mechanical power output at the nominal voltage (including short-term operation).
Max efficiency	η_{max}		%	The maximum efficiency achievable in converting electrical to mechanical power.

Figure 13.13: A sample motor data sheet, with values to be filled in.

Terminal inductance L There are a number of ways to measure inductance. One approach is to add a capacitor in parallel with the motor and measure the oscillation frequency of the resulting RLC circuit. For example, you could build the circuit shown in Figure 13.14, where a good choice for C may be 0.01 μF or 0.1 μF. The motor acts as a resistor and inductor in series; back-emf will not be an issue, because the motor will be powered by tiny currents at high frequency and therefore will not move.

Use a function generator to put a 1 kHz square wave between 0 and 5 V at the point indicated. The 1 kΩ resistor limits the current from the function generator. Measure the voltage with an oscilloscope where indicated. You should be able to see a decaying oscillatory response to the square wave input when you choose the right scales on your scope. Measure the frequency of the oscillatory response. Knowing C and that the natural frequency of an RLC circuit is $\omega_n = 1/\sqrt{LC}$ in rad/s, estimate L.

Let's think about why we see this response. Say the input to the circuit has been at 0 V for a long time. Then your scope will also read 0 V. Now the input steps up to 5 V. After some time, in steady state, the capacitor will be an open circuit and the inductor will be a closed circuit (wire), so the voltage on the scope

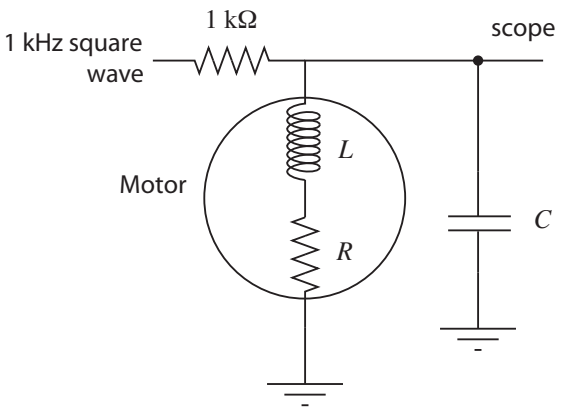


Figure 13.14: Using a capacitor to create an RLC circuit to measure motor inductance.

will settle to 5 V $\times$ $(R/(1000+R))$—the two resistors in the circuit set the final voltage. Right after the voltage step, however, all current goes to charge the capacitor (as the zero current through the inductor cannot change discontinuously). If the inductor continued to enforce zero current, the capacitor would charge to 5 V. As the voltage across the capacitor grows, however, so does voltage across the inductor, and therefore so does the rate of change of current that must flow through the inductor (by the relation $V_L + V_R = V_C$ and the constitutive law $V_L = L\, dI/dt$). Eventually the integral of this rate of change dictates that all current is redirected to the inductor, and in fact the capacitor will have to provide current to the inductor, discharging itself. As the voltage across the capacitor drops, though, the voltage across the inductor will eventually become negative, and therefore the rate of change of current across the inductor will become negative. And so on, to create the oscillation. If R were large, i.e., if the circuit were heavily damped, the oscillation would die quickly, but you should be able to see it.

Note that you are seeing a damped oscillation, so you are actually measuring a damped natural frequency. But the damping is low if you are seeing at least a couple of cycles of oscillation, so the damped natural frequency is nearly indistinguishable from the undamped natural frequency.

Electrical time constant T_e The electrical time constant can be calculated from L and R as $T_e = L/R$.

Rotor inertia J The rotor inertia can be estimated from measurements of the mechanical time constant T_m, the torque constant k_t, and the resistance R. Alternatively, a ballpark esimate can be made based on the mass of the motor, a guess at the portion of the mass that belongs to the spinning rotor, a guess at the radius of the rotor, and a formula for the inertia of a uniform density cylinder. Or, more simply, consult a data sheet for a motor of similar size and mass.

Mechanical time constant T_m The time constant can be measured by applying a constant voltage to the motor, measuring the velocity, and determining the time it takes to reach 63% of final speed. Alternatively, you could make a reasonable estimate of the rotor inertia J and calculate $T_m = JR/k_t^2$.

Friction Friction torque arises from the brushes sliding on the commutator and the motor shaft spinning in its bearings, and it may depend on external loads. A typical model of friction includes both Coulomb friction and viscous friction, written

$$\tau_{\text{fric}} = b_0 \,\text{sgn}(\omega) + b_1 \omega,$$

where b_0 is the Coulomb friction torque ($\text{sgn}(\omega)$ just returns the sign of ω) and b_1 is a viscous friction coefficient. At no load, $\tau_{\text{fric}} = k_t I_0$. An estimate of each of b_0 and b_1 can be made by running the motor at two different voltages with no load.

Nominal voltage $V_{\mathbf{nom}}$ This is the specification you are most likely to know for an otherwise unknown motor. It is sometimes printed right on the motor itself. This voltage is just a recommendation; the real issue is to avoid overheating the motor or spinning it at speeds beyond the recommended value for the brushes or bearings. Nominal voltage cannot be measured, but a typical no-load speed for a brushed DC motor is between 3000 and 10,000 rpm, so the nominal voltage will often give a no-load speed in this range.

Power rating P The power rating is the mechanical power output at the max continuous torque.

No-load speed ω_0 You can determine ω_0 by measuring the unloaded motor speed when powered with the nominal voltage. The amount that this is less than V_{nom}/k_t can be attributed to friction torque.

No-load current I_0 You can determine I_0 by using a multimeter in current measurement mode.

Stall current $I_{\mathbf{stall}}$ Stall current is sometimes called starting current. You can estimate this using your estimate of R. Since R may be difficult to measure with a multimeter, you can instead stall the motor shaft and use your multimeter in current sensing mode, provided the multimeter can handle the current.

Stall torque $\tau_{\mathbf{stall}}$ This can be obtained from k_t and I_{stall}.

Max mechanical power $P_{\mathbf{max}}$ The max mechanical power occurs at $\frac{1}{2}\tau_{\text{stall}}$ and $\frac{1}{2}\omega_0$. For most motor data sheets, the max mechanical power occurs outside the continuous operation region.

Max efficiency $\eta_{\mathbf{max}}$ Efficiency is defined as the power out divided by the power in, $\tau_{\text{out}}\omega/(IV)$. The wasted power is due to coil heating and friction losses. Maximum efficiency can be estimated using the no-load current I_0 and the stall current I_{stall}, as discussed in Section 13.4.

13.8 Chapter Summary

- The Lorentz force law says that a current-carrying conductor in a constant magnetic field feels a net force according to
$$\mathbf{F} = \ell\mathbf{I} \times \mathbf{B}.$$
- A brushed DC motor consists of multiple current-carrying coils attached to a rotor, and magnets on the stator to create a magnetic field. Current is transmitted to the coils by two brushes connected to the stator sliding over a commutator ring attached to the rotor. Each coil attaches to two different commutator segment. This commutation scheme assures that the current direction switches at the appropriate rotor angles.
- The voltage across a motor's terminals can be expressed as
$$V = k_t\omega + IR + L\frac{dI}{dt},$$
where k_t is the torque constant and $k_t\omega$ is the back-emf.
- The speed-torque curve is obtained by plotting the steady-state speed as a function of torque for a given motor voltage V,
$$\omega = \frac{1}{k_t}V - \frac{R}{k_t^2}\tau.$$
The maximum speed (at $\tau = 0$) is called the no-load speed ω_0 and the maximum torque (at $\omega = 0$) is called the stall torque τ_{stall}.
- The continuous operating region of a motor is defined by the maximum current I the motor coils can conduct continuously without overheating due to I^2R power dissipation.

- The mechanical power $\tau\omega$ delivered by a motor is maximized at half the stall torque and half the no-load speed, $P_{\max} = \frac{1}{4}\tau_{\text{stall}}\omega_0$.
- A motor's electrical time constant $T_e = L/R$ is the time needed for current to reach 63% of its final value in response to a step input in voltage.
- A motor's mechanical time constant $T_m = JR/k_t^2$ is the time needed for the motor speed to reach 63% of its final value in response to a step change in voltage.

13.9 Exercises

1. Assume a DC motor with a five-segment commutator. Each segment covers 70° of the circumference of the commutator circle. The two brushes are positioned at opposite ends of the commutator circle, and each makes contact with 10° of the commutator circle.
   (a) How many separate coils does this motor likely have? Explain.
   (b) Choose one of the motor coils. As the rotor rotates 360°, what is the total angle over which that coil is energized? (For example, an answer of 360° means that the coil is energized at all angles; an answer of 180° means that the coil is energized at half of the motor positions.)
2. Figure 13.15 gives the data sheet for the 10 W Maxon RE 25 motor. The columns correspond to different windings.
   (a) Draw the speed-torque curve for the 12 V version of the motor, indicating the no-load speed (in rad/s), the stall torque, the nominal operating point, and the rated power of the motor.
   (b) Explain why the torque constant is different for the different versions of the motor.
   (c) Using other entries in the table, calculate the maximum efficiency $\eta_{\max}$ of the 12 V motor and compare to the value listed.
   (d) Calculate the electrical time constant T_e of the 12 V motor. What is the ratio to the mechanical time constant T_m?
   (e) Calculate the short-circuit damping B for the 12 V motor.
   (f) Calculate the motor constant k_m for the 12 V motor.
   (g) How many commutator segments do these motors have?
   (h) Which versions of these motors are likely to be in stock?
   (i) (Optional) Motor manufacturers may specify slightly different continuous and intermittent operating regions than the ones described in this chapter. For example, the limit of the continuous operating region is not quite vertical in the speed-torque plot of Figure 13.15. Come up with a possible explanation, perhaps using online resources.
3. There are 21 entries on the motor data sheet from Section 13.7. Let's assume zero friction, so we ignore the last entry. To avoid thermal tests, you may also assume a maximum continuous power that the motor coils can dissipate as heat before overheating. Of the 20 remaining entries, under the assumption of zero friction, how many independent entries are there? That is, what is the minimum number N of entries you need to be able to fill in the rest of the entries? Give a set of N independent entries from which you can derive the other $20 - N$ dependent entries. For each of the $20 - N$ dependent entries, give the equation in terms of the N independent entries. For example, V_{nom} and R will be two of the N independent entries, from which we can calculate the dependent entry $I_{\text{stall}} = V_{\text{nom}}/R$.
4. This exercise is an experimental characterization of a motor. For this exercise, you need a low-power motor (preferably without a gearhead to avoid high friction) with an encoder. You also need a multimeter, oscilloscope, and a power source for the encoder and motor. Make sure the power source for the motor can provide enough current when the motor is stalled. A low-voltage battery pack is a good choice.

Stock program
Standard program
Special program (on request)

Article Numbers

	Motor Data		118740	118741	118742	118743	118744	118745	118746	118747	118748
	Values at nominal voltage										
1	Nominal voltage	V	4.5	8	9	12	15	18	24	32	48
2	No load speed	rpm	5360	5320	5230	4850	4980	4790	5190	5510	5070
3	No load current	mA	79.7	44.4	38.7	26.3	21.8	9.88	14.4	11.7	6.96
4	Nominal speed	rpm	4980	4520	4220	3800	3920	3710	4130	4450	4000
5	Nominal torque (max. continuous torque)	mNm	11.4	20.9	23.9	28.6	28.2	28.7	28	27.9	27.9
6	Nominal current (max. continuous current)	A	1.5	1.5	1.5	1.24	1.01	0.811	0.652	0.516	0.317
7	Stall torque	mNm	131	132	119	129	131	126	136	144	132
8	Starting current	A	16.5	9.23	7.31	5.5	4.57	3.52	3.1	2.61	1.47
9	Max. efficiency	%	87	87	86	87	87	90	87	87	87
	Characteristics										
10	Terminal resistance	Ω	0.273	0.867	1.23	2.18	3.28	5.11	7.73	12.3	32.6
11	Terminal inductance	mH	0.0275	0.0882	0.115	0.238	0.353	0.551	0.832	1.31	3.48
12	Torque constant	mNm/A	7.99	14.3	16.3	23.5	28.6	35.8	43.9	55.2	89.9
13	Speed constant	rpm/V	1200	668	584	406	334	267	217	173	106
14	Speed / torque gradient	rpm/mNm	40.9	40.5	44	37.7	38.3	38.2	38.3	38.5	38.6
15	Mechanical time constant	ms	4.99	4.4	4.37	4.25	4.23	4.22	4.22	4.22	4.23
16	Rotor inertia	gcm²	11.7	10.4	9.49	10.8	10.6	10.6	10.5	10.5	10.5

Specifications

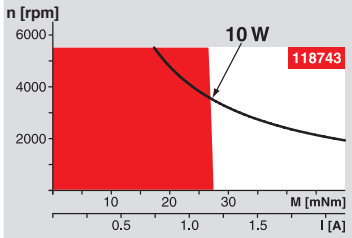
	Thermal data	
17	Thermal resistance housing-ambient	14 K/W
18	Thermal resistance winding-housing	3.1 K/W
19	Thermal time constant winding	12.5 s
20	Thermal time constant motor	612 s
21	Ambient temperature	-20…+85°C
22	Max. permissible winding temperature	+100°C
	Mechanical data (ball bearings)	
23	Max. permissible speed	5500 rpm
24	Axial play	0.05 - 0.15 mm
25	Radial play	0.025 mm
26	Max. axial load (dynamic)	3.2 N
27	Max. force for press fits (static)	64 N
	(static, shaft supported)	800 N
28	Max. radial loading, 5 mm from flange	16 N
	Other specifications	
29	Number of pole pairs	1
30	Number of commutator segments	11
31	Weight of motor	130 g

CLL = Capacitor Long Life

Values listed in the table are nominal.
Explanation of the figures on page 49.

Option
Preloaded ball bearings

Operating Range



Comments

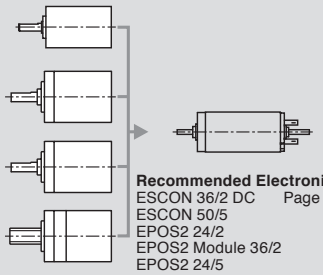
Continuous operation
In observation of above listed thermal resistance (lines 17 and 18) the maximum permissible winding temperature will be reached during continuous operation at 25°C ambient.
= Thermal limit.

Short term operation
The motor may be briefly overloaded (recurring).

Assigned power rating

maxon Modular System — Overview on page 16 - 21

Planetary Gearhead
Ø26 mm
0.5 - 4.5 Nm
Page 231/232
Planetary Gearhead
Ø32 mm
0.75 - 6.0 Nm
Page 234/235/237
Koaxdrive
Ø32 mm
1.0 - 4.5 Nm
Page 240
Spindle Drive
Ø32 mm
Page 255/256/257



Recommended Electronics:

ESCON 36/2 DC	Page 292
ESCON 50/5	292
EPOS2 24/2	312
EPOS2 Module 36/2	312
EPOS2 24/5	313
EPOS2 50/5	313
EPOS2 P 24/5	316
EPOS3 70/10 EtherCAT	319
Notes	**18**

Encoder MR
128 - 1000 CPT,
3 channels
Page 272
Encoder Enc
22 mm
100 CPT, 2 channels
Page 274
Encoder HED_ 5540
500 CPT,
3 channels
Page 276/278
DC-Tacho DCT
Ø22 mm
0.52 V
Page 286

Figure 13.15: The data sheet for the Maxon RE 25 motor. The columns correspond to different windings for different nominal voltages.

(a) Spin the motor shaft by hand. Get a feel for the rotor inertia and friction. Try to spin the shaft fast enough that it continues spinning briefly after you let go of it.

(b) Now short the motor terminals by electrically connecting them. Spin again by hand, and try to spin the shaft fast enough that it continues spinning briefly after you let go of it. Do you notice the short-circuit damping?

(c) Try measuring your motor's resistance using your multimeter. It may vary with the angle of the shaft, and it may not be easy to get a steady reading. What is the minimum value you can get reliably? To double-check your answer, you can power your motor and use your multimeter to measure the current as you stall the motor's shaft by hand.

(d) Attach one of your motor's terminals to scope ground and the other to an analog input. Spin the motor shaft by hand and observe the motor's back-emf.

(e) Power the motor's encoder, attach the A and B encoder channels to your oscilloscope, and make sure the encoder ground and scope ground are connected together. Do *not* power the motor. (The motor inputs should be disconnected from anything.) Spin the motor shaft by hand and observe the encoder pulses, including their relative phase.

(f) Now power your motor with a low-voltage battery pack. Given the number of lines per revolution of the encoder, and the rate of the encoder pulses you observe on your scope, calculate the motor's no-load speed for the voltage you are using.

(g) Work with a partner. Couple your two motor shafts together by tape or flexible tubing. (This may only work if your motor has no gearhead.) Now plug one terminal of one of the motors (we'll call it the *passive* motor) into one channel of a scope, and plug the other terminal of the passive motor into GND of the same scope. Now power the other motor (the *driving* motor) with a battery pack so that both motors spin. Measure the speed of the passive motor by looking at its encoder count rate on your scope. Also measure its back-emf. With this information, calculate the passive motor's torque constant k_t.

5. Using techniques discussed in this chapter, or techniques you come up with on your own, create a data sheet with all 21 entries for your nominal voltage. Indicate how you calculated the entry. (Did you do an experiment for it? Did you calculate it from other entries? Or did you do estimate by more than one method to cross-check your answer?) For the friction entry, you can assume Coulomb friction only—the friction torque opposes the rotation direction ($b_0 \neq 0$), but is independent of the speed of rotation ($b_1 = 0$). For your measurement of inductance, turn in an image of the scope trace you used to estimate ω_n and L, and indicate the value of C that you used.

   If there are any entries you are unable to estimate experimentally, approximate, or calculate from other values, simply say so and leave that entry blank.

6. Based on your data sheet from above, draw the speed-torque curves described below, and answer the associated questions. Do not do any experiments for this exercise; just extrapolate your previous results.

   (a) Draw the speed-torque curve for your motor. Indicate the stall torque and no-load speed. Assume a maximum power the motor coils can dissipate continuously before overheating and indicate the continuous operating regime. Given this, what is the power rating P for this motor? What is the max mechanical power $P_{\max}$?

   (b) Draw the speed-torque curve for your motor assuming a nominal voltage four times larger than in Exercise 6a. Indicate the stall torque and no-load speed. What is the max mechanical power $P_{\max}$?

7. You are choosing a motor for the last joint of a new direct-drive robot arm design. (A direct-drive robot does not use gearheads on the motors, creating high speeds with low friction.) Since it is the last joint of the robot, and it has to be carried by all the other joints, you want it to be as light as possible. From the line of motors you are considering from your favorite motor manufacturer, you know that the mass increases with the motor's power rating. Therefore you are looking for the lowest power motor that works for your specifications. Your specifications are that the motor should have a stall torque of at least 0.1 Nm, should be able to rotate at least 5 revolutions per second when providing 0.01 Nm, and the motor should be able to operate continuously while providing 0.02 Nm. Which motor do you choose from Table 13.1? Give a justification for your answer.

Assigned power rating	W	1.6	3.2	4.5	10	20	90
Nominal voltage	V	15	15	15	15	15	15
No load speed	rpm	8080	6010	14000	4990	9640	7070
Stall torque	mNm	4.2	14.9	31.4	134	222	872
Speed/torque gradient	rpm/mNm	1970	406	453	37.5	45.2	8.45
No load current	mA	5.69	3.72	37.1	24.2	60.7	245
Starting current	mA	243	628	3120	4680	15600	44900
Terminal resistance	Ohm	61.8	23.9	4.81	3.2	0.964	0.334
Max permissible speed	rpm	11000	7600	16000	5500	11000	8200
Max continuous current	mA	135	230	514	1030	1500	4000
Max continuous torque	mNm	2.34	5.45	5.17	29.36	21.45	77.7
Max power out at nominal voltage	mW	872	2330	11300	17400	52500	152000
Max efficiency	%	72.1	85.5	79.2	86.3	82.3	80.6
Torque constant	mNm/A	17.3	23.7	10.1	28.5	14.3	19.4
Speed constant	rpm/V	551	403	948	334	669	491
Mechanical time constant	ms	10.1	4.96	5.92	4.07	4.77	5.8
Rotor inertia	gcm^2	0.489	1.17	1.25	10.4	10.1	65.5
Terminal inductance	mH	1.15	0.99	0.17	0.35	0.09	0.09
Thermal resistance housing-ambient	K/W	35	30	30	14	14	6.2
Thermal resistance rotor-housing	K/W	8.2	8.5	8.5	3.1	3.1	2

Table 13.1: Motors to choose from.

Chapter 14

Gearing and Motor Sizing

The mechanical power produced by a DC motor is a product of its torque and angular velocity at the output shaft. Even if a DC motor provides enough power for a given application, it may rotate at too high a speed (up to thousands of rpm), and too low a torque, to be useful for a typical application. In this case, we can add a gearhead to the output shaft to decrease the speed by a factor of $G > 1$ and to increase the torque by a similar factor. In rare cases, we can choose $G < 1$ to actually increase the output speed.

In this chapter we discuss options for gearing the output of a motor, and how to choose a DC motor and gearing combination that works for your application.

14.1 Gearing

Gearing takes many forms, including different kinds of rotating meshing gears, belts and pulleys, chain drives, cable drives, and even methods for converting rotational motion to linear motion, such as racks and pinions, lead screws, and ball screws. All transform torques/forces and angular/linear velocities while ideally preserving mechanical power. For specificity, we refer to a gearhead with an input shaft (attached to the motor shaft) and an output shaft.

Figure 14.1 shows the basic idea. The input shaft is attached to an input gear A with N teeth, and the output shaft is attached to an output gear B with GN teeth, where typically $G > 1$. The meshing of these teeth enforce the relationship

$$\omega_{\text{out}} = \frac{1}{G}\omega_{\text{in}}.$$

Ideally the meshing gears preserve mechanical power, so $P_{\text{in}} = P_{\text{out}}$, which implies

$$\tau_{\text{in}}\omega_{\text{in}} = P_{\text{in}} = P_{\text{out}} = \frac{1}{G}\omega_{\text{in}}\tau_{\text{out}} \quad \rightarrow \quad \tau_{\text{out}} = G\tau_{\text{in}}.$$

It is common to have multiple stages of gearing (Figure 14.2(a)), so the output shaft described above has a second, smaller gear which becomes the input of the next stage. If the gear ratios of the two stages are G_1

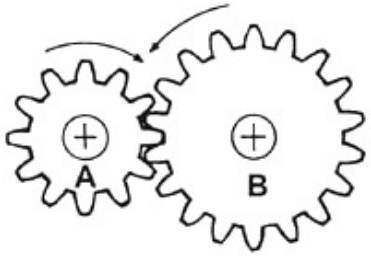


Figure 14.1: The input gear A has 12 teeth and the output gear B has 18, making the gear ratio $G = 1.5$.

and G_2, the total gear ratio is $G = G_1 G_2$. Multi-stage gearheads can make huge reductions in speed and increases in torque, up to ratios of hundreds or more.

14.1.1 Practical Issues

Efficiency. In practice, some power is lost due to friction and impacts between the teeth. This is often modeled by an efficiency coefficient $\eta < 1$, such that $P_{\text{out}} = \eta P_{\text{in}}$. Since the teeth enforce the ratio G between input and output velocities, the power loss must show up as a decrease in the available output torque, i.e.,

$$\omega_{\text{out}} = \frac{1}{G}\omega_{\text{in}} \quad \tau_{\text{out}} = \eta G \tau_{\text{in}}.$$

The total efficiency of a multi-stage gearhead is the product of the efficiencies of each stage individually, i.e., $\eta = \eta_1 \eta_2$ for a two-stage gearhead. As a result, high ratio gearheads may have relatively low efficiency.

Backlash. *Backlash* refers to the angle that the output shaft of a gearhead can rotate without the input shaft moving. Backlash arises due to tolerance in manufacturing; the gear teeth need a little bit of play to avoid jamming when they mesh. An inexpensive gearhead may have backlash of a degree or more, while more expensive precision gearheads have nearly zero backlash. Backlash typically increases with the number of gear stages. Some gear types, notably harmonic drive gears (see Section 14.1.2) are specifically designed for near-zero backlash, usually by making use of flexible elements.

Backlash can be a serious issue in controlling endpoint motions, due to the limited resolution of sensing the gearhead output shaft angle using an encoder attached to the motor shaft (the input of the gearhead).

Backdrivability. *Backdrivability* refers to the ability to drive the output shaft of a gearhead with an external device (or by hand), i.e., to backdrive the gearing. Typically the motor and gearhead combination is less backdrivable for higher gear ratios, due to the higher friction in the gearhead and the higher apparent inertia of the motor. Backdrivability also depends on the type of gearing. In some applications we don't want the motor and gearhead to be backdrivable (e.g., if we want the gearhead to act as a kind of brake that prevents motion when the motor is turned off), and in others backdrivability is highly desirable (e.g., in virtual environment haptics applications, where the motor is used to create forces on a user's hand).

Input and output limits. The input and output shafts and gears, and the bearings that support them, are subject to practical limits on how fast they can spin and how much torque they can support. Gearheads will often have maximum input velocity and maximum output torque specifications reflecting these limits. For example, you can't assume that you get a 10 Nm actuator by adding a $G = 10$ gearhead to a 1 Nm motor; you must make sure that the gearhead is rated for 10 Nm output torque.

14.1.2 Examples

Figure 14.2 shows several different gear types. Not shown are cable, belt, and chain drives, which can also be used to implement a gear ratio while transmitting torques over distances.

Spur gearhead. Figure 14.2(a) shows a multi-stage *spur* gearhead. To keep the spur gearhead package compact, typically each stage has a gear ratio of only 2 or 3; larger gear ratios would require large gears.

Planetary gearhead. A planetary gearhead has an input rotating a *sun* gear and an output attached to a *planet carrier* (Figure 14.2(b)). The sun gear meshes with the planets, which also mesh with an internal gear. An advantage of the planetary gearhead is that more teeth mesh, allowing higher torques.

Bevel gears. Bevel gears (Figure 14.2(c)) can be used to implement a gear ratio as well as to change the axis of rotation.

Worm gears. The screw-like input *worm* interfaces with the output *worm gear* in Figure 14.2(d), making for a large gear ratio in a compact package.

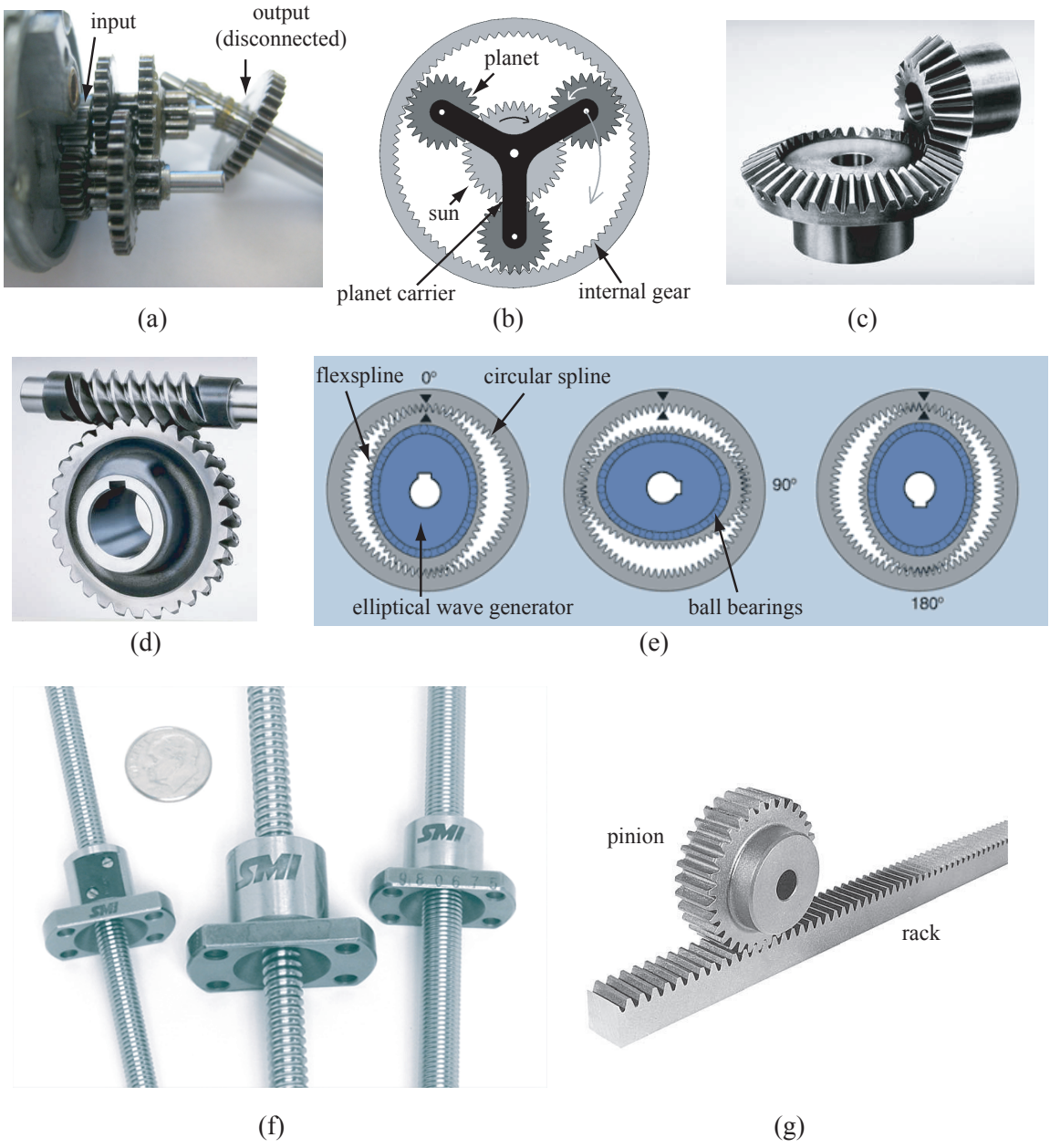


Figure 14.2: (a) Multi-stage spur gearhead. (b) A planetary gearhead. (c) Bevel gears. (d) Worm gears. (e) Harmonic drive gearhead. (f) Ball screws. (g) Rack and pinion.

Harmonic drive. The *harmonic drive* gearhead (Figure 14.2(e)) has an elliptical *wave generator* attached to the input shaft and a flexible *flexspline* attached to the output shaft. Ball bearings between the wave generator and the flexibility of the flexspline allow them to move smoothly relative to each other. The flexspline teeth engage with a rigid external *circular spline*. As the wave generator completes a full revolution, the teeth of the flexspline may have moved by as little as one tooth relative to the circular spline. Thus the harmonic drive can implement a high gear ratio (for example $G = 50$ or 100) in a single stage with essentially zero backlash. Harmonic drives can be quite expensive.

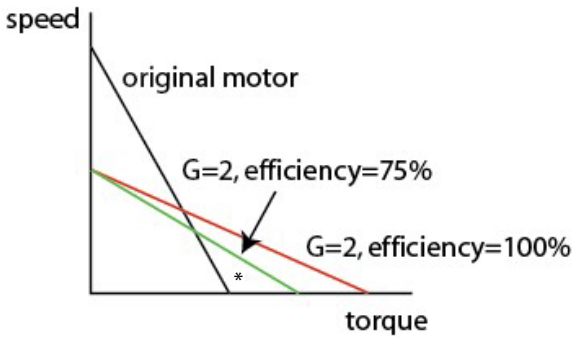


Figure 14.3: The effect of gearing on the speed-torque curve. The operating point * is possible with the gearhead, but not without.

Ball screw and lead screw. A ball screw or lead screw (Figure 14.2(f)) is aligned with the axis of, and coupled to, the motor's shaft. As the screw rotates, a nut on the screw translates along the screw. The nut is prevented from rotating (and therefore must translate) by linear guide rods passing through the nut. The holes in the nuts in Figure 14.2(f) are clearly visible. A lead screw and a ball screw are basically the same thing, but a ball screw has ball bearings in the nut to reduce friction with the screw.

Ball and lead screws convert rotational motion to linear motion. The ratio of the linear motion to the rotational motion is specified by the *lead* of the screw.

Rack and pinion. The rack and pinion (Figure 14.2(g)) is another way to convert angular to linear motion. The rack is typically mounted to a part on a linear slide.

14.2 Choosing a Motor and Gearhead

14.2.1 Speed-Torque Curve

Figure 14.3 illustrates the effect of a gearhead with $G = 2$ and efficiency $\eta = 0.75$ on the speed-torque curve. The continuous operating torque also increases by a factor ηG, or 1.5 in this example. When choosing a motor and gearing combination, the expected operating points should lie under the geared speed-torque curve, and continuous operating points should have torques less than $\eta G \tau_c$, where τ_c is the continuous torque of the motor alone.

14.2.2 Inertia and Reflected Inertia

If you spin the shaft of a motor by hand, you can feel its rotor inertia directly. If you spin the output shaft of a gearhead attached to the motor, however, you feel the *reflected inertia* of the rotor through the gearbox. Say I_m is the inertia of the motor, ω_m is the angular velocity of the motor, and $\omega_{\text{out}} = \omega_m / G$ is the output velocity of the gearhead. Then we can write the kinetic energy of the motor as

$$KE = \frac{1}{2} I_m \omega_m^2 = \frac{1}{2} I_m G^2 \omega_{\text{out}}^2 = \frac{1}{2} I_{\text{ref}} \omega_{\text{out}}^2,$$

and $I_{\text{ref}} = G^2 I_m$ is called the reflected (or apparent) inertia of the motor.

Commonly the gearbox output shaft is attached to a rigid body load. For a rigid body consisting of point masses, the inertia I_{load} about the axis of rotation is calculated as

$$I_{\text{load}} = \sum_{i=1}^{N} m_i r_i^2,$$

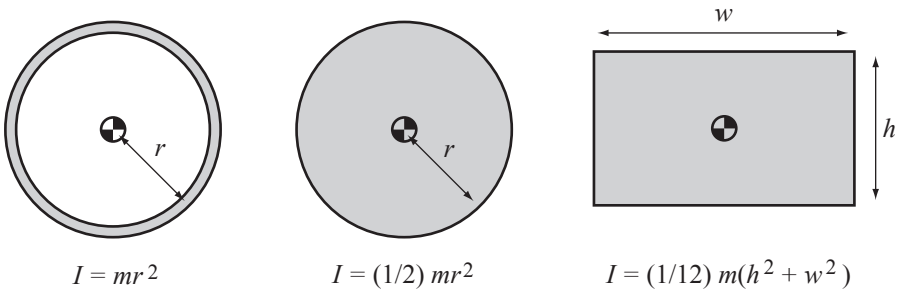


Figure 14.4: Inertia for an annulus, a solid disk, and a rectangle, each of mass m, about an axis out of the page and through the center of mass.

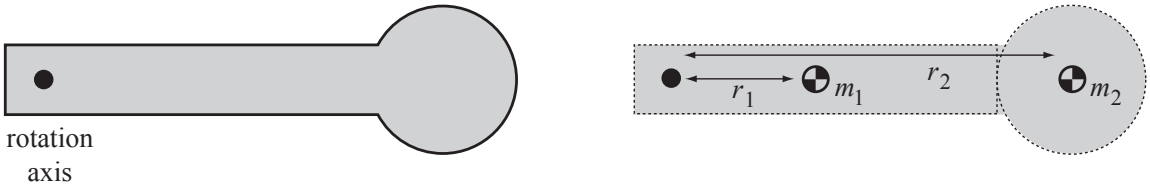


Figure 14.5: The body on the left can be approximated by the rectangle and disk on the right. If the inertias of the two bodies (about their centers of mass) are I_1 and I_2, then the approximate inertia of the compound body about the rotation axis is $I = I_1 + m_1 r_1^2 + I_2 + m_2 r_2^2$ by the parallel-axis theorem.

where m_i is the mass of point i and r_i is its distance from the axis of rotation. In the case of a continuous body, the discrete sum becomes the integral

$$I_{\text{load}} = \int_V \rho(\mathbf{r}) r^2 dV(\mathbf{r}),$$

where $\mathbf{r}$ refers to the location of a point on the body, r is the distance of that point to the rotation axis, $\rho(\mathbf{r})$ is the mass density at that point, V is the volume of the body, and dV is a differential volume element. Solutions to this equation are given in Figure 14.4 for some simple bodies of mass m and uniform density.

If the inertia of a body about its center of mass is I_{cm}, then the inertia I' about a parallel axis a distance r from the center of mass is

$$I' = I_{\text{cm}} + mr^2.$$

This is called the *parallel-axis theorem*. With the parallel-axis theorem and the formulas in Figure 14.4, we can approximately calculate the inertia of a load consisting of multiple bodies (Figure 14.5). Typically I_{load} is significantly larger than I_m, but with the gearing, the reflected inertia of the motor $G^2 I_m$ may be as large or larger than I_{load}.

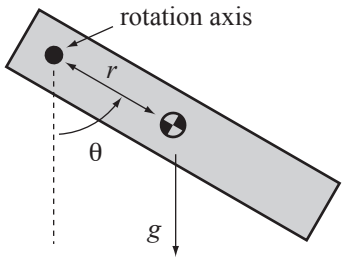


Figure 14.6: A load in gravity.

Given a load of mass m and inertia I_{load} (about the gearhead axis) in gravity as shown in Figure 14.6, and a desired acceleration $\alpha > 0$ (counterclockwise), we can calculate the torque needed to achieve the acceleration:

$$\tau = (G^2 I_m + I_{\text{load}})\alpha + mgr\sin\theta.$$

Given angular velocities at which we would like this acceleration to be possible, we have a set of speed-torque points that must lie under the speed-torque curve (transformed by the gearhead).

14.2.3 Choosing a Motor and Gearhead

To choose a motor and gearing combination, consider the following factors:

- The motor can be chosen based on the peak mechanical power required for the task. If the motor's power rating is sufficient, then theoretically we can follow by choosing a gearhead to give the necessary speed and torque. Our choice of motor might also be constrained by the voltage supply available for the application.
- The maximum velocity needed for the task should be less than ω_0/G, where ω_0 is the no-load speed of the motor.
- The maximum torque needed for the task should be less than $G\tau_{\text{stall}}$, where τ_{stall} is the motor's stall torque.
- Any required operating point (τ, ω) must lie below the gearing-transformed speed-torque curve.
- If the motor will be used continuously, then the torques during this continuous operation should be less than $G\tau_c$, where τ_c is the maximum continuous torque of the motor.

To account for the efficiency η of the gearhead and other uncertain factors, it is a good idea to oversize the motor by a fudge factor of 1.5 or 2.

Subject to the hard constraints specified above, we might wish to find an "optimal" design, e.g., to minimize the cost of the motor and gearing, its weight, or the electrical power consumed by the motor. One type of optimization is called *inertia matching*.

Inertia Matching Given the motor inertia I_m and the load inertia I_{load}, the system is inertia matched if the gearing G is chosen so that the load acceleration α is maximized for any given motor torque τ_m. We can express the load acceleration as

$$\alpha = \frac{G\tau_m}{I_{\text{load}} + G^2 I_m}.$$

The derivative with respect to G is

$$\frac{d\alpha}{dG} = \frac{(I_{\text{load}} - G^2 I_m)\tau_m}{(I_{\text{load}} + G^2 I_m)^2}$$

and solving $d\alpha/dG = 0$ yields

$$G = \sqrt{\frac{I_{\text{load}}}{I_m}},$$

or $G^2 I_m = I_{\text{load}}$, hence the term "inertia matched." With this choice of gearing, half of the torque goes to accelerating the motor's inertia and half goes to accelerating the load inertia.

14.3 Chapter Summary

- For gearing with a gear ratio G, the output angular velocity is $\omega_{\text{out}} = \omega_{\text{in}}/G$ and the ideal output torque is $\tau_{\text{out}} = G\tau_{\text{in}}$, where ω_{in} and τ_{in} are the input angular velocity and torque, respectively. If the gear efficiency $\eta < 1$ is taken into account, the output torque is $\tau_{\text{out}} = \eta G\tau_{\text{in}}$.
- For a two-stage gearhead with gear ratios G_1 and G_2 and efficiencies η_1 and η_2 for the individual stages, the total gear ratio is $G_1 G_2$ and total efficiency is $\eta_1\eta_2$.

- Backlash refers to the amount the output of the gearing can move without motion of the input.
- The reflected inertia of the motor (the apparent inertia of the motor from the output of the gearhead) is $G^2 I_m$.
- A motor and gearing system is inertia matched with its load if

$$G = \sqrt{\frac{I_{\text{load}}}{I_m}}.$$

14.4 Exercises

1. You are designing gearheads using gears with 10, 15, and 20 teeth. When the 10- and 15-teeth gears mesh, you have $\eta = 85\%$. When the 15-and 20-teeth gear mesh, you have $\eta = 90\%$. When the 10- and 20-teeth gear mesh, you have $\eta = 80\%$.
   (a) For a one-stage gearhead, what gear ratios $G > 1$ can you achieve, and what are their efficiencies?
   (b) For a two-stage gearhead, what gear ratios $G > 1$ can you achieve, and what are their efficiencies? Consider all possible combinations of one-stage gearheads.
2. The inertia of the motor's rotor is I_m, and its load is a uniform solid disk, which will be centered on the gearhead output shaft. The disk has a mass m and a radius R. For what gear ratio G is the system inertia matched?
3. The inertia of the motor's rotor is I_m, and its load is a propeller with three blades. You model the propeller as a simple planar body consisting of a uniform-density solid disk of radius R and mass M, with each blade a uniform-density solid rectangle extending from the disk. Each blade has mass m, length ℓ, and (small) width w.
   (a) What is the inertia of the propeller? (Since a propeller must push air to be effective, ideally our model of the propeller inertia would include the *added mass* of the air being pushed, but we leave that out here.)
   (b) What gear ratio G provides inertia matching?
4. You are working for a startup robotics company designing a small differential-drive mobile robot, and your job is to choose the motors and gearing. A diff-drive robot has two wheels, each driven directly by its own motor, as well as a caster wheel or two for balance. Your design specs say that the robot should be capable of continuously climbing a 20° slope at 20 cm/s. To keep things simple, assume that the mass of the whole robot, including motor amplifiers, motors, and gearing, will be 2 kg, regardless of the motors and gearing you choose. Further assume that the robot must overcome a viscous damping force of (10 Ns/m)$*v$ when it moves forward at a constant velocity v, regardless of the slope. The radius of the wheels has already been chosen to be 4 cm, and you can assume they never slip. If you need to make other assumptions to complete the problem, clearly state them.

   You will choose among the 15 V motors in Table 13.1, as well as gearheads with $G = 1$, 10, 20, 50, or 100. Assume the gearing efficiency η for $G = 1$ is 100%, and for the others, 75%. (Do not combine gearheads! You get to use only one.)
   (a) Provide a list of all combinations of motor and gearhead that satisfy the specs, and explain your reasoning. (There are 30 possible combinations: 6 motors and 5 gearheads.) "Satisfy the specs" means that the motor and gearhead can provide at least what is required by the specifications. Remember that each motor only needs to provide half of the total force needed, since there are two wheels.
   (b) To optimize your design, you decide to use the motor with the lowest power rating, since it is the least expensive. You also decide to use the lowest gear ratio that works with this motor. (Even though we are not modeling it, a lower gear ratio likely means higher efficiency, less backlash, less mass in a smaller package, a higher top-end speed [though lower top-end torque], and lower cost.) Which motor and gearing do you choose?

(c) Instead of optimizing the cost, you decide to optimize the power efficiency—the motor and gearing combination that uses the least electrical power when climbing up the 20° slope at a constant 20 cm/s. This is in recognition that battery life is very important to your customers. Which motor and gearhead do you choose?

(d) Forget about your previous answers, satisfying the specs, or the limited set of gear ratios. If the motor you choose has rotor inertia I_m, half of the mass of the robot (including the motors and gearheads) is M, and the mass of the wheels is negligible, what gear ratio would you choose to achieve inertia matching? If you need to make other assumptions to complete the problem, clearly state them.

Chapter 15

DC Motor Control

Driving a brushed DC motor with variable speed and torque requires variable high current. A microcontroller is capable of neither variable analog output nor high current. Both problems are solved through the use of digital PWM and an H-bridge. The H-bridge consists of a set of switches that are rapidly opened and closed by the microcontroller's PWM signal, alternately connecting and disconnecting high voltage to the motor. The effect is similar to the time-average of the voltage. Motion control of the motor is achieved using motor position feedback, typically from an encoder.

15.1 The H-bridge and Pulse Width Modulation

Let's consider a series of improving ideas for driving a motor. By attempting to fix their problems, we arrive at the H-bridge.

Direct Driving from a Microcontroller Pin (Figure 15.1(a)) The first idea is to simply connect a microcontroller digital output pin to one motor lead and connect the other motor lead to a positive voltage. The pin can alternate between low (ground) and high impedance (disconnected, or "tristated"). This is a bad idea, of course, since a typical digital output can only sink a few milliamps, and most motors must draw much more than that to do anything useful.

Current Amplification with a Switch (Figure 15.1(b)) To increase the current, we can use the digital pin to turn a switch on and off. The switch could be an electromechanical relay or a transistor, and it allows a much larger current to flow through the motor.

Consider, however, what happens when the switch has been closed for a while. A large current I_0 is flowing through the motor, which electrically behaves like a resistor and inductor in series. When the switch opens, what happens? The voltage across the inductor is governed by

$$V_L = L\frac{dI}{dt},$$

so the instantaneous drop in current from I_0 to zero means that a large (theoretically infinite) voltage develops across the motor leads. The large voltage means that a spark will occur between the switch and the motor lead it was recently attached to. This is certainly not a good thing for the microcontroller.

Adding a Flyback Diode (Figure 15.1(c)) To prevent the instantaneous change in current and sparking, a *flyback diode* can be added to the circuit. Now when the closed switch is opened, the motor's current has a path to flow through—the diode in parallel with the motor. The voltage across the motor instantaneously changes from +V to the negative of the forward bias voltage of the diode, but that's OK, there is nothing in the resistor-inductor-diode circuit that tries to prevent that. With the switch open, the energy stored in the motor's inductance is dissipated by the current flowing through the motor's resistance, and the initial current

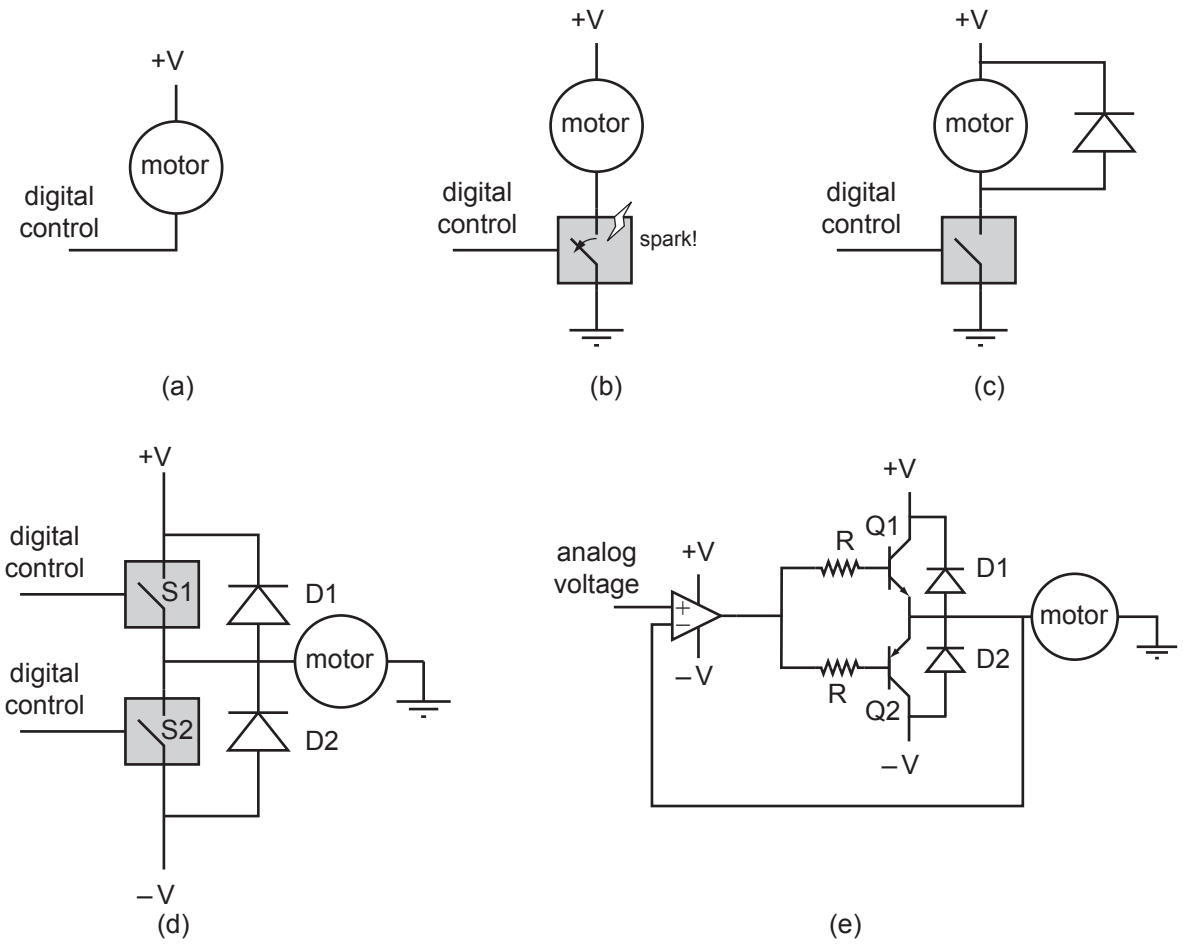


Figure 15.1: A progression of ideas to drive a motor, from worst to better. (a) Attempting to drive directly from a digital output. (b) Using a digital output to control a switch that allows more current to flow. (c) Adding a flyback diode to prevent sparking. (d) Using two switches and a bipolar supply to run the motor bidirectionally. (e) Using an analog control signal, an op-amp, and two transistors to make a linear push-pull amplifier.

I_0 will drop smoothly. Assuming the diode's forward bias voltage is zero, and treating the motor as a resistor and inductor in series, Kirchoff's voltage law tells us that the voltage around the closed loop satisfies

$$L\frac{dI}{dt}(t) + RI(t) = 0,$$

and the current through the motor after opening the switch can be solved as

$$I(t) = I_0 e^{-\frac{R}{L}t},$$

a first-order exponential drop from the initial current I_0 to zero, with time constant equal to the electrical time constant of the motor, $T_e = L/R$.

When the switch is closed, the flyback diode has no effect on the circuit. Flyback diodes should be capable of carrying a lot of current, should be fast switching, and should have low forward bias voltage.

This circuit represents a viable approach to controlling a motor with variable speed: by opening and closing the switch rapidly, it is possible to create a variable average voltage across, and current through, the motor, depending on the duty cycle of the switching. The current always has the same sign, though, so the motor can only be driven in one direction.

Bidirectional Operation with Two Switches and a Bipolar Supply (Figure 15.1(d)) By using a bipolar power supply (+V, −V, and GND) and two switches, each controlled by a separate digital input, it is possible to achieve bidirectional motion. With the switch S1 closed and S2 open, current flows through the motor from +V to GND. With S2 closed and S1 open, current flows in the opposite direction, from GND to −V. Two flyback diodes are used to provide current paths when switches transition from closed to open. For example, if S2 is open and S1 switches from closed to open, the motor current that was formerly provided by S1 now comes from −V through the diode D2.

To prevent a short circuit, S1 and S2 should never be closed simultaneously.

Bidirectional average voltages between +V and −V are determined by the duty cycle of the rapidly opening and closing switches.

A drawback of this approach is that a bipolar power supply is needed. This issue is solved by the H-bridge. But before discussing the H-bridge, let's consider a commonly-used variation of the circuit in Figure 15.1(d).

Linear Push-Pull Amplifier (Figure 15.1(e)) Figure 15.1(e) shows a linear push-pull amplifier. The control signal is a low-current analog voltage to an op-amp configured with negative feedback. Because of the negative feedback and the effectively infinite gain of the op-amp, the op-amp output does whatever it can to make sure that the signals at the inverting and non-inverting inputs are equal. Since the inverting input is connected to one of the motor leads, the voltage across the motor is equal to the control voltage at the non-inverting input, except with higher current available due to the output transistors. Only one of the two transistors is active at a time: either the NPN bipolar junction transistor Q1 "pushes" current from +V, through the motor, to GND, or the PNP BJT Q2 "pulls" current from GND, through the motor, to −V. Thus the op-amp provides a high impedance input and voltage following of the low-current control voltage, while the transistors provide current amplification.

As an example, if +V = 10 V, and the control signal is at 6 V, then the op-amp ensures 6 V across the motor. To double-check that our circuit works as we expect, we calculate the current that would flow through the motor when it is stalled. If the motor's resistance is 6 Ω, then the current $I_e = 6$ V/6 Ω = 1 A must be provided by the emitter of Q1. If the transistor is capable of providing that much current, we then check if the op-amp is capable of providing the base current $I_b = I_e/(\beta + 1)$ required to activate the transistor, where β is the transistor gain. If so, we are in good shape. The voltage at the base of Q1 is a PN diode drop higher than the voltage across the motor, and the voltage at the op-amp output is that base voltage plus $I_b R$. Note that Q1 is dissipating power approximately equal to the 4 V between the collector and the emitter times the 1 A emitter current, or approximately 4 W. This power goes to heating the transistor, so the transistor must be heat-sinked to allow it to dissipate the heat without burning up.

An example application of a linear push-pull amplifier would be using a rotary knob to control a motor's bidirectional speed. The ends of a potentiometer in the knob would be connected to +V and −V, with the wiper voltage serving as the control signal.

If the op-amp by itself can provide enough current, the op-amp output can be connected directly to the motor and flyback diodes, eliminating the resistors and transistors. Power op-amps are available, but they tend to be expensive relative to using output transistors to boost current.

We could instead eliminate the op-amp by connecting the control signal directly to the base resistors of the transistors. The drawback is that neither transistor would be activated for control signals between approximately −0.7 V and 0.7 V, or whatever the base-emitter voltage is when the transistors are activated. We have a "deadband" from the control signal to the motor voltage.

Some issues with the linear push-pull amp, addressed by the H-bridge, include:

- A bipolar power supply is required.
- The control signal is an analog voltage, which is generally not available from a microcontroller.
- The output transistors operate in the linear regime, with significant voltage between the collectors and emitters. A transistor in the linear mode dissipates power approximately equal to the current through the transistor multiplied by the voltage across it. This heats the transistor and wastes power.

Linear push-pull amps are sometimes used when power dissipation and heat are not a concern. They are also common in speaker amplifiers. (A speaker is a current-carrying coil moving in a magnetic field, essentially

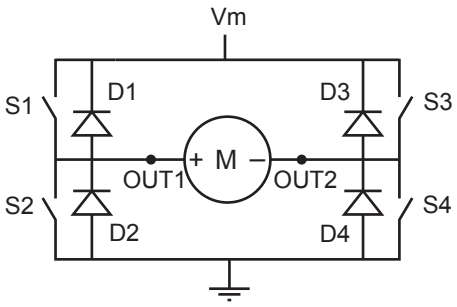


Figure 15.2: An H-bridge constructed of four switches and four flyback diodes. OUT1 and OUT2 are the H-bridge outputs, attached to the two terminals of the DC motor.

a linear motor.) There are many improvements to, and variations on, the basic circuit in Figure 15.1(e), and audio applications have raised amplifier circuit design to an art form. You can use a commercial audio amplifier to drive a DC motor, but you would have to remove the high-pass filter on the amplifier input. The high-pass filter is there because we can't hear sound below 20 Hz, and low-frequency currents simply heat up the speaker coil without producing audible sound.

15.1.1 The H-bridge

For most motor applications, the preferred amplifer is an H-bridge (Figure 15.2). An H-bridge uses a unipolar power supply (Vm and GND), is controlled by digital pulse width modulation pulse trains that can be created by a microcontroller, and has output transistors (switches) operating in the saturated mode, therefore with little voltage drop across them and relatively little power wasted as heat.

An H-bridge consists of four switches, S1–S4, typically implemented by MOSFETs, and four flyback diodes D1–D4.[1] An H-bridge can be used to run a DC motor bidirectionally, depending on which switches are closed:

closed switches	voltage across motor
S1, S4	positive (forward rotation)
S2, S3	negative (reverse rotation)
S1, S3	zero (short-circuit braking)
S2, S4	zero (short-circuit braking)
none or one	open circuit (coasting)

Switch settings not covered in the table (S1 and S2 closed, or S3 and S4 closed, or any set of three or four switches closed) result in a short circuit and should obviously be avoided!

While you can build your own H-bridge out of discrete components, it is often easier to buy one packaged in an integrated circuit, particularly for low-power applications. Apart from reducing your component count, these ICs also make it impossible for you to accidentally cause a short circuit. An example H-bridge IC is the Texas Instruments DRV8835.

The DRV8835 has two full H-bridges, labeled A and B, each capable of providing up to 1.5 A continuously to power two separate motors. The two H-bridges can be used in parallel to provide up to 3 A to drive a single motor. The DRV8835 works with motor supply voltages (to power the motors) of up to 11 V and logic supply voltages (for interfacing with the microcontroller) between 2 V and 7 V. It offers two modes of operation: the IN/IN mode, where the two inputs for each H-bridge control whether the H-bridge is in the forward, reverse, braking, or coasting mode, and the PHASE/ENABLE mode, where one input controls whether the H-bridge is enabled or braking and the other input controls forward vs. reverse if the H-bridge is enabled. We will focus on the PHASE/ENABLE mode.

[1] MOSFETs themselves allow reverse currents, acting somewhat as flyback diodes, but typically flyback diodes are incorporated for better performance.

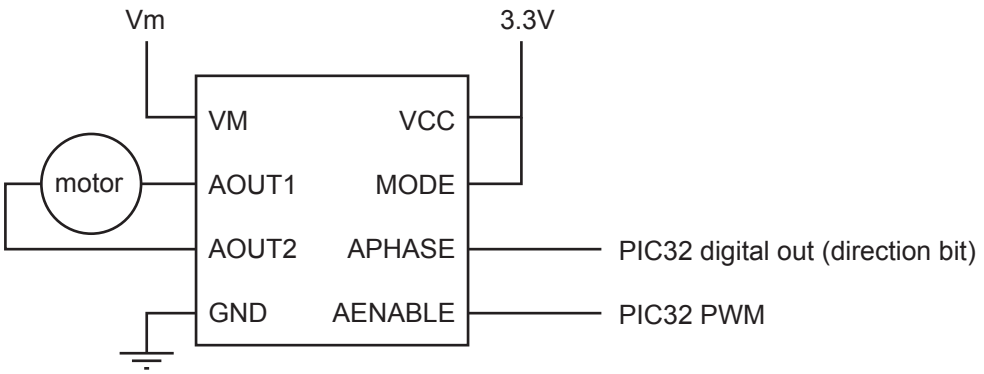


Figure 15.3: Wiring the DRV8835 to use H-bridge A. Not shown is a recommended 10 μF capacitor from VM to GND and a recommended 0.1 μF capacitor from VCC to GND, which may be included already on a DRV8835 breakout board.

In the PHASE/ENABLE mode, chosen by setting the MODE pin to logic high, the following truth table determines how the logic inputs (0 and 1) of one H-bridge controls its two outputs:

MODE	PHASE	ENABLE	OUT1	OUT2	function
1	x	0	L	L	short-circuit braking (S2, S4 closed)
1	0	1	H	L	forward (S1, S4 closed)
1	1	1	L	H	reverse (S2, S3 closed)

When the ENABLE pin is low, OUT1 and OUT2 are held at the same (low) voltage, causing the motor to brake by its own short-circuit damping. When ENABLE is high, then if PHASE is low, switches S1 and S4 are closed, putting a positive voltage across the motor trying to drive it in the forward direction. When ENABLE is high and PHASE is high, switches S2 and S3 are closed, putting a negative voltage across the motor trying to drive it in the reverse direction. PHASE sets the direction of current through the motor, and the duty cycle of the PWM on ENABLE determines the average voltage across the motor, which switches between approximately +Vm (or −Vm) and zero.

Figure 15.3 shows the wiring of the DRV8835 to use H-bridge A, where Vm is the voltage to power the motor. The logic high voltage VCC is 3.3 V. If the two H-bridges of the DRV8835 are used in parallel for more current, the following pins should be connected to each other: APHASE and BPHASE; AENABLE and BENABLE; AOUT1 and BOUT1; and AOUT2 and BOUT2.

15.1.2 Control with PWM

Rapidly switching ENABLE from high to low can effectively create an average voltage V_{ave} across the motor. Assuming PHASE = 0 (forward), then if DC is the duty cycle of ENABLE, where $0 \leq \text{DC} \leq 1$, and if we ignore voltage drops due to flyback diodes and resistance at the MOSFETs, then the average voltage across the motor is

$$V_{\text{ave}} \approx \text{DC} * \text{Vm}.$$

Ignoring the details of the motor's inductance charging and discharging, this yields an approximate average current through the motor of

$$I_{\text{ave}} \approx (V_{\text{ave}} - V_{\text{emf}})/R,$$

where $V_{\text{emf}} = k_t\omega$ is the back-emf. Since the period of a PWM cycle is typically much shorter than the motor's mechanical time constant T_m, the motor's speed ω (and therefore V_{emf}) is approximately constant during a PWM cycle.

Figure 15.4 shows a motor with positive average current (from left to right). When switches S1 and S4 are closed and S2 and S3 are open (ENABLE is 1, OUT1 is high, and OUT2 is low), S1 and S4 carry current from Vm, through the motor, to ground. When the PWM on ENABLE becomes 0, S2 and S4 are closed

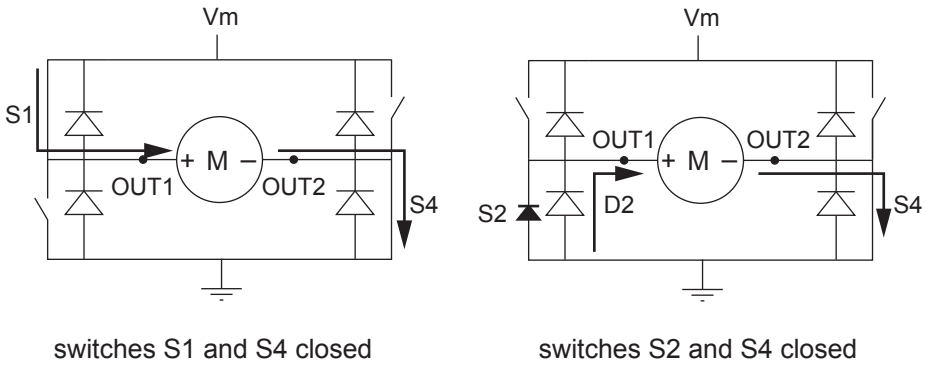


Figure 15.4: Left: The closed switches S1 and S4 provide current to the motor, from Vm to GND. Right: When S1 is opened and S2 is closed, the motor's need for a continuous current is satisfied by the flyback diode D2 and the switch S4. Current could also flow through the MOSFET S2, which acts like a diode to reverse current, but the flyback diode is designed to carry the current.

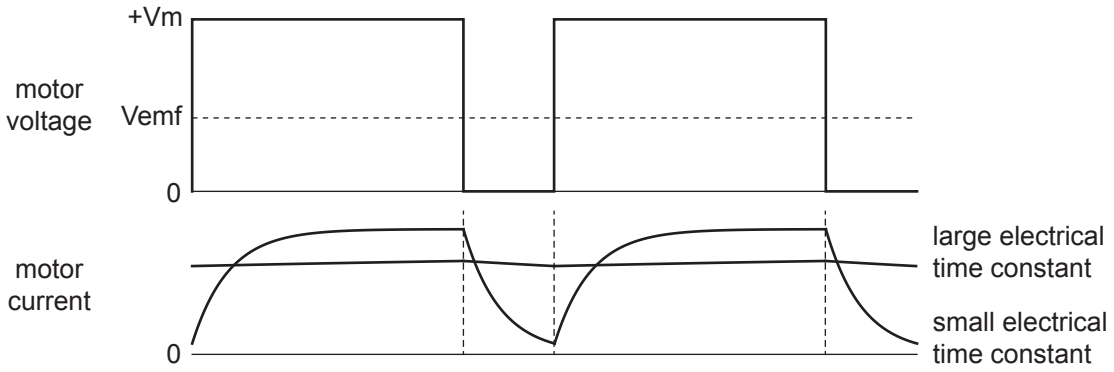


Figure 15.5: A PWM signal puts Vm across the motor for 75% of the PWM cycle and 0 V for 25% of the cycle. This causes a nearly constant positive current for a motor with large L/R, while the current for a motor with small L/R varies significantly.

and S1 and S3 are open (OUT1 and OUT2 are both low). Because the motor's inductance requires that the current not change instantaneously, the current must flow from ground, through the flyback diode D2, through the motor, then through switch S4 back to ground. (The "closed switch" S2 MOSFET is represented as a diode, since a MOSFET behaves similarly to a diode when current tries to flow in the reverse direction. But the flyback diode is designed to carry this current, so current flows through D2.)

During the period when OUT1 and OUT2 are low, the voltage across the motor is approximately zero, and the motor current $I(t)$ satisfies

$$0 = L\frac{dI}{dt}(t) + RI(t) + V_{\text{emf}},$$

causing an exponential drop in $I(t)$ with time constant $T_e = L/R$, the electrical time constant of the motor. Figure 15.5 shows an example of the voltage across the motor during two cycles of PWM, and the resulting current for two different motors: one with a large T_e and one with a small T_e. A large T_e results in a nearly constant current during a PWM cycle, while a small T_e results in a fast-changing current. The nearly constant motor velocity during a PWM cycle gives a nearly constant V_{emf}.

To understand the behavior of the electromechanical system under PWM control, we need to consider three time scales: the PWM period T, the electrical time constant T_e, and the mechanical time constant T_m. The PWM frequency $1/T$ should be chosen to be much higher than $1/T_m$, to prevent significant speed variation during a PWM cycle. Ideally the PWM frequency would also be much higher than $1/T_e$, to minimize

current variation during a cycle. One reason for this is that more-constant current results in less power wasted heating the motor coils. To see this, consider a motor with resistance $R = 1\ \Omega$ powered by a constant current of 2 A vs. a current alternating with 50% duty cycle between 4 A and 0 A, for a time-average of 2 A. Both provide the same average torque, but the average power to heating the coils in the first case is $I^2R = 4$ W while it is $0.5(4\text{ A})^2(1\ \Omega) = 8$ W in the second.

Another consideration is audible noise: to make sure the switching is not audible, the PWM frequency $1/T$ should be chosen at 20 kHz or larger.

On the other hand, the PWM frequency should not be chosen too high, as it takes time for the H-bridge MOSFETS to switch. During switching, when larger voltages are across the active MOSFETS, more power is wasted to heating. If switching occurs too often, the H-bridge may even overheat. The DRV8835 takes approximately 200 ns to switch, and its maximum recommended PWM frequency is 250 kHz.

Trading off the considerations above, common PWM frequencies are in the range 20–40 kHz.

15.1.3 Regeneration

When the voltage across the motor and the current through the motor have the same sign, the motor is consuming electrical power ($IV > 0$), converting some of it to mechanical power. When the voltage across the motor and the current through the motor have opposite signs ($IV < 0$), then the motor is converting mechanical power to electrical power. This is called *regeneration*. Regeneration may occur when braking a motor, for example. Regnerative braking is used in hybrid and electric cars to convert some of the car's kinetic energy into battery energy, instead of just wasting it heating the brake pads.

For concreteness, consider an H-bridge powered by 10 V, flyback diodes with a forward bias voltage of 0.7 V, and a motor with a resistance of 1 Ω and a torque constant of 0.01 Nm/A (0.01 Vs/rad). Consider these two examples of regeneration.

1. **Forced motion of the motor output.** Assume all H-bridge switches are open and an external power source spins the motor shaft. The external source could be water falling over the blades of a turbine in a hydroelectric dam, for example. If the motor shaft spins at a constant $\omega = 2000$ rad/s, then the back-emf is $k_t\omega = (0.01\text{ Vs/rad})(2000\text{ rad/s}) = 20$ V. The flyback diodes cap the voltage across the motor to the range [−11.4 V, 11.4 V], however, so current must be flowing through the motor. Assuming the flyback diodes D1 and D4 are conducting, we have

$$11.4\text{ V} = k_t\omega + IR = 20\text{ V} + I(1\ \Omega).$$

   Solving for I, we get a current of $I = -8.6$ A, flowing from ground through D4, the motor, and D1 to the 10 V supply (Figure 15.6). The motor is generating $(8.6\text{ A})(11.4\text{ V}) = 98.04$ W of electrical power.

   (If we had assumed the flyback diodes D2 and D3 were conducting instead, putting −11.4 V across the motor, and solved for I, we would have seen that the required negative current cannot be provided by D2 and D3. Therefore, D2 and D3 are not conducting.)

2. **Motor braking.** Assume the motor has a positive current of 2 A through it (left to right, carried by switches S1 and S4), then all switches are opened. Immediately after the switches are opened, the only option to continue providing 2 A to the motor is from ground, through D2, the motor, and D3 to the 10 V supply. The voltage across the motor must therefore be −11.4 V: two diode drops and the 10 V supply voltage. This means the motor acts as a generator, providing $(2\text{ A})(11.4\text{ V}) = 22.8$ W of power just after the switches are opened.

As these examples show, regeneration dumps current back into the power supply, charging it up, whether it wants to be charged or not. Some batteries can directly accept the regeneration current. For a wall-powered supply, however, a high-capacitance, high-voltage, typically polarized electrolytic capacitor at the power supply outputs can act as storage for energy dumped back into the power supply. While such a capacitor may be present in a linear power supply, a switched-mode power supply is unlikely to have one, so an external capacitor would have to be added. If the power supply capacitor voltage gets too high, a switch can allow the back-current to be redirected to a "regen" power resistor, which is designed to dissipate electrical energy as heat.

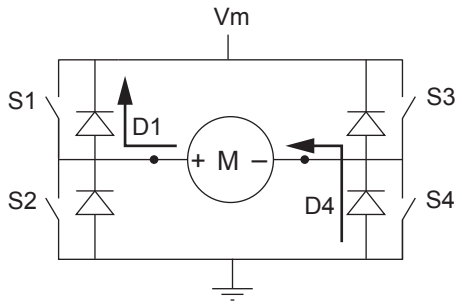


Figure 15.6: One example of regeneration, where the current through and voltage across a motor have opposite signs. The motor is forced to spin forward, by an external source, at a speed ω such that the back-emf $k_t\omega$ is larger than Vm. This forces a negative current to flow through the motor, carried by the flyback diodes D1 and D4. Electrical power is dumped into the power supply.

15.1.4 Other Practical Considerations

Motors are noisy devices, creating both electromagnetic interference (EMI), e.g., induced currents on sensitive electronics due to changing magnetic fields induced by large changing motor currents, as well as voltage spikes, due to brush switching and changing PWM current and voltage. These effects can disrupt the functioning of your microcontroller, cause erroneous readings on your sensor inputs, etc. Dealing with EMI is beyond our scope, but it can be minimized by keeping the motor leads short and far from sensitive circuitry, and by using shielded cable or twisted wires for motor and sensor leads.

Optoisolators can be used to separate noisy motor power supplies from clean logic voltage supplies. An optoisolator consists of an LED and a phototransistor. When the LED turns on, the phototransistor is activated, allowing current to flow from its collector to its emitter. Thus a digital on/off signal can be passed between the logic circuit and the power circuit using only an optical connection, eliminating an electrical connection. In our case, the PIC32's H-bridge control signals would be applied to the LEDs and converted by the phototransistors to high and low signals to be passed to the inputs of the H-bridge. Optoisolators can be bought in packages with multiple optoisolators. Each LED-phototransistor pair uses four pins: two for the internal LED and two for the collector and emitter of the phototransistor. Thus you can get a 16-pin DIP chip with four optoisolators, for example.

It is also common to directly solder a nonpolarized capacitor across the motor terminal leads, effectively turning the motor into a capacitor in parallel with the resistance and inductance of the motor. This capacitor helps to smooth out voltage spikes due to brushed commutation.

Finally, the H-bridge chip should be heat-sinked to prevent overheating. The heat sink dissipates heat due to MOSFET switching and MOSFET output resistance (on the order of hundreds of mΩ for the DRV8835).

15.2 Encoder Feedback

Motor angles can be measured using a potentiometer, or, most commonly, an encoder. There are two major types of encoders: *incremental* and *absolute*. By far the more common is the incremental encoder.

15.2.1 Incremental Encoder

An incremental encoder creates two pulse trains, A and B, as the encoder shaft rotates a *codewheel*. These pulse trains can be created by magnetic field sensors (Hall effect sensors) or light sensors (LEDs and phototransistors or photodiodes). The latter technique, used in *optical encoders*, is more common and is illustrated in Figure 15.7. The codewheel could be an opaque material with slots or a transparent material (glass or plastic) with opaque lines.

The relative phase of the A and B pulses determines whether the encoder is rotating clockwise or counterclockwise. A rising edge on B after a rising edge on A means the encoder is rotating one way, and a

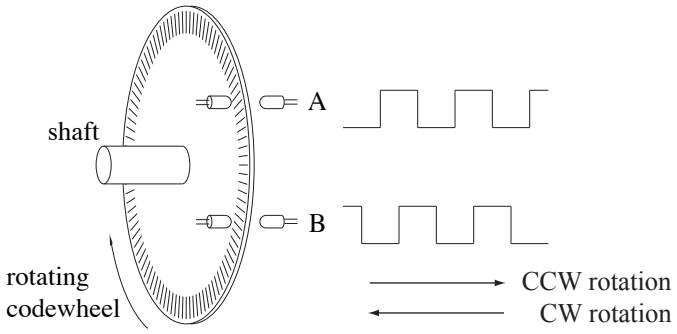


Figure 15.7: A rotating encoder creates 90 degree out-of-phase pulse trains on A and B using LEDs and phototransistors.

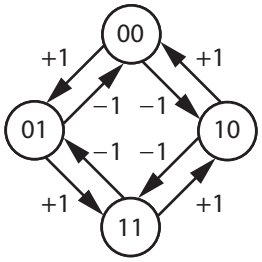


Figure 15.8: 4x decoding of A/B quadrature encoder channels. Each node of the state machine shows the digital A/B signals as a 2-bit number. When the signals change, the encoder count is either incremented or decremented according to the transition.

rising edge on B after a falling edge on A means the encoder is rotating the opposite direction. A rising edge on B followed by a falling edge on B (with no change in A) means that the encoder has undergone no net motion. The out-of-phase A and B pulse trains are known as *quadrature* signals.

Apart from the direction, the pulses can be counted to determine how far the encoder has rotated. The encoder signals can be "decoded" at 1x, 2x, or 4x resolution, where 1x resolution means that a single count is generated for each full cycle of A and B (e.g., on the rising edge of A), 2x resolution means that 2 counts are generated for each full cycle (e.g., on the rising and falling edges of A), and 4x means that a count is generated for every rising and falling edge of A and B (four counts per cycle, Figure 15.8). Thus an encoder with "100 lines" or "100 pulses per revolution" can be used to generate up to 400 counts per revolution of the encoder. If the motor has a 20:1 gearhead, the encoder (attached to the motor shaft) generates $400 \times 20 = 8000$ counts per revolution of the gearhead output shaft.

Some encoders offer a third output channel called the *index* channel, usually labeled I or Z. The index channel creates one pulse per revolution of the motor and can be used to determine when the motor is at a "home" position. Some encoders also offer differential outputs $\bar{A}$, $\bar{B}$, and $\bar{Z}$, which are always opposite A, B, and Z, respectively. That way, if the encoder is powered by 5 V, the change in voltage on a rising edge on A can be measured as $\Delta A - \Delta\bar{A} = 10$ V, as compared to the single-ended measurement $\Delta A = 5$ V. This is for noise immunity in electrically noisy environments.

The A and B encoder outputs can be fed to a decoder chip, such as the US Digital LS7183, which converts the pulses to "up" pulses and "down" pulses to be read directly by an external counter chip or counter/timers on the PIC32. Alternatively, the signals can be read by a standalone decoder/counter chip which keeps the count. This count can then be queried by a microcontroller using SPI or I^2C.

Incremental encoders also come in linear versions to measure linear motion.

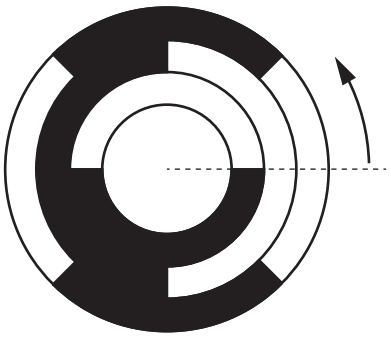


Figure 15.9: A 3-bit Gray code codewheel. If the innermost ring corresponds to the most significant bit, then as we proceed counterclockwise around the codewheel, the count is 000, 001, 011, 010, 110, 111, 101, and 100.

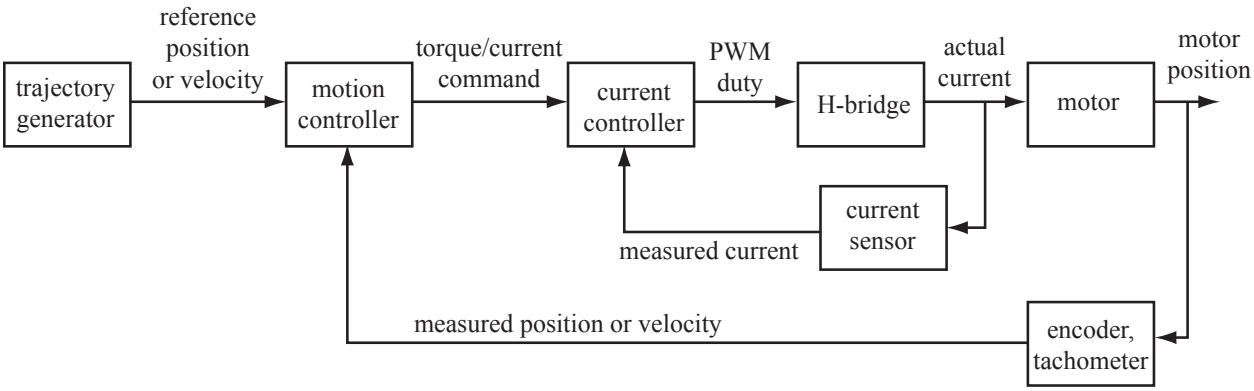


Figure 15.10: A block diagram for motion control.

15.2.2 Absolute Encoder

An incremental encoder can only tell you how far the motor has moved since the encoder was turned on. An absolute encoder can tell you where the motor is at any time, regardless of the position of the motor at power on. To do this, an absolute encoder uses many more LED/phototransistor pairs, and each one provides a single bit of information on the motor's position. For example, an absolute encoder with 8 channels can distinguish the absolute orientation of a motor up to a resolution of $360°/(2^8) = 1.4°$.

As the codewheel rotates, the binary count represented by the 8 channels increments according to *Gray code*, not the typical binary code, so that at each increment, only one of the 8 channels changes signal. This removes the need for the infinite manufacturing precision needed to to make two signals switch at exactly the same angle. Compare the following two three-bit sequences, for example (Figure 15.9):

Decimal	0	1	2	3	4	5	6	7
Binary code	000	001	010	011	100	101	110	111
Gray code	000	001	011	010	110	111	101	100

Absolute encoders tend to be more expensive than incremental encoders yielding similar resolution.

15.3 Motion Control of a DC Motor

An example block diagram for control of a DC motor is shown in Figure 15.10.[2] A trajectory generator creates a reference position as a function of time. To drive the motor to follow this reference trajectory, we use two nested control loops: an outer motion control loop and an inner current control loop. These two loops are roughly motivated by the two time scales of the system: the mechanical time constant of the motor and load and the electrical time constant of the motor.

- **Outer motion control loop.** This outer loop runs at a lower frequency, typically a few hundred Hz to a few kHz. The motion controller takes as input the desired position and/or velocity, as well as the motor's current position, as measured by an encoder or potentiometer, and possibly the motor's current velocity, as measured by a tachometer. The output of the controller is a commanded current I_c. The current is directly proportional to the torque. Thus the motion control loop treats the mechanical system as if it has direct control of motor torque.
- **Inner current control loop.** This inner loop typically runs at a higher frequency, from a few kHz to tens of kHz, but no higher than the PWM frequency. The purpose of the current controller is to deliver the current requested by the motion controller. To do this, it monitors the actual current flowing through the motor and outputs a commanded average voltage V_c (expressed as a PWM duty cycle) to compensate error.

Traditionally a mechanical engineer might design the motion control loop, and an electrical engineer might design the current control loop. But you are a mechatronics engineer, so you will do both.

15.3.1 Motion Control

Feedback Control

Let θ and $\dot{\theta}$ be the actual position and velocity of the motor, and r and $\dot{r}$ be the desired position and velocity. Define the error $e = r - \theta$, error rate of change $\dot{e} = \dot{r} - \dot{\theta}$, and error sum (approximating an integral) $e_{\text{int}} = \sum_k e(k)$. Then a reasonable choice of controller would be a PID controller,

$$I_{\text{c,fb}} = k_p e + k_i e_{\text{int}} + k_d \dot{e}, \tag{15.1}$$

where $I_{\text{c,fb}}$ is the commanded feedback current. The $k_p e$ term acts as a virtual spring that creates a force proportional to the error, pulling the motor to the desired angle. The $k_d \dot{e}$ term acts as a virtual damper that creates a force proportional to the "velocity" of the error, driving the error rate of change toward zero. The $k_i e_{\text{int}}$ term creates a force proportional to the time integral of error. See Chapter 11 for more on PID control.

In the absence of a model of the motor's dynamics, a reasonable commanded current I_c is simply $I_c = I_{\text{c,fb}}$.

An alternative form of the feedback controller (15.1) is

$$\ddot{\theta}_d = k_p e + k_i e_{\text{int}} + k_d \dot{e}, \tag{15.2}$$

where the feedback gains set a desired corrective acceleration of the motor $\ddot{\theta}_d$ instead of a current. This alternative form of the PID controller is used in conjunction with a system model in the next section.

Feedforward Plus Feedback Control

If you have a decent model of the motor and its load, a model-based controller can be combined with a feedback controller to yield better performance. For example, for an unbalanced load as in Figure 15.11, you could choose a feedforward current command to be

$$I_{\text{c,ff}} = \frac{1}{k_t}(J\ddot{r} + mgd\sin\theta + b_0\,\text{sgn}(\dot{\theta}) + b_1\dot{\theta}),$$

[2] A simpler block diagram would have the Motion Controller block directly output a PWM duty cycle to an H-bridge, with no inner-loop control of the actual motor current. This is sufficient for most applications. However, the block diagram in Figure 15.10 is more typical of industrial implementations.

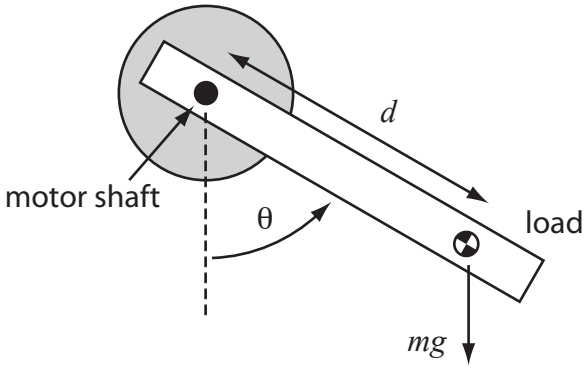


Figure 15.11: An unbalanced load in gravity.

where k_t is the torque constant, J is the motor and load inertia about the motor's axis, the planned motor acceleration $\ddot{r}$ can be obtained by finite differencing the desired trajectory, mg is the weight of the load, d is the distance of the load center of mass from the motor axis, θ is the angle of the load from vertical, b_0 is Coulomb friction torque, and b_1 is a viscous friction coefficient. To compensate for errors, the feedback current command $I_{\mathrm{c,fb}}$ can be combined with the feedforward command to yield

$$I_{\mathrm{c}} = I_{\mathrm{c,ff}} + I_{\mathrm{c,fb}}. \tag{15.3}$$

Alternatively, the motor acceleration feedback law (15.2) could be combined with the system model to yield the current controller

$$I_{\mathrm{c}} = \frac{1}{k_t}(J(\ddot{r} + \ddot{\theta}_d) + mgd\sin\theta + b_0\,\mathrm{sgn}(\dot{\theta}) + b_1\dot{\theta}). \tag{15.4}$$

15.3.2 Current Sensing and Current Control

Current Sensor

To measure the current flowing through the motor, a current sensing resistor can be placed in series with it. Current flowing through this resistor creates a voltage drop across it, which is then measured. To have minimum effect on the motor current, the sensing resistance should be small. For good accuracy, the resistor should have a tight tolerance on its resistance, and the resistor's power rating should be high enough to allow it to survive the largest current that can flow through it. For example, a 15 mΩ resistor used with a motor that may draw up to 5 A should be rated for at least $(5\text{ A})^2\ 0.015\ \Omega = 0.375$ W to ensure that it won't burn up.

The voltage across a current sense resistor is intended to be small, e.g., $5\text{ A} \times 0.015\ \Omega = 0.075$ V for 5 A through a 15 mΩ resistor. A specialized current-sense amplifier chip can be used to turn this small signal into a signal usable by a microcontroller. One such chip is the Maxim Integrated MAX9918 current-sense amplifier (Figure 15.12). The voltage across the current sense resistor is registered by pins RS+ and RS−. The analog output voltage OUT is given by

$$\text{OUT} = \text{G} * (\text{RS+} - \text{RS-}) + \text{REFIN}$$

where the gain G is set by the external feedback resistors R1 and R2 as $\text{G} = 1 + (\text{R2/R1})$. To implement this equation, the chip uses a non-inverting instrumentation amplifier along with a level shifting circuit (Figure 15.12).

The circuit shows REFIN as 1.65 V, so that zero current through the sense resistor reads as 1.65 V at OUT. This offset voltage allows OUT voltages less than 1.65 V to represent negative currents. The R3 voltage divider resistors feeding the reference voltage REFIN should be relatively small, perhaps a few hundred ohms,

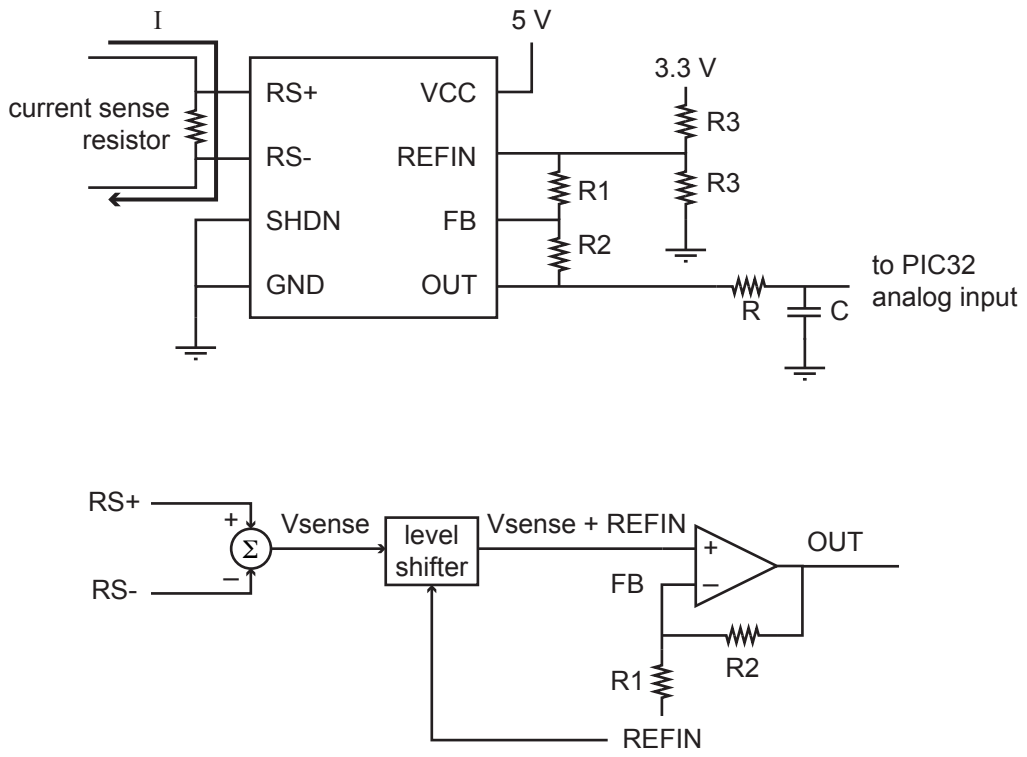


Figure 15.12: Top: Wiring the MAX9918 current-sense amplifier. Bottom: Effective internal circuit, showing how the R1 and R2 resistors are used to set the gains of a non-inverting amplifier.

to prevent small currents in the feedback resistor network from affecting REFIN.[3] Lowering R3 further would waste power unnecessarily.

The gain G should be chosen so that the maximum expected current through the motor gives the maximum voltage at the output. For example, if the maximum expected current is 2 A, then the maximum expected voltage across the 15 mΩ resistor is ±0.03 V. To use the full resolution of the PIC32's ADC, this should map to ±1.65 V, meaning G = 1.65 V/0.03 V = 55. Then a current of 2 A reads as 3.3 V at OUT and a current of −2 A reads as 0 V. The feedback resistors R1 and R2 should be relatively high resistance, so as not to load the REFIN voltage divider. See Exercise 13.

Finally, the signal OUT is low-pass RC filtered to average any current measurement variations due to the switching PWM (e.g., Figure 15.5).[4] A good choice for the RC time constant would give a filter cutoff frequency $f_c = 1/(2\pi RC)$ of a few hundred Hz. This is approximately 100 times less than a typical PWM frequency, cutting off most variation due to the PWM, but without being so sluggish as to adversely affect current control.

Current Control

The output of the current controller is V_c, the commanded average voltage (to be converted to a PWM duty cycle). The simplest current controller would be

$$V_c = k_V I_c$$

for some gain k_V. This would be a good choice if your load were only a resistance R, in which case you would choose $k_V = R$. Even if not, if you do not have a good mechanical model of your system, achieving a

[3] Ideally the voltage reference to REFIN would be from a lower-impedance source, like a buffered output, but here we are trying to keep the component count down.

[4] A better solution would be a high-impedance input active filter, using an op-amp.

Figure 15.13: The Copley Controls Accelus amplifier.

particular current/torque may not matter anyway. You can just tune your motion control PID gains, use $k_V = 1$, and not worry about what the actual current is, eliminating the inner control loop.

On the other hand, if your battery pack voltage changes (due to discharging, or changing batteries, or changing from a 6 V to a 12 V battery pack), the change in behavior of your overall controller will be significant if you do not measure the actual current in your current controller. More sophisticated current controller choices might be a mixed model-based and integral feedback controller

$$V_{\rm c} = I_{\rm c} R + k_t \dot{\theta} + k_{I,i}\, e_{I,\rm int}$$

or, recognizing that the electrical system is a first-order system (using voltage to control a current through a resistor and inductor), a PI feedback controller

$$V_c = k_{I,p}\, e_I + k_{I,i}\, e_{I,\rm int},$$

where e_I is the error between the commanded current $I_{\rm c}$ and the measured current, $e_{I,\rm int}$ is the integral of current error, R is the motor resistance, k_t is the torque constant, $k_{I,p}$ is a proportional current control gain, and $k_{I,i}$ is an integral current control gain. A good current controller would closely track the commanded current.

15.3.3 An Industrial Example: The Copley Controls Accelus Amplifier

Copley Controls, http://www.copleycontrols.com, is a well known manufacturer of amplifiers for brushed and brushless motors for industrial applications and robotics. One of their models is the Accelus, pictured in Figure 15.13. The Accelus supports a number of different operating modes. Examples include control of motor current or velocity to be proportional to either an analog voltage input or the duty cycle of a PWM input. A microcontroller on the Accelus interprets the analog input or PWM duty cycle and implements a controller similar to that in Figure 15.10.

The mode most relevant to us is the Programmed Position mode. In this mode, the user specifies a few parameters to describe a desired rest-to-rest motion of the motor. The controller's job is to drive the motor to track this trajectory.

When the amplifier is first paired with a motor, some initialization steps must be performed. A GUI interface on the host computer, provided by Copley, communicates with the microcontroller on the Accelus using RS-232.

1. **Enter motor parameters.** From the motor's data sheet, enter the inertia, peak torque, continuous torque, maximum speed, torque constant, resistance, and inductance. These values are used for initial guesses at control gains for motion and current control. Also enter the number of lines per revolution of the encoder.

2. **Tune the inner current control loop.** Set a limit on the recent current to avoid overheating the motor. This limit is based on the integral $\int_{T_1}^{T_2} I^2(t)dt$, which is a measure of how much energy the

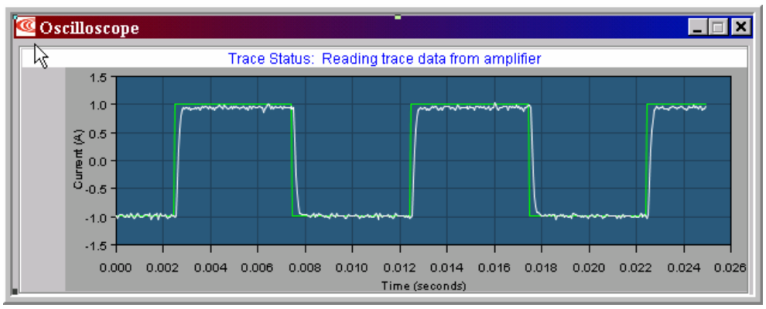


Figure 15.14: A plot of the reference square wave current and the actual measured current during PI current controller tuning.

motor coils have dissipated recently. (When this limit is exceeded, the motor current is limited to the continuous operating current until the history of currents indicates that the motor has cooled.) Also, tune the values of P and I control gains for a PI current controller. This tuning is assisted by plots of reference and actual currents as a function of time. See Figure 15.14, which shows a square wave reference current of amplitude 1 A and frequency 100 Hz. The zero average current and high frequency of the reference waveform ensure that the motor does not move during current tuning, which focuses on the electrical properties of the motor.

The current control loop executes at 20 kHz, which is also the PWM frequency (i.e., the PWM duty cycle is updated every cycle).

3. **Tune the outer motion control loop with the load attached.** Attach the load to the motor and tune PID feedback control gains, a feedforward acceleration term, and a feedforward velocity term to achieve good tracking of sample reference trajectories. This process is assisted by plots of reference and actual positions and velocities as a function of time. The motion control loop executes at 4 kHz.

Once the initial setup procedures have been completed, the Accelus microcontroller saves all the motor parameters and control gains to nonvolatile flash memory. These tuned parameters then survive power cycling and are available the next time you power up the amplifier.

Now the amplifier is ready for use. The user specifies a desired trajectory using any of a number of interfaces (RS-232, CAN, etc.), and the amplifier uses the saved parameters to drive the motor to track the trajectory.

15.4 Exercises

1. The switch in Figure 15.1(b), with no flyback diode, has been closed for a long time, and then it is opened. The voltage supply is 10 V, the motor's resistance is $R = 2\ \Omega$, the motor's inductance is $L = 1$ mH, and the motor's torque constant is $k_t = 0.01$ Nm/A. Assume the motor is stalled.

   (a) What is the current through the motor just before the switch is opened?
   (b) What is the current through the motor just after the switch is opened?
   (c) What is the torque being generated by the motor just before the switch is opened?
   (d) What is the torque being generated by the motor just after the switch is opened?
   (e) What is the voltage across the motor just before the switch is opened?
   (f) What is the voltage across the motor just after the switch is opened?

2. The switch in Figure 15.1(c), with the flyback diode, has been closed for a long time, and then it is opened. The voltage supply is 10 V, the motor's resistance is $R = 2\ \Omega$, the motor's inductance is $L = 1$ mH, and the motor's torque constant is $k_t = 0.01$ Nm/A. The flyback diode has a forward bias voltage drop of 0.7 V. Assume the motor is stalled.

(a) What is the current through the motor just before the switch is opened?

(b) What is the current through the motor just after the switch is opened?

(c) What is the torque being generated by the motor just before the switch is opened?

(d) What is the torque being generated by the motor just after the switch is opened?

(e) What is the voltage across the motor just before the switch is opened?

(f) What is the voltage across the motor just after the switch is opened?

(g) What is the rate of change of current through the motor dI/dt just after the switch is opened? (Make sure to use a correct sign, relative to your current answers above.)

3. In Figure 15.1(d), the voltage supplies are ± 10 V, the motor's resistance is $R = 5\ \Omega$, the motor's inductance is $L = 1$ mH, and the motor's torque constant is $k_t = 0.01$ Nm/A. The flyback diodes have a forward bias voltage drop of 0.7 V. Switch S1 has been closed for a long time, with no voltage drop across it, and the motor is stalled. Switch S2 is open. Then switch S1 is opened while switch S2 remains open. Immediately after S1 opens, which flyback diode conducts current? What is the voltage across the motor? What is the current through the motor? What is the rate of change of the current through the motor?

4. Give some advantages of using an H-bridge over a linear push-pull amplifier to drive a DC motor. Give at least one advantage of using a linear push-pull amplifer over an H-bridge. (Hint: consider the case of low PWM frequency or low motor inductance.)

5. Explain why an initially spinning motor comes to rest faster if the two motor leads are shorted to each other rather than left disconnected. Derive the result from the motor voltage equation.

6. Provide a circuit diagram showing the DRV8835 configured to drive a single motor with more than 2 A continuous.

7. Consider a motor in an H-bridge with all switches opened (motor is unpowered). The motor rotor is rotated by external forces (e.g., rushing water in a hydroelectric dam spinning the blades of a turbine). If the H-bridge is connected to a battery supply of voltage Vm, and the forward bias voltage of the flyback diodes is Vd, give a mathematical expression for the rotor speed at which the battery begins to charge. When this speed is exceeded, and assuming that the battery voltage Vm is constant (e.g., it acts somewhat like a very high capacitance capacitor, accepting current without changing voltage), give an expression for the current through the motor as a function of the rotor speed. Also give an expression for the power lost due to the heating the windings.

   How does the presence of the hydrogenerator affect the total energy of a bucket of water at the top of the dam compared to the total energy just before that water splashes into the river at the bottom of the dam? (The total energy is the potential energy plus the kinetic energy.)

8. To create an average current I through a motor, you could send I constantly, or you could alternate quickly between kI for $100\%/k$ of a cycle and zero current for the rest of the cycle. Provide an expression for the average power dissipated by the motor coils for each of these cases.

9. Imagine a motor with a 500 line incremental encoder on one end of the motor shaft and a gearhead with $G = 50$ on the other end. If the encoder is decoded in 4x mode, how many encoder counts are counted per revolution of the gearhead output shaft?

10. A simple outer-loop motion controller is to command a torque (current) calculated by a PID controller. A more sophisticated controller would attempt to use a model of the motor-load dynamics to get better trajectory tracking. One possibility is the control law (15.3). Another possibility is the control law (15.4). Describe any advantages of (15.4) over (15.3).

11. Choose R2/R1 in Figure 15.12 so the output voltage OUT swings full range (0 to 3.3 V) for a current range of ± 1 A.

12. Choose an example R and C in Figure 15.12 to create a cutoff frequency of 200 Hz.

13. For the current sensor amplifier in Figure 15.12, the reference voltage at REFIN should be constant. We know that the currents into and out of the high-impedance op-amp inputs REFIN and FB are negligible, as is the current into the PIC32 analog input. But the currents through the external resistors R1 and R2 may not be small. As a result, REFIN actually varies as a function of the the sensed input voltage and the output voltage OUT. We should choose the resistances R1, R2, and R3 so the voltage variation at REFIN is small.

    Pay attention only to the op-amp portion of the circuit in the bottom of Figure 15.12. Now assume that OUT is 3.3 V. What is the voltage at REFIN as a function of R1, R2, and R3? (Note that it is not exactly 1.65 V.) Use your equation to comment on how to choose the relative values of R1, R2, and R3 to make sure that REFIN is close to 1.65 V. Explain other practical constraints on the absolute values (minimum and maximum values) of R1, R2, and R3.

14. Due to natural variations in resistor values within their tolerance ranges, the gain G, and the voltage at REFIN, that you designed for your current sensor circuit will not be exactly realized. Due to this and other variations, you need to calibrate your current sensor by running some experiments. Clearly explain what experiments you would do, and how you would use the results to interpret analog voltages read at the PIC32 as motor currents. Be specific; the interpretation should be easy to implement in software.

15. Clearly outline how you would implement the motor controller in Figure 15.10 in software on a PIC32. Indicate what peripherals, ISRs, NU32 functions, and global variables you might use. You may use concepts from the LED brightness control project, if it is helpful. In particular, indicate how you would implement:

    - **The trajectory generator.** Assume that desired trajectories are sent from Matlab on the host computer, and trajectory tracking results are plotted in Matlab.
    - **The outer-loop motion controller.** Assume that an external decoder/counter chip maintains the encoder count, and that communication with the chip occurs using SPI. How would you collect trajectory tracking data to be plotted in Matlab?
    - **The inner-loop current controller**, using the current sensor described in this chapter. How would you collect data on tracking the desired current to be plotted in Matlab?

Chapter 16

A Motor Control System

Imagine combining the power and convenience of a personal computer with the peripherals of a microcontroller. Motors moving in response to keyboard commands. Plots showing the motor's positional history. Interactive controller tuning. By combining your knowledge of microcontrollers, motors, and control theory you can accomplish these goals.

It starts with a menu. Although they seem old-fashioned, menus provide a simple user interface: a list of commands that you can choose by pressing a key. We will begin with an empty menu. By the end of this project, it will have nearly 20 options: everything from reading encoders to setting gains to plotting trajectories. Much work lies ahead, but don't panic! We guide you through the implementation, explaining how to structure the code to reduce programming errors and increase maintainability. We have also selected the major electronic components that you need, so you can focus on programming. Let's examine the features you will implement in greater detail.

16.1 Features

We want to be able to specify a trajectory, have the motor execute that trajectory, and view the results. Rather than attempting this goal all at once, we divide it into several supporting features, accessible from a menu running on your computer. This approach enables you to test the project incrementally and creates a more useful final product.

Accomplishing trajectory tracking requires a control strategy. Although many trajectory tracking methods exist, we follow the path of Ch. 15 by using a nested control structure. The trajectory consists of a series of positions, which an outer position control loop attempts to follow. The outer loop, based on position feedback, calculates control effort in terms of motor current. The desired current becomes a reference signal for an inner control loop, which compares it to current sensor readings and adjusts the PWM motor control signal appropriately. These control loops also record data that we send to the client to display.

Prior to implementing these control loops, we need to implement some more basic features. The current control loop must output a PWM signal to the motor and read current sensor values, and the position control loop needs encoder feedback. To test these basic features the user must be able to to view sensor readings and specify PWM duty cycles. Another option should allow us to tune the current controller independently of the position controller. The ability to specify gains without needing to recompile the program will ease this process. All of the features we have discussed will be accessible from the client menu that you will create, depicted in Figure 16.2.

Now that you know what you will be implementing, we will describe the hardware you will use to accomplish these tasks. Next, we explain the way you will structure your software. Finally, the guide to building this project begins.

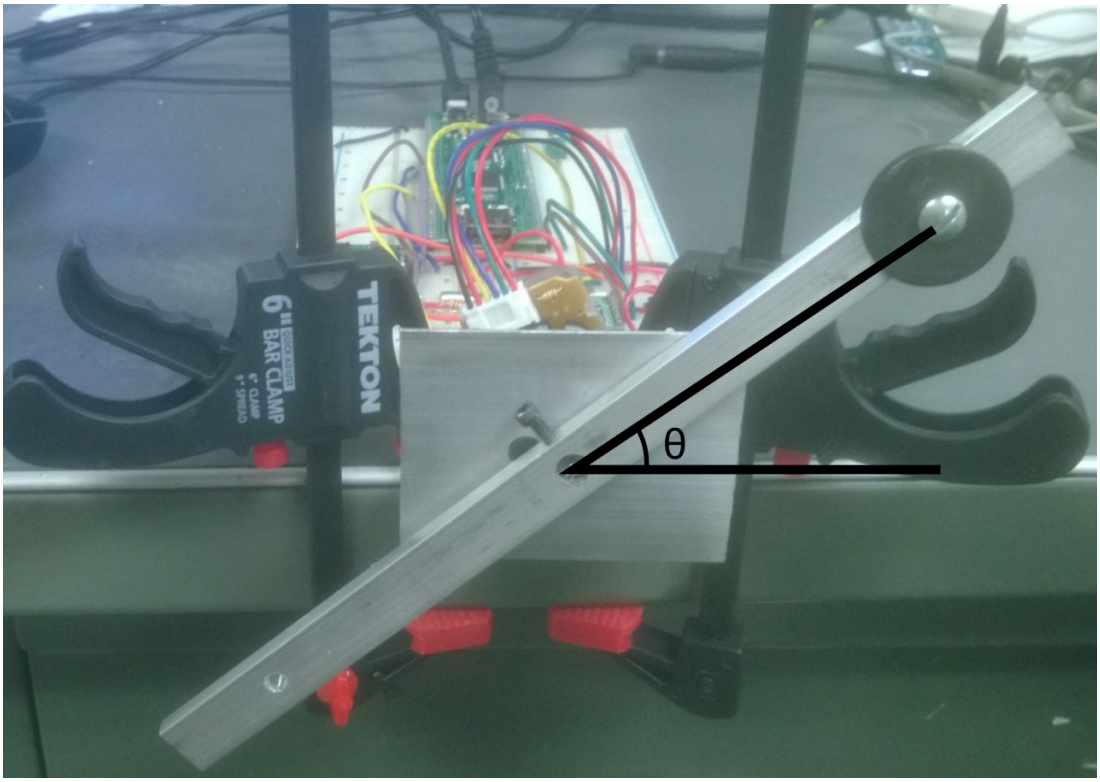


Figure 16.1: Complete motor setup.

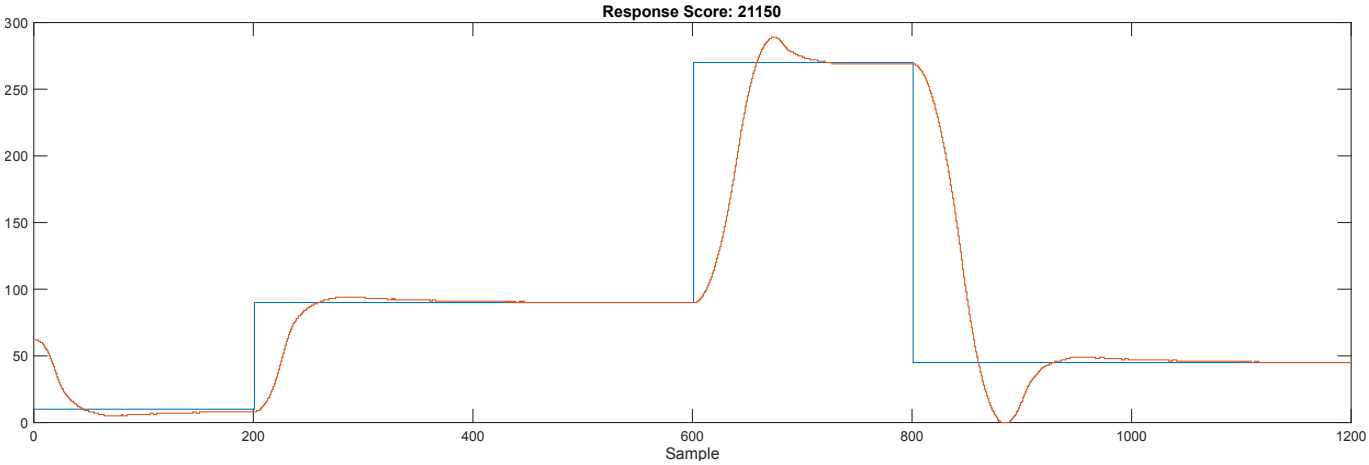
16.2 Hardware

We have chosen the major hardware components necessary to complete this project. You still must choose appropriate resistors and capacitors for some filters and analog gains; we guide you through that process when necessary. In addition to the electronics, we have provided you with a motor mounting bracket and a bar with weights that can serve as a load for the motor. Using the motor with the bar and weights makes control more challenging, but also more rewarding. For reference, here is a list of the hardware. Data sheets, where applicable, can be found on the book's website http://hades.mech.northwestern.edu/index.php/Pic32book.

- motor, including bracket, bar, and weights
- quadrature decoder and counter chip
- current sensor
- H-bridge
- resistors and capacitors

16.3 Software

This project requires you to write C code that runs on the PIC32 and client code that runs on your computer. We focus primarily on the C code here. We have designed this code from the top down, dividing the main task into individual modules. The goals of this decomposition are twofold: to isolate unrelated components and to provide defined roles for these components. Modular design is especially important in embedded programming because multiple tasks require access to the same peripherals; therefore, defining which code




```
a: Read current sensor (ticks)
b: Read current sensor (mA)
c: Read encoder (ticks)
d: Read encoder (deg)
e: Reset encoder
f: Set PWM
g: Set current gains
h: Get current gains
i: Set position gains
j: Get position gains
k: Tune current loop
l: Set recording samples
m: Go to angle
n: Load step trajectory
o: Load cubic trajectory
p: Execute trajectory
q: Quit
r: Get state

Enter Command:
```

Figure 16.2: Motor control menu and data plot.

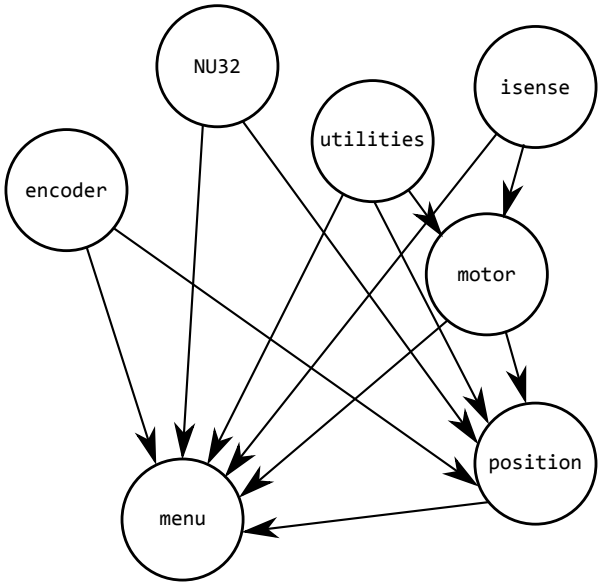
uses which hardware at what time becomes crucial for avoiding conflicts. After reading this section you will understand how to structure the code.

A module consists of one `.c` and one `.h` file. The `.h` file contains the module's *interface* and the `.c` file contains the module's implementation. The `.c` file should always `#include` the corresponding `.h` file. The interface consists of function prototypes and data types that other modules can use. You can use functions from module `A` in module `B` by having `B.c` include `A.h`: this situation means that module `B` depends on module `A`. The implementation contains the code that makes the interface functions work, as well as any `static` functions and variables that the module depends on but wants to hide from other modules. By hiding implementations in the `.c` file and only exposing certain functions in the `.h` file you decrease the dependencies between modules, making maintenance easier and helping prevent bugs. For those readers familiar with an object-oriented programming language such as C++ or Java, these concepts roughly mirror the ideas of public and private class members.

In an embedded system, hardware resources are globally accessible; therefore dividing the code into modules is only the first step in a good design. To maintain proper module separation, we document which modules *own* which peripherals. If a module owns a peripheral, only it should access that peripheral directly. If a module needs to access a peripheral it does not own, it must call a function in the owning module. These rules, enforced by your vigilance rather than the compiler, will help you more easily isolate bugs and fix them as they occur. We now explain the structure that we suggest you use to complete this project. Figure 16.3 shows the dependencies between modules and provides a brief summary of their roles.

16.3.1 Utilities

The `utilities` module maintains the state of the PIC32 and provides buffers for communicating between the control loops and the PC. The utilities module does not access any peripherals directly; however, it may



- `NU32` Provides functions for communicating with the client via the UART.
- `encoder` Provides functions for reading the encoders.
- `isense` Uses the ADC to measure the current sensor output.
- `motor` Directly interfaces with the motor. Runs the current control loop.
- `position` Runs the position control loop. Indirectly controls the motor by using `motor`.
- `menu` Receives commands from the client and dispatches them to other modules. Also contains `main`.
- `utilities` Maintains the system state and implements a communication buffer.

Figure 16.3: The modules. An arrow from `A` to `B` indicates that `A` is used by `B`, and `B.c` should `#include A.h`.

communicate with the client using `NU32`. The `position` and `motor` modules query the current system state to decide what to do: run a specified PWM directly, tune the current control, or track a trajectory. The `menu` module uses `utilities` to select the proper state.

16.3.2 Current Sensor

The `isense` module handles the current sensor. The current sensor generates a voltage proportional to the current flowing through the motor, which we read using the ADC. The `isense` module owns the ADC, and uses it to provide current readings in both ADC ticks and in milliamps.

16.3.3 Encoder

To convert encoder pulses into useful information we use an external encoder decoding chip, with which we communicate over SPI. The encoder module, therefore, owns an SPI peripheral. The encoder module makes encoder readings available both as ticks and motor angle. It also provides a method for resetting the encoder, allowing the user to establish the zero angle.

16.3.4 NU32

You can think of the NU32 library as a separate module in this project. This module owns UART1, providing methods to read and write data over the UART. It also provides direct access to LED1, LED2, and the USER button.

16.3.5 Motor

The `motor` module directly controls the motor and implements the current control loop. The current control loop runs from a timer ISR, implemented in `motor.c`. Depending on the system state, the current control loop may run the current controller, execute a tuning routine, or set a specific PWM duty cycle. To accomplish these tasks, the `motor` module owns one timer to implement a fixed-frequency current control ISR and one timer and one output compare for the PWM output. It also owns a digital output pin to control the motor's direction.

The `motor` module uses the `isense` module to read the current and the `utilities` module to record data. Within the `motor` module itself, we will follow the convention that, after initialization, only the current control loop ISR will ever modify the motor PWM signal; this will help prevent conflicts between the different portions of code that will eventually need to control the motor.

16.3.6 Position Control

The `position` module contains the position control loop. It can also load reference trajectories from the client and track them. This module owns a timer to create the control loop ISR. The `encoder` module provides position measurements and the `motor` module converts current into PWM signals. This module, at least indirectly, depends on every other module except `menu`, so we will implement it last.

16.3.7 The Menu

The `menu` module accepts commands from the client, initiates the appropriate action, and then returns the results. No modules depend on `menu`; therefore it does not need its own `.h` file. The `menu` module depends on every other module, which makes sense because it needs to coordinate all of them. It also ensures that the PIC32 is in the correct state; no other module should modify the state. We have chosen to place the `main` function in `menu` because the menu belongs in the infinite `while` loop typically found in `main`.

16.3.8 The Client

The client runs on the computer and is written in MATLAB. It presents a menu to the user and also displays information received from the PIC32. It also allows the user to specify trajectories in a human-friendly format.

16.4 Project Guide

We now begin the guide to this project. This guide focuses on explaining what to code and in what order; however, the details are left to you. You will need to refer to other chapters to understand the details required to implement certain steps: where necessary we will point you to the appropriate chapter. We designed the procedure so that you can create the project incrementally, testing as you go. Although there is no single correct way to create this project, we suggest that you test thoroughly after completing each task, as later tasks depend on earlier ones. Remember, the earlier you catch a bug, the easier it is to fix!

16.4.1 Decisions, Decisions

Before beginning, you have some decisions to make. It may be helpful to record these decisions somewhere, either in a text file or on paper. First, decide which specific peripherals each module will use, noting that `NU32` already uses UART1. The `encoder` module requires an SPI port. The `isense` module needs an analog

input. The `motor` module needs a timer for the current control loop, a timer for PWM, an output compare for PWM, and a digital output. The `position` module needs timer for the position control loop.

After you have assigned peripherals to modules, you need to choose frequencies for the PWM, the current control loop, and the position control loop. For this project, you will choose 20 kHz, 5 kHz, and 200 Hz, respectively. Also, consider the priorities for the interrupts: faster tasks need higher priority.

Before continuing, record the following information and answer the following questions:

1. The peripherals you will use in each module
2. The pins you will use in each module
3. What factors should be considered when choosing control loop and PWM frequencies?

16.4.2 Establishing Communication

This project's heart is the menu. By creating a menu you can run various functions on the PIC32 without recompiling, which makes testing easier. We start with the some familiar code for `menu.c`. In this project, the role of `menu.c` is to coordinate all of the other modules, dispatching commands that the PIC32 receives from the client to the appropriate function.

Code Sample 16.1. `menu.c` Basic code for setup and communication.

```
#include "NU32.h"          // config bits, constants, funcs for startup and UART
// include other module headers here

#define BUF_SIZE 200

int main()
{
  char buffer[BUF_SIZE];
  NU32_Startup();  // cache on, min flash wait, interrupts on, LED/button init, UART init
  NU32_LED1 = 1;        // turn off the LEDs
  NU32_LED2 = 1;
  __builtin_disable_interrupts();
  // initialize other modules here
  __builtin_enable_interrupts();

  while(1)
  {
    NU32_ReadUART1(buffer,BUF_SIZE);     // we expect the next character to be a menu command
    NU32_LED2 = 1;                        // clear the error led
    switch (buffer[0]) {
      case 'd':                           // dummy command for demonstration purposes
      {
        int n = 0;
        NU32_ReadUART1(buffer,BUF_SIZE);
        sscanf(buffer, "%d", &n);
        sprintf(buffer,"%d\r\n", n + 1);       // return the number + 1
        NU32_WriteUART1(buffer);
        break;
      }
      case 'q':
      {
        // handle q for quit. Later you may want to return to idle state here
        break;
      }
      default:
      {
```

```
        NU32_LED2 = 0;    // turn on LED2 to indicate an error
        break;
      }
    }
  }
  return 0;
}
```

Notice the infinite while loop. It reads from UART1, expecting a single character. That character is processed by a `switch` statement to determine what action to perform. If the character matches a known menu entry, we first retrieve additional parameters from the client. The specific format for these parameters depends on the particular command, but in this case, after receiving a "d," we expect an integer and store it in `n`. The command then increments the integer and sends the result to the client. Each `case` statement ends with a `break`. The `break` prevents the code from falling through to the next case. Make sure that every `case` has a corresponding `break`.

Unrecognized commands trigger the default case. The default case illuminates LED2 to indicate an error. We could have designed the communication protocol to allow the PIC32 to return error conditions to the client. Although crucial in the real world, proper error handling complicates the communication protocol and obscures the goals of this project; therefore we omit it.

Before proceeding further, test the menu code:

1. Compile and load `menu.c`, then connect to the PIC32 with a terminal emulator.
2. Issue the "d" command, followed by a number. Ensure that the command works as expected.
3. Issue an unknown command and verify that LED2 illuminates.

We have also provided you with some basic client code. This code expects a single character of keyboard input and sends it to the PIC32. Then, depending on the command sent, the user can be prompted to enter more information and that information can be sent to the PIC32.

Code Sample 16.2. `client.m` MATLAB client code.

```
function client(port)
%   provides a menu for accessing PIC32 motor control functions
%
%   client_menu(port)
%
%   Input Arguments:
%       port - the name of the com port.  This should be the same as what
%               you use in screen or putty in quotes ' '
%
%   Example:
%       client_menu('/dev/ttyUSB0') (Linux/Mac)
%       client_menu('COM3') (PC)
%
%   For convenience, you may want to embed this in a script that
%   contains your port number

% Opening COM connection
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
```

```
fprintf('Opening port %s....\n',port);

% settings for opening the serial port. baud rate 230400, hardware flow control
% wait up to 120 seconds for data before timing out
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware','Timeout',120);
% opens serial connection
fopen(mySerial);
% closes serial port when function exits
clean = onCleanup(@()fclose(mySerial));

has_quit = false;
% menu loop
while ~has_quit
    % display the menu options
    fprintf('d: Dummy Command    q: Quit\n');
    % read the users choice
    selection = input('Enter Command: ', 's');

    % send the command to the PIC32
    fprintf(mySerial,'%c\n',selection);


    % take the appropriate action
    switch selection
        case 'd'                              % example operation
            n = input('Enter number: '); % get the number to send
            fprintf(mySerial, '%d\n',n);    % send the number
            n = fscanf(mySerial,'%d');
            fprintf('Read: %d\n',n);
        case 'q'
          has_quit = true;              % exit matlab
        otherwise
            fprintf('Invalid Selection %c\n', selection);
    end
end

end
```

Test the client, as follows, before proceeding:

1. Exit the terminal emulator (if it is still open).
2. Run `client.m` in MATLAB and issue the "d" command.
3. Quit the client by using "q."
4. Implement a command on the PIC32 that accepts two integers, adds them, and returns the sum to the client.
5. Test the new command by using the terminal emulator and entering the information manually.
6. Add an entry to the client menu that prompts the user for two integers, issues the sum command, and prints the result.

After completing the above tasks, you should be familiar with the menu system. Adding menu entries and determining a communication protocol for these commands will become routine as you proceed through this project. If you encounter problems, you can open the terminal emulator and enter the commands manually to narrow down whether the issue is on the client or on the PIC32. Remember, do not attempt to simultaneously open the serial port in the terminal emulator and MATLAB. Also, a mismatch between the data that the

PIC32 expects and what MATLAB sends, or vice versa may cause one or the other to freeze while waiting for data. You can force-quite the client in MATLAB by typing CTRL-C. When you are comfortable with how the menu system works, you can remove the dummy command and the sum command, as they are no longer needed.

16.4.3 Encoder

Now that you have established communication with the PIC32 and can add menu items, we will use the menu to read the encoder from the computer. For more details about encoders, refer to Ch. 15. Before involving the PIC32, we will examine the encoder using an oscilloscope.

1. Power the encoders by connecting the Vcc and GND pins, as depicted in Fig. 16.4.
2. Attach two oscilloscope probes to the outputs EA and EB.
3. Slowly twist the motor in both directions and make sure you see a signal from both encoder channels.

Next, connect the encoders to the quadrature chip and connect the chip to the SPI peripheral of your choosing, as per Fig. 16.4. The quadrature chip uses SPI to communicate: the details about how to communicate with this chip are provided in Ch. **??**. After connecting the encoder chip, you should verify that it works by attaching the oscilloscope to pin B2 and checking for the 1 Khz heartbeat signal.

Once you have connected all the components, you will need to implement the `encoder.h` module. We have provided you with a partial header file, you will implement most of the functions. We have also provided you with some implementation to get you started. The implementation assumes that you used SPI4.

Code Sample 16.3. `encoder.h` Encoder header file.

```
#ifndef ENCODER__H__
#define ENCODER__H__

void encoder_init();          // initialize the encoder module

int encoder_ticks();          // read the encoder, in ticks

void encoder_reset();         // reset the encoder position

int encoder_angle();          // read the encoder angle, in tenths of a degree

#endif
```

Code Sample 16.4. `encoder.c` Encoder implementation.

```
#include "encoder.h"
#include <xc.h>

static int encoder_command(int read) { // send  a command to the encoder chip
  SPI4BUF = read;                 // send the command

  while (!SPI4STATbits.SPIRBF) { ; } // wait for the response

  SPI4BUF;                        // garbage was transfered over, ignore it
  SPI4BUF = 5;                    // write garbage, but the corresponding read will have the data

  while (!SPI4STATbits.SPIRBF) { ; }

  return SPI4BUF;
```

```
}

void encoder_init(void) {
  // SPI initialization for reading from the encoder chip
  SPI4CON = 0;            // stop and reset SPI4
  SPI4BUF;               // read to clear the rx buffer
  SPI4BRG = 0x4;            // bit rate to 8MHz, SPI4BRG = 80000000/(2*desired)-1
  SPI4STATbits.SPIROV = 0;  // clear the overflow
  SPI4CONbits.MSTEN = 1;    // master mode
  SPI4CONbits.MSSEN = 1;    // Slave select enable
  SPI4CONbits.MODE16 = 1;   // 16 bit mode
  SPI4CONbits.MODE32 = 0;
  SPI4CONbits.SMP = 1;      // sample at the end of the clock
  SPI4CONbits.ON = 1;       // turn spi on
}

int encoder_ticks(void) {
  encoder_command(1);         // we need to read twice to get a valid reading
  return encoder_command(1);
}
```

The function `encoder_init()` initializes the SPI peripheral. Call this function from the beginning of `main`, while interrupts are disabled. [1] The SPI peripheral uses a baud of 8 Mhz, 16-bit operation, sampling on the falling edge, and automatic slave select.

The next function is not declared in the header: rather, it should be declared as `static` within `encoder.c`, making it inaccessible to other modules. The function, `encoder_command`, sends a command to the quadrature chip and returns a response. Valid commands are 0x01, which reads the tick count, and 0x00, which resets the encoder. Note that every time you write to SPI you must also read, even if you do not need the data. Due to the protocol with this particular encoder chip, `encoder_command` should initiate two read/write sequences, returning the value of the second read.

You should use `encoder_command` to implement both `encoder_ticks` and `encoder_reset`. The first function returns the number of ticks from the encoder: the tick count can be read by using `encoder_command` to send 0x01 to the quadrature chip twice, and returning the second result. To reset the encoder, simply send a 0x00 using `encoder_command`.

Finally, you should declare and implement `encoder_angle`, which should read the tick count from the quadrature chip and return the angle. For maximum efficiency, we have written it to return the angle as an integer, in units of tenths of a degree. Be careful with rounding errors when doing the conversion. When reset, the encoder chip will read 32768: this should correspond to the angle zero. You can convert tick counts to angle based on the resolution of the encoder, and knowing that there are for ticks for each line, provided by the encoder data sheet. Do not worry about having angles wrap around, the controller will consider 0, 360, -360, and 720 as different angles. More details about encoders are found in Ch. 15.

After implementing these functions, add menu entries in `menu.c` so that you can issue the `encoder_reset`, `encoder_ticks`, and `encoder_angle` commands from your computer. For example, to add a "Read encoder ticks" option, add a menu entry in `menu.c`

```
  case 'r':
  {
    sprintf(buffer,"\%d", encoder_ticks());
    NU32_WriteUART1(buffer);
    break;
  }
```

Notice that when the menu receives an "r" from the client, it will return the `encoder_ticks()` value.

Verify that the commands by performing the following steps, using the terminal emulator to issue commands (make sure you have exited the MATLAB menu).

[1] Don't forget to include `encoder.h` in `menu.c`!

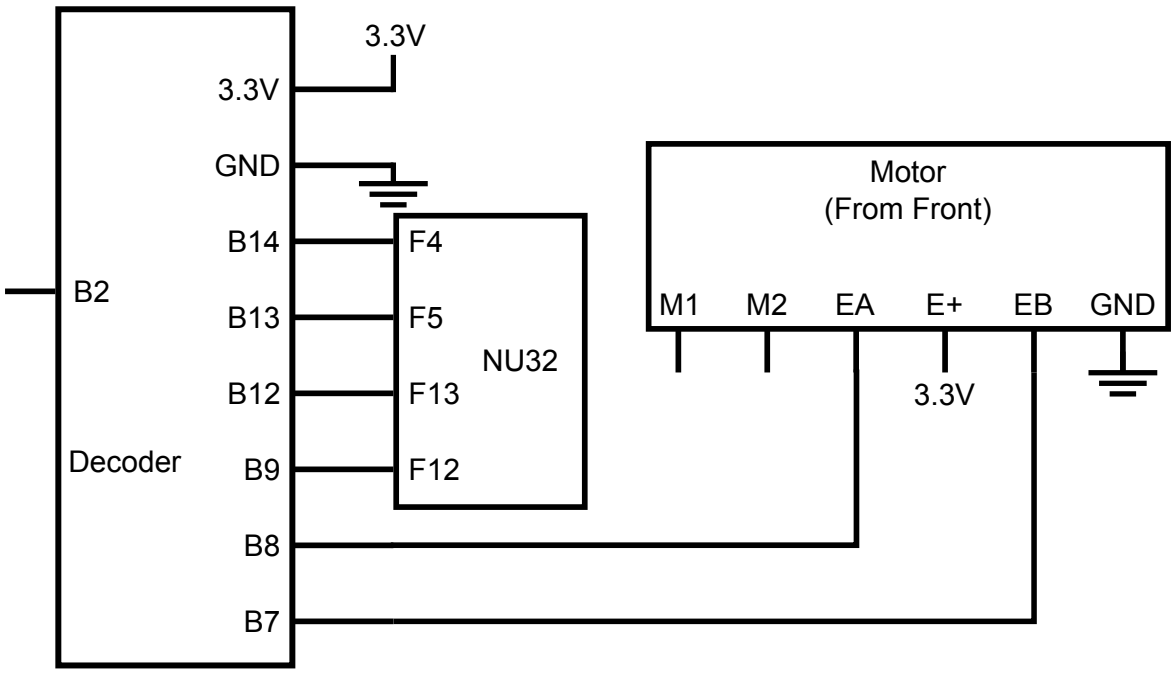


Figure 16.4: Encoder counter circuit

1. Read the encoder count by issuing the read command.
2. Twist the motor counterclockwise, and ensure that the encoder count increases. If it decreases, swap EA and EB.
3. Twist the motor clockwise, and ensure that the encoder count decreases.
4. Reset the encoders and read the count: it should be 32768.
5. Read the encoder angle; it should be 0 degrees.
6. Twist the motor 180 degrees counter clockwise. Verify that the encoder angle is near 180 degrees.
7. Twist the motor 360 degrees clockwise. Verify that the encoder angle is near -180 degrees.

Next add an entry to the client menu. You should probably print an additional instruction such as "r: Encoder Ticks" to remind yourself of the command. Then, process the "r" character as follows:

```
case 'r'
  ticks = fscanf(mySerial,'\%d');
  fprintf('Encoder Ticks: \%d', ticks)
```

Close the terminal emulator and use the client to issue encoder commands. After verifying that the client works, proceed to the next section.

16.4.4 State

One of the roles of the `utilities` module is to maintain the system's state, allowing the control loops to behave differently depending on the user's commands. We have provided you with the header file, `utilities.h`, which declares functions and data types for handling the state of the system. The other functions are used for managing the buffer, which we discuss later.

Code Sample 16.5. `utilities.h` Utilities header file.

```
#ifndef util__H__
#define util__H__

typedef enum { IDLE } state_t;          // the various states

typedef struct {                        // structure used for storing control loop data
  int reference;
} control_data;

state_t util_state_get();               // get the current state

void util_state_set(state_t s);         // atomically set the current state.

void util_buffer_init(unsigned int n);  // initialize the buffer to record n samples

void util_buffer_write(control_data d); // write data to the current buffer position, if not full

void util_buffer_output();              // wait for the buffer to be full and output it to the client

#endif
```

The first declaration in the file is an `enum`, `state_t`, which contains constants for the possible system states. Currently, we have only defined `IDLE`. By default, the system will be in the `IDLE` state. The motor will always remain stationary when in IDLE mode. You will add additional states as needed. The `state_t enum` is just a data type declaration, like a `struct`. You still need to declare a variable to hold the current system state.

Create `utilities.c` and add the declaration

```
static volatile state_t state = IDLE;
```

Notice that `state` is both `static` and `volatile`. The system state will be modified by mainline code and read in interrupts, so it should be volatile. We declare it `static` to prevent other modules from accessing it directly. Instead of direct access, you will provide two functions: `util_state_get` and `util_state_set` to manipulate the `state`. The function `util_state_get` should return the value of the current state, while the function `util_state_set` should disable interrupts, set the state, and re-enable interrupts. Disabling interrupts ensures that the `state` variable will be set atomically, meaning that an interrupt will not occur in the middle of changing the state variable.

Atomicity could also be guaranteed by generating an *atomic* operation, but this would require writing some inline assembly code. [2] By using these accessors, we ensure that no code outside of the `utilities` module can modify `state` without first disabling interrupts.

1. Implement `util_state_get` and `util_state_set`
2. Add a menu item to query the current state, and verify that it works.
3. Update the "quit" menu entry to return the PIC32 to the IDLE state. Eventually this will cause the motor to stop when you exit the client.

16.4.5 Current Sensor

The current sensor, as used in this project, requires three external circuits to work properly: a gain, an offset, and an RC filter. The output of the sensor is a voltage that you will read using the ADC. Before building the current sensor circuit, we will write and test the necessary software. Verifying the software independently of the complicated current sensor circuitry will make debugging the software and the hardware easier.

[2] The compiler will probably generate an atomic operation here anyway, but it is not guaranteed to do so.

Software

The `isense` module owns the ADC; therefore, this module should provide functions for reading the current sensor. The ADC will use manual sampling but automatic conversion, which will allow you to initiate a sample and wait for a complete reading before continuing. You want to get readings as fast as possible so sample for the shortest permissible time (you will need to check the data sheet to find this), and set the conversion clock to be as fast as possible. For more details about the ADC, see Ch. 10. Remember, put the function prototype in `isense.h` and the implementation in `isense.c`.

1. Implement a function that initializes the ADC. Call it from `main`.
2. Implement a function that reads the ADC and waits for the conversion to finish.
3. Implement a function that performs a few ADC reads and averages them: this will help reduce noise.
4. Add a menu item to access the averaged ADC reading and return it to the client.
5. Using voltage dividers, connect various voltages to the analog input and read the ADC on the client. Verify that the ADC returns the correct count for the given voltages.

Once confident that the ADC software works, you can begin to design the current sensing circuit.

Hardware

Section 15.3.2 contains the information needed to choose components for the current sensor and connect it to the circuit; refer to that section to design the circuit. After you have determined the necessary component values, you must perform an additional calibration step, to convert ADC readings into current readings. The most direct calibration method involves using an ammeter. You will need to attach various resistors R to the circuit to change the current flowing through it. If you use small resistors, you may dissipate more power than the typical $\frac{1}{4}W$ rating; therefore, for small resistances, you must use a resistor with a higher power rating.

1. Disconnect the sense resistor from everything.
2. Read and record the ADC value; this reading corresponds to zero current and should be near the middle of the ADC range.
3. Attach a resistor and the ammeter to the circuit, as per Fig. 16.5.
4. Record the ADC reading and the current.
5. Using these two data points, find a linear fit, relating current to ADC ticks.
6. Write a function that reads the ADC and returns the current, in milliamps
7. Add a menu item to read the current sensor, in milliamps

Next, you will read different currents and make sure that your results match the ammeter reading.

1. Test the calibration by replacing the R resistor with another value.
2. Swap the 3.3V and ground connections and ensure that you read negative current.

If you want a more accurate fit, you can try taking many measurements and performing a least-squares fit to determine the calibration parameters.

If you lack a ammeter, you can estimate the fit by analyzing the circuit. First, derive the equation relating the current through the sense resistor to the ADC ticks. Assume DC current so that you can ignore the effect of the low-pass filter. By substituting the real component values into this equation you can derive a reasonable estimate for the current based on the ADC count.

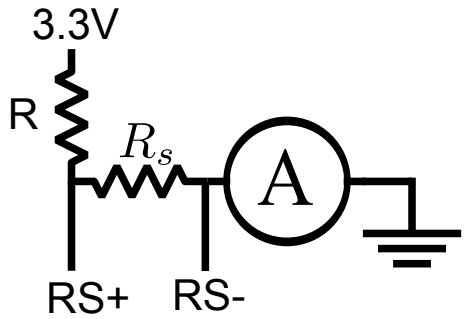


Figure 16.5: Current sensor calibration circuit.

16.4.6 Motor

By now you should have both the encoder and current sensor working. The next step is to provide low-level motor control. First we will implement part of the current control loop. After ensuring that the control loop behaves properly we will connect the H-bridge and the motor. When you finish this section you will be able to control the motor PWM signal from the client.

Software

The `motor` module, consisting of `motor.h` and `motor.c`, implements the current control loop and direct PWM control. Create a `motor_init` function to initialize the peripherals needed by the `motor` module. The `motor` module owns one timer for the control loop, another timer for the PWM signal, an output compare peripheral, and a digital output. As usual, call `motor_init` from `main`. Refer to Chs. **??**, 9, and 7 for more information about timers, output compare, and digital output.

After implementing `motor_init`, create the current control loop ISR. For now, the ISR should set the PWM duty cycle to 50 percent and toggle the output pin. After completing these coding tasks, you should verify everything with an oscilloscope.

1. Attach one probe to the digital output pin to ensure that the ISR frequency is correct. Remember, the pin flips at half the ISR frequency.

2. Attach another probe to the output compare pin, and verify that it produces 50 percent duty cycle PWM signal at the desired frequency.

Next, we will implement a function that converts a signed control effort into a valid PWM rollover count and a direction. The H-bridge we have chosen has two modes. In one mode, you use two PWM signals, one for each direction. The other mode uses one PWM signal to control the motor's speed and a digital output to control it's direction. We will use the mode that requires only one PWM signal. To convert a signed control effort into the appropriate PWM rollover count and output direction, create a function, `static set_pwm(int u)`. The units of `u` are in PWM counts: the function will saturate the signal if `u` is larger than the maximum PWM count. This function is `static` because only the `motor` module should set the PWM. Other modules can technically access the output compare peripheral directly, but this action would violate the module ownership rules. You should always use `set_pwm` to change the PWM value. By following this rule, you would need to change only one function if you, for example, decided to use the alternative H-bridge interface.

Not only will `set_pwm` be inaccessible from other modules, we will agree to only use `set_pwm` from within the current control loop ISR. By honoring this agreement, we will always know where the motor control signal is set, which will help you isolate problems. We also, however, want the user to be able to specify a PWM signal from the client. To implement this feature, declare the following global variable, which will hold a desired PWM reference:

```
static volatile int pwm_ref;
```

Next, implement a function, `current_set_duty(int duty)`, which accepts a signed duty cycle percentage (nominally from -100 to 100), converts it to a PWM control effort, and stores it in `pwm_ref`. Finally, remove the previous code from the ISR and update it as follows:

```
switch(util_state_get())
{
  case IDLE:
    set_pwm(0);
    break;
  case PWM:
    set_pwm(pwm_effort);
    break;
  default:
    NU32_LED2 = 0;  // unknown state, this is an error
  }
```

The current ISR now queries the state to determine what action to take. In IDLE mode, the motor will be stopped, while in PWM mode a desired PWM value will be sent to the motor. Include `utilities.h` to use `util_get_state`.

By following the rule that only the current control loop ISR modifies the PWM duty cycle, we guarantee that whenever the PIC32 returns to the IDLE state the motors will be stopped almost immediately (within a maximum of one current control loop period). Since the PIC32 puts the system into the IDLE state when you issue the "quit" command, quitting the client will stop the motors.

To set the PWM value from the client, add a new menu entry that prompts the user for a PWM signal and sends it to the PIC32. In response to a PWM command, the PIC32 menu should call `current_set_duty` to set the appropriate duty cycle reference, followed by `util_state_set` to set PWM mode. You can optionally add a command that allows you to explicitly set the IDLE state, or interpret a PWM signal of zero as meaning you want to return to IDLE mode. Before continuing, you should verify that you can properly set the motor PWM.

1. Set the PWM to 75. Verify the duty cycle, record the value of the direction pin.
2. Read the system state and verify that it is in PWM mode.
3. Set the PWM to -50. Verify the duty cycle, and make sure that the direction pin has changed.
4. Quit the client menu and restart it. Verify that the PWM duty cycle is 0 and that the system state is IDLE.

Hardware

Now that we can ostensibly control the motor direction and PWM from the client, it is time to prove it by connecting the H-bridge and motor. We use the DRV8835 H-bridge, described in Sec. 15.1.1; however, since this is surface-mount component we use it via a breakout board from Pololu. Figure 16.6 displays the circuit diagram for the H-bridge and Ch. 15 discusses how H-bridges work. Notice that one H-bridge output connects directly to the motor, while the other connects through the current sensing resistor. After constructing the circuit, turn the battery pack on and use the menu to test the motor. In addition to making sure that the motor spins properly, you should also check that the motor, encoder, and current sensor all have consistent directions. You could compensate for discrepancies in software, but it is easier to know that positive PWM yields increasing encoder counts and positive current, while negative PWM yields the opposite. You may need to swap the encoder channels, motor leads, or current sensor orientation to make these readings consistent. Firmly attach the bar to the and run the following tests:

1. Reset the encoder.
2. Set the PWM to 50% read the angle, and make sure that it increases. Read the current and make sure that it is positive.
3. Set the PWM to 100% and make sure that the motor spins faster than at 50%.

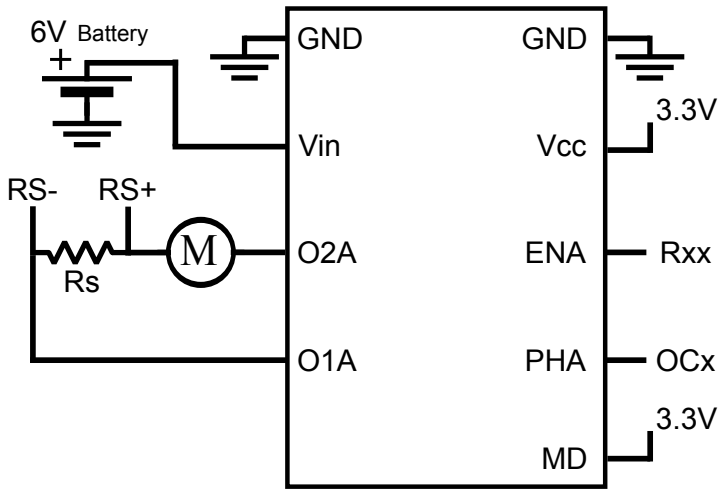


Figure 16.6: H-bridge circuit.

4. Set the PWM to -50% and then -100%, make sure it spins in the opposite direction. Make sure the current is negative and the angle decreases.

5. Set the PWM to 0% and make sure that the motor stops.

Congratulations! You fully control the low-level features of the hardware.

16.4.7 PI Current Control

We now implement a PI current controller. The PI controller makes the current sensor reading match a reference signal by adjusting the PWM signal and motor direction. For details about PI controllers see Ch. 11.

Gains

PI controllers rely on two gains: the proportional gain K_p and the integral gain K_i. You must choose between using an integral or a floating point data type to store the gains. Floating point gains are easier to use but much slower than integer gains. With floating point gains, you can, for example, compute the proportional part of the control law directly using $K_p * e$, where e is the error. Integer gains, however, require more care. For example, if you wanted to use $K_p = 1.54$, you would need to specify K_p as 154 and then divide by 100 later in the code. Dividing in different, but mathematically equivalent, orders could yield different results due to truncation; for example $(K_p/100) * e$ versus $K_p * e/100$ generally result in different answers when using integer arithmetic.

After deciding on the data type, you need to determine other properties of the variables that hold the gains. Will they be local or global? Static? Volatile? Remember that you must set the gains from the menu and that you may want to have different variables of the same name in the `position` module. After deciding on how you will declare the variables to store the gains, add menu options that allow the client to store and retrieve them. You will likely use this option many times while tuning the controller.

1. Declare variables to store your gains in `motor.c`.

2. Write functions to get and set the gains.

3. Add the ability to read and write gains to the menu.

4. Verify the menu items by reading and writing some gains.

Tuning Reference

To tune the controller, you need to provide an internally generated reference signal. By telling the controller to follow this reference and plotting the results on the client, you will be able to adjust the gains appropriately. First, add a TUNE state to the project. When in TUNE mode, the current control loop should track a 100 Hz, 200 mA amplitude square wave reference.

There are multiple ways to implement the reference signal. One method stores the desired reference in an array indexed by time-step. Another method examines the time-step directly to determine the appropriate reference. You should consider the advantages and disadvantages of these methods prior to implementation. In the control loop, when in TUNE mode, you should advance through the reference signal, repeating it from the beginning when you reach the end of one period. You should also create a menu item that allows you to tune the current controller. For now, it will simply reset the tuning reference to the beginning and place the PIC32 into TUNE mode. Later, it will also return the tuning data to the client.

Once you implement the tuning reference, you need to verify it by plotting it on the client. Serial communication, however, is too slow to perform directly from the control loop. Instead, the control loop will write the data it wants to send to a buffer. Meanwhile, the menu code will wait for the buffer to be full. When the buffer is full, the menu code will send the data to the client. This mode of processing is known as batch processing because we collect all of the data as a "batch" prior to using it. The implementation of this buffer is similar to what you did in Ch. **??**; however, it will be different because we want to implement it as part of the `utilities` module. We use the `utilities` module so that both the current control loop and the position loop can use the same buffer code.

Before proceeding you should

1. Add a TUNE state.
2. Add menu item to put the PIC32 into the TUNE state. Verify by reading the state.
3. Implement a reference signal for current loop tuning.

Buffer

Both the position and current control loops use a buffer in the `utilities` module to send data to the client. The data type used by the buffer is defined by the `struct control_data` in `utilities.h`. Currently, `control_data` has only one member, which can be used to store the reference signal. If your reference is a float rather than an int, you should change the type in `control_data`. In addition to the `control_data` struct, the utilities module uses three functions to manage the buffer: `util_buffer_init`, `util_buffer_write`, and `util_buffer_output`. You will eventually implement these functions, but first we will describe how they are used.

To initiate data recording, the menu code calls `util_buffer_init`. This function tells `utilities` how many samples to record. Next, the menu code sets the appropriate system state; for instance, TUNE, to initiate tuning mode. The menu then calls `util_buffer_output`, which waits until the buffer is full before sending the number of samples and the recorded data to the client. Meanwhile, based on the state the appropriate control loop will begin writing data into the buffer using `util_buffer_write`. Finally, the menu should restore the system to the appropriate state.

Here is an example that initiates recording of 10 samples of tuning data, and sends it to the PC:

```
util_buffer_init(10);
util_state_set(TUNE);
util_buffer_output();
util_state_set(IDLE);
```

The function `util_buffer_output()` will return after the control loop has stored 10 elements using `util_buffer_write`.

On the client, you can create a function to read the elements from the PIC32 and plot the data. For example, if the `control_data struct` contained two integer and one floating point elements, you could use the following MATLAB code to receive and plot the data:

Code Sample 16.6. `read_plot_matrix.m` Reads a matrix from the PIC32 and plots the results. It also computes a score, to help you evaluate the controller.

```
function data = read_plot_matrix(mySerial)
  nsamples = fscanf(mySerial,'%d');
  data = zeros(nsamples,3);
  for i=1:nsamples
    data(i,:) = fscanf(mySerial,'%d %d %d');
  end
  if nsamples > 1
    stairs(1:nsamples,data(:,1:2));
  else
    fprintf('Only 1 sample received\n')
    disp(data);
  end
  % compute the control score, assuming that the first column of data
  % is the reference and the second is the sensor reading
  score = norm(data(:,1)-data(:,2),1);
  fprintf('\nScore: %d\n',int32(score));
  title(sprintf('Response Score: %d',int32(score)));
  xlabel('Sample');
end
```

Note that if the first and second columns are the reference and measurement, then score represents a measure of how well your controller performed: the lower the better (like golf). You should compute the score for all of your plots and try to achieve as low a score as possible.[3]

You should now implement the three `util_buffer_` functions. Essentially `util_buffer_init` resets the buffer module and prepares it to collect samples. The `util_buffer_write` function stores new data in the buffer, as long as it is not full. Finally, `util_buffer_output` waits for the buffer to be full and then outputs the data to the client.

Decide how long to make the buffer. The size of the buffer determines how long you can record data for a given run, and is limited by the available RAM. To record data for longer periods of time you can decimate the data by having `util_buffer_write` only actually save data every n-th time it is called.

Remember that any variable shared between ISRs and mainline code should be declared volatile. Also, other modules should not need to access the internal `utilities` variables, so they can be declared `static`. You can safely assume that `util_buffer_init` will only be called from the menu code and `util_buffer_write` will only be called from within an ISR, and never from two ISRs simultaneously. Additionally, by always changing the system state after you initialize the buffer, you prevent the ISRs from calling `util_buffer_write` while mainline code calls `util_buffer_init`.

After you have implemented the buffer, test it by outputting the reference signal from the current ISR when in TUNE mode. When the client selects the "Tune current controller" option the client can either prompt the user for the number of samples to collect or you can hard-code this value in the client. The client will then send the number of samples it expects to the PIC32. The client then receives the number of samples that the PIC32 sends, followed by the samples themselves. Note that if you do not call `util_buffer_write` enough times in the interrupt, then util_buffer_output will freeze the PIC32. When you successfully implement everything you should be able to view the square wave reference signal.

Before you have completed this section, you should

1. Implement `util_buffer_init`, which sets the number of samples to record.

2. Implement `util_buffer_write`, which, if not all samples have been recorded, stores a sample. Optionally, this function may decimate the data.

[3] Judging the performance of a controller is application dependent and should not be boiled down to one number. Score presented is for entertainment only.

3. Implement `util_buffer_output`, which waits for all samples to be recorded. Then it outputs the number of samples, followed by each sample, one sample per line.

4. Use the `util_buffer_` functions to return the tuning reference to the client.

5. Adjust `read_plot_matrix` to accept the appropriate data.

6. Issue the tuning command and plot the results on the client. Verify the reference signal.

Current Control Law

You should now implement the current control law. To use the current control to both follow the tuning reference and respond to references from the position control loop, implement the control law as a separate function. The control law function should accept the current measurement and reference (in mA) and return a control effort. Use `set_pwm` from within the current ISR to send the control signal to the motors. Also, expand the `control_data` struct to include fields for the measurement and the reference and record these values using `util_buffer_write`.

The client you should plot the measurement and the reference to see how well the controller performs. Sometimes, for debugging purposes, viewing the control effort or other values may be useful. Feel free to add these quantities to the `control_data` struct and plot them. To tune the controller, focus on determining K_p while keeping $K_i = 0$. Next, add K_i to eliminate the steady-state error. For more information about PI controllers, refer to Ch. **??**.

To complete the `motor` module, you should provide a function that allows other modules to specify a current reference. This function should saturate the requested signal at a reasonable current level. You should also add another state, TRACK. In the TRACK state, the current module will attempt to track the current reference. Technically, you could provide a client function that allowed the user to set the current reference manually; however, this feature would not be useful unless you apply an external torque to the motor (why?).

Prior to continuing ensure that

1. You can issue a tuning command and view the results.

2. You have tuned the control loop and found suitable gains.

3. You have added the TRACK state, and the necessary function to allow the position loop to use current as its control effort value.

16.4.8 Position Controller

You will now implement the position controller inside the `position` module. The steps you need to take are similar for those used to implement current control. First, write the `position_init` function to configure the position control loop at the desired frequency. Like the current control ISR, the position control ISR should query the state and take the appropriate action.

Reference Trajectory

The position controller will advance through a reference trajectory specified by the user and then hold itself at the final position. By conceiving of the control problem in this manner, holding at a single position requires executing a trajectory with just one position specified. For convenience, we will provide separate menu options for holding a position and executing a trajectory; but they can share the same implementation.

The `position` module will store reference trajectories in an array, with each element corresponding to the desired position at the given time-step. It should also store the trajectory length and the current position. A function, `position_load`, will load trajectories. First, it will read the number of trajectory samples from the client. Next, it will read each position, one per line, and store it in the array. If the number of samples exceeds the size of the trajectory array, the function will not store the subsequent samples and illuminate LED2 to indicate an error. Finally, `position_load` will return the trajectory length.

Another function, `position_reset` will reset the current position to the beginning of the trajectories. This function will enable you to execute a trajectory multiple times, without needing to load it each time. You

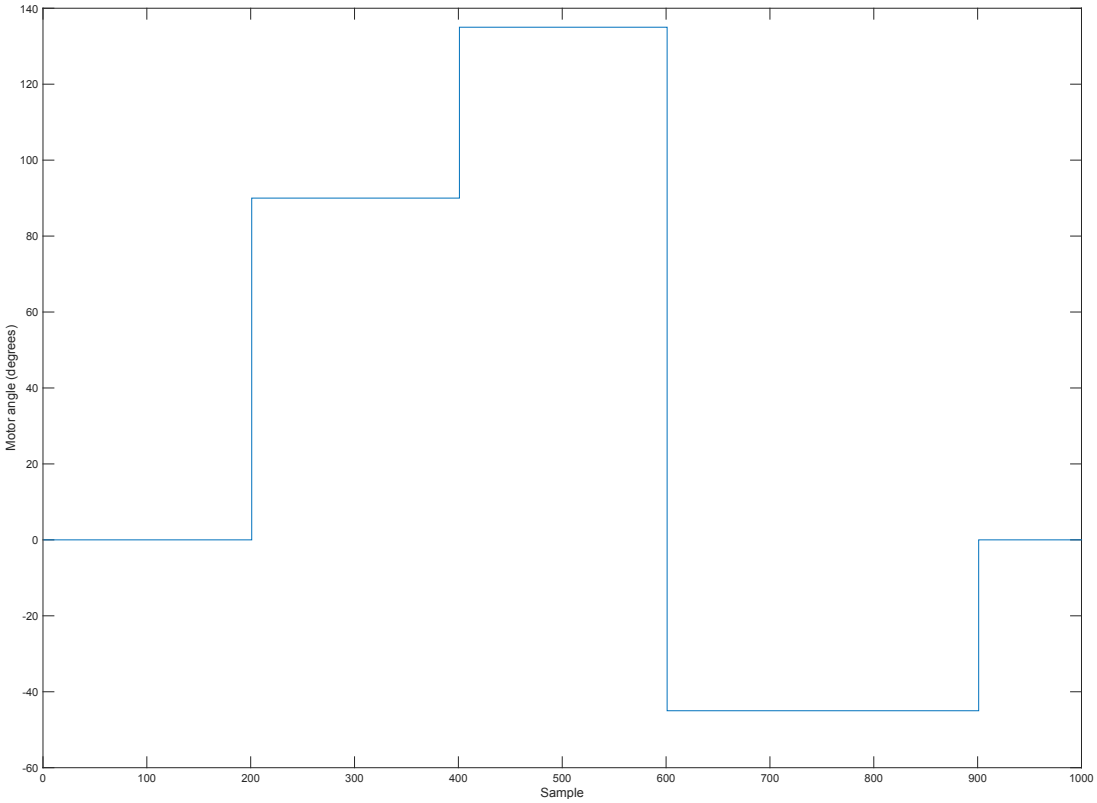


Figure 16.7: Step (zero order) reference trajectory.

should only call `position_reset` and `position_load` from the IDLE state, to avoid overwriting a currently executing trajectory.

Specifying trajectories as series of positions, one for each time-step can be cumbersome. We will, therefore, use client code to translate between simpler data entry methods and the format expected by the PIC32. The client will provide three commands for loading trajectories, all of which will send trajectory length followed by each desired position to the PIC32. The first command will tell the PIC32 to go to a specific angle. The PIC32 will attempt to hold the specified position and respond with data from the controller, and the client will plot the results. The other two commands will accept a matrix of times and angles, converting them into a trajectory using either zero order or third order interpolation. Figures 16.7 and 16.8 show example trajectories. Unlike goto, these commands will only load, not execute the trajectory, allowing you to execute the same trajectory multiple times without re-entering it.

You should also create two additional menu commands. One command will cause the PIC32 to execute the current trajectory and return the data to the client for plotting. The other command will determine how many time-steps worth of data the PIC32 returns, beyond the length of the trajectory. The client needs a method for knowing how many samples it expects to receive. You can either have the PIC32 communicate this information or keep track of the current trajectory length and additional samples in the client.

The procedure on the PIC32 in response to all three of these commands is similar; therefore you should create functions in `menu.c` to avoid duplicating code. To load a trajectory, set the IDLE state to stop any pending trajectories and avoid overwriting the current position target. Next, call `position_load` to read the trajectory from the client. You should store the return value, as this, combined with another parameter, will let you know how many samples to record.

To execute a trajectory, set the state to IDLE mode. Next, call `position_reset` to reset the current trajectory. Initialize data recording using `util_buffer_init`. Set the state TRACK to begin tracking. Finally, call `util_buffer_output` to send the recorded data to the client. After receiving a trajectory command, the PIC32 will remain in the TRACK state, holding the last position.

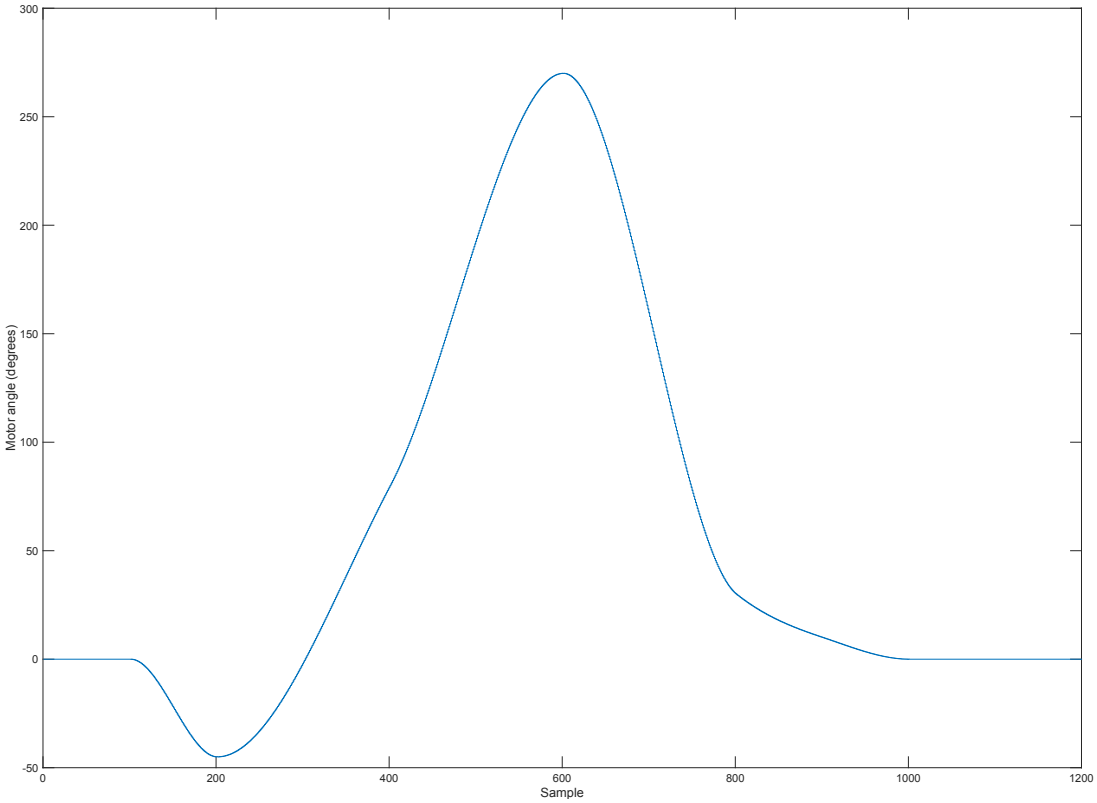


Figure 16.8: Cubic (third order) reference trajectory.

To help you implement parts of the client, we have provided you with the function `gen_ref.m`. This function converts the frequency of the motion control loop and an $n \times 2$ array of times and positions and returns a trajectory suitable for sending to the PIC32 by using interpolation. All the methods provided by the MATLAB function `interp1` are available. The two methods you should make accessible from your menu are `'previous'` and `'pchip'`. The `'previous'` method will generate a step trajectory where the position remains at a given angle until the next specified point. The `'pchip'` method will perform cubic interpolation between the trajectory points.

Code Sample 16.7. MATLAB code to generate a reference trajectory

```
function ref = gen_ref(freq,time_angles, method)
%   Generates a trajectory given a control loop frequency and a series
%   of times and angles. Angles are interpolated between the given points
%   using one of the interpolation methods from interp1 (see interp1 help
%   for details).  Also plots the trajectory.  Good methods to try are
%   'previous' which generates a 'step' trajectory and 'cubic' which
%   performs cubic interpolation
%
%   ref = gen_ref(freq, angles, method)
%
%   Input Arguments:
%       freq - the control loop frequency (in Hz)
%       time_angles - n x 2 matrix, first column is time (in seconds)
%                     second column is the angle to reach at the given time.
%                     The output trajectory is relative to the first time
%                     (i.e., the samples are shifted as if the first time
%                     were zero)
```

```
%       method - the interpolation method. 'previous' will create a step
%                trajectory, 'cubic' will use cubic interpolation
%
%   Output Arguments:
%       ref - a list of angles (in deg), one for each timestep.  If followed at freq
%             these angles represent the positions specified by time_angles
%             at the specified time
%   Example:
%       ref = gen_ref(2, [0, 0; 1.0, 90; 3.0, 45; 4.0, 0],'previous');
%
%       ref stores the angles at each timestep.  Since freq=2Hz these
%       timesteps correspond to the times [0,.5,1,1.5,2,2.5,3,3.5,4]
%       At 0 seconds, the angle is 0, at 1 second the angle is 90,
%       At 3 seconds, the angle is 45, and at 4 seconds the angle is 0.
%       In between the specified times, the previous angle is held, so
%       for example, at time .5, the angle is 0 and at time 2 the angle is
%       90
[numpos,numvars] = size(time_angles);

if (numpos < 1) || (numvars ~= 2)
  error('Input must be of form [t1,p1; ... tn,pn] for n >= 1.');
end

times = time_angles(:,1);       % target times
angles = time_angles(:,2);      % target angles
T = 1/freq;                     % period
samples = times(1):T:times(end);% the specific sampling times

ref = interp1(times,angles,samples,method); % perform the interpolation
str = sprintf('The trajectory takes %5.3f seconds and consists of %d samples at %7.2f Hz.', ...
               time_angles(end,1),length(ref),freq);
disp(str);
stairs(ref);
ylabel('Motor angle (degrees)'); xlabel('Sample number');
xlabel('Sample')

ref = round(ref.*10); %convert to decidegrees, rounded to integral values
```

Note, you can have the user enter a matrix directly by using `A = input('Enter Trajectory:  ')`. Before continuing you should test the commands by plotting the position reference signal from `TRACK` mode. We will refer to plotting this signal as "executing" the trajectory: however, the motor will not move until after you implement a controller. Also, the user enters angles in degrees, but the PIC32 uses tenths of a degree. Therefore, `gen_ref` performs a unit conversion before returning the trajectory. You should also perform a unit conversion when sending angles from the PIC32 to the client. It is easier for the user to always use degrees, even though, internally, the PIC32 uses one-tenth of a degree.

Test the following:

1. Specify the number of time-steps, beyond the end of the trajectory, for which you will collect data.
2. Test the go-to command by specifying an angle and viewing the reference signal.
3. Load a step trajectory.
4. Execute the step trajectory and verify the reference signal.
5. Load a cubic trajectory.
6. Execute the cubic trajectory and verify the reference signal.

After you have added the entries for the three trajectory entry methods, the execute trajectory command, and the data collection length command you should test them. You have not implemented the position controller yet, but you can call `util_buffer_write` to plot the reference signal when in TRACK mode and make sure that it appears as you expect it to.

PID Control

The final step in the project is to implement PID control. First, create a method for setting the gains K_p, K_i and K_d, to make tuning the control loop easier. The gains can be either integers or floats: the same advantages and disadvantages that applied for current control apply here. To tune the control loop, first find a K_p value while leaving K_i and K_d at zero. Then introduce some derivative control to increase damping, and finally some integral control to achieve zero steady state error for step inputs. Remember to include anti-windup protection for the integral error. You can find more information about the control loops in Ch. 11. After you have implemented the PID controller you need to tune it. First, make sure that the current loop is tuned. Then, tune the position controller so that it can complete the following test trajectories successfully. All of these trajectories assume that zero degrees indicates that the bar is horizontal. You should view a plot for all of the trajectories.

1. Set the data to record for 600 time-steps beyond the trajectory length.
2. From zero degrees, move to 90 degrees. Also, try disturbing the bar.
3. Set the data to record for 200 time-steps beyond the trajectory length.
4. Load and execute the following step trajectory `[0,0; 0.5, 0; 1, 90; 2, 135; 3, -45; 4.5, 0; 5,0]`. The controller should achieve zero steady-state error.
5. Load and execute the following cubic trajectory `[0,0; 0.5,0; 1, -45; 2, 80; 3, 270; 4, 30; 4.5, 10; 5, 0; 6,0]`.

Congratulations! You now have a complete motor control system. Of course, there is room for additional features. Some ideas are discussed in the next section.

16.4.9 Extensions

Now that you have a fully functioning motor control system you can begin to add. Here are some ideas.

Gains

Currently you must re-enter gains every time you reset the PIC32; this quickly becomes annoying. You can, however, store the gains in flash memory, which will allow them to persist, even when the PIC32 is powered off. To see how to access flash memory, refer to Ch. **??**.

LCD Display

Connect an LCD display to the PIC32. Use it to show information about the current system state, the gains, and whatever else you want.

Feed-forward Control

In Ch. **??**, we learned how to characterize a motor. Notice that we have mostly ignored the motor parameters when implementing the control law. Our ability to do this demonstrates the power of feedback control; however, incorporating a model of the motor into the control loop could improve the motor's performance. Also, you need not restrict yourself to PID control; even though widely used and useful, there are many other control methods out there.

Real-time Data

Currently we employ batch processing to retrieve motor data. This severely limits how long you can collect data before running out of memory. What if you want to see the motor data in real-time? A data structure called a circular (a.k.a. ring) buffer can help. The circular buffer has two positions indices; one for reading and one for writing. Data is added to the array at the write position, which is subsequently advanced. If the end of the array is reached, the write position wraps around to the beginning. Data is read from the read position and the read index advanced, also wrapping around. If the read position and write position are the same, the buffer is empty. If the write position is one behind the read position, the buffer is full. When using a circular buffer in this project, either the current loop or position loop will add data to the buffer. Rather than waiting for the buffer to be full, `util_buffer_output` will loop for the number of expected data samples. In loop iteration, `util_buffer_output` waits for the buffer to be non-empty, then reads one item from the buffer and sends it to the client.

16.4.10 Conclusion

This project required you to combine your knowledge of electronics, programming, and control theory; using it to create a comprehensive motor control system. We started the project by dividing it into smaller pieces, from the top down. We then built the project from the bottom up, testing incrementally. These approaches go hand-in-hand: partitioning the project enabled incremental testing. Attempting to create a project without such an approach would be even more difficult, especially in an embedded environment where the tools available for tracking down bugs are limited. Of course, what appeared to be a series of (hopefully) straightforward steps was actually the result of an iterative process of design and testing. Compromises occurred between performance, clarity, and complexity. When working on your own projects, remember that everything in engineering involves trade-offs and that the initial design will change as you learn from experience.

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.

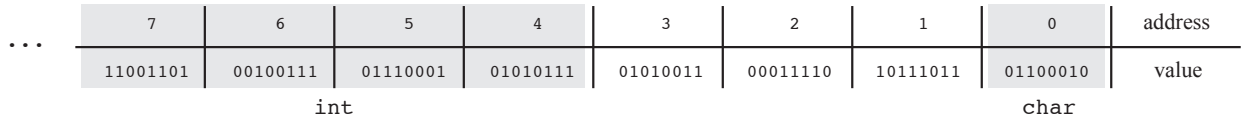
...	7	6	5	4	3	2	1	0	address
	11001101	00100111	01110001	01010111	01010011	00011110	10111011	01100010	value

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?

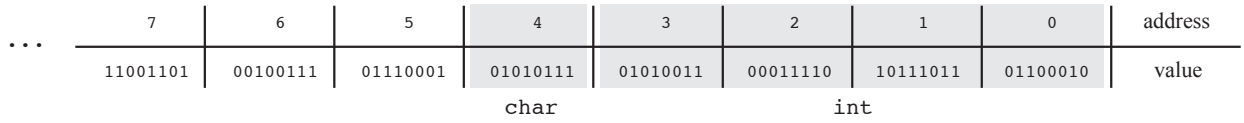
The answer lies in the **data type** of the variable at that address. In C, before you use a variable, you have to *define* it and its type. This tells the compiler how many bytes to set aside for the variable and how to write or interpret 0's and 1's at the address(es) used by that variable. The most 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
```

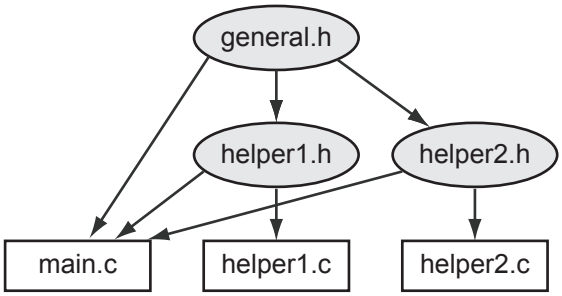


Figure A.1: An example project consisting of three C files and three header files. Arrows point from header files to files that include them.

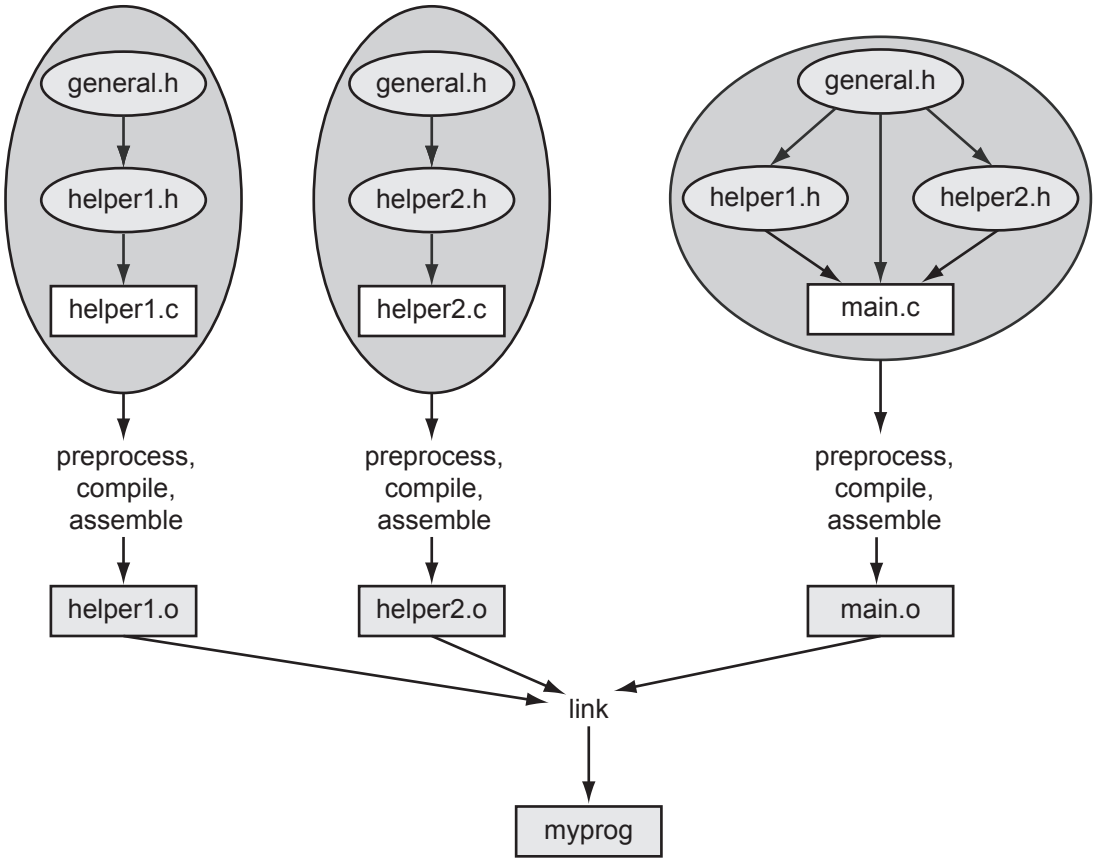


Figure A.2: The building of the project in Figure A.1.

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);
}
```