Embedded Computing in C with the PIC32 Microcontroller
DRAFT
March 2013 version

Chapter 1

Quickstart

Microchip PIC32s are powerful microcontrollers that can be purchased for less than $10 in quantities of one. The PIC32 combines, in a single chip, a 32-bit central processing unit (CPU), RAM data memory, flash nonvolatile program memory, and many *peripherals* useful for embedded control, such as several channels for analog-to-digital conversion, digital I/O, synchronous and asynchronous serial communication, pulse-width modulated output, etc. The RAM, flash memory, and peripherals are what distinguish a microcontroller, like the PIC32, from a microprocessor.

The "32" in PIC32 refers to the 32-bit CPU architecture: the CPU operates directly on 32-bit instructions and registers, and the instruction and data buses are 32 bits wide. This means that 32 bits can be fetched from RAM simultaneously, for example. While 8-bit and 16-bit microcontrollers continue to be popular, a primary advantage of 32-bit microcontrollers is the greater computational horsepower they offer.

There are three families of PIC32: the 1xx/2xx, 3xx/4xx, and 5xx/6xx/7xx families. Each family consists of a number of different chip models, differing in the number of pins, the amount of RAM and flash available, and the number and type of peripherals available. Representatives of the three families include, respectively, the PIC32MX250F128D, the PIC32MX460F512L, and the PIC32MX795F512L. The chips in the 3xx/4xx and 5xx/6xx/7xx families have 64 pins or 100 pins and are available only in surface mount packages, while those in the 1xx/2xx families are available in 28, 36, and 44 pin variants, some of which are "dual inline packages" (DIPs) that can be plugged directly into a solderless breadboard. The 3xx/4xx family is the original PIC32 family; the 5xx/6xx/7xx family offers additional support for CAN bus and ethernet communication, while the most recent 1xx/2xx family, while generally having fewer peripherals available, offer interfaces for connecting to audio devices and capacitive-based touch sensors. The 1xx/2xx family also offers the most flexible mapping of different pins to different functions (Peripheral Pin Select).

While the three families share most features, and most of this book applies to all families, where there are differences we will focus on the 5xx/6xx/7xx family. Also, we will occasionally find it convenient to cite specific numbers, such as the amount of RAM and flash memory available, and in these cases we will use the PIC32MX795F512L as our model. This is the PIC32 used on the NU32 development board as well as the Microchip PIC32 USB Starter Kit II and PIC32 Ethernet Starter Kit.

The PIC32MX795F512L features a max clock frequency of 80 MHz, 512 KB program memory (flash), and 128 KB data memory (RAM). It also features 16 10-bit analog-to-digital input lines (multiplexed to a single analog-to-digital converter, or ADC), many digital I/O channels, 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 a number of pins that can generate interrupts based on external signals, among other features.

The purpose of this chapter is to make sure you have everything you need to run your first simple programs. A deeper exploration of the hardware and software is left to the following chapters.

1.1 What You Need

Detailed instructions accompanying this chapter can be found on the wiki's quickstart page, `http://hades.mech.northwestern.edu/index.php/NU32:_Quickstart`.

In this chapter we start with the PIC32 by getting some simple code up and running quickly. To do this, you need the following three things:

1. **A host computer.** The host computer is used to create PIC32 programs. Any operating system is fine.

2. **A PIC32 board.** The PIC32 needs some external electronics to function, and these are typically provided by a small printed circuit board on which the PIC32 is mounted. These boards often have LEDs and buttons for simple input and output. Example boards include:

   (a) **Starter kits.** Microchip's various PIC32 Starter Kits, such as the PIC32 Starter Kit, PIC32 USB Starter Kit II, PIC32 Ethernet Starter Kit, Microstick II, and PIC32MX1/MX2 Starter Kit consist of boards with the PIC32, some I/O devices for user interface, and an onboard **programmer** that programs the flash memory of the PIC32 using a USB connection to the host computer.

   (b) **Development boards.** A development board is similar to a starter kit, except that there is no programmer onboard. The PIC32's flash memory is programmed by a separate external programmer device, such as Microchip's ICD 3 or PICkit 3, which connects to a USB port of the host computer. Examples of development boards include Microchip's PIC32 plug-in modules coupled with the Explorer 16 Development Board, as well as the NU32 and UBW32 development boards.

   (c) **Development boards with a PIC32 with an installed "bootloader."** If the PIC32 on the development board already has a **bootloader** installed in program memory, the PIC32 can be programmed without an external programmer device. When the PIC32 is powered on, the bootloader runs, and depending on whether or not the user is pushing a button, it either (1) jumps to a "user" program that has already been installed somewhere else in program memory, or (2) receives a new user program from the host computer (e.g., via a USB connection) and writes the new program to program memory. You can use the bootloader to change the user program as often as you want. There is no need for an external programmer device, only a program on the host computer (the "bootloader communication utility") that talks to the bootloader. Examples of development boards with preloaded bootloaders include the NU32 and the UBW32.

3. **Software to program the PIC32.** See Section 1.1.1.

Most of the examples in this book use the NU32 development board with an installed bootloader.

1.1.1 Software to Download

Regardless of the particular PIC32 development board, you need the following free Microchip software:

- **The Microchip XC32 compiler, which includes Microchip's C software library.** The XC32 compiler (also called the XC32++ compiler) is used to turn your C programs into programs that can be installed directly into PIC32 program memory.

- **The Microchip MPLAB X IDE.** The MPLAB X Integrated Development Environment provides a front end to the XC32 compiler.

For the examples in this book, you also need the following software:

- **The FTDI Virtual COM Port (VCP) Driver.** Download and install the driver appropriate to your operating system from FTDI Chip's website. This driver allows you to use a USB port as a "virtual COM port" to talk to the NU32 board.

- **The bootloader communication utility.** The bootloader communication utility is a program for your host computer that talks to the bootloader on the NU32, allowing installation of a program in flash memory without an external programmer.

- **(Optional) A terminal emulator program.** A terminal emulator provides a simple I/O interface to a virtual COM port. Characters you type on your keyboard can be received by the PIC32, and output from the PIC32 can be displayed on your screen. One option available on all operating systems is CoolTerm. Alternatively, for Windows, you can try downloading the PuTTY terminal emulator. For Linux and Mac OS X, you can use the built-in `screen` emulator from the command line.

1.1.2 Reading Material

It is also a good idea to download reference documentation from the Microchip website for future consultation. Much of this documentation has been copied to the wiki for convenience.

- **The PIC32 Family Data Sheet.** We will focus on the PIC32MX5XX/6XX/7XX Family Data Sheet, but the PIC32MX1/2 and PIC32MX3/4 families share many common features.

- **The individual chapters of the PIC32 Reference Manual.** Search for "Microchip Reference Manual," and on the Microchip page search for the chapters corresponding to the PIC32MX795F512L. You'll find around 30 chapters to download. The wiki has an early version of the Reference Manual as a single file.

When learning about a particular capability of the PIC32, it is usually best to first consult the relevant section in the Data Sheet, and then if you need more information, the more detailed Reference Manual chapter. Some of the information in the Reference Manual is not relevant to all PIC32s, however; for example, not all PIC32s have all the *special function registers* (see Chapter 2) mentioned in each chapter. Also, the Microchip documentation is not very easy to get started with. Therefore, the purpose of this book is to be your introduction to the PIC32, and to help you get comfortable exploring the Data Sheet and Reference Manual.

1.2 Loading Your First Program

To get a program running on the PIC32, we first write the program in C, then compile to an executable file, then load the executable onto the PIC32. As our first step, however, let's start with a pre-compiled executable file and simply load it onto the NU32.

1.2.1 NU32 with a Bootloader

For an NU32 board with a bootloader installed on the PIC32, download `NU32test.hex` from the wiki. Using the instructions on the wiki, write `NU32test.hex` to the PIC32 flash memory using your bootloader communication utility. When the program is running, the NU32's LEDs flash when you press the USER button and cease flashing when you let go.

1.2.2 NU32 Standalone (No Bootloader)

For an NU32 board without a bootloader installed on the PIC32, download `NU32test_standalone.hex` from the wiki. Using an external programmer device like an ICD 3 or a PICkit 3, and following the instructions on the wiki, write `NU32test_standalone.hex` to the PIC32 flash memory. When the program is running, the NU32's LEDs flash when you press the USER button and cease flashing when you let go.

1.3 Compiling Your First Program

Now that you have confirmed your ability to load an executable on the PIC32, the last step is to create your own executable. In this section you will write a program that turns LEDs on and off and responds to a pushbutton. Section 1.4 gives a program for the NU32 that takes input from the host computer keyboard and displays output on the computer screen.

1.3.1 NU32 with a Bootloader

The C program is called `simplePIC.c`. Download the following from the wiki:

- The `simplePIC.c` file, the C source code for our program.
- The `NU32bootloaded.ld` file. This is a custom "linker script" that specifies where the executable program should be placed in the PIC32's program memory. We will learn more about linker scripts in Chapter 3.

In the MPLAB X IDE, create a project called `simplePIC`. The wiki gives instructions on how to create the project, add the source code to the project, build the executable, load it onto the PIC32, and run it. Once the program is running on the PIC32, you see that the two LEDs alternate on and off, pausing when the USER button is pressed. We will learn more about the operation of this program in Chapter 3.

The program listing is given in Code Sample 1.1.

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

```
#include <plib.h>

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 the two lights
  }
  return 0;
}

void delay(void) {
  int j;
  for (j=0; j<1000000; j++) { // number is 1 million
    while(!PORTCbits.RC13);   // Pin C13 is the USER switch, low if pressed.
  }
}
```

Now that you can build and load the program, try changing the program, rebuilding, and reloading. For example, you could change the number of cycles in the `delay` loop to 2 million.

1.3.2 NU32 Standalone

If you have an external programmer like the ICD 3 or PICkit 3, you can program an NU32 without a bootloader (or, if there is a bootloader already loaded in the PIC32 flash memory, you can write over it). This is called a *standalone* program.

In MPLAB X, create a project called `simplePIC_standalone`. You need only one file, `simplePIC_standalone.c`. You do not need a custom linker script, as the default linker script suffices for a standalone program. The program listing is given below.

Code Sample 1.2. `simplePIC_standalone.c`. Standalone version of bootloaded `simplePIC.c`.

```
#include <plib.h>

// configuration bits are not set by a bootloader, so set here
#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

#define SYS_FREQ 80000000          // 80 million Hz

void delay(void);

int main(void) {

  SYSTEMConfig(SYS_FREQ, SYS_CFG_ALL);
  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 the two lights
  }
  return 0;
}

void delay(void) {
  int j;
  for (j=0; j<1000000; j++) { // number is 1 million
    while(!PORTCbits.RC13);   // Pin C13 is the USER switch, low if pressed.
  }
}
```

Instructions for building the executable, loading it onto the PIC32, and running it are all given on the wiki. Just as for the bootloaded version, once the program is running, the two LEDs alternate on and off, pausing when the USER button is pressed.

1.4 Communicating with the Host

Now you are ready for a second program, `talkingPIC.c`, that provides keyboard input to the NU32 and prints output to the host computer screen. These capabilities are quite useful for user interaction and debugging. To

gain access to these capabilities, we will compile `talkingPIC.c` with the NU32 library, `NU32.c` and `NU32.h`. This library will be discussed in Chapter 4.

1.4.1 NU32 with a Bootloader

For this project, download the source files `talkingPIC.c`, `NU32.c`, and `NU32.h`, as well as the custom linker script `NU32bootloaded.ld`. Create a project `talkingPIC` in the MPLAB X IDE and add the four files to the project, as indicated on the wiki. Build and load the executable as in Section 1.3.1. Once the program is running, to talk to the PIC32 you must either connect through the bootloader communication utility, as described on the wiki, or through a terminal emulator. Once connected, the NU32 echoes whatever you type back to your own computer screen, demonstrating two-way communication.

Code Sample 1.3. `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 <plib.h>
#include "NU32.h"
#define MAX_MESSAGE_LENGTH 200

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

  NU32_Startup();
  sprintf(message, "Hello World!\r\n");
  NU32_WriteUART1(message);
  while (1) {
    sprintf(message, "Hello?  What do you want to tell me?  ");
    NU32_WriteUART1(message);
    NU32_ReadUART1(message, MAX_MESSAGE_LENGTH);
    NU32_WriteUART1(message);
    NU32_WriteUART1("\r\n");
    NU32LED1 = !NU32LED1; // toggle the LEDs
    NU32LED2 = !NU32LED2;
  }
  return 0;
}
```

1.4.2 NU32 Standalone

The standalone version of `talkingPIC.c` requires the same files as the bootloaded version in Section 1.4.1, with the exception that the custom `NU32bootloaded.ld` linker script should not be used, since the default linker script is appropriate for standalone programs. Also, you must uncomment the first line of `talkingPIC.c`, so that it now reads

```
#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
```

This tells `NU32.h` that the program is being compiled as a standalone version.

Create a project `talkingPIC_standalone` in the MPLAB X IDE and build and load the executable as in Section 1.3.2. Once the program is running, to talk to the PIC32 you must either connect through the bootloader communication utility, as described on the wiki, or through a terminal emulator. Once connected, the NU32 echoes whatever you type back to your own computer screen, demonstrating two-way communication.

1.5 Chapter Summary

- A microcontroller, like the PIC32, differs from a microprocessor in that it has onboard flash nonvolatile program memory, data RAM, and peripherals for sensing, communication, and control.
- A PIC32 can be programmed using a separate programming device (standalone program) or by using a bootloader. A bootloader is a program installed in the PIC32 flash memory that can communicate with a host computer (via a USB cable, for example), accept a new user program, and write it to program flash. Typically the bootloader knows to enter this "write new program" mode if a button is being pressed when the PIC32 resets. Otherwise, the bootloader simply calls a program that has already been installed in memory.
- To program a PIC32, you need a PIC32 development board or starter kit and a host computer with Microchip's XC32 compiler and MPLAB X IDE. To use the NU32 development board, you also need the FTDI virtual COM port driver, which allows your USB ports to talk to the PIC32 over a USB cable. If you are bootloading your programs, you also need the bootloader communication utility. Other useful software includes a terminal emulator program, which allows writing data to the PIC32 and displaying responses on the screen, and the Processing programming language.
- Creating a new executable for the PIC32 involves creating a project in the MPLAB X IDE, creating (or adding) files to the project, building the executable, and loading it onto the PIC32 (via a bootloader communication utility for bootloaded executable programs or via the MPLAB X IDE's programming function for standalone programs). Bootloaded programs for the NU32 use the custom `NU32bootloaded.ld` linker script, while the default linker script suffices for standalone projects.

1.6 Problems

1. Create the executables for `simplePIC.c` (or `simplePIC_standalone.c`) and `talkingPIC.c`, load them, and verify that they work. Try modifying them and verify that you see the expected behavior.

Chapter 2

Looking Under the Hood: Hardware

Now that we've got our first programs running, it's time to take a look under the hood. We begin with the PIC32 hardware, then move to the NU32 development board it sits on.

2.1 The PIC32

2.1.1 Pin Functions and Special Function Registers (SFRs)

The PIC32MX795F512L features a max clock frequency of 80 MHz, 512 KB program memory (flash), and 128 KB data memory (RAM). It also features 16 10-bit analog-to-digital input lines (multiplexed to a single analog-to-digital converter, or ADC), many digital I/O channels, 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 a number of pins that can generate interrupts based on external signals, among other features.

To cram so much functionality into 100 pins, many of the pins serve multiple functions. See the pinout diagram for the PIC32MX795F512L (Figure 2.1). As an example, pin 21 can serve as 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 briefly summarizes some of the pin functions. Some of the most important functions for embedded control are indicated 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 some memory address. The values of the SFR bits, 0 or 1, control the functions of the pins as well as other functions of the PIC32.

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 set the SFRs that control this pin to disable the OC4 function and to choose the RD3 function as digital output instead of digital input. Looking at the PIC32 Family Reference Manual section on Output Compare, we see that the 32-bit SFR named "OC4CON" determines whether OC4 is enabled or not. Specifically, for bits numbered $0 \ldots 31$, 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 a 0, OC4 is disabled, and if it is a 1, OC4 is enabled. So we clear this bit to 0. (Bits can be "cleared to 0," or simply "cleared" for short, or "set to 1," or simply "set" for short.) Now, referring to the I/O Ports section of the Reference Manual, we see that bit 3 of the SFR TRISD, i.e., TRISD$\langle 3 \rangle$, should be cleared to 0 to make pin 78 a digital output.

In fact, according to the Memory Organization chapter of the Data Sheet, OC4CON$\langle 15 \rangle$ is cleared to 0 by default on reset of the PIC32, so this step was not necessary. On the other hand, TRISD$\langle 3 \rangle$ is set to 1 on reset, making pin 78 a digital input by default. (This is for safety, to make sure the PIC32 does not impose an unwanted voltage on anything it is connected to on startup.)

We will see SFRs again and again as we learn how to work with the PIC32.

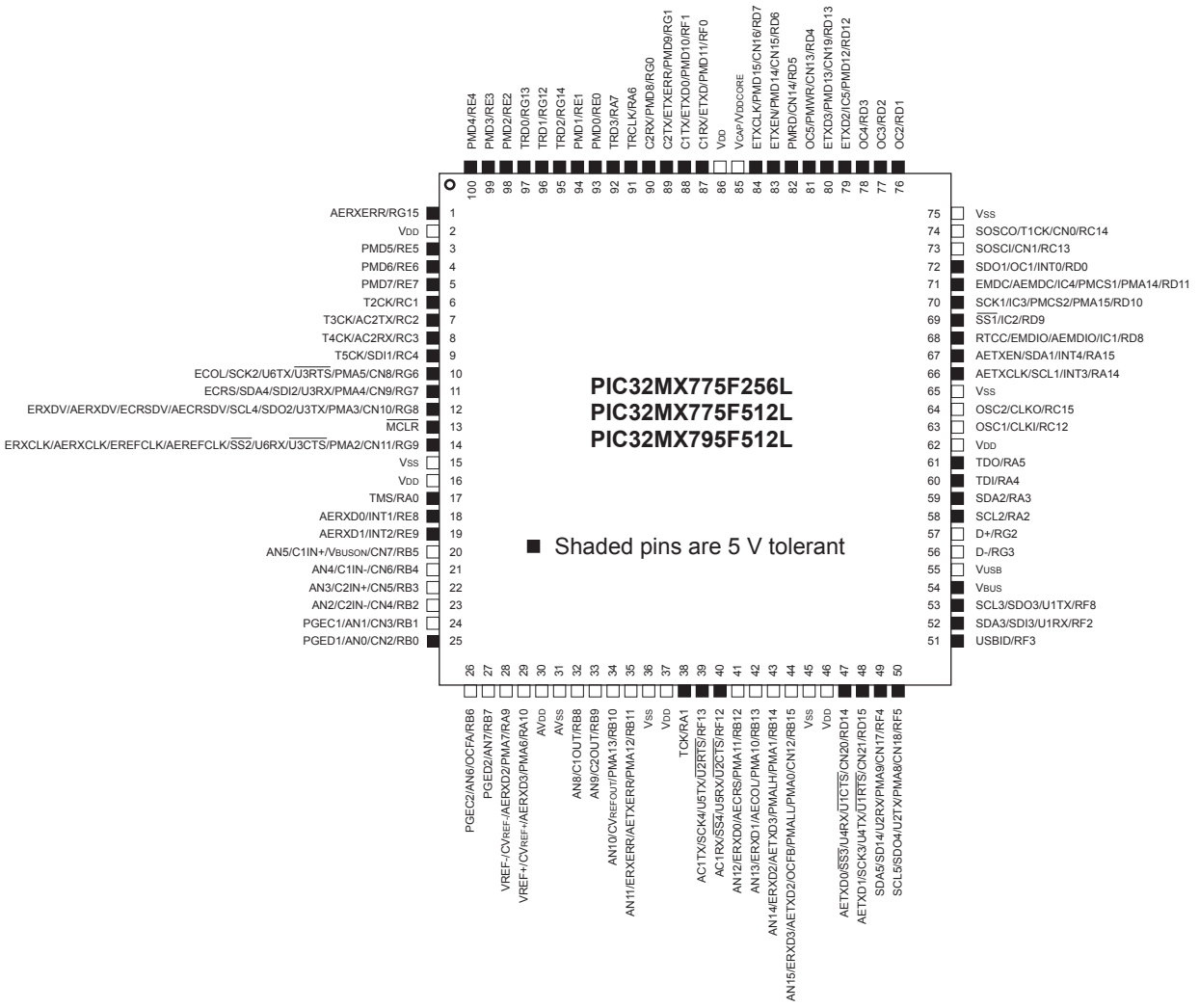


Figure 2.1: The pinout of the PIC32MX795F512L used on the NU32.

2.1.2 PIC32 Architecture

Figure 2.2 is a block diagram of the architecture of the PIC32. 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 year-month-day-time; and two comparators, each of which determines which of two analog inputs has a higher voltage.

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

Pin Label	Function
ANx (x=0-15)	analog-to-digital (ADC) inputs
AVDD, AVSS	positive supply and ground reference for ADC
BCLK1, BLCK2	clocks for infrared (IrDA) comm encoding and decoding for 2 UARTs
CxIN-, CxIN+, CxOUT (x=1,2)	comparator negative and positive input and output
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.

communication. The timing generation block that creates these clock signals and other elements of the architecture in Figure 2.2 are briefly described below.

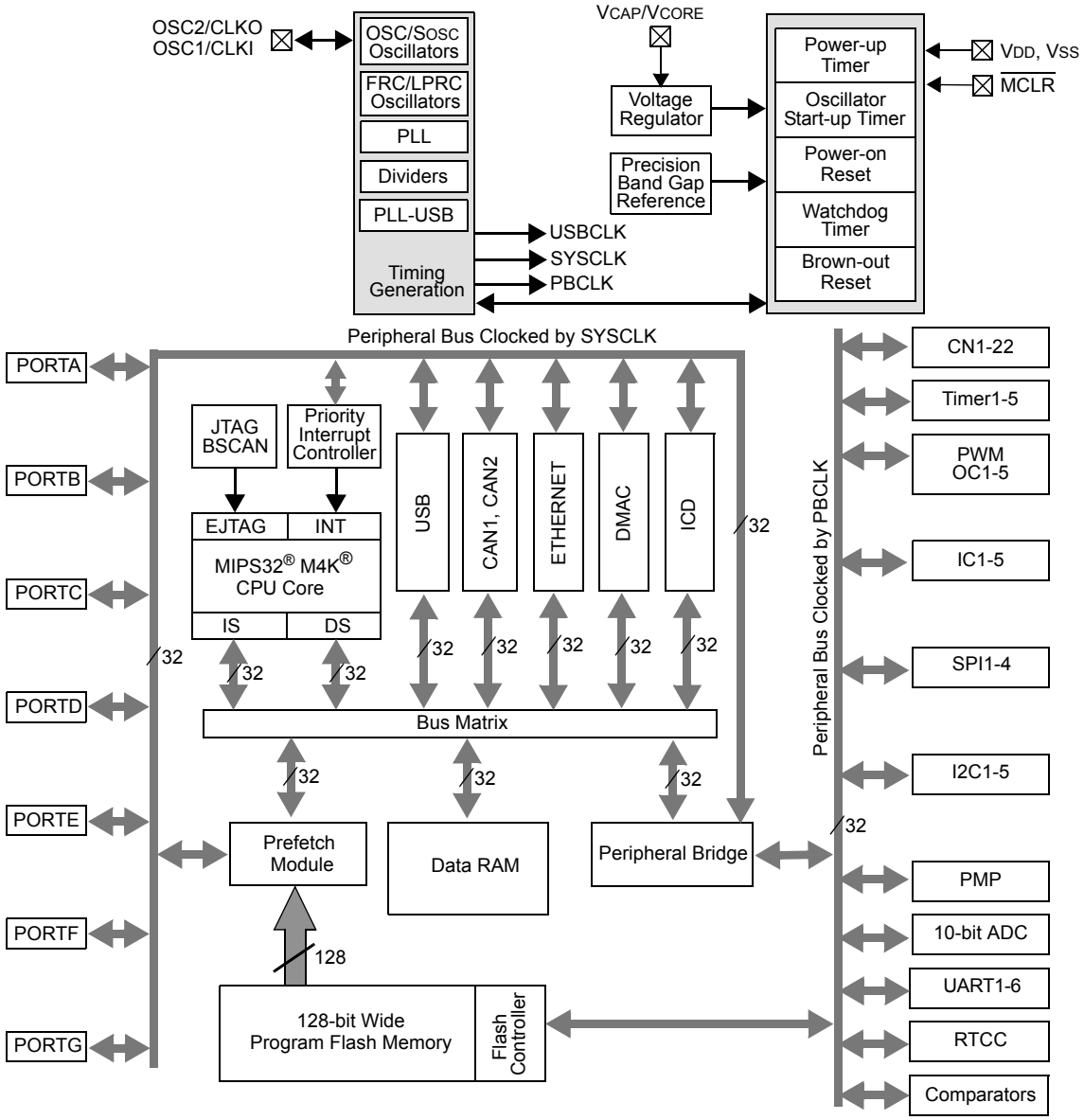


Figure 2.2: The PIC32MX5XX/6XX/7XX architecture.

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 MIPS® 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 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.

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.

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

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 allow it 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 1024 different voltage levels. Conversions are theoretically possible at a maximum rate of 1 million samples per second.

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^{2}C Serial Communication The Inter-Integrated Circuit protocol I^{2}C (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 PIC32 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 read from (to get sensor input, for example), and *boot flash*. Of these, we have not yet seen "boot flash." This is an extra 12 KB of flash 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.

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):

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

Physical Memory Start Address	Size (bytes)	Memory Type
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 PIC, 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*. These configuration bits set a number of important properties of how your 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 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.

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 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, 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 to your computer's USB port to a dual USB-to-RS232 chip that allows your PIC32 to speak RS232 to your computer's USB port. Two RS232 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.

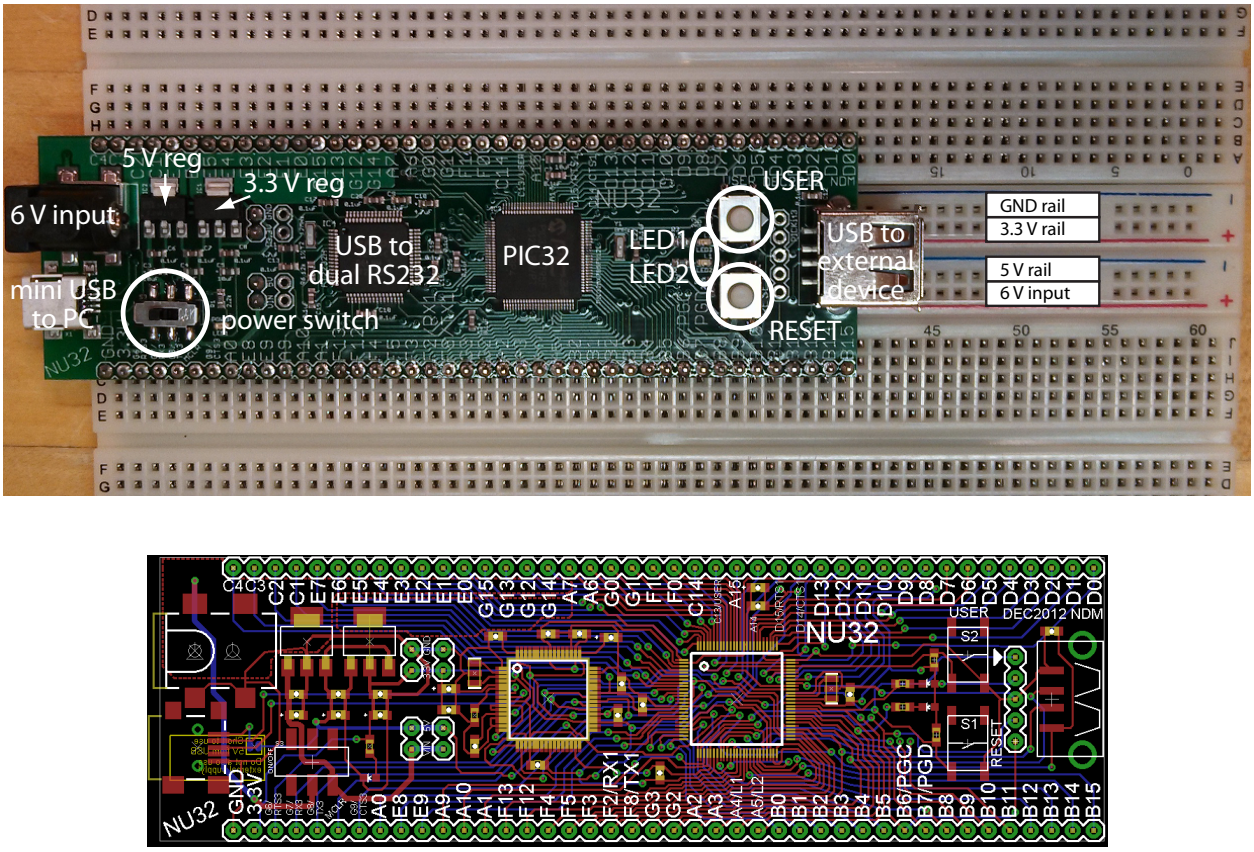


Figure 2.3: The NU32 development board: photo and PCB silkscreen.

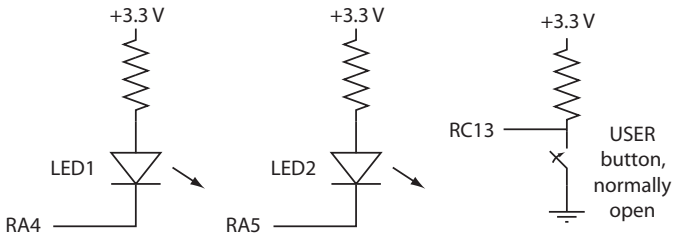


Figure 2.4: The NU32 connection of pins RA4, RA5, and RC13 to LED1, LED2, and the USER button, respectively.

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 RC13 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.

Function		Board			Function
GND		GND	C4	√	T5CK/SDI1/C4
3.3 V		3.3 V	C3	√	T4CK/C3
SCK2/U6TX/U3RTS/CN8/G6	√	**G6/RTS3**	C2	√	T3CK/C2
SDA4/SDI2/U3RX/CN9/G7	√	**G7/RX3**	C1	√	T2CK/C1
SCL4/SDO2/U3TX/CN10/G8	√	**G8/TX3**	E7	√	PMD7/E7
MCLR	√	MCLR	E6	√	PMD6/E6
SS2/U6RX/U3CTS/CN11/G9	√	**G9/CTS3**	E5	√	PMD5/E5
A0	√	A0	E4	√	PMD4/E4
INT1/E8	√	E8	E3	√	PMD3/E3
INT2/E9	√	E9	E2	√	PMD2/E2
VREF-/CVREF-/A9		A9	E1	√	PMD1/E1
VREF+/CVREF+/A10		A10	E0	√	PMD0/E0
A1	√	A1	G15	√	G15
SCK4/U5TX/U2RTS/F13	√	F13	G13	√	G13
SS4/U5RX/U2CTS/F12	√	F12	G12	√	G12
SDA5/SDI4/U2RX/CN17/F4	√	F4	G14	√	G14
SCL5/SDO4/U2TX/CN18/F5	√	F5	A7	√	A7
USBID/F3	√	F3	A6	√	A6
SDA3/SDI3/U1RX/F2	√	**F2/RX1**	G0	√	C2RX/PMD8/G0
SCL3/SD03/U1TX/F8	√	**F8/TX1**	G1	√	C2TX/PMD9/G1
D-/G3		G3	F1	√	C1TX/PMD10/F1
D+/G2		G2	F0	√	C1RX/PMD11/F0
SCL2/A2	√	A2	C14		T1CK/CN0/C14
SDA2/A3	√	A3	**C13/USER**		CN1/C13
A4	√	**A4/L1**	A15	√	SDA1/INT4/A15
A5	√	**A5/L2**	A14	√	SCL1/INT3/A14
PGED1/AN0/CN2/B0		B0	**D15/RTS1**	√	SCK3/U4TX/U1RTS/CN21/D15
PGEC1/AN1/CN3/B1		B1	**D14/CTS1**	√	SS3/U4RX/U1CTS/CN20/D14
AN2/C2IN-/CN4/B2		B2	D13	√	PMD13/CN19/D13
AN3/C2IN+/CN5/B3		B3	D12	√	IC5/PMD12/D12
AN4/C1IN-/CN6/B4		B4	D11	√	IC4/D11
AN5/C1IN+/CN7/B5		B5	D10	√	SCK1/IC3/D10
PGEC2/AN6/OCFA/B6		**B6/PGC**	D9	√	SS1/IC2/D9
PGED2/AN7/B7		B7/PGD	D8	√	RTCC/IC1/D8
AN8/C1OUT/B8		B8	D7	√	PMD15/CN16/D7
AN9/C2OUT/B9		B9	D6	√	PMD14/CN15/D6
AN10/CVREFOUT/B10		B10	D5	√	CN14/D5
AN11/B11		B11	D4	√	OC5/CN13/D4
AN12/B12		B12	D3	√	OC4/D3
AN13/B13		B13	D2	√	OC3/D2
AN14/B14		B14	D1	√	OC2/D1
AN15/OCFB/CN12/B15		B15	D0	√	SDO1/OC1/INT0/D0

Table 2.2: The NU32 pinout. 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.

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.
- 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 PIC32 requires external electronics for power, communication, etc. 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 Problems

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 OC2CON controls the behavior of Output Compare module 2. Referring to the Memory Organization section of the Data Sheet, what is the physical address of OC2CON?

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 (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? 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 begin to explore the compilation process and the XC32 compiler installation.

3.1 The Virtual Memory Map

In the last chapter we learned about the PIC32 physical memory map. The physical memory map is relatively easy to understand: the CPU can access any SFR, or any location in data RAM, program flash, or boot flash, by a 32-bit address that it puts on the bus matrix. Since we don't really have 2^{32} bytes, or 4 GB, to access, many choices of the 32 bits don't address anything.

In this chapter we focus on the virtual memory map. This is because almost all software refers only to virtual memory addresses. Virtual addresses (VAs) specified in software are translated to physical addresses (PAs) by the fixed mapping translation (FMT) unit in the CPU, which is simply

PA = VA & 0x1FFFFFFF

This bitwise AND operation simply clears the first three bits, the three most significant bits of the most significant hex digit.

If we're just throwing away those three bits, what's the point of them? Well, those first three bits are used by the CPU and the prefetch module we learned about in the previous chapter. If the first three bits of the virtual address of a program instruction are 100 (so the corresponding most significant hex digit of the VA is an 8 or 9), then that instruction can be cached. If the first three bits are 101 (corresponding to an A or B in the leftmost hex digit of the VA), then it cannot. 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]

You don't need to worry about the mysteries of cacheable vs. noncacheable instructions. Suffice to say that your program instructions will be made cacheable, speeding up execution.

The relationship of the physical memory map to the virtual memory map is illustrated in Figure 3.1. One important thing to note from the figure is that the SFRs are not included in the KSEG0 cacheable virtual memory segment. This is because the SFRs correspond to physical devices, e.g., peripherals, and their values cannot be cached. For example, if port B is configured as a digital input port, then the SFR PORTB contains the current input values. When your program asks for these values, it needs the current values; it cannot pull them from the cache.

For the rest of this chapter we will deal only with virtual addresses like 0x9D000000 and 0xBD000000, and you should realize that these refer to the same physical address. Since virtual addresses start at 0x80000000,

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

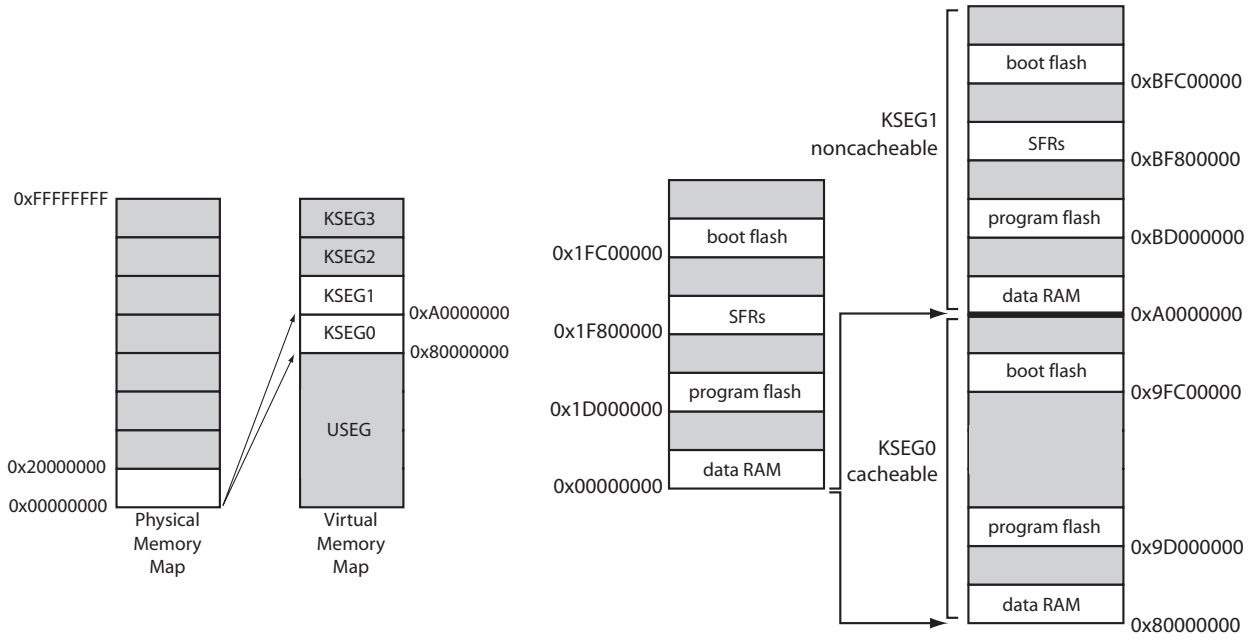


Figure 3.1: (Left) The 4 GB physical and virtual memory maps are divided into 512 MB segments. The mapping of the valid physical memory addresses to the virtual memory regions KSEG0 and KSEG1 is illustrated. The PIC32 does not use the virtual memory segments KSEG2 and KSEG3, which are allowed by the 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, 0x9FC02FF0 to 0x9FC02FFF in KSEG0 and 0xBFC02FF0 to 0xBFC02FFF in KSEG1, correspond to the device configuration words DEVCFG0 to DEVCFG3. Memory regions are not drawn to scale.

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: The Bootloaded `simplePIC.c` Program

Let's build the `simplePIC.c` bootloaded executable from Chapter 1.3.1. For convenience, here is the program again:

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

```
#include <plib.h>

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 the two lights
  }
  return 0;
}

void delay(void) {
  int j;
  for (j=0; j<1000000; j++) { // number is 1 million
    while(!PORTCbits.RC13);   // Pin C13 is the USER switch, low if pressed.
  }
}
```

Following the same procedure as in Chapter 1, "Build" the program and load it onto your NU32. Remember to use the linker script `NU32bootloaded.ld`.

When you have the program loaded and running, the NU32's two LEDs should alternate on and off, and stop while you press the USER button. This program refers to SFRs named TRISA, LATAINV, etc. These names align with the SFR names in the Data Sheet and Reference Manual sections on I/O Ports. We will consult the Data Sheet and Reference Manual often when programming the PIC32. We will come back to understanding the use of these SFRs shortly.

3.3 What Happens When You Compile?

Now let's begin to understand how you created the `.hex` file in the first place. Figure 3.2 gives a schematic of what happens when you press "Build" in your MPLAB X IDE.

First the **preprocessor** strips out comments and inserts `#include`d header files. 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 that are directly understood by the PIC32's CPU. So while some of your C code may be easily portable to other microprocessors, your assembly code will not be. These assembly files are readable by a text editor, and it is possible to program the PIC32 directly in assembly language.

The **assembler** then 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 for us!

Finally, the **linker** takes multiple object files and links them together into a single executable file, with all data and 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 as to where to place the data and program instructions in virtual memory. The end result is an executable and linkable format (`.elf`) file, a standard format. This file contains a plethora of information that is useful for debugging and *disassembling* the file into the assembly code produced by the compiler (Chapter 5.1.3). In fact, `simplePIC.c` has resulted in a `.elf` file that is hundreds of kilobytes! A final step creates a stripped-down `.hex` file of less than 10 KB that is suitable for placing directly into the memory of your PIC32.

Although the entire process consists of several steps, it is often referred to as "compiling" for short. "Build" is perhaps more accurate, and that is the term the MPLAB X IDE uses.

3.4 What Happens When You Reset the PIC32?

You've got your program running. Now you hit the RESET button on the NU32. What happens next?

The first thing your PIC32 does is jump to the first address in boot flash, 0xBFC00000, and begin executing instructions there.[2] For an NU32 with a bootloader installed, the preloaded bootloader program

[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.

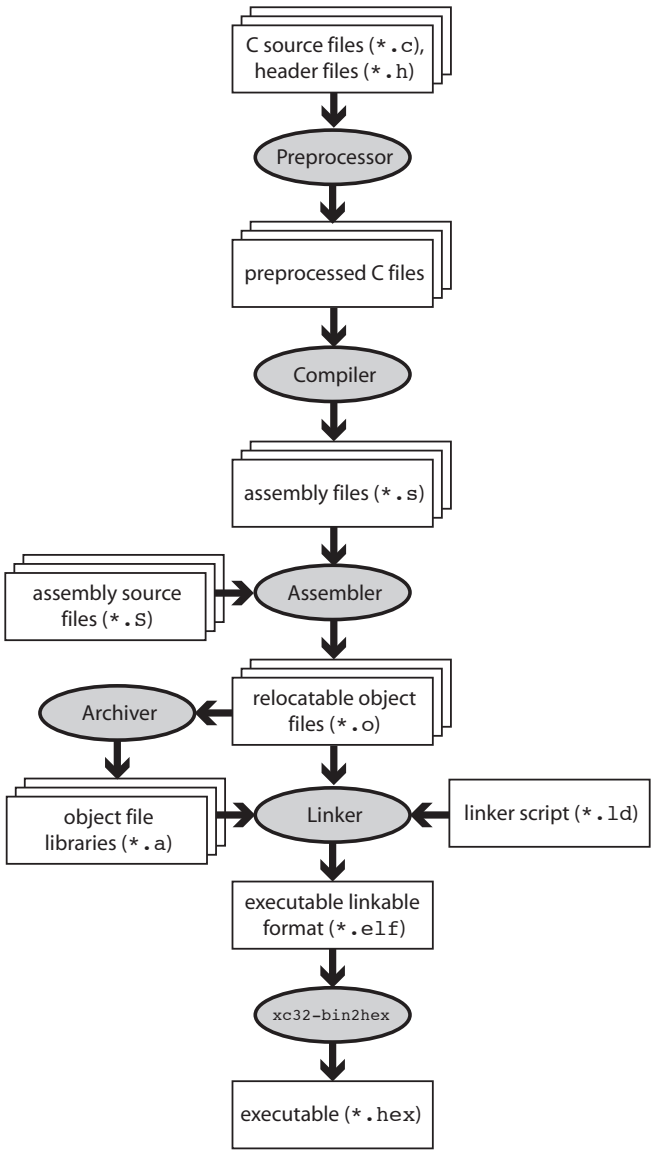


Figure 3.2: The "compilation" process.

sits at that location in memory. The bootloader first checks to see if you are pressing the USER button. If so, it knows that you want to reprogram your PIC32, so the bootloader attempts to establish communication with the bootloader communication utility on your computer. When communication is established, the bootloader receives your executable `.hex` file and writes it to your PIC32's program flash. (See question 3.) Let's call the virtual address where your program is installed `_RESET_ADDR`.

Note: The PIC32's reset address 0xBFC00000 is fixed in hardware and cannot be changed. On the other hand, there is nothing too special about the choice of the program flash address where the bootloader writes your program.

Let's say 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.

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.

3.5 Understanding `simplePIC.c`

OK, let's get back to understanding `simplePIC.c`. The `main` function is very simple. It 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 simply goes through a `for` loop a million times. Each time through the `for` loop we enter a `while` loop, which checks the value of (!PORTCbits.RC13). If PORTCbits.RC13 is 0 (FALSE), then the expression (!PORTCbits.RC13) evaluates to TRUE, and the program stays stuck here, doing nothing but checking the expression (!PORTCbits.RC13). When this evaluates to FALSE, the `while` loop is exited, and we continue with the `for` loop. After a million times through the `for` loop, control returns to `main`.

Special Function Registers (SFRs) The only reason this program is even a little interesting is that TRISA, LATA, and PORTC all refer to peripherals that interact with the outside world. Specifically, TRISA and LATA correspond to port A, an input/output port, and PORTC corresponds to port C, another input/output port.[3] We can start our exploration 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, consists of 12 different pins, and port C, with pins named RC1 to RC15, has 8 pins. These are in contrast to port B, which has a full 16 pins, labeled RB0 to RB15.

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

If you're curious what 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 upon reset. There are a lot of SFRs! But after a bit of searching, you 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

$$\text{0x0000C6FF} = 0000\ 0000\ 0000\ 0000\ 1100\ 0110\ 1111\ 1111.$$

The leftmost four hex digits (two bytes, or 16 bits) are all 0. This is because those bits don't exist, technically. 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 have a value 1, while the rest have value 0. The bits with value 1 correspond precisely to the pins we have available. (For example, there is no pin 8 on Port A.) So all of our available pins are configured as inputs, by default. This is for safety reasons; when we power on the PIC32, each pin will take its default direction before the program has a chance to 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 be fighting 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

[3] DDPCON corresponds to JTAG debugging, which we do not use in this book. The `DDPCONbits.JTAGEN = 0` command simply 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.

```
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 simply 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;
```

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 PORTCbits.RC13, for example.

The two other basic SFRs in this program are LATA and PORTC. 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, PORTC contains the digital inputs on the port C pins. (Notice we didn't configure port C as input; we relied on the fact that it's the default.) PORTCbits.RC13 is 0 if 0 V is present on pin RC13 and 1 if approximately 3.3 V is present.

Pins RA4, RA5, and RC13 on the NU32 Figure 2.4 shows how pins RA4, RA5, and RC13 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, RC13 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 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 go back 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; they are consecutive in the memory map. Since LATA is at 0xBF886020, LATACLR, LATASET, and LATAINV are at 0xBF886024, 0xBF886028, and 0xBF88602C, respectively.

OK, we should now have a pretty good understanding of 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; they must be declared somewhere. The only place they could be declared is in the included file `plib.h`. We've been ignoring that `#include <plib.h>` statement until now. Time to take a look.[4]

[4]Microchip often makes changes to the software it distributes, so there may be differences in details, but the broad strokes described here will be the same.

3.5.1 Down the Rabbit Hole

But where do we find `plib.h`? If our program had the preprocessor command `#include "plib.h"`, the preprocessor would look for `plib.h` in the same directory as the C file including it. But we had `#include <plib.h>`, and the `<...>` notation means that the preprocessor will look in directories specified in your *include path*. This include path was generated for you automatically when you installed the XC32 compiler. For us, the default include path means that the compiler finds `plib.h` sitting in the directory path

```
microchip/xc32/v1.20/pic32mx/include/plib.h
```

Your path or version number might be slightly different. In this book the directory separator character is `/`, consistent with Linux, Unix, and Mac OS X. On Windows, the directory separator character is `\`.

Including `plib.h` gives us access to many data types, variables, constants, macros, and functions that Microchip has provided for our convenience. While `plib.h` stands for "peripheral library," including `plib.h` provides functionality beyond just the peripherals.

Before we open up `plib.h`, let's look at the directory structure that was created when we installed the XC32 compiler. 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 `microchip/xc32/v1.20` and summarize what's in the nested set of directories, without being exhaustive.

1. `bin`: This 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.
5. `pic32-libs`: This directory is notable because it contains the `.c` C files, `.h` header files, and `.S` assembly files needed to create the object code for a number of pre-built PIC32 functions, particularly for the peripherals. Later we might want to look at these more closely, as they are some of the best documentation on the peripheral library functions. A few notable subdirectories:
   (a) `peripheral`: The subdirectories under this directory contain the C code and for peripheral library functions.
   (b) `include/peripheral`: This directory contains header files for peripheral library functions. Though there is a `plib.h` here, it is not the one the compiler finds when building `simplePIC.c`.
   (c) `dsp`: This directory contains some C functions that call vector math, filter, and Fourier transform code in the MIPS Digital Signal Processing library.
   (d) `libpic32`: This directory contains a number of C and assembly files for basic PIC32 functions, such as code that should be executed at the beginning of any executable (see next).
   (e) `libpic32/startup/crt0.S`: This "C run-time startup" assembly code gets inserted at the beginning of every program we create. This code takes care of a number of initialization tasks. For example, if your program uses uninitialized global variables, `crt0` writes zeros into their data RAM locations. If your global variables are initialized at the time of definition, e.g., `int k=3`, then `crt0` copies the initialized values from program flash to data RAM.
6. `pic32mx`: This directory has a number of files we are interested in.
   (a) `bin`: We can ignore these executables.
   (b) `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. `crt0.o`: This is the compiled object code of `crt0.S`. The linker combines this code with our program's object code and makes sure that it is executed first.

ii. `libmchp_peripheral.a`: This library contains the `.o` object code versions of the `.c` core timer functions in the top-level `pic32-libs` library.

iii. `libmchp_peripheral_32MX795F512L.a`: This library contains the `.o` object code versions of the `.c` peripheral library functions in the top-level `pic32-libs` library. There are versions of this file for every type of PIC32.

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

v. `ldscripts/elf32pic32mx.x`: For a standalone program, this is the default linker script that gives the linker rules as to where it is allowed to finally place the relocatable object codes in virtual memory. This script includes `procdefs.ld`, below.
If your program is built to be bootloaded, the linker uses your custom linker script instead, such as `NU32bootloaded.ld`.

vi. `proc/32MX795F512L/procdefs.ld`: This file is included by the default linker script, above. It declares segments of data RAM and program flash where the linker can place data and instructions, and it is specific to your particular PIC32 model. It also includes `processor.o`, below.

vii. `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. This tells the linker that it must use these addresses when making final address assignments. This makes sense; 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.

viii. `proc/32MX795F512L/configuration.data`: This file describes some constants used in setting the configuration bits in DEVCFG0 to DEVCFG3 (Chapter 2.1.4). The main purpose of these constants is to make your code more readable. For example, a standalone program like `simplePIC_standalone.c` from Chapter 1.3.2 has the following code:

```
#pragma config DEBUG = OFF         // Background Debugger disabled
#pragma config FPLLMUL = MUL_20    // PLL Multiplier: Multiply by 20
```

These `#pragma`s are nonstandard C code, and for our particular compiler, they are used to write to the DEVCFGx bits the values defined by constants like `MUL_20`, defined in `configuration.data`. Many of these `#pragma`s are used to set up the timing generation circuit that turn our 8 MHz resonator into an 80 MHz SYSCLK, an 80 MHZ PBCLK, and a 48 MHz USBCLK. You can learn more about the DEVCFGx configuration bits in the Special Features section of the Data Sheet.
For bootloaded code, the configuration bits are set by the bootloader program, so these `#pragma`s are not necessary.

(c) `include`: This directory contains a number of `.h` header files, including the one we've been looking for, `plib.h`.

i. `plib.h`: This is the file that was found in our compiler's include path. If we open it up, we find that it includes a bunch of other files! One of them is `peripheral/ports.h`, so let's open that one up.

ii. `peripherals/ports.h`: This file provides constants, macros, and function prototypes for library functions that work with the I/O ports. More importantly, for now, is that it includes `p32xxxx.h`. This file is found one directory up in the directory tree. Let's open that next.

iii. `p32xxxx.h`: This file basically just includes `xc.h`.

iv. `xc.h`: This file does a few different things, such as defining constants, macros, and including files defining MIPS constants and macros, but the most important for the moment is that it includes `proc/p32mx795f512l.h` because of the lines

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

Why does this code include `proc/p32mx795f512l.h`? When you were setting up your `simplePIC` project in the first place, you had to specify the particular PIC32 you were using. The MPLAB X IDE passed your answer to the compiler by "defining" the constant `__32MX795F512L__`. This allows the preprocessor to include the right files for your PIC32. Let's open `proc/p32mx795f512l.h`.

v. `proc/p32mx795f512l.h`: Whoa! This file is over 40,000 lines long. It also includes one other file in the same directory, `ppic32mx.h`, which is over 1000 lines long. With these we have reached the bottom of our include chain. Let's pop out of this big directory tree we are sitting in and look at those two files in a little more detail.

3.5.2 The Include Files `p32mx795f512l.h` and `ppic32mx.h`

The first 30% of `p32mx795f512l.h`, about 14,000 lines, consists of code like this:

```
extern volatile unsigned int        TRISA __attribute__((section("sfrs")));
typedef union {
  struct {
    unsigned TRISA0:1;   // TRISA0 is bit 0 (1 bit long), interpreted as an unsigned int
    unsigned TRISA1:1;   // bits are in order, so the next bit, bit 1, is called TRISA1
    unsigned TRISA2:1;   // ...
    unsigned TRISA3:1;
    unsigned TRISA4:1;
    unsigned TRISA5:1;
    unsigned TRISA6:1;
    unsigned TRISA7:1;
    unsigned :1;         // don't give a name to bit 8; it's unimplemented
    unsigned TRISA9:1;   // bit 9 is called TRISA9
    unsigned TRISA10:1;
    unsigned :3;         // skip 3 bits, 11-13
    unsigned TRISA14:1;
    unsigned TRISA15:1;  // later bits are not given names
  };
  struct {
    unsigned w:32;       // w is a field referring to all 32 bits; the 16 above, and 16 more
  };
} __TRISAbits_t;
extern volatile __TRISAbits_t TRISAbits __asm__ ("TRISA") __attribute__((section("sfrs")));
extern volatile unsigned int        TRISACLR __attribute__((section("sfrs")));
extern volatile unsigned int        TRISASET __attribute__((section("sfrs")));
extern volatile unsigned int        TRISAINV __attribute__((section("sfrs")));
```

The first line, beginning `extern`, indicates that TRISA is an `unsigned int` variable that has been defined elsewhere; the compiler does not have to allocate space for it.[5] The `processor.o` file is the one that actually

[5]The `volatile` keyword, applied to all the SFRs, means that the value of this variable could change without the CPU knowing it. Thus the CPU should reload it every time it is needed, not assume that its value is unchanged just because the CPU has not changed it.

defines the VA of the symbol TRISA, as mentioned earlier. (The `__attribute__` syntax tells the linker that TRISA is in the `sfrs` section of memory.)

The next section of code defines a data type called `__TRISAbits_t`. The purpose of this is to provide a struct that gives easy access to the bits of the SFR. After defining this type, a variable named TRISAbits is declared of this type. Again, since it is an `extern` variable, no memory is allocated, and, in fact, the `__asm__ ("TRISA")` syntax means that TRISAbits is at the same VA as TRISA. The definition of the *bit field* TRISAbits allows us to use TRISAbits.TRISA0 to refer to bit 0 of TRISA. In general, fields do not have to be one bit long; for example, TRISA.w is the `unsigned int` created from all 32 bits, and 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.

After the declaration of TRISA and TRISAbits, we see declarations of TRISACLR, TRISASET, and TRISAINV. The presence of these declarations in this included header file 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 VAs.

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

```
TRISA = 0xFFCF;
LATAINV = 0x0030;
while(!PORTCbits.RC13)
```

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 memory 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 17,500 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 Timer section of the Reference Manual, 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—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
                  // AND MASKing clears all bits, except bits 4 and 5, which are unchanged
```

Another example usage is in `pic32mx/include/peripheral/timer.h`, where we find the constant definition

```
#define T1_PS_1_64    (2 << _T1CON_TCKPS_POSITION)    /* 1:64 */
```

T1_PS_1_64 is set to the value of 2, or binary 0b10, left-shifted by _T1CON_TCKPS_POSITION positions, yielding 0b100000. If this is bitwise OR'ed with other constants, you can specify the properties of Timer 1 using code that is readable without consulting the Reference Manual. For example, you could use the statement

```
T1CON = T1_ON | T1_PS_1_64 | T1_SOURCE_INT;
```

to turn the timer on, set the prescaler to 1:64, and set the source of the timer to be the internal PBCLK. Of course you have to read the file `timer.h` to know what the available constants are! You might find it easier to consult the Reference Manual and assign the bit values based on the information there.

The definitions of the `POSITTION`, `LENGTH`, and `MASK` constants take up most of the rest of the file. 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 PIC32's MIPS 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).

Finally, `p32mx795f512l.h` concludes by including `ppic32mx.h`, which defines a number of other constants, again with the intent to help you write more readable code. These constants are common for all PIC32 types, unlike those defined in `p32mx795f512l.h`.

3.5.3 The `NU32bootloaded.ld` Linker Script

Examining the `NU32bootloaded.ld` linker script with a text editor, we see the following three lines near the beginning:

```
INPUT("processor.o")
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512L.a")
```

The first line tells the linker to load the `processor.o` file specific to your PIC32. This allows the linker to resolve references to SFRs to actual addresses. The next two lines tell the linker to give our code access to the `.o` object codes for the PIC32 peripheral library.

The rest of the `NU32bootloaded.ld` linker script has information such as the amount 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
  /* 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 we link to.

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 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 0x9D001970. 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`.

3.6 Summarizing the Build

Section 3.5 was a long one, so let's summarize by looking at what happens when you press "Build" on the IDE. If you look at your IDE's Output window after the build, you will see the actual command line commands that were executed. They look something like this:

```
xc32-gcc -g -x c -c -mprocessor=32MX795F512L -MMD -MF build/default/production/simplePIC.o.d
  -o build/default/production/simplePIC.o simplePIC.c

xc32-gcc -mprocessor=32MX795F512L -o dist/default/production/simplePIC.X.production.elf
  build/default/production/simplePIC.o -Wl,--defsym=__MPLAB_BUILD=1,--script="NU32bootloaded.ld"

xc32-bin2hex dist/default/production/simplePIC.X.production.elf
```

The first line compiles and assembles the `simplePIC.c` program, the second links the resulting `.o` file to make a `.elf` file, and the third line converts the `.elf` file to a `.hex` file.[6]

Using the XC32 compiler guide, in the `docs` directory of our XC32 distribution, we see that the `-g` flag in the first line tells the compiler to produce debugging information, which is useful for examining the assembly code compiled from your source code; the `-x c` sequence indicates that the input file is C source code; the `-c` flag tells the compiler to compile and assemble, but not to link, and instead produce a `.o` object file; `-mprocessor=32MX795F512L` indicates the device so that the proper device-specific files are used in compilation; `-MMD -MF` indicate that a file named `simplePIC.o.d` should be created indicating which user-created `.h` files the `.c` file depends on; and the `-o` flag indicates the name of the output file. The last argument, `simplePIC.c`, is the name of the file to be compiled.

In the second line, the linker guide tells us that `-mprocessor=32MX795F512L` specifies the device type (e.g., which `processor.o` file to load); `-o` indicates the name of the output file; and `-Wl,--defsym=__MPLAB_BUILD=1,--script="NU32bootloaded.ld"` tells the linker to put the symbol `__MPLAB_BUILD` at address 1 and to use the linker script `NU32bootloaded.ld`.

Note that in the first line, `xc32-gcc` compiles and assembles, based on the specified options, while in the second line, `xc32-gcc` links because the input file is object code.

The purpose of mentioning the command lines above is to point out that nothing magical is going on in the build process. MPLAB X is simply invoking the three command line commands. In fact, you could bypass the MPLAB X IDE completely and simply use a text editor to create your `simplePIC.c` program and `xc32-gcc` and `xc32-bin2hex` from the command line to create your `.hex` program.

To summarize, here is what you need to know about the creation of your final `.hex` executable from your `simplePIC.c` source file:

- **IDE setting of the processor type.** By choosing the processor type when we created the project, the IDE knows which `-mprocessor` to specify to the `xc32-gcc` command which preprocesses, compiles, and assembles, and to the `xc32-gcc` command which links to create the executable.

- **Including the Microchip `plib.h` file.** By including `plib.h` at the beginning of your program, we get access to variables for all the SFRs, as well as a number of other constants, macros, and prototypes for functions that Microchip provides. The `simplePIC.c` file, which contains references to these, will now compile and assemble successfully, and these references will be resolved at the linking stage.

- **Linking.** The object code `simplePIC.o` is linked with (a) the `crt0.o` C run-time startup library, which performs functions such as initializing global variables; (b) the `processor.o` object code with the SFR VAs; and (c) code from object code libraries such as `libpic32.a`, `libmchp_peripheral.a`, and `libmchp_peripheral_32MX795F512L.a`. The linker script `NU32bootloaded.ld`, which is specific to the bootloader and the PIC32MX795F512L, provides information to the linker on the allowable absolute virtual addresses for the program instructions and data. The result of the linker is a fully self-contained executable in `.elf` format, which is then converted to `.hex` format by `xc32-bin2hex`. The address of the first instruction in the executable is the same address the bootloader jumps to.

- **Installing the program.** The last step is to use the NU32 bootloader and the host computer's bootloader communication 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.

[6] You can see even more information about the build by specifying the `--verbose` option for the compiler and linker in the IDE. Type `--verbose` under Additional Options at both Run > Set Project Configuration > Customize > xc32-gcc and Run > Set Project Configuration > Customize > xc32-ld. At the command line, put `--verbose` at the end of the compile command and `,--verbose` immediately at the end of the `-Wl` options, with no space in front of the comma.

3.7 Building simplePIC_standalone.c

In the case of the standalone version simplePIC_standalone.c in Code Sample 1.2 in Chapter 1.3.2, the build process is very similar to that of the bootloaded version in Section 3.6, with the following exceptions:

- **Source code differences.** The source code has the following additions compared to the bootloaded version:

  1. **Configuration bits.** A number of lines beginning `#pragma config` define the configuration bits of the Device Configuration Registers (Chapter 2.1.4). These bits define fundamental operating behavior of the PIC32 that should not be changed during execution. Examples include bits that control the conversion of the external oscillator frequency into the SYSCLK and PBCLK. These XC32-specific preprocessor commands will cause the final `.hex` file to contain values to be written to the Device Configuration Registers. The constants used in these `#pragma`s, like `MUL_20`, are defined in the `configuration.data` file. You can learn more about the configuration bits in the Special Features section of the Data Sheet.

     When using a bootloader, the configuration bits are set by the bootloader, so the `#pragma`s are not needed.

  2. **Configuring the cache and flash wait cycles.** The other additions to the source code are the preprocessor command

     ```
     #define SYS_FREQ 80000000          // 80 million Hz
     ```

     and the code statement

     ```
     SYSTEMConfig(SYS_FREQ, SYS_CFG_ALL);
     ```

     The preprocessor command simply defines the constant `SYS_FREQ` for use in `SYSTEMConfig()`. Defining `SYS_FREQ` does not actually affect `SYSCLK`; `SYSCLK` is determined by the configuration bits and the frequency of the external oscillator. We must make sure `SYS_FREQ` is consistent with these.

     The command `SYSTEMConfig()` is defined in `pic32mx/include/peripheral/system.h`. Its purpose is to maximize performance by turning on the prefetch cache module and setting the smallest possible number of flash wait cycles. The number of flash wait cycles is the number of cycles that the CPU must stall while waiting for an instruction to load from flash. Based on the SYSCLK frequency (80 MHz) and the maximum frequency of flash access (30 MHz), the number of flash wait cycles is set to 2. This wait time is set in the CHECON SFR, described in the Prefetch Cache Module chapter of the Reference Manual.

     When using a bootloader, the bootloader configures the cache and flash wait cycles.

- **The linker script.** The standalone version of the program uses the default linker script `elf32pic32mx.x`, not `NU32bootloaded.ld`. Opening `elf32pic32mx.x`, we see the command `INCLUDE procdefs.ld`. If there happens to be a `procdefs.ld` file in the same folder as `simplePIC_standalone.c`, that file will be included in the linker script. Otherwise, the linker will find the file in the directory `pic32mx/lib/proc/MX795F512L/`. This file contains default definitions of the size of RAM and flash memory. Since there is no bootloader, there is no concern of the new executable overwriting a bootloader. Your standalone executable will be installed beginning at the hardware-defined reset address, 0xBFC00000.

  Since the linker script `NU32bootloaded.ld` is no longer needed, we do not have the `--script` option to `-Wl` in the linker command:

  ```
  xc32-gcc -mprocessor=32MX795F512L -o dist/default/production/simplePIC_standalone.X.production.elf
    build/default/production/simplePIC_standalone.o -Wl,--defsym=__MPLAB_BUILD=1
  ```

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 some of them are invoked by the MPLAB X IDE. We have already seen the first two of these utilities, as described in Section 3.6:

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
```

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

```
xc32-objdump -D file.o
xc32-objdump -d -S file.elf                // change -d to -D to see debugging info
xc32-objdump -d -S file.elf > file.disasm   // send output to the file file.disasm
```

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-`. You can search online for more information on these utilities. To learn the options available for a command called `xc32-cmdname`, you can type `xc32-cmdname ?` at the command line.

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 `plib.h` initiates a chain of included files that give our program access to variables, data types, constants, macros, and prototypes of functions that significantly simplify programming. C source files can be found in `pic32-libs/peripheral`, header files in `pic32mx/include`, and compiled libraries in `pic32mx/lib`.

- 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 and Reference Manual.
- All programs are linked with `pic32mx/lib/crt0.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. Standalone executables have their first instruction (of the `crt0` startup code) at this address. For a PIC32 with a bootloader, the `crt0` 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.
- If the PIC32 has a bootloader, 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. If a program is standalone, it must have code to perform these functions.
- 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. A standalone program uses the default `elf32pic32mx.x` linker script, which makes use of a default `procdefs.ld` for the particular processor.
- Building an executable can be performed by a series of command line commands, bypassing the MPLAB X IDE.
- Command line utilities like `xc32-ar`, `xc32-nm`, `xc32-objdump`, and `xc32-readelf` allow us to learn more about our compiled code, outside of the MPLAB X IDE.

3.10 Problems

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. Explain the differences between a standalone PIC32 program and a program that is meant to be loaded with a bootloader. Which commands or functions must appear in a standalone program that are not needed in a program loaded by a bootloader? What differences are there in the linker scripts used by a standalone program and a program loaded by a bootloader?
3. Look at the linker script used with bootloaded programs for the NU32. Where does the bootloader install your program in virtual memory? (Hint: look at the `_RESET_ADDR`.)
4. 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.
5. Let's build `simplePIC.hex` using only command line tools. We will follow the template from your IDE's Output window (summarized, for example, in Section 3.6). Copy `simplePIC.c` and `NU32bootloaded.ld` to a new directory with nothing else in it.

(a) Use `xc32-gcc` to create the object file `simplePIC.o` in the same directory. Do not create a `.d` dependency file. Give the command line command you use to create the object file.

(b) Create the file `simplePIC.elf`, in the same directory, from the `simplePIC.o` file. Do not insert the symbol `__MPLAB_BUILD`. Give the command line command you use to create the `.elf` file.

(c) Give the command you use to convert `simplePIC.elf` to `simplePIC.hex`, also in the same directory.

(d) Load your new `simplePIC.hex` on your PIC32 and confirm that it works as expected.

6. In the previous problem, you showed that you could create a `.hex` file from the command line. But it required you to type in a few different commands. In this problem we will use a simple `makefile` to do the same thing. Create a directory with only `simplePIC.c` and `NU32bootloaded.ld`. In that same directory, create a file called `makefile` with the following contents (you can leave out the comments if you wish):

```
# 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.

# make the hex file from the elf file
simplePIC.hex:  simplePIC.elf
        xc32-bin2hex simplePIC.elf

# make the elf file from the object file (link)
simplePIC.elf:  simplePIC.o
        xc32-gcc -mprocessor=32MX795F512L -o simplePIC.elf simplePIC.o -Wl,--script="NU32bootloaded.ld"

# make the object file from the C source file (compile and assemble)
simplePIC.o:  simplePIC.c
        xc32-gcc -g -x c -c -mprocessor=32MX795F512L -o simplePIC.o simplePIC.c

# "all" is what is made by "make" by default
all:  simplePIC.hex

# "make clean" throws away all previously made files to ensure make from scratch
clean:
        rm -f *.o *.elf *.hex
```

With this `makefile`, you should be able to type the command `make` to "build" the `simplePIC.hex` file; `make clean` to get rid of all the `.o`, `.elf`, and `.hex` files; or `make simplePIC.o`, for example, to just create that target. Verify that this works for you.

If you are interested to learn more about makefiles, you can find a number of online resources.

7. 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.

8. 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.

9. Give the VAs and reset values of the following SFRs. (a) I2C2CON. (b) TRISC.

10. The `processor.o` file linked with your `simplePIC` project is much larger than your final `.hex` file. Explain how that is possible.

11. 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, list four things that the startup code does.

    (b) Copy the library file `pic32mx/libmchp_peripheral_32MX795F512L.a` to a directory where you can experiment. Using the archiver command `xc32-ar`, find the object codes that belong to the library by the command

    ```
    xc32-ar -t libmchp_peripheral_32MX795F512L.a
    ```

    You should see that one of the included object codes is `pcache.o`. Now extract this object file using

    ```
    xc32-ar -x libmchp_peripheral_32MX795F512L.a pcache.o
    ```

    Finally we can disassemble the object code to see the corresponding assembly code using

    ```
    xc32-objdump -D pcache.o
    ```

    The two functions of interest are `CheKseg0CacheOn` and `CheKseg0CacheOff`, which are used to turn on and off the cache. Find the C source file corresponding to the object code under `pic32-libs/peripheral`. For the function `CheKseg0CacheOff`, give the one line of C code in the C source file that corresponds to the two lines of assembly code beginning with `and` (a bitwise and) and `ori` (a bitwise or).

    (c) Using the command `xc32-nm -n processor.o`, give the names and addresses of the five SFRs with the highest addresses.

    (d) Give five files included by `pic32mx/include/plib.h`.

    (e) In `pic32mx/include/peripheral/ports.h`, explain how the macro `mPORTAReadBits()` works and give an example call to this macro that reads bits 0 and 3.

    (f) 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?

12. 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

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

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

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

Let's start with some basics of helper C source files and their header files. Then we'll discuss the NU32 library, a dot matrix LCD character display library, and peripheral libraries.

4.1 `.h` Header Files and `.c` Helper Files

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. As an example, Figure 4.1 illustrates a project consisting of one C source file with a `main` function and two helper C source files without a `main` function. Each of the helper C files has its own header file, and we call a C file and its header file together a library. This project also has one other header file, `general.h`, for general definitions that are not specific to either library or the `main` C file. The arrows indicate that the pointed-to file includes the pointed-from header file.

The project in Figure 4.1 can be built by the following five command line commands, which create three object files (one for each source file), link them together into `project.elf`, then convert to a hex file:

```
xc32-gcc -g -x c -c -mprocessor=32MX795F512L -o main.o main.c
xc32-gcc -g -x c -c -mprocessor=32MX795F512L -o helper1.o helper1.c
xc32-gcc -g -x c -c -mprocessor=32MX795F512L -o helper2.o helper2.c
xc32-gcc -mprocessor=32MX795F512L -o project.elf main.o helper1.o helper2.o -Wl,--script="NU32bootloaded.ld"
xc32-bin2hex project.elf
```

The `-Wl,--script="NU32bootloaded.ld"` option is not used for a standalone project. In the MPLAB X IDE, the three C source files and the three header files should be added to the appropriate sections of your project, and the `NU32bootloaded.ld` linker script should be added to your Linker Files if you are building a bootloaded version.

The build is illustrated in Figure 4.2. Each C file is compiled independently and requires the constants, macros, data types, and function prototypes needed to successfully compile into an object file. During compilation of a single C file to a single object code file, the compiler does not have (nor need) access to

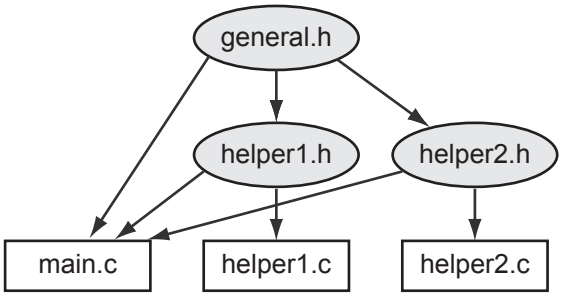


Figure 4.1: An example project consisting of three C files and three header files. Arrows point from header files to files that include them.

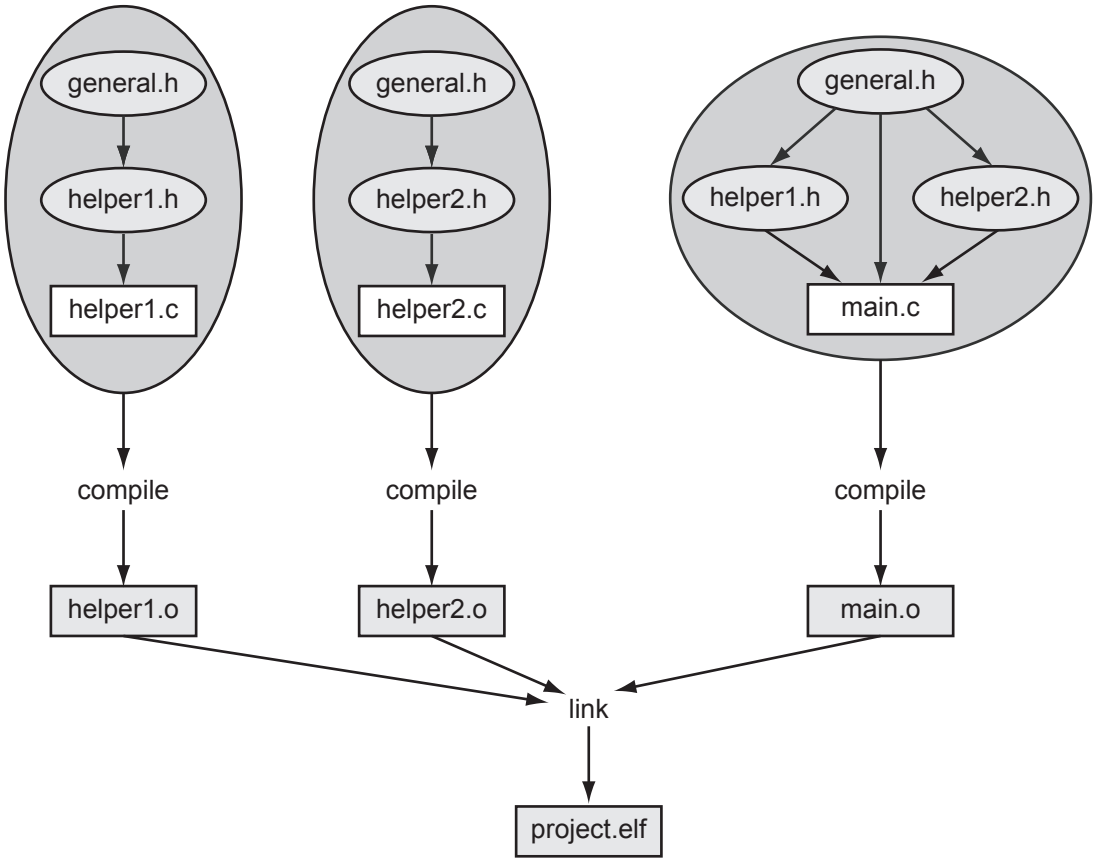


Figure 4.2: The compiling and linking of the project in Figure 4.1.

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 when it creates `main.o`. Calls to the function from `main.o` are linked to the actual function in `helper1.o` at the linker stage.

According to Figures 4.1 and 4.2, `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 4.1; `helper1.h` has a `#include "general.h"` command). Since `general.h` has already been copied in, it should not be copied again; otherwise we will have multiple copies of the same function prototypes, constant definitions, etc. To prevent this, the header file `general.h` should have an **include guard** which prevents the preprocessor from copying it a second time during compilation of a single `.o` file. We will see an example in a moment.

In summary, the `helper1.h` and `helper2.h` header files contain definitions that are made public to source files including them. In other words, we might see the following items in the `helper1.h` header file:

- 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 source 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`

A header file should *not* contain variables, function prototypes, constants, etc., that are meant to be private to the helper C file. If it is private, just give the prototype in the C source file. Also, global variables should not be defined (space allocated) in a header file which could be included by more than one C file in a project. A global should be defined in exactly one C file (e.g., `int Global;`), and then this variable can be made accessible to other files by adding the non-allocating declaration `extern int Global;` to a header file.

As an example, here is a portion of the `pic32mx/include/peripheral/ports.h` header file associated with I/O port functions, with comments added:

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

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

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

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

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

#endif                // end _PORTS_H_ include guard
```

This file demonstrates a typical include guard in the first two lines and last line; the defined constant used in the include guard should be chosen based on the name of the header file. For any file that includes it, `ports.h` provides the header file `xc.h`, provides a new data type `IoPortId`, provides a prototype that allows the use of the function `PORTReadBits`, and provides the macro `mPORTAReadBits()` and the constant `DEBUG_JTAGPORT_ON`.

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 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`, or functions in `helper1.c` that do not have prototypes in `helper1.h`, are private to `helper1.c` and cannot be accessed by other files.

4.2 The NU32 Library

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

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

```
#include <plib.h>

#ifndef __NU32_H
#define __NU32_H

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

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

#endif // NU32_STANDALONE

#define NU32LED1 LATAbits.LATA4
#define NU32LED2 LATAbits.LATA5
#define NU32USER PORTCbits.RC13
#define SYS_FREQ 80000000           // 80 million Hz

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

#endif // __NU32_H
```

The `__NU32_H` include guard, consisting of the first two lines and the last line, ensure that `NU32.h` is not included twice when compiling any single C file. The test `#ifdef NU32_STANDALONE` checks to see if the C file has defined the constant `NU32_STANDALONE`. If so, the device configuration bits are set by the header file; if not, the bootloader has already set them. The next three lines define the mnemonic constants `NU32LED1` and `NU32LED2` for the two LEDs and `NU32USER` for the USER button. With these we can use the C statements

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

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

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

Example use:

```
NU32_Startup();
```

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

Example use:

```
char message[100];
NU32_ReadUART1(message, 100);
```

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

Example use:

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

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

Example use:

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

  char data;

  if (INTGetFlag(INT_SOURCE_UART_RX(UART1))) {  // RX interrupt?
    data = UARTGetDataByte(UART1);              // get the data
    PutCharacter(UART1,data);                   // do something, your choice
    INTClearFlag(INT_SOURCE_UART_RX(UART1));    // clear interrupt flag
  }
  if (INTGetFlag(INT_SOURCE_UART_TX(UART1))) {  // ignore TX interrupts
    INTClearFlag(INT_SOURCE_UART_TX(UART1));
  }
}

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

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

Example use:

```
NU32_DisableUART1Interrupt();
```

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

Example use:

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

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

Example use:

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

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

4.3 Bootloaded and Standalone Programs Throughout the Book

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

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

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

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

While other C files and header files might include `NU32.h` to gain access to its contents and function prototypes, no file except the C file with the `main` function should define `NU32_STANDALONE` or call `NU32_Startup()`.

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

- the constants `NU32LED1`, `NU32LED2`, `NU32USER`, and `SYS_FREQ` are made available
- access is given to the NU32 library commands described above
- the prefetch cache is enabled and the flash wait cycles are set to the minimum (this is redundant for bootloaded programs)
- pins RA4 and RA5 are configured as outputs to control LED1 and LED2
- interrupts are enabled and UART1 and UART3 are set up for communication with the host
- if `NU32_STANDALONE` is defined, the device configuration bits are set in `NU32.h`

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

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

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

4.4 An LCD Library

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

The pinout of the HD44780 is given below:

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

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

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

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

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

which writes the string `str` starting at `row` 1 or 2 and `col` 1 to 16. The header file and C file that make these functions available are given below.

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

```
#ifndef LCD_H
#define LCD_H

// Initialize the LCD
void LCDSetup();

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

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

#endif
```

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

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

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

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

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

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

void LCDClear(int line) {
  switch(line) {
    case 1:                         // clear line 1
        LCDWriteString("                ", 1, 1);
        break;
    case 2:                         // clear line 2
        LCDWriteString("                ", 2, 1);
```

```
        break;
    default:                        // clear both lines
        LCDcommand(0,0,0,0,0,0,0,0,1);
  }
}

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

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

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

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

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

```
Echo!!__________
__Command___1___
```

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

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

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

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

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

4.5 The Peripheral Library

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

Not only does the peripheral library provide functions and macros that would be difficult to write yourself, it also makes your code more readable and more portable to different PIC32s. Many of the library macros and functions perform several SFR setting and reading steps that are always performed together, so using the library functions will minimize mistakes of leaving steps out. Using the library code, particularly building from existing sample code, could reduce the number of times you have to refer to the Data Sheet and Reference Manual. Also, some of the library functions execute assembly instructions for the CPU's CP0 register, which you can't easily learn about in the Reference Manual and Data Sheet.

On the other hand, some of the library macros and functions use verbose syntax to perform rather simple manipulations of the SFRs. Also, the documentation of the peripheral library is spotty; often the best way to learn what the functions do is to dig through the `.c` and `.h` files. A huge number of constants is defined in the peripheral library, and you will have to look these up in the many `.h` files. Plus the software library has evolved over time, and in some cases there are multiple library functions and macros that do the same thing.

Because of the spotty documentation, new PIC32 programmers often learn to program the PIC32 by modifying existing sample code using the library functions. This encourages tweaking the code without really understanding what is going on, making it very difficult to debug.

From a learning standpoint, it is useful to know what is happening at the hardware level with the SFRs, rather than having it hidden by some function that you don't really understand. The Data Sheet and the Reference Manual are often much clearer in their descriptions of how the PIC32 and its peripherals work than the peripheral library documentation. Therefore, for many PIC32 peripherals, you may find it easier to simply read the Data Sheet and Reference Manual and read or write the SFRs directly. Of course you should always comment your code, so you and others don't have to consult the Reference Manual to understand what you're doing!

In these chapters, you will see both direct SFR manipulation and peripheral library functions used in the sample code, often in the same snippet. The bias is toward direct SFR manipulation using information from the Data Sheet and Reference Manual, for learning purposes. There is no way to avoid consulting these references, so we might as well embrace it!

Regardless of the extent to which we use the library functions, all of our programs will have

```
#include <plib.h>
```

somewhere in the include chain, as this gives us access to SFR definitions and other functions we need.

4.6 Navigating the Peripheral Library

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

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

4.7 Chapter Summary

- A library is often considered a `.a` archive of `.o` object codes and an associated `.h` header file that gives user programs access to function prototypes, constants, macros, data types, and variables associated with the object codes. In this chapter we call a `.c` helper file and an associated `.h` file a library.

- For a helper library, the `helper.h` header file can be included by both `helper.c` as well as the `main` C file which uses the helper library. The header file `helper.h` should contain function prototypes, constants, etc., that are meant to be public. Function prototypes and variables that are meant to be private to `helper.c` should be defined in `helper.c`, not `helper.h`.

- For a project with multiple C files, each C file is compiled independently with the aid of its included header files. Compiling a C file does not require the actual definitions of helper functions in other helper C files; only the prototypes, provided by the header files, are needed. The function calls are resolved to the proper virtual address when the multiple object codes are linked. If more than one object code has a `main()` function, the linker will fail.

- The NU32 library provides functions for initializing the PIC32 and communicating with the host computer. The LCD library provides functions to write to a 16×2 character dot matrix LCD screen.

- The Microchip "peripheral library" consists of a large set of header files and object code to be used with your programs. Some peripheral library functions and macros, such as those that execute several steps or manipulate the CPU's CP0 register, are particularly useful. Other functions and macros are less useful; it may be easier to set the values of SFRs based on reading the Data Sheet and Reference Manual than to find the equivalent peripheral library functions.

4.8 Problems

1. Explain what can go wrong if a header file contains the global variable definition `int i=2;` if that header file is included by more than one C file in the same project.

2. Identify which, if any, functions, constants, and global variables in `NU32.c` are private to `NU32.c`.

3. You will create your own libraries.

   (a) Strip out all the comments from `invest.c` in the Appendix. Now modify it to work on the NU32 using the NU32 library. You will need to replace all instances of `printf` and `scanf` with appropriate combinations of `sprintf`, `sscanf`, `NU32_ReadUART1` and `NU32_WriteUART1`. Verify that you can provide data to the PIC32 with your keyboard and display the results on your computer screen. Turn in your code for all the files, with comments where you altered the input and output statements.

   (b) Now break `invest.c` into two C files, `main.c` and `helper.c`, and one header file, `helper.h`. `helper.c` contains all functions other than `main`. Which constants, function prototypes, data type definitions, etc., should go in each file? Build your project and verify that it works. For safety of future users of the `helper` library, make sure to put an include guard in `helper.h`. Turn in your code and a separate paragraph justifying your choice for where to put the various definitions.

   (c) Now break `invest.c` into three files: `main.c`, `io.c`, and `calculate.c`. Any function which deals with input or output should be in `io.c`. Think about which prototypes, data types, etc., are needed for each C file and come up with a good choice of a set of header files and how to include them. Again, for safety, use include guards on your header files. Verify that your code works. Turn in your code and a separate paragraph justifying your choice of header files.

   If you prefer, you are welcome to first solve the tasks using a C installation on your computer, then modify the input/output functions for the NU32.

4. When you try to build and run a program, you could run into (at least) three different kinds of errors: a compiler error, a linker error, or a run-time error. A compiler or linker error would prevent the building of an executable, while a run-time error would only become evident when the program doesn't behave as expected. Say you're building a program with no global variables and two C files, exactly one of which has a `main()` function. For each of the three types of errors, give simple code that would lead to it.

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

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.

5.1 Time and the Disassembly File

5.1.1 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. 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 5.1.3. 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.

5.1.2 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 your program includes `plib.h`, you can use statements such as the following:

```
unsigned int elapsedticks, elapsedns;

WriteCoreTimer(0);              // set the core timer counter to 0; in pic32-libs/peripheral/timer
   ...                          // some code you want to time
elapsedticks = ReadCoreTimer(); // read the core timer
elapsedns = elapsedticks * 25;  // for 80 MHz SYSCLK
```

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 overhead for writing to and reading from the core timer to the bare minimum. Looking up the source for the `WriteCoreTimer()` and `ReadCoreTimer()` functions in `pic32-libs/peripheral/timer/source`, we see that each is implemented by a single assembly command instruction to the CPU.

```
unsigned int elapsed, start=0;

asm volatile("mtc0   %0, $9": "+r"(start));     // WriteCoreTimer(0);
   ...                                          // some code you want to time
asm volatile("mfc0   %0, $9" : "=r"(elapsed));  // elapsed = ReadCoreTimer();
```

The `asm` command constructs a line of assembly code to be directly inserted by the compiler.

In the next section we look more systematically at the assembly code created by our C code. See also Problem 3.

5.1.3 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"          // plib.h, config bits, constants, funcs 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++) {}
}

void toggleLight(void) {
  LATAINV = 0x20;
  // LATAbits.LATA5 = !LATAbits.LATA5;
}
```

After building it in the IDE, go to Window > Output > Disassembly Listing File.[1] You will see a listing showing your C code and, below each C line, the assembly code it generated. Each assembly line has the actual virtual address where the assembly instruction was 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`:

```
23:                   LATAINV = 0x20;
9D00336C  3C02BF88   LUI V0, -16504
9D003370  24030020   ADDIU V1, ZERO, 32
9D003374  AC43602C   SW V1, 24620(V0)
24:                   // LATAbits.LATA5 = !LATAbits.LATA5;
```

We see that the `LATAINV = 0x20` command has expanded to three assembly statements. Without going into details[2], the `ADDIU` stores the value 0x20 (the sum of ZERO and 32 in decimal) into the 32-bit CPU register V1, and the `SW` stores this word into the memory address corresponding to LATAINV.

If instead we comment out the `LATAINV = 0x20` command and replace it with the bit manipulation version, we get the following disassembly:

```
23:                   // LATAINV = 0x20;
24:                   LATAbits.LATA5 = !LATAbits.LATA5;
9D00336C  3C02BF88   LUI V0, -16504
9D003370  8C426020   LW V0, 24608(V0)
9D003374  30420020   ANDI V0, V0, 32
9D003378  2C420001   SLTIU V0, V0, 1
9D00337C  304400FF   ANDI A0, V0, 255
9D003380  3C03BF88   LUI V1, -16504
9D003384  8C626020   LW V0, 24608(V1)
9D003388  7C822944   INS V0, A0, 5, 1
9D00338C  AC626020   SW V0, 24608(V1)
```

[1] At the command line, you can instead use `xc32-objdump -d -S filename.elf > filename.disasm` and inspect the file `filename.disasm`. The results may not be quite as easy to interpret, however.

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

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. With the LATAINV syntax, this is done in hardware.

Although one method of manipulating the SFR bit appears three times slower than the other, we don't yet know how many processor 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 5.1.4 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:

```
17:                 void delay(void) {
9D003310  27BDFFF0  ADDIU SP, SP, -16         // manipulate the stack pointer on ...
9D003314  AFBE000C  SW S8, 12(SP)             // ... entering the function (see text)
9D003318  03A0F021  ADDU S8, SP, ZERO
18:                   int i;
19:                   for (i=0; i<DELAYTIME; i++) {}
9D00331C  AFC00000  SW ZERO, 0(S8)            // initialization of i in RAM to 0
9D003320  0B400CCD  J 0x9D003334              // jump to 9D003334 (skip adding 1 to i)
9D003324  00000000  NOP                       // "no operation," let jump complete
9D003328  8FC20000  LW V0, 0(S8)              // start of the loop; load RAM i into register V0
9D00332C  24420001  ADDIU V0, V0, 1           // add 1 to V0 ...
9D003330  AFC20000  SW V0, 0(S8)              // ... and store it to i in RAM
9D003334  8FC30000  LW V1, 0(S8)              // load i into V1
9D003338  3C0202FA  LUI V0, 762               // these two lines ...
9D00333C  3442F080  ORI V0, V0, -3968         // ... load the constant 50,000,000 into V0
9D003340  0062102A  SLT V0, V1, V0            // store "true" (1) in V0 if V1 < V0
9D003344  1440FFF8  BNE V0, ZERO, 0x9D003328  // if V0 does not equal 0, branch to top of loop
9D003348  00000000  NOP                       // branch delay slot is executed before branch
20:                 }
9D00334C  03C0E821  ADDU SP, S8, ZERO         // manipulate the stack pointer on exiting
9D003350  8FBE000C  LW S8, 12(SP)
9D003354  27BD0010  ADDIU SP, SP, 16
9D003358  03E00008  JR RA                     // jump to return address RA stored by JAL
9D00335C  00000000  NOP
```

There are nine instructions in the delay loop itself, starting with `LW V0, 0(S8)` and ending with the `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 these are 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 50 million $\times$ 9 instructions $\times$ 12.5 ns/instruction = 5.625 seconds. When we timed by a stopwatch, we got about 6.25 seconds, which implies 10 cycles per loop. So our cache module has the CPU executing one assembly instruction almost every cycle.

You might notice a couple of ways you could have written the assembly code for the delay function more time-efficiently. 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! Also, more efficient assembly code can be generated from your C code by employing certain compiler optimizations (Section 5.1.5). (Not to mention that `delay` was designed to waste time, so no need to be efficient!)

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 memory that holds function 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 the stack is too small for the local variables defined in currently-called functions.

The overhead due to 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.

5.1.4 The Prefetch Cache Module

In the previous section, we saw that our bootloaded `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. (See Chapters 3.7 and 4.2.)

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: it keeps recent instructions in the cache, ready if the CPU requests the instruction at that address (allowing the cache to completely store small loops); and 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

```
CHECONCLR = 0x30;    // SFR in prefetch cache section of Reference Manual
CheKseg0CacheOff();  // defined in pic32-libs/peripheral/pcache/source/pcache.c
```

right after `NU32_Startup()` in `main`. Everything else stays the same. Consulting the section on the prefetch cache module in the Reference Manual, we see that bits 4 and 5 of the SFR CHECON determine whether instructions are prefetched, and that clearing both bits disables predictive prefetch. The second line is a library function that uses MIPS32 assembly instructions to turn off the cache.

Rerunning `timing.c` with these two commands, we find that the LED stays on for approximately 17 seconds, compared to approximately 6.25 seconds before. We are seeing the effect of the flash wait cycles—the CPU has to wait two cycles before receiving requested instructions from flash.

If we comment out the first line, so that the prefetch is enabled but the cache is off, and rerun, we find that the LED stays on for about 7.5 seconds, or 12 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 second line but leave the first line, so that the prefetch is disabled but the cache is on, we recover our original performance of approximately 6.25 seconds. The reason is that the entire loop can be stored in the cache, so prefetch is not necessary.

5.1.5 Optimization

Compilers can sometimes recognize ways to increase the speed of execution (or decrease program size) by trimming redundant code, eliminating code that doesn't do anything, and many other ways.[3] For example, the function `delay` in `timing.c` accomplishes nothing but wasting time. In our case, that was the desired effect. If we chose a compiler option to optimize running time, however, we shouldn't be surprised if that code were to be optimized away. Depending on the compiler, something like this could happen even if we didn't choose to optimize for execution time. If you want to implement a delay with predictable behavior, consider using the core timer.

5.1.6 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 Problems, but here are a few rules of thumb for efficient math:

- There is no floating point unit on the PIC32, 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

[3]See the Command Line chapter of the MPLAB XC32/XC32++ Compiler User's Guide in your `docs` directory if you are interested. Certain optimizations are not available in the free version of the XC32 compiler.

math, scaling the variables as necessary to maintain precision, and only convert back 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.

5.2 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`. The stack holds 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 easiest way to keep track of the amount of flash and global variable RAM used by your program is to look at Window > Dashboard after a build. If your program attempts to put too many local variables on the stack (stack overflow), however, 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 specific 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. For an executable created from two object code files `file1.o` and `file2.o`, we can create a map file at the command line (Section 3.6) with a linker command of the form

```
xc32-gcc -mprocessor=32MX795F512L -o proj.elf file1.o file2.o -Wl,--script="NU32bootloaded.ld",-Map="proj.map"
```

or in the MPLAB X IDE under Run > Set Project Configuration > Customize > xc32-ld > Diagnostics, where you set the name of the map file. The map file can be opened with a text editor.

Let's create a map file `timing.map` for `timing.c` as shown in Code Sample 5.1. 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                    0x9d001e08          0xacc        2764  App's exec code
.rodata                  0x9d0028d4          0x7d4        2004  Read-only const
.text                    0x9d0030a8          0x168         360  App's exec code
.text.general_exception  0x9d003210           0xd0         208
.text                    0x9d0032e0           0xac         172  App's exec code

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

.text                    0x9d003630           0x38          56  App's exec code

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

.text.UARTSetLineContro 0x9d003864           0x28          40
```

```
.rodata                 0x9d00388c           0x1c          28  Read-only const
.text.INTRestoreInterru 0x9d0038a8           0x1c          28
.text.CheKseg0CacheOff  0x9d0038c4           0x18          24
.text.CheKseg0CacheOn   0x9d0038dc           0x18          24
.rodata                 0x9d0038f4           0x18          24  Read-only const
.text                   0x9d00390c           0x18          24  App's exec code
.dinit                  0x9d003924           0x10          16
.text._bootstrap_except 0x9d003934            0xc          12
.text._general_exceptio 0x9d003940            0xc          12
.text.INTDisableInterru 0x9d00394c            0x8           8
.text.INTEnableInterrup 0x9d003954            0x8           8
.text._on_reset         0x9d00395c            0x8           8
.text._on_bootstrap     0x9d003964            0x8           8
      Total kseg0_program_mem used  :      0x1b64        7012  1.4% 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 exception_mem report ...]]]

.vector_63              0x9d0009e0            0x8           8  Interrupt Vector 63
          Total exception_mem used  :       0x210         528  12.9% of 0x1000

kseg1 Boot-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  ------------------------  -----------
.reset                  0xbd001970          0x1f4         500  Reset handler
.bev_excpt              0xbd001cf0           0x10          16  BEV-Exception
         Total kseg1_boot_mem used  :       0x204         516  44.2% of 0x490
        ------------------------------------------------------------------------
         Total Program Memory used  :      0x1f78        8056  1.5% of 0x80000
        ------------------------------------------------------------------------
```

The kseg0 program memory usage report tells us that 7012 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 2764 bytes, described as `App's exec code`, and installed starting at VA 0x9D001E08. 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 `.rodata`, standing for "read-only data." An example of read-only data is the string on the right-hand side of the following initialized `char` array:

```
char str[] = "my initialized string";
```

Searching the map file for the memory address 0x9D0028D4, we see that the read-only data in question is information about the interrupt table (Chapter 6).

Continuing down, we see other `.text` program instruction sections, including ones named for the specific C functions that were compiled to make the object code in that section. Some of these are functions that are

called by `NU32_Startup()` in `NU32.c`; others are functions added automatically to the program by `crt0.o`. The `.text` section that is 172 bytes long corresponds to `timing.o`, and searching the map file for the address 0x9D0032E0 reveals

```
.text           0x9d0032e0       0xac
 .text          0x9d0032e0       0xac build/default/production/timing.o
                0x9d0032e0                main
                0x9d003310                delay
                0x9d003360                toggleLight
```

our functions `main`, `delay`, and `toggleLight` of `timing.o` are stored consecutively in memory. The addresses agree with our disassembly file from Section 5.1.3.

Continuing, the kseg0 boot memory report indicates that no code is placed in this memory region. The exception memory report indicates that instructions corresponding to interrupts occupy 528 bytes. Finally, the kseg1 boot memory report indicates that `crt0.o` installs reset functions that occupy 516 bytes. The `.reset` section is the first section that the bootloader jumps to.

In all, 8056 bytes of program memory are used.

Continuing further in the map file, we see

```
kseg1 Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
.bss                     0xa0000000            0x20          32  Uninitialized data
         Total kseg1_data_mem used  :        0x20          32  0.0% of 0x20000
        ------------------------------------------------------------------------
            Total Data Memory used  :        0x20          32  0.0% of 0x20000
        ------------------------------------------------------------------------

Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  --------------------------  -----------
heap                     0xa0000028               0           0  Reserved for heap
stack                    0xa0000040         0x1ffb8      131000  Reserved for stack
```

The heap size is zero and almost all data memory is reserved for the stack. Only 32 bytes of kseg1 data memory are set aside for global variables, beginning at the origin of data memory, 0xA0000000. This section is called `.bss`, which is for uninitialized data. These 32 bytes are the 32 bytes reserved by `NU32.c` in the statement

```
char NU32_RS232OutBuffer[32];
```

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            0x84         132  Uninitialized data
.data                    0xa0000098            0x34          52  Initialized data
         Total kseg1_data_mem used  :        0xca         202  0.2% of 0x20000
```

```
-------------------------------------------------------------------------
    Total Data Memory used  :        0xca         202  0.2% of 0x20000
-------------------------------------------------------------------------
```

Our global variables now occupy 202 bytes of data memory. 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 build/default/production/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 must be 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.

Startup code initializes all global variables with their initial values and fills all uninitialized global variables with zeros.

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) section of the kseg0 program memory report has grown from 16 bytes to 116, meaning that our total program memory used is now 8156 bytes compared to 8056 before.

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            0x84         132  Uninitialized data
.data                    0xa0000090            0x34          52  Initialized data
         Total kseg1_data_mem used  :        0xc2         194  0.1% of 0x20000
        -------------------------------------------------------------------------
            Total Data Memory used  :        0xc2         194  0.1% of 0x20000
        -------------------------------------------------------------------------
```

Now looking at the program memory report

```
kseg0 Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                  ----------  ------------------------  -----------
.text                    0x9d001e08          0xacc        2764  App's exec code
.rodata                  0x9d0028d4          0x7d4        2004  Read-only const
.text                    0x9d0030a8          0x168         360  App's exec code
.text.general_exception 0x9d003210           0xd0         208
```

```
.text                   0x9d0032e0          0xc4         196  App's exec code

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

      Total kseg0_program_mem used  :      0x1be0        7136  1.4% of 0x7e200
```

we see that our `timing.o` code is now 196 bytes as compared to 172 before. This is because the assignment of `my_cat_string` is now taken care of by assembly commands in our code, not by the global variable initialization in `.dinit`. Correspondingly, the global data initialization section `.dinit` shrinks from 116 bytes to 108 bytes.

Finally, we might wish to reserve some RAM for dynamic memory allocation using `malloc` or `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. By default, MPLAB X assumes you will not use dynamic memory allocation and sets the heap size to zero. To set a nonzero heap size, go to Run > Set Project Configuration > Customize > xc32-ld > General. If we choose 4 KB, or 4096 bytes, the map file after building shows

```
Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                  ----------  -------------------------  -----------
heap                      0xa00000c8         0x1000        4096  Reserved for heap
stack                     0xa00010e0        0x1ef10      126736  Reserved for stack
```

A heap can also be allocated at the command line (Section 3.6) using a linker command of the form

```
xc32-gcc -mprocessor=32MX795F512L -o proj.elf file1.o file2.o
        -Wl,--script="NU32bootloaded.ld",-Map="proj.map",--defsym=_min_heap_size=4096
```

The heap is allocated just after the global variables, starting in this case at address 0xA00000C8. The stack grows "down" from the end of RAM—as local variables are added to the stack, the stack pointer address decreases, and when local variables are discarded after exiting a function, the stack pointer address increases.

5.3 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 `WriteCoreTimer(0);` and `unsigned int dt = ReadCoreTimer();` can be used to measure the execution time of the code in between to within a few SYSCLK cycles.
- Use Window > Output > Disassembly Listing File to see how your C code is compiled to assembly code.
- 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 local variables. The heap is zero bytes by default.
- A summary of program flash and global variable data RAM usage can be found at Window > Dashboard after a build.
- A map file provides a more detailed summary of memory usage. A map file can be created during the build by choosing a map file name in the MPLAB X IDE under Run > Set Project Configuration > Customize > xc32-ld > Diagnostics.

- 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 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.

5.4 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 a stopwatch, verify the time taken by the delay loop. Do your results agree with Section 5.1.3?
3. When you look at Window > Output > Disassembly Listing File, it shows you the disassembly of your code, not Microchip object code that you may have linked with. You can try the command-line command

   ```
   xc32-objdump -d - S filename.elf > filename.disasm
   ```

   to create a disassembly listing `filename.disasm` of your entire executable. (The listing will look somewhat different than what you see in MPLAB X.) Let's do this for two different versions of some timing code.

   (a) Write a short program that uses `WriteCoreTimer(0)` and `elapsed = 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 `WriteCoreTimer` command? How many cycles does it take to complete the `ReadCoreTimer` command? Approximately how much error will you have in your estimate of the timed code? (It's certainly not a sum of the two.)

   (b) Now replace the `WriteCoreTimer(0)` and `elapsed = ReadCoreTimer()` with their assembly equivalents, as in Section 5.1.2. 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 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 makes 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 usually 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 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 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 (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.

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." If interrupts are piling up (one ISR interrupts another which has interrupted another, etc., before any of them finish), then the PIC32 could potentially run out of RAM, causing a stack overflow and the program to crash. These errors can be very difficult to debug, so it is a good idea to ensure that your ISRs execute quickly to prevent pile-up.

6.2 Details

The address of an ISR in virtual memory is determined by the *interrupt vector* associated with the IRQ. The PIC32 supports up to 64 unique interrupt vectors (and therefore 64 ISRs). For `timing.c` in Chapter 5.2, the virtual addresses of the interrupt vectors can be seen in the exception memory region of the map file (an interrupt is also known as an "exception"):

```
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 exception_mem report ...]]]

.vector_63               0x9d0009e0             0x8           8  Interrupt Vector 63
```

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 the mode set after executing `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 IPC0$\langle 17{:}16 \rangle$ are the two subpriority bits.

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

Interrupt Source(1)	IRQ	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	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
SPI3E – SPI3 Fault						
I2C3B – I2C3 Bus Collision Event						
U1RX – UART1 Receiver	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>
SPI3RX – SPI3 Receive Done						
I2C3S – I2C3 Slave Event						
U1TX – UART1 Transmitter	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
SPI3TX – SPI3 Transfer Done						
I2C3M – I2C3 Master Event						
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>
AD1 – ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

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

Interrupt Source(1)	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
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>
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>

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

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

Interrupt Source(1)	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
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, Table 2 and Table 3 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. (In this table, however, lower IRQ numbers 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 PIC32'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 will see examples of this in Section 6.5. Obviously we cannot expect 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 Reference Manual. 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 8-1: 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 P = Programmable bit
-n = Bit Value at POR:('0', '1', x = 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 8-4: IFSx: Interrupt Flag Status Register(1)

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	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = 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 8-5: IECx: Interrupt Enable Control Register(1)

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	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = 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 8-6: IPCx: Interrupt Priority Control Register(1)

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	P = Programmable bit
-n = Bit Value at POR:('0', '1', x = 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

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 6.5.

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`, which is only available for priority level 7, makes use of the shadow register set. 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. Configure the CPU to receive interrupts in multi-vector mode and enable the CPU to process interrupts.

Steps 2 and 7 involve assembly commands to the CPU, so we use peripheral library functions to implement them. These steps are often unnecessary. Steps 4–6 involve manipulation of SFRs, which can be accomplished directly or by using peripheral library functions.

6.4 Library Functions

Libary functions can be found in `pic32-libs/peripheral/int/source`, particularly in `int_tbl_lib.c`, `int_enable_mv_int_lib.c`, and `int_disable_interrupts_lib.c`. Constants and function prototypes can be found in `pic32mx/include/peripheral/int.h`, and mnemonic constants for IRQ sources and interrupt

vectors can be found in `p32mx795f512l.h`. Examples include the IRQ `_CHANGE_NOTICE_IRQ` (defined as 32) and `_EXTERNAL_2_VECTOR`, for the interrupt input INT2, defined as 11. These agree with the table given earlier.

Below are some sample functions. The first two are quite useful for their assembly language commands. The next four are of lesser importance, since they just manipulate the SFRs according to the Reference Manual. But they act like self-commented code, and they make it less likely that you make a mistake manipulating the SFR bits yourself.

INTEnableSystemMultiVectoredInt() This command does three things: it sets INTCON⟨12⟩ to choose multi-vector interrupts, sets the IV bit in the CPU's CP0 Cause register to use the interrupt vector table (not the general exception vector), and sets the IE bit in the CPU's CP0 Status register to cause the CPU to start processing interrupts.

INTDisableInterrupts() Disables interrupts at the CPU. You can ignore the return value.

INTSetVectorPriority(vector, priority) An example `vector` is `_EXTERNAL_2_VECTOR` (or 11) and an example `priority` is `INT_PRIORITY_LEVEL_2` (or simply 2).

INTSetVectorSubPriority(vector, subpriority) An example `subpriority` is `INT_SUB_PRIORITY_LEVEL_3` (or simply 3).

INTClearFlag(IRQ) Clears the interrupt flag for `IRQ`. An example of `IRQ` is `_CHANGE_NOTICE_IRQ`, or, equivalently, 32.

INTEnable(IRQ, value) `value` could be `INT_DISABLED` or `INT_ENABLED`, or, equivalently, 0 or 1, respectively.

6.5 Sample Code

6.5.1 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"          // plib.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
  INTClearFlag(_CORE_TIMER_IRQ);                     // clear the interrupt flag
  LATAINV = 0x20;                                    // invert pin RA5 only
  WriteCoreTimer(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

  INTDisableInterrupts();                            // step 2: disable interrupts at CPU
  _CP0_SET_COMPARE(CORE_TICKS);                      // step 3: CP0_COMPARE register set to 40 M
```

```
  INTSetVectorPriority(_CORE_TIMER_VECTOR,7);        // step 4: interrupt priority
  INTSetVectorSubPriority(_CORE_TIMER_VECTOR,0);     // step 4: subp is 0, which is the default
  INTClearFlag(_CORE_TIMER_IRQ);                     // step 5: clear interrupt flag
  INTEnable(_CORE_TIMER_IRQ,INT_ENABLED);            // step 6: enable core timer interrupt
  INTEnableSystemMultiVectoredInt();                 // step 7: CPU enabled for mvec interrupts

  WriteCoreTimer(0);                                 // set core timer counter to 0
  while(1);                                          // infinite loop
}
```

Following our seven steps in using an interrupt, we have:

Step 1. The ISR.

```
void __ISR(_CORE_TIMER_VECTOR, IPL7SRS) CoreTimerISR(void) {  // step 1: the ISR
  INTClearFlag(_CORE_TIMER_IRQ);                     // clear the interrupt flag
  ...
}
```

We are allowed to call our ISR whatever we want, and in this example we call it `CoreTimerISR`. The `__ISR` syntax is Microchip-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), 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, and `IPLnSOFT` otherwise. You don't specify subpriority in the ISR.

The ISR must clear the interrupt flag, `_CORE_TIMER_IRQ`, defined in `p32mx795f512l.h` as 0 to agree with the table.

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

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

Disabling interrupts is for safety in general practice, but in most cases it is not needed.

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.

```
  INTSetVectorPriority(_CORE_TIMER_VECTOR,7);        // step 4: interrupt priority
  INTSetVectorSubPriority(_CORE_TIMER_VECTOR,0);     // step 4: subp is 0, which is the default
```

These two commands set the appropriate bits in IPCy (y=0 here). The priority should agree with the ISR priority. The second command is not strictly necessary, since zero is the default priority and subpriority. We could have used the constants `INT_PRIORITY_LEVEL_7` and `INT_SUB_PRIORITY_LEVEL_0` instead of 7 and 0, but we preferred to save space!

Step 5. Clearing the interrupt flag status bit.

```
  INTClearFlag(_CORE_TIMER_IRQ);                    // step 5: clear interrupt flag
```

This command clears the appropriate bit in IFSx (x=0 here).

Step 6. Enabling the core timer interrupt.

```
  INTEnable(_CORE_TIMER_IRQ,INT_ENABLED);           // step 6: enable core timer interrupt
```

This command sets the appropriate bit in IECx (x=0 here).

Step 7. Configure for multi-vector interrupts and tell the CPU to accept interrupt requests.

```
  INTEnableSystemMultiVectoredInt();                // step 7: CPU enabled for mvec interrupts
```

Sets the CPU to process multi-vectored interrupts.

In this example we used a number of peripheral library functions. In the next one we directly manipulate SFRs where possible.

6.5.2 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 sub-priority 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 C13/USER pin to the D0/INT0 pin. When the USER button is pressed, there is a falling edge on digital input C13 (see the wiring diagram in Figure 2.4) and therefore INT0, which causes the LEDs to flash. You might also notice the issue of switch *bounce*: when you release the button, nominally creating a rising edge, you might see the LEDs flash again. This is because there may be a brief period of chattering when mechanical contact between two conductors is established or broken, creating spurious rising and falling edges. This can be addressed by a *debouncing* circuit or software; see Problem 16.

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"          // plib.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
  WriteCoreTimer(0);
  while(ReadCoreTimer()<10000000);   // delay for 10 M core ticks, 0.25 s
  LATASET = 0x30;                    // set pins RA4 and RA5 high
  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
  INTDisableInterrupts();            // step 2: disable interrupts at the CPU
  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 interrupt flag
  IEC0SET = 1 << 3;                  // step 6: enable INT0 by setting IEC0<3>
  INTEnableSystemMultiVectoredInt(); // step 7: CPU enabled for mvec interrupts
  while(1);                          // infinite loop
}
```

6.5.3 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 `IPL2SOFT` 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 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):

```
IPL2SOFT in  33 out  40 total  72 time 1800 ns
 IPL7SRS in  22 out  29 total  45 time 1125 ns
```

For context save and restore, it takes 33 core clock ticks (about 66 SYSCLK ticks) to begin executing statements in the ISR; the last ISR statement completes about 7 (14) ticks later; and finally control is returned to `main` approximately 72 (144) total ticks, or 1.8 microseconds, after the interrupt flag is set. For the SRS, the first ISR statement is executed after about 22 (44) ticks; 7 (14) ticks are needed to complete the last ISR statement; and a total of approximately 45 (90) ticks, or 1.125 microseconds, 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, a few things can be noted:

- The ISR is not entered immediately after the flag is set. Instructions in `main` may be executed after the flag is set.
- The SRS saves time in entering and exiting the ISR, approximately 0.7 microseconds total in this case.
- Simple ISRs can be completed in just over 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"          // plib.h, config bits, constants, funcs for startup and UART
#define DELAYTIME 40000000 // 40 million core clocks, or 1 second

void delay();

int Entered,Exited;

void __ISR(_EXTERNAL_0_VECTOR, IPL2SOFT) Ext0ISR(void) {
  Entered = ReadCoreTimer();  // record time ISR begins
  IFS0CLR = 1 << 3;           // clear the interrupt flag
  NU32LED2 = 1;               // turn off LED2
  Exited = ReadCoreTimer();   // record time ISR ends
}

void __ISR(_EXTERNAL_1_VECTOR, IPL7SRS) Ext1ISR(void) {
  Entered = ReadCoreTimer();
  IFS0CLR = 1 << 7;
  NU32LED2 = 0;               // turn on LED2
  Exited = ReadCoreTimer();
}
```

```
int main(void) {
  int dt;
  char msg[128];

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  INTDisableInterrupts();  // 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 |= 10 << 24;        // step 4: set INT0 to priority 2 subpriority 2
  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
  INTEnableSystemMultiVectoredInt();  // step 7: CPU enabled for mvec interrupts
  while(1) {
    delay();               // delay, so results sent back at reasonable rate
    WriteCoreTimer(0);     // start timing
    IFS0bits.INT0IF = 1;   // artificially set the INT0 interrupt flag
    while(IFS0bits.INT0IF);// wait until the interrupt clears the flag
    dt = ReadCoreTimer();  // get elapsed time
    sprintf(msg,"IPL2SOFT in %3d out %3d total %3d time %4d ns\r\n",Entered,Exited,dt,dt*25);
    NU32_WriteUART1(msg);  // send times to the host

    delay();               // same as above, except for INT1
    WriteCoreTimer(0);
    IFS0bits.INT1IF = 1;   // trigger INT1 interrupt
    while(IFS0bits.INT1IF);
    dt = ReadCoreTimer();
    sprintf(msg," IPL7SRS in %3d out %3d total %3d time %4d ns\r\n",Entered,Exited,dt,dt*25);
    NU32_WriteUART1(msg);
  }
}

void delay() {
  WriteCoreTimer(0);
  while(ReadCoreTimer()<DELAYTIME);
}
```

6.6 Chapter Summary

- The PIC32 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 `IPL7SRS`. `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.

- 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.

6.7 Problems

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 sitting and 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 a priority level 4, subpriority level 2 vector 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 interrupt enable, flag status, and priority and subpriority 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 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. 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.

10. 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.

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

12. 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.

13. 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.

14. 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.

15. Build `INT_timing.c` in Code Sample 6.3. Now disassemble it at the command line using the command

    ```
    xc32-objdump -d -S filename.elf > disasm.txt
    ```

    and open `disasm.txt` with a text editor. (Do not use the IDE's disassembly window, as it does not show you the disassembly of all the code in your `.elf` file.) 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` runs from 0 to 63. 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) The bootstrap exception section ends by a jump to an address set by the previous two commands, which set the upper two bytes and the lower two bytes of the address, respectively. (Note that the last four hex characters of these two eight-character assembly commands are the first four hex characters and last four hex characters of the eight-character address.) Search for this address in the file to find where the bootstrap exception vector jumps to. What happens to the PIC32 after this jump?

    (b) Most of the `.vector_x` sections jump to the same address. Find this address in the file. What happens to the PIC32 if it jumps to this address? (The assembly command `di` disables interrupts.)

    (c) Which `.vector_x` sections jump to addresses different from the one above? What are these addresses, and what is installed there?

    (d) Find the `Ext0ISR` and `Ext1ISR` functions. How many assembly commands are before the first `ReadCoreTimer()` command in each function? How many assembly commands are after the last `ReadCoreTimer()` command in each function? What is the purpose of the commands that account for the majority of the difference in the number of commands? (You might find it helpful to do a web search for these assembly commands if you are uncertain of their purpose.) Explain why the two functions are different even though their C code is essentially identical.

(e) Modify the `INT_timing.c` code by commenting out the ISR for INT1. Build the project and disassemble the resulting `.elf` file using the `xc32-objdump` command. Look at the interrupt vector sections. According to the disassembly, what happens to the PIC32 when an INT1 interrupt is generated? Now load and run the `.hex` file. Does the behavior agree with your prediction?

16. 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.

17. Using your solution to debouncing the USER button (Problem 16), write a stopwatch program using an ISR based on INT2. Connect a wire from the C13 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 consider using polling to write out the current time to the user's screen every second so the user can see the running time.

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

19. 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 IRQ priority to 3 and subpriority to 0.

20. (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.

21. The NU32 library, Chapter 4.3, allows the programmer to use interrupt-based receiving of data from the host computer, rather than just sitting and waiting for the user to send a carriage return as with `NU32_ReadUART1()`. By including the command

    ```
    NU32_EnableUART1Interrupt();
    ```

    in your `main` function, you can create a UART1 ISR that receives a character and does something based on it. See the example ISR in Chapter 4.3.

    Write a program using a UART1 ISR to receive a character, add one to the ASCII representation, and send it back to the host screen.

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). 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 will pull the output up to this voltage. 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).

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), 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 (left floating). Otherwise, if an input pin is not connected to anything, we cannot be certain what the input will read.

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 what has changed.

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, 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, 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.

To activate the change notification interrupt, the interrupt flag status IFS1⟨0⟩ (or IFS1bits.CNIF) should be initially cleared, the priority should be placed in IPC6⟨28:26⟩ (or IPC6bits.CNIP), the subpriority should be placed in IPC6⟨25:24⟩ (or IPC6bits.CNIS), and the interrupt enable control bit IEC1⟨0⟩ (or IEC1bits.CNIE) must be set. 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) disable interrupts at the CPU using `INTDisableInterrupts()`; (2) clear the interrupt flag IFS1bits.CNIF, set the priority IPC6bits.CNIP and subpriority IPC6bits.CNIS, enable the appropriate pins using CNEN, and choose any desired pull-up resistors with CNPUE; (3) read all the ports of pins involved in the change notification scan; (4) enable the interrupt by setting IEC1bits.CNIE; and (5) re-enable interrupts at the CPU using `INTEnableSystemMultiVectoredInt()`.

7.3 Library Functions

C functions for the DIO ports can be found in `pic32-libs/peripheral/ports`, and macros and constants can be found in `pic32mx/include/peripheral/ports.h`. Most of these perform simple SFR manipulation. Below are some macros from `ports.h`.

mPORTBSetPinsAnalogIn(bits) Configures TRISB and AD1PCFG so that pins with a 1 in `bits` are analog inputs.

mPORTxSetPinsDigitalOut(bits) Configures TRISx and AD1PCFG so that pins of port x with a 1 in `bits` are digital outputs.

mPORTxSetPinsDigitalIn(bits) Configures TRISx and AD1PCFG so that pins of port x with a 1 in `bits` are digital inputs.

mPORTxOpenDrainOpen(bits) Configures TRISx and ODCx so that pins with a 1 in `bits` are open-drain outputs.

mPORTxOpenDrainClose(bits) Configures TRISx and ODCx so that pins with a 1 in `bits` are digital inputs.

mPORTxREAD() Reads the digital inputs from port x. Equivalent to `PORTx`.

mPORTxWrite(bits) Writes digital outputs to port x. Equivalent to `LATx = bits`.

mCNOpen(config, pins, pullups) Choosing `config = 0` or `CN_ON` simply enables CN interrupts in CNCON. CNEN is set to `pins` and CNPUE is set to `pullups`.

Apart from these, you can use the interrupt setting functions `INTSetVectorPriority(vec, pri)`, `IntSetVectorSubPriority(vec, subpri)`, `INTClearFlag(IRQ)`, and `INTEnable(IRQ, val)` from Chapter 6, using the IRQ name `_CHANGE_NOTICE_IRQ` and the vector name `_CHANGE_NOTICE_VECTOR`.

7.4 Sample Code

In this example, we set 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 following code is written with direct SFR manipulation.

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

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

unsigned oldB, oldF, newB, newF;   // 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
  IFS1CLR = 1;            // clear the interrupt flag
}

void 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 = 0x00B0;       // 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 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

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

  while(1);               // infinite loop
}
```

The following code implements the same functionality, but this time using peripheral library functions.

Code Sample 7.2. `DIO_plib.c`. Digital input, output, and change notification using the peripheral library.

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

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

void __ISR(_CHANGE_NOTICE_VECTOR, IPL3SOFT) CNISR(void) { // INT step 1
  newB = mPORTBRead();    // since pins on port B and F are being monitored ...
  newF = mPORTFRead();    // ... 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
  INTClearFlag(_CHANGE_NOTICE_IRQ);    // clear the interrupt flag
}

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

  mPORTBSetPinsAnalogIn(BIT_8|BIT_9|BIT_10|BIT_11|BIT_12|BIT_13|BIT_14|BIT_15);
  mPORTBSetPinsDigitalOut(BIT_4 | BIT_5 | BIT_6 | BIT_7);
  mPORTBSetPinsDigitalIn(BIT_0 | BIT_1 | BIT_2 | BIT_3);
  mPORTBOpenDrainOpen(BIT_6 | BIT_7);

  INTDisableInterrupts(); // INT step 2
  // INT step 3: turn on CN and choose pins for CN (also choose pull-ups)
  mCNOpen(CN_ON, CN2_ENABLE | CN17_ENABLE | CN18_ENABLE,
          CN2_PULLUP_ENABLE | CN3_PULLUP_ENABLE | CN17_PULLUP_ENABLE);

  oldB = mPORTBRead();
```

```
  oldF = mPORTFRead();
  mPORTBWrite(0x0050);

  INTSetVectorPriority(_CHANGE_NOTICE_VECTOR, 3);    // INT step 4: priority
  INTSetVectorSubPriority(_CHANGE_NOTICE_VECTOR, 2); // INT step 4: subpriority
  INTClearFlag(_CHANGE_NOTICE_IRQ);                  // INT step 5
  INTEnable(_CHANGE_NOTICE_IRQ, INT_ENABLED);        // INT step 6
  INTEnableSystemMultiVectoredInt();                 // INT step 7

  while(1);              // infinite loop
}
```

7.5 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 trigger an interrupt. The change notification pins also 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.
- In the change notification ISR, ports with enabled change notification pins must be read to clear the interrupt.

7.6 Problems

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; you can just write 0 for those bits.)

Chapter 8

Counter/Timers

The basic function of counters is quite simple: they count pulse rising edges. 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. The pulses may come from the internal peripheral bus clock or external sources. 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 $P \leq 2^{16} - 1 = 65535 = 0$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 fixed frequency function (a control loop, for example).

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 `WriteCoreTimer()` and `ReadCoreTimer()` are used in Chapter 5, except that peripheral timers can be set to increment once every PBCLK cycle instead of 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 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 20).

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 15). A timer can also be used to automatically 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 Library Functions

The macros below can be found in `pic32mx/include/peripheral/timer.h` along with some constants for code readability.

OpenTimerx(config, period), x = 1 to 5 Sets TMRx to 0, the 16-bit PRx to `period`, and TxCON to `config`.

OpenTimerxy(config, period), x = 2 or 4, y = x+1 Sets the 32-bit count in TMRx to 0, the 32-bit PRx to `period`, and TxCON to `config`, making sure TxCONbits.ON = 1 and TxCONbits.T32 = 1.

CloseTimerx(), x = 1 to 5 Disables the interrupt associated with Timerx and turns it off (clears TxCON to 0).

CloseTimerxy(), x = 2 or 4, y = x+1 Disables the interrupt and turns off both timers.

WriteTimerx(value), x = 1 to 5 Simply performs `TMRx = value`, where `value` is a 16-bit value.

WriteTimerxy(value), x = 2 or 4, y = x+1 Simply performs `TMRx = value`, where `value` is a 32-bit value.

ReadTimerx(), x = 1 to 5 Returns `TMRx`.

ReadTimerxy(), x = 2 or 4, y = x+1 Returns `TMRx`.

WritePeriodx(value), x = 1 to 5 Simply performs `PRx = value`, where `value` is a 16-bit value.

WritePeriodxy(value), x = 2 or 4, y = x+1 Simply performs `PRx = value`, where `value` is a 32-bit value.

ReadPeriodx(value), x = 1 to 5 Returns `PRx`.

ReadPeriodxy(), x = 2 or 4, y = x+1 Returns `PRx`.

8.4 Sample Code

8.4.1 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"          // plib.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
}

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

  INTDisableInterrupts();             // 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
  INTEnableSystemMultiVectoredInt();  // INT step 7: enable interrupts at CPU

  while (1);
  return 0;
}
```

8.4.2 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 uses both the peripheral library as well as direct SFR manipulation. 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.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.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;

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

  TRISDbits.TRISD1 = 0;       // make pin 1 of Port D an output

  // plib function to set up and turn on Timer2.  We use the full period, 0xFFFF,
  // for counting external pulses.  In the config info, only T2_SOURCE_EXT and T2_ON
  // are necessary; others for 1:1 prescaler, no gate, and 32-bit mode off are defaults.
  OpenTimer2(T2_SOURCE_EXT | T2_ON | T2_PS_1_1 | T2_GATE_OFF | T2_32BIT_MODE_OFF, 0xFFFF);

  INTDisableInterrupts();     // INT step 2: disable interrupts at CPU
                              // INT step 3: setup Timer45:  T4CON, TMR4, PR4, INT for T5
  T4CON = 1 << 15 | 1 << 3;   //             turn on Timer4 in 32-bit; others default
  PR4 = 39999;                //             set period register for 2 kHz INT
  TMR4 = 0;                   //             initialize count to 0
  IPC5bits.T5IP = 4;          // INT step 4: priority for Timer5 (INT for Timer45)
  IFS0bits.T5IF = 0;          // INT step 5: clear interrupt flag
  IEC0bits.T5IE = 1;          // INT step 6: enable interrupt
  INTEnableSystemMultiVectoredInt();  // INT step 7: enable interrupts at CPU

  while (1) {
    sprintf(message,"Elapsed time:  %5u ms\r\n",TMR2);
    NU32_WriteUART1(message);
    for (i=0; i<10000000; i++); // for loop to 10M; delay print statements
  }
  return 0;
}
```

8.4.3 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 500 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 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"          // plib.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];

  IFS0bits.T3IF = 0;          // clear interrupt flag
  sprintf(msg,"The count was %10u, or %10.8f seconds.\r\n",TMR2,TMR2/80000000.0);
  NU32_WriteUART1(msg);
  TMR2 = 0;                   // reset Timer23
}

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

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  TRISDbits.TRISD1 = 0;       // make pin 1 of Port D an output

  // plib function to set up and turn on the 32-bit Timer23.  We use the full period,
  // 0xFFFFFFFF, to avoid unnecessary rollovers in timing durations.  In the config
  // we specify only the gated mode and to turn Timer23 on; other defaults are fine.
  OpenTimer23(T23_ON | T23_GATE_ON, 0xFFFFFFFF);

  INTDisableInterrupts();     // INT step 2: disable interrupts at CPU
                              // INT step 3: setup Timer45:  T4CON, TMR4, PR4, INT for T5
  T4CON = 1 << 15 | 1 << 3;   //             turn on Timer4 in 32-bit; others default
  PR4 = 79999999;             //             set period register for 1 Hz INT
  TMR4 = 0;                   //             initialize count to 0
  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
  INTEnableSystemMultiVectoredInt();  // INT step 7: enable interrupts at CPU

  while (1);
  return 0;
}
```

8.5 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).

8.6 Problems

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 one or two fixed values, 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 (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 is the particular module). The timer 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 count is compared to a single 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 of the Timer base, and the pulse duration determined by OCxR.

In the dual compare modes, the count 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 roll-over of the timer base. 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. If the time base of the OCx peripheral is the 16-bit Timery, then the duty cycle of the pulse train, as a percentage, is

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

The two PWM modes also offer the use of a fault protection input. If this is chosen, then the OCFA pin (corresponding to OC1 through OC4) or the OCFB 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 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 and alterations to the OC SFRs can be made.

OCxCON⟨5⟩, or OCxCONbits.OC32: 1 = use all 32 bits of OCxR and OCxRS to compare to the 32-bit timer source, 0 = use only OCxR⟨15:0⟩ and OCxRS⟨15:0⟩ to compare to a 16-bit timer source.

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.

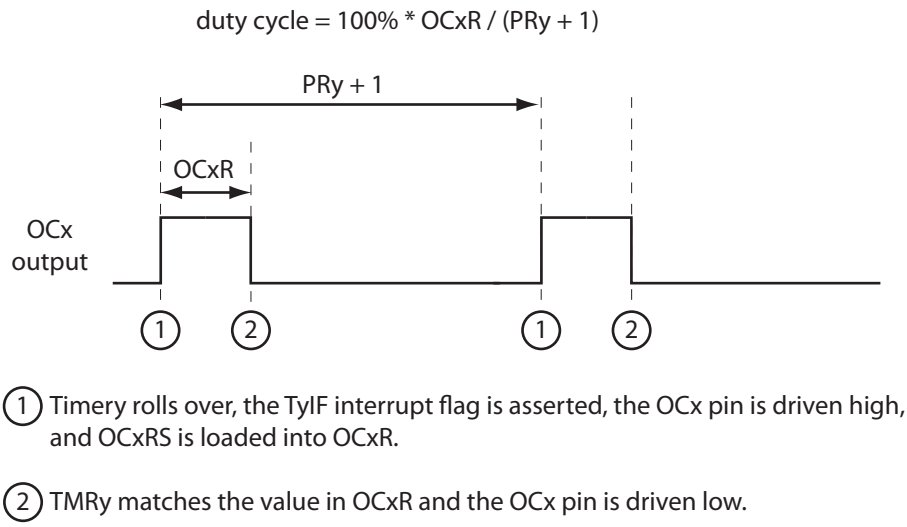


Figure 9.1: A PWM waveform from OCx using Timery as the time base.

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 proportional to 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 Library Functions

The file `pic32mx/include/peripheral/outcompare.h` contains macros and readable constants for the output compare peripheral.

OpenOCx(config, val1, val2) Sets OCxCON to `config`, OC1RS to `val1`, and the initial OC1R to `val2`.

SetDCOCxPWM(hightime) Sets the duration of the high portion of the PWM signal. Equivalent to `OCxRS = hightime`.

9.4 Sample Code

9.4.1 Generating a Pulse Train with PWM

Below is sample code using OC1 with Timer2 to generate a 10 kHz 50% duty cycle PWM signal with no fault pin.

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

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.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
  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; then it is read-only
  T2CONbits.ON = 1;        // turn on Timer2
  OC1CONbits.ON = 1;       // turn on OC1

  // some code ...

  OC1RS = 1000;            // set duty cycle to 50%
  while (1);
  return 0;
}
```

Here's the same code written with library functions.

Code Sample 9.2. `OC_PWM_plib.c` Generating 10 kHz PWM with 50% duty cycle using library functions.

```
//#define NU32_STANDALONE  // uncomment if program is standalone, not bootloaded
#include "NU32.h"          // plib.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

  // some defaults could be relied on below
  OpenTimer2(T2_ON | T2_PS_1_4, 1999);
  OpenOC1(OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE, 1500, 1500);

  // some code ...

  SetDCOC1PWM(1000);  // set duty cycle to 50%
  while (1);
  return 0;
}
```

9.4.2 Analog Output

We can use the PWM output to generate an analog output by low-pass filtering it. This essentially averages the high and low voltages of the wave form, weighted by the duty cycle

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

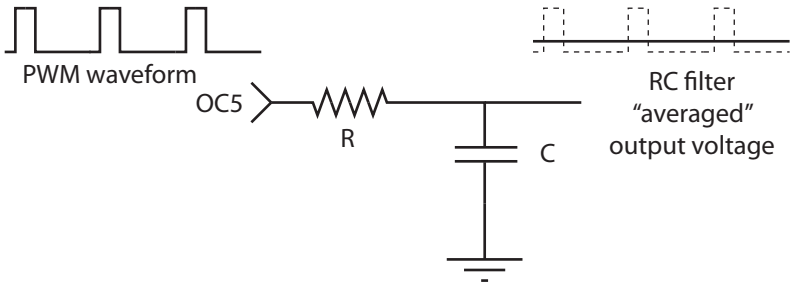


Figure 9.2: An RC low-pass filter "averaging" the PWM output from OC5.

assuming that the PIC32 is powered by 3.3 V.

A simple way to low-pass filter PWM output is to use an RC filter, as shown in Figure 9.2. Averaging is achieved because the resistor R limits the amount of current that can charge and discharge the capacitor. This current allows the capacitor to build up or dump electrical charge, and the voltage across the capacitor is proportional to the charge and the inverse of the capacitance of the capacitor. The larger the value of R or C, the slower the voltage across the capacitor changes.

The *cutoff frequency* of the RC filter, in Hz, is $f_{\text{cutoff}} = 1/(2\pi RC)$. If a sinusoid of frequency f_{cutoff} is input to the RC filter, its magnitude at the output is $1/\sqrt{2}$ of its original frequency. Higher frequencies are further attenuated, while lower frequencies pass through the filter with less attenuation.

We are faced with a tradeoff: we want to choose RC high enough that the high-frequency of the PWM is not visible at the filter output, but we don't want to choose RC so high that changes to the PWM duty cycle are too slowly reflected at the output. A reasonable rule of thumb is to choose f_{cutoff} to be 100 times lower than the frequency of the PWM. At this frequency, only about 1% of the signal at the PWM frequency makes it through the filter

Another consideration is the resolution of the averaged voltage we wish to achieve. For example, are four bits of analog voltage ($2^4 = 16$ voltage levels) sufficient for our application? Eight bits? Twelve? Clearly higher is better, but (1) the device receiving the analog input will have some limit to its analog input sensing resolution, and (2) the transmission lines for an analog signal may be subject to some electromagnetic noise, making extremely high resolution worthless.

Assume we want 10 bits of analog voltage, or 1024 voltage levels, matching the analog-to-digital converter resolution on the PIC32 (Chapter 10). Then the Timerx base must have a PRx of at least 1023, allowing the duration of the high pulse to be anywhere from 0 cycles (0% duty cycle) to 1024 (100% duty cycle). Since the PBCLK is 80 MHz, a Timerx with a period of 1024 counts means that it rolls over at 80 MHz/1024 = 78125 Hz. This is the PWM frequency f_{PWM}. A good f_{cutoff} would be around 780 Hz. Reasonable RC choices could be $R = 2$ kΩ and $C = 0.1$ μF, giving $f_{\text{cutoff}} = 795$ Hz. If the duty cycle is sinusoidally varied from 0% to 100% at 795 Hz, only about 0.7 of the full 0 to 3.3 V amplitude would make it through the filter. In addition, the signal would be phase delayed by about 45°. To largely avoid these effects, we could limit the duty cycle variation to have frequencies of 100 Hz or less.

In summary, the higher the duty cycle resolution, the lower the f_{PWM}, and therefore the lower the frequency at which the duty cycle can be reasonably varied. You can choose between high resolution, low frequency analog outputs or low resolution, high frequency analog outputs. In summary, the three frequencies to consider are f_{PWM} (78125 Hz in our case), f_{cutoff}, and f_{DC}, the maximum frequency component of the desired analog output (and hence the duty cycle). These frequencies should satisfy $f_{\text{PWM}} \gg f_{\text{cutoff}} \gg f_{\text{DC}}$.

Below is code to generate PWM at 78125 Hz with a 10-bit duty cycle based on Timer2. With a suitable resistor and capacitor attached to OC5, 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_{cutoff} significantly lower than 795 Hz.

Code Sample 9.3. `OC_analog_out.c` Using Timer2, OC5, and an RC low-pass filter to create analog output.

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

#define PERIOD 1024        // this is PR2 + 1
#define MAXVOLTAGE 3.3     // in case another voltage is used

int getUserPulseWidth(void) {
  char msg[100];
  float f;
  int hightime;

  sprintf(msg,"Enter the desired voltage, from 0 to %3.1f (volts): ",MAXVOLTAGE);
  NU32_WriteUART1(msg);
  NU32_ReadUART1(msg,10);
  sscanf(msg,"%f",&f);
  if (f>MAXVOLTAGE) f = MAXVOLTAGE;
  if (f<0.0) f = 0.0;
  sprintf(msg,"\r\nSending %5.3f volts.\r\n",f);
  NU32_WriteUART1(msg);
  hightime = (int) (0.5 + PERIOD*f/MAXVOLTAGE);  // convert volts to counts
  return(hightime);
}

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 OC5, PR2 defines PWM frequency
  TMR2 = 0;
  T2CON = 1<<15;  // turn Timer2 on, all defaults fine
  // below, only OC_ON is needed for config, others are defaults.
  // initial pulse width is 0 counts (0 volts out)
  OpenOC5(OC_ON | OC_PWM_FAULT_PIN_DISABLE | OC_TIMER2_SRC | OC_TIMER_MODE16, 0, 0);
  while (1) {
    OC5RS = getUserPulseWidth();
  }
  return 0;
}
```

9.5 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_{cutoff}, allows the generation of analog outputs. There is a fundamental tradeoff between the resolution of the analog output quanitization and the maximum possible frequency component f_{DC} of the generated analog signal. If the PWM frequency is f_{PWM}, then generally the frequencies should satisfy $f_{\text{PWM}} \gg f_{\text{cutoff}} \gg f_{\text{DC}}$.

9.6 Problems

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

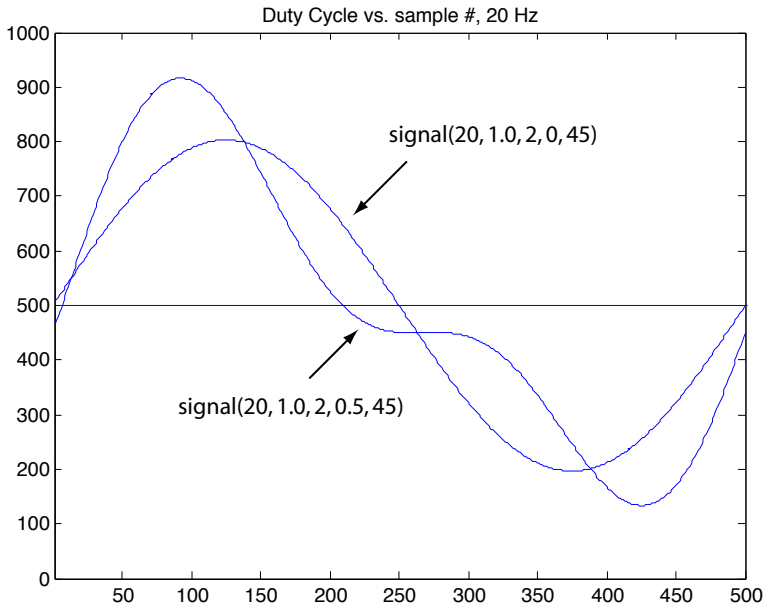


Figure 9.3: Example analog output waveforms, plotted as the duration 0 to 1000 of the high portion of the PWM waveform, which has a period of 1000.

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:

   (a) Write a function that forms a sampled approximation of

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

   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 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. Example waveforms are shown in Figure 9.3.

```
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('Error: 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 = (PHASE/360)*numsamps;      % 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 a float vec in volts; dutyvec is an int vec taking values 0...1000

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

(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 9.3, 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 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

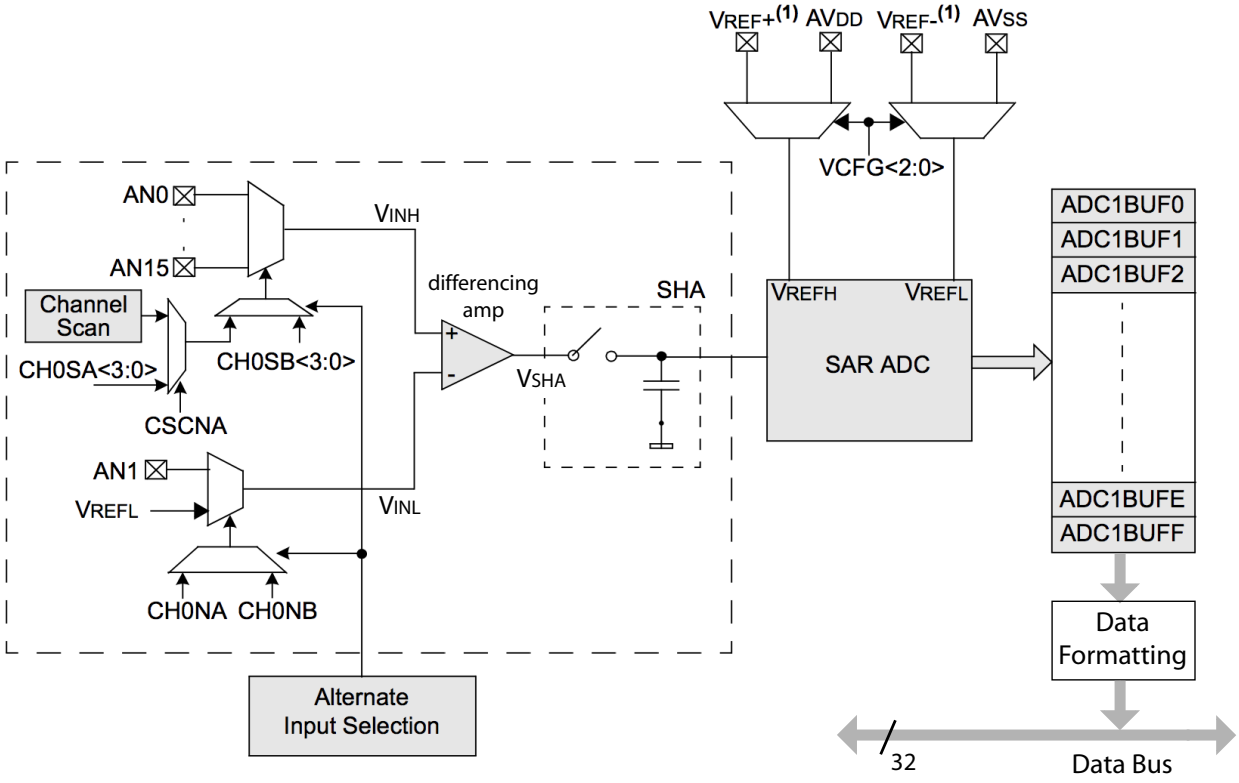


Figure 10.1: A simplified schematic of the ADC module.

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. You should see that 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 Table 31-37 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×(PBCLK period), where k is any integer from 1 to 256. Since the PBCLK period 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:

- 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 V − 1 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 is configured as an analog input. If a bit is 1, it is digital I/O.

AD1CON1 Determines whether 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 VREF+, negative reference could be GND or VREF-); 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., VREF- 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 Library Functions

Relevant macros and constants can be found in `pic32-libs/include/peripheral/adc10.h`.

OpenADC10(config1, config2, config3, configport, configscan) The bits set as 1 in `configport` correspond to pins configured as analog inputs. The bits set as 0 in `configscan` are inputs that are included in a scan (AD1CSSL is the bitwise NOT of `configscan`). AD1CON3, AD1CON2, and AD1CON1 are set as `config3`, `config2`, and `config1`, respectively.

SetChanADC10(config) Sets the AD1CHS SFR to `config` (i.e., which pins are sampled).

EnableADC10() Sets the ON bit in AD1CON1 that indicates that the ADC is activated.

AcquireADC10() Sets AD1CON1bits.SAMP to 1, which causes the SHA to sample.

ConvertADC10() Clears AD1CON1bits.SAMP to 0, which causes the SHA to hold and conversion to begin.

BusyADC10() Returns the AD1CON1bits.DONE bit, where 1 indicates that the ADC conversion is done and 0 indicates that it is not.

ReadADC10(bufindex) Reads the conversion result stored in the `bufindex`'th element of the buffer ADC1BUF.

ReadActiveBufferADC10() Returns 0 if conversions are currently being written into words 0-7 of ADC1BUF, and 1 if they are being written into words 8-15. Only relevant in the dual buffer mode.

CloseADC10() Turns off the ADC (clears the ON bit in ADC1CON1) and disables the interrupt.

10.4 Sample Code

10.4.1 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 $6 \times$ 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"          // plib.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

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

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

  AD1PCFG = 0x3FFF;         // bits 14 and 15 are 0, so AN14 and AN15 are AN inputs
  AD1CON3bits.ADCS = 2;     // ADC clock period is Tad = 2 * (ADCS+1) * Tpb
  AD1CON1bits.ADON = 1;     // turn on A/D converter

  while (1) {
    WriteCoreTimer(0);
    AD1CHSbits.CH0SA = 14;  // connect pin AN14 to MUXA for sampling
    AD1CON1bits.SAMP = 1;   // start sampling
    elapsed = ReadCoreTimer();
    finishtime = elapsed + SAMPLE_TIME;
    while (ReadCoreTimer() < finishtime); // sample for more than 200 ns
    AD1CON1bits.SAMP = 0;   // stop sampling and start converting
    while (!AD1CON1bits.DONE); // wait for the conversion process to finish
    sample14 = ADC1BUF0;    // read the buffer with the result

    AD1CHSbits.CH0SA = 15;  // connect pin AN15 to MUXA for sampling
    AD1CON1bits.SAMP = 1;   // start sampling
    elapsed = ReadCoreTimer();
    finishtime = elapsed + SAMPLE_TIME;
    while (ReadCoreTimer() < finishtime); // sample for more than 200 ns
    AD1CON1bits.SAMP = 0;   // stop sampling and start converting
    while (!AD1CON1bits.DONE); // wait for the conversion process to finish
    sample15 = ADC1BUF0;    // read the buffer with the result
    elapsed = ReadCoreTimer();

    // 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);
    WriteCoreTimer(0);      // make the messages less frantic!
    while(ReadCoreTimer()<DELAY_TICKS);
  }
}
```

10.5 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.

10.6 Problems

Chapter 11

Brushed Permanent Magnet DC Motors

Essentially all electric motors operate on the same principle: current flowing through a magnetic field creates a force on the conductor carrying the current. Because of this relationship between current and force, electric motors can be used to convert electrical energy to mechanical energy. They can also be used to convert mechanical energy to electrical energy; 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 DC motor. For brevity, we will 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 will achieve 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 equipped with a sensing device, most commonly an encoder, to track the position and speed of the motor, and a gearhead to increase the output torque (also reducing output speed).

11.1 Motor Physics

DC motors exploit the *Lorentz force law*,

$$\mathbf{F} = \ell \mathbf{I} \times \mathbf{B}, \tag{11.1}$$

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 11.1.

Now let's replace the conductor by a loop of wire, and constrain that loop to rotate about its center. See Figures 11.2 and 11.3. 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 11.3, 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

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

This torque changes as a function of θ. For $-90^\circ < \theta < 90^\circ$, 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 11.4(a). The torque is zero at

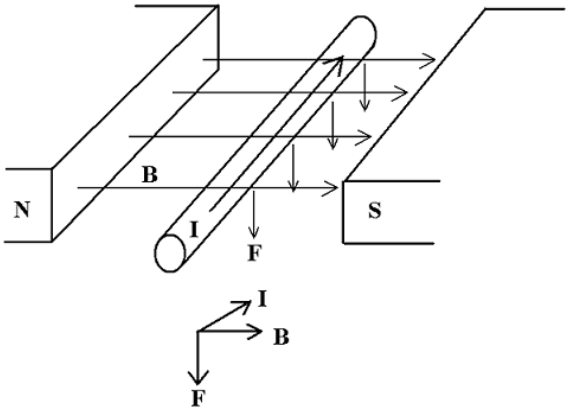


Figure 11.1: Two magnets create a magnetic field **B**, and a current **I** along the conductor causes a force **F** on the conductor.

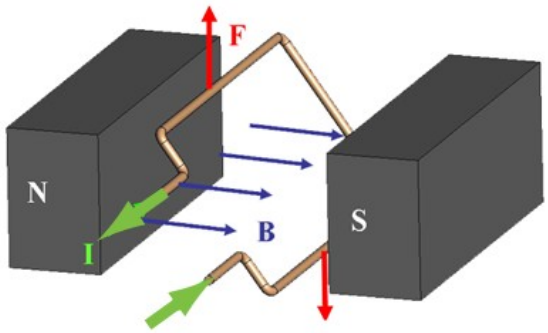


Figure 11.2: A current-carrying loop of wire in a magnetic field.

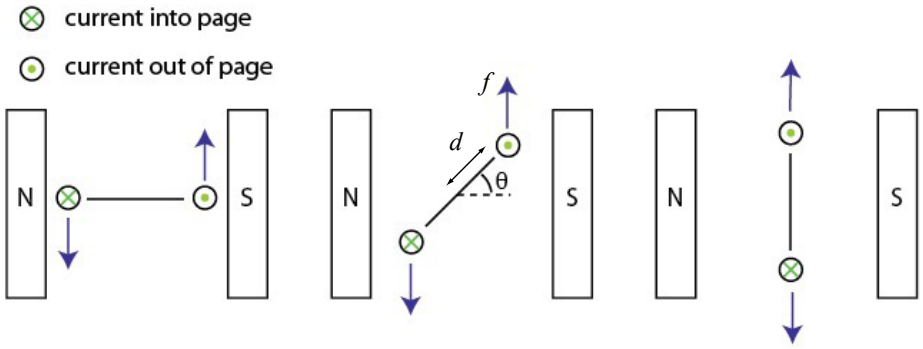


Figure 11.3: A loop of wire in a magnetic field, viewed end-on. Current flows into the page on one side of the loop and out of the page on the other, creating forces of opposite directions on the two halves of the loop. These opposite forces create torque on the loop about its center at most angles θ of the loop.

$\theta = \{-90°, 90°\}$, 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 11.4(b)). The torque is still zero at

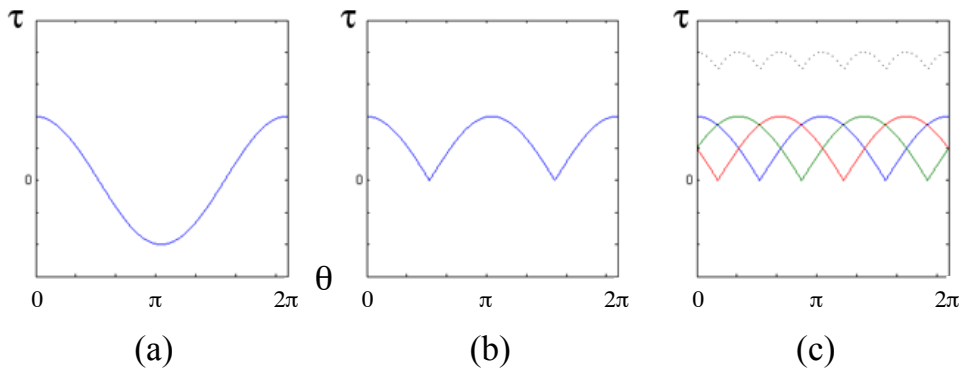


Figure 11.4: (a) The torque on the loop of Figure 11.3 as a function of its angle for a constant current. (b) If we reverse the current direction at the angles $\theta = -90°$ and $\theta = 90°$, we can make the torque nonnegative at all θ. (c) If we use several loops offset from each other, the sum 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 undergoes a large variation as a function of θ. To make the torque more constant as a function of θ, so that the motor's behavior doesn't vary so much with angle, we can introduce more loops of wire, each offset from the others in angle, and each reversing their current direction at appropriate angles. Figure 11.4(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 11.5 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 carbon-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*, and therefore they switch the current direction flowing through the coils. Figure 11.5 shows only two commutator segments and a single coil to make the idea clear, but a real DC motor must have at least three commutator segments, and most good motors have many more than that. Typically there is a coil between any two commutator segments that may be in contact with different brushes at the same time. (Obviously no single commutator segment can be in contact with both brushes at the same time; that would cause a short circuit.)

And that is how brushed DC motors work. The basic design hasn't changed much since the late 1800's. *Brushless* motors are more recent, have permanent magnets forming the rotor and coils (armature) attached to the interior of the motor housing (stator), and use external circuitry and motor angle measurement using Hall sensors to switch the current direction electronically instead of mechanically. 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,
- friction due to the brushes,
- particles due to the wearing brushes,
- electrical noise due to the abrupt switching events, and
- lower continuous current ratings, as brushless motors can use the motor housing to help transfer heat from the coils and therefore can sustain higher continuous currents.

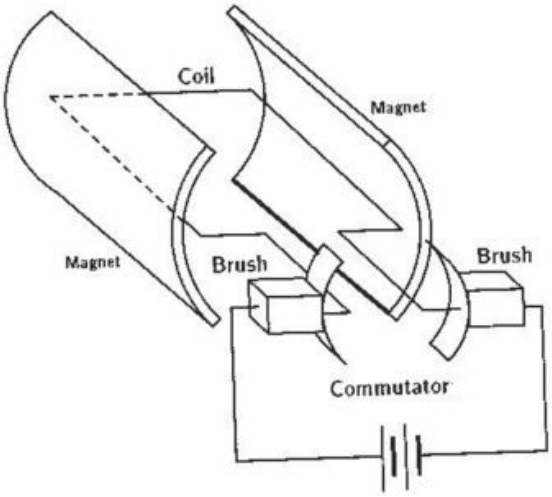


Figure 11.5: Brushes and a two-segment commutator.

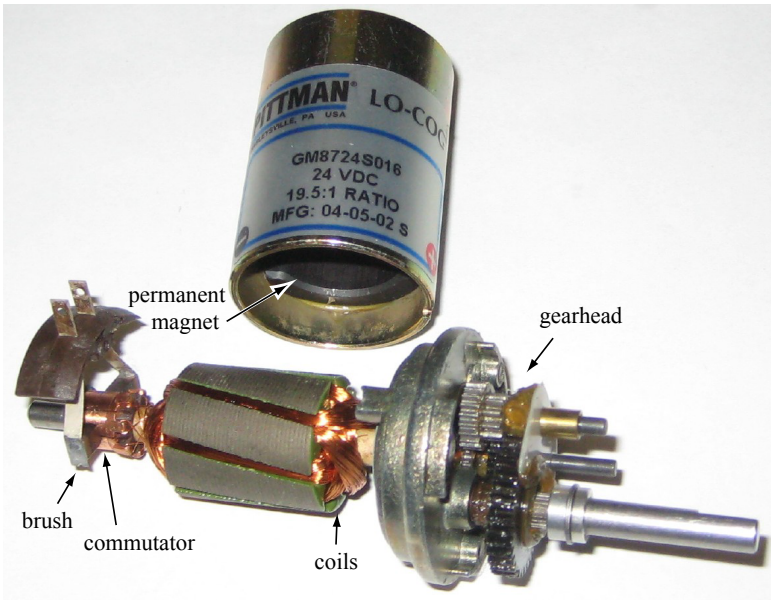


Figure 11.6: The two brushes of this disassembled DC motor are attached to the motor housing, which has otherwise been removed. One of the two permanent magnets is visible inside the housing. Coils are often wrapped around iron or other ferromagnetic material to increase magnetic permeability.

Brushless motors have the disadvantage of being more expensive, and the issues above are not serious limitations for many applications.

Figure 11.6 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.

11.2 Governing Equations

To derive an equation to model the motor's behavior, we 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 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 all 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 these last terms, we have our motor model, written in terms of power:

$$IV = \tau\omega + I^2R + LI\frac{dI}{dt}. \tag{11.2}$$

From Equation (11.2) we can derive all other relationships of interest. For example, dividing both sides of (11.2) by I, we get

$$V = \frac{\tau}{I}\omega + IR + L\frac{dI}{dt}. \tag{11.3}$$

The ratio τ/I is a constant, an expression of the Lorentz force law for the particular motor design. This constant is called the *motor constant* k_m, and it is one of the most important properties of the motor:

$$k_m = \frac{\tau}{I} \quad \text{or} \quad \tau = k_m I. \tag{11.4}$$

The torque produced by a motor is proportional to the current through it. The SI units of k_m 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 datasheet we will see shortly.) The term *torque constant* k_t is also commonly used, to emphasize that the torque is proportional to the current. Equation (11.3) also shows that the SI units for k_m 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*. While you should recognize these other terms, we will stick with "motor constant" k_m in most places to emphasize that all of these refer to the same thing.

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

$$V = k_m\omega + IR + L\frac{dI}{dt}. \tag{11.5}$$

You should remember, or be able to quickly rederive, the power equation (11.2), the motor constant (11.4), and the voltage equation (11.5).

The term $k_m\omega$, 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 (11.5) reduces to

$$V = k_m\omega.$$

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 cars works.

The existence of this back-emf term also means that if we put a constant voltage V across a free-spinning motor, after some time it will reach a constant speed V/k_m. At this speed, by (11.5), the current I drops to zero, meaning there is no more torque τ to accelerate the motor.

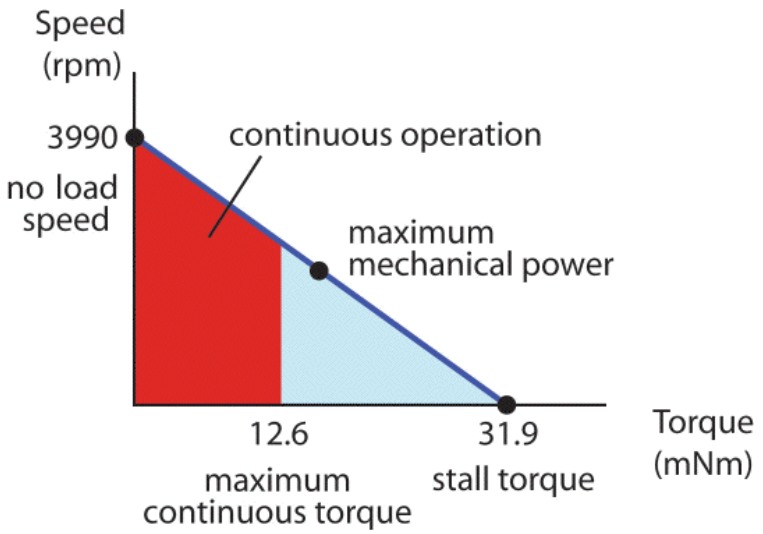


Figure 11.7: The speed-torque curve for the 24 V Maxon motor 2140.937-22.116-050.

11.3 The Speed-Torque Curve

If we assume the motor is at steady state, i.e., $dI/dt = 0$, Equation (11.5) reduces to

$$V = k_m\omega + IR. \tag{11.6}$$

Using (11.4), we get the equivalent form

$$\omega = \frac{1}{k_m}V - \frac{R}{k_m^2}\tau. \tag{11.7}$$

Equation (11.7) gives ω as a linear function of τ for a given constant V. This line, of slope $-R/k_m^2$, is called the *speed-torque curve* (Figure 11.7).

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

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

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

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

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 output torque and current ($I_{\text{stall}} = \tau_{\text{stall}}/k_m = V/R$) are maximized due to the lack of back-emf. Which point along the speed-torque curve the motor actually operates at is determined by what the motor shaft is attached to.

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

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

and we can find the value of c that maximizes the power output 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 $\tau = (1/2)\tau_{\text{stall}}$ and $\omega = (1/2)\omega_0$. This maximum output power is

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

The efficiency of converting electrical to mechanical energy is

$$\eta = \frac{\tau\omega}{IV}.$$

The efficiency depends on the operating point (τ, ω), and it is zero at $\tau = 0$ and $\omega = 0$, as there is no mechanical power output.

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 *continuous operating region* under the speed-torque curve. This is the operating region where, in a typical ambient environment, the motor's housing can transfer the heat to the environment at a rate that will prevent the motor from overheating. The motor can be intermittently operated outside of the continuous operating region, but if it is operated continuously at high currents, the heat will cause the insulation of the coil wires to melt. In Figure 11.7, we see that the maximum power output condition, as well as stall, are outside the continuous operating region.

Since the continuous operating region constraint depends only on torque (current), it is bounded by a vertical line (see Figure 11.7). Some motors also have a specified *maximum permissible speed*, which creates a horizontal line constraint on the permissible operating range. This speed is determined by the shaft bearings, which may have a maximum speed associated with them, or properties of the brushes sliding over the commutator. The maximum permissible speed is typically larger than the no-load speed.

The speed-torque curve for a motor is drawn based on a *nominal* voltage. This is a "safe" voltage that the manufacturer recommends. Typically it is a voltage that will not make the motor spin beyond its maximum permissible speed, and one where intermittent operation at stall will not damage the motor. There is no problem with using a higher voltage, however, provided you do not continuously operate the motor at currents larger than the continuous operating current, nor spin the motor faster than the maximum permissible speed.

11.4 Motor Data Sheet

Motor manufacturers summarize motor properties in a speed-torque curve and in a datasheet similar to the one below (in addition to the mass and dimensions of the motor). 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.

Motor Property	Symbol	Value	Units	Comments
Nominal voltage	V_{nom}		V	Should be chosen so the no-load speed is safe for brushes, commutator, and bearings.
Power rating	P		W	Usually the max continuous electrical input power without overheating.
Terminal resistance	R		Ω	Resistance of the motor windings. May change as brushes slide over commutator segments. Increases with heat.
Terminal inductance	L		H	Inductance due to the coils.
Motor constant	k_m		Nm/A	The constant ratio of torque produced to current through the motor. Also called the torque constant.
Rotor inertia	J		kgm^2	Often given in units gcm^2.
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.
Max continuous current	I_c		A	Max continuous current without overheating.
Max continuous torque	τ_c		Nm	Max continuous torque without overheating.
Starting current	I_s		A	Same as stall current, V_{nom}/R.
Stall torque	τ_{stall}		Nm	The maximum torque achievable at the nominal voltage. Occurs at stall.
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.
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.
Electrical time constant	T_e		s	The time for the motor current to reach 63% of its final value. Equal to L/R.
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_m^2.
Short-circuit damping	B		Nms/rad	Not included in some data sheets, but useful for motor braking (and haptics).
Friction				Not included in most data sheets. See explanation.

11.4.1 Getting to Know Your Brushed DC Motor

Here are some things to try to get to know your motor and encoder:

1. 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.

2. Now short the motor terminals by plugging the wires into the same row of a breadboard. 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 a difference?

3. Work with a partner. Connect your two motors together so the + terminals of the motors are attached to each other, and the − terminals are attached to each other. (Doesn't matter which terminal you call + or −. Connect the wires through the breadboard.) No batteries or external power. Now try spinning one of the motor shafts by hand. What do you predict will happen to the other motor, if anything? What actually happens?

4. Try measuring your motor's resistance using your multimeter. Notice that it varies 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 with a 6 V battery pack and use your multimeter to measure the current as you stall the motor's shaft by hand.

5. Attach your motor's encoder cable to +5 V (red wire) and GND (black) of your NUscope. Attach the other two encoder wires to two of your scope's digital inputs. Attach one of your motor terminals to channel A on your scope and the other to scope ground. Spin the shaft by hand and observe the encoder pulses, including their relative phase. Also observe the back-emf on channel A.

6. Now disconnect your two motor terminals from the scope, and instead power your motor using a 6 V battery pack. Your motor's encoder has 99 lines. Given this, and the rate of the encoder pulses you observe on your scope, calculate the motor's no-load speed and enter it on your data sheet.

7. Work with a partner. Couple your two motor shafts together using some tubing. Now plug one terminal of one of the motors (we'll call it the *passive* motor) into channel A 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 6 V so that both motors spin. Measure the speed of the passive motor by looking at its encoder count rate. Also measure its back-emf on channel A. With this information, calculate the passive motor's motor constant k_m. (**Note:** You may need to use your multimeter to measure the back-emf, instead of the scope, if the back-emf exceeds the range of the scope.)

You can now continue with the problems at the end of this chapter. Some methods for completing the problems are suggested below.

11.4.2 Characterizing a Brushed DC Motor

Below are some ideas for characterizing a motor using mostly your NUscope, multimeter, encoder (or any way to measure the motor speed), and resistors and capacitors. There is more than one way to measure most of these motor parameters. Feel free to come up with your own methods.

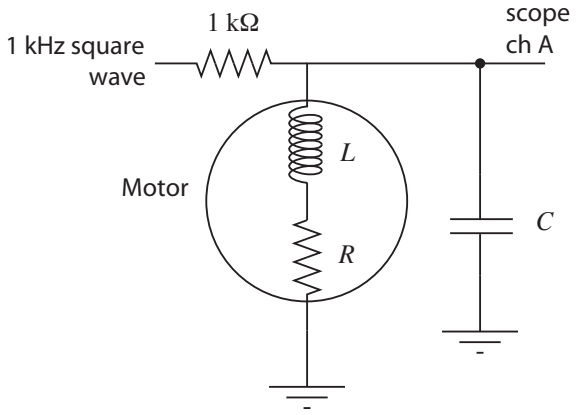
Nominal voltage This voltage is just a recommendation. Exceeding the nominal voltage is fine, though damage to the brushes and commutator, or to the bearings, is possible at high speeds. Nominal voltage cannot be measured, but a typical no-load speed for a brushed DC motor is between 3000 and 10,000 rpm, so the nominal voltage will usually give a no-load speed in this range. For our data sheet, we'll use a nominal voltage of 6 V. (The motor is rated for 24 V, but we want to keep current low.)

Power rating The power rating is the maximum continuous electrical input power and it depends on thermal characteristics of the motor (i.e., the rate at which heat produced in the coils can be dissipated to the environment). It is typically based on a "standard" temperature environment (like 25° C). Given a maximum temperature for the coils before melting the insulation, the power rating is determined by the maximum current that can be continuously sent through the coils without exceeding this temperature in steady state. To avoid doing these measurements, let's assume that the coils can dissipate up to 5 W continuously before overheating.

Terminal resistance 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. See the description of starting current.

Terminal inductance There are a number of ways we could measure inductance. We propose adding a capacitor in parallel with the motor and measuring the oscillation frequency of the resulting RLC circuit. (If you'd like to try something else, be our guest, but (1) make sure to document your method, and (2) if you are driving the motor with a current from your NUscope, be sure to use a 1 kΩ resistor in series with the motor, to limit the current the scope has to provide.)

We'll use the NUscope to generate a 1 kHz square wave to pulse an RLC circuit constructed of the motor, which acts as a resistor and inductor, and an added capacitor. Build the circuit shown below, where a good choice for C may be 0.01 μF or 0.1 μF.



Use the NUscope to put a 1 kHz square wave at the point indicated. The 1 kΩ resistor limits the current from the scope. The motor is modeled as a resistor and inductor. (Back-emf will not be an issue, because the motor will not move, as it is being driven with tiny current at high frequency.) Measure the voltage with your NUscope where indicated. You should be able to see a decaying oscillatory response to the square wave transitions when you choose the right V/div and s/div 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 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., 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 indistinguishable from the undamped natural frequency.

Motor constant 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.

Rotor inertia The rotor inertia can be estimated from measurements of the mechanical time constant T_m, the motor constant k_m, and the resistance R. Alternatively, a ballpark esimate can be made from 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.

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

No-load current You can determine I_0 by using a multimeter in current measurement mode. Friction torque is $k_m I_0$.

Max continuous current For our motor, assume that the motor coils can dissipate up to 5 W continuously before overheating. Use your estimate of motor resistance to then estimate the max continuous current.

Max continuous torque This is determined by thermal considerations. We will assume that the motor coils can dissipate up to 5 W continuously.

Starting current Starting current is equivalent to stall 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 This can be obtained from k_m and I_{stall}.

Max mechanical power 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 regime.

Max efficiency Efficiency is defined as the power out divided by the power in, $\tau\omega/(IV)$. The wasted power is due to coil heating and friction losses. Motors are most efficient at converting electrical power to mechanical power at high speeds and low torque. The maximum efficiency is approximately

$$\eta_{\max} = \left(1 - \sqrt{\frac{I_0}{I_s}}\right)^2,$$

occurring typically at high speed and low torque.

Electrical constant Identical to the motor constant. When written in SI units, k_m and k_e have the same numerical values, but often k_m will be given in mNm/A or English units like oz-in/A, and often k_e will be given in V/rpm.

Speed constant Just the inverse of the electrical constant.

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

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

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

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

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

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

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

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

or

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

with a time constant of $T_m = JR/k_m^2$.

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 an approximation.

Short-circuit damping When the terminals of the motor are shorted together, you get a viscous damping torque τ opposing the shaft angular velocity ω, satisfying the equation $\tau = -B\omega$, where $B > 0$. The damping B can be calculated from the torque constant and terminal resistance.

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

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

where b_0 is the Coulomb friction torque, indicating that the friction torque opposes the direction of motion ($\text{sgn}(\omega)$ just returns the sign of ω), and b_1 is a viscous friction coefficient. At no load, $\tau_{\text{fric}} = k_m I_0$.

11.5 Problems

1. There are 20 entries on the motor datasheet. Let's assume zero friction, so we ignore the last entry. (We will also assume that the motor coils can dissipate a maximum of 5 W continuously without overheating.) Of the 19 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 $19 - N$ dependent entries. For each of the $19 - N$ 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 $I_s = V_{\text{nom}}/R$.

2. Based on your experiments, create a data sheet with all 20 entries for $V_{\text{nom}} = 6$ V. For the remaining 19 entries, 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 a screen snap of the scope trace you used to estimate ω_n and L, and indicate the value of C that you used.

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

3. Based on your data sheet, draw the speed-torque curves described below, and answer the associated questions.

   (a) Draw the speed-torque curve for your motor assuming a nominal voltage of 6 V. Indicate the stall torque and no-load speed. If the motor coils can dissipate a maximum of 5 W continuously before overheating, indicate the continuous operating regime. What is the power rating P for this motor? What is the max mechanical power P_{max}?

   (b) **(Do not do any more experiments for the remainder of this problem; just extrapolate your previous results.)** Draw the speed-torque curve for your motor assuming a nominal voltage of 24 V. Indicate the stall torque and no-load speed. If the motor coils can dissipate a maximum of 5 W continuously before overheating, indicate the continuous operating regime. What is the power rating P for this motor? What is the max mechanical power P_{max}?

   (c) Often DC motors spin at speeds that are too high, and torques that are too low, to be useful. If we put a $G = 10$, or 10:1, gearhead on the output of our motor, however, the speed of the motor is reduced by a factor of 10 ($\omega_{\text{out}} = \omega_{\text{in}}/G$) and the torque is increased by a factor of 10 ($\tau_{\text{out}} = G\tau_{\text{in}}$). Draw the speed-torque curve for the 24 V motor (the previous curve you drew) with a 10:1 gearhead and indicate the no-load speed and stall torque.

   (d) Gearheads are not 100% efficient; some power is lost due to friction and impact between gear teeth. Now assume our 10:1 gearhead from the previous example is $\eta = 80\%$ efficient. The relation $\omega_{\text{out}} = \omega_{\text{in}}/G$ must be preserved (it's enforced by the teeth), so we will use the relation $\tau_{\text{out}} = \eta G\tau_{\text{in}}$, giving $P_{\text{out}} = \tau_{\text{out}}\omega_{\text{out}} = \eta\tau_{\text{in}}\omega_{\text{in}} = \eta P_{\text{in}}$. Draw the speed-torque curve for the 24 V motor with an 80% efficient 10:1 gearhead.

Chapter 12

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.

12.1 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 attempting to preserve mechanical power. For specificity, we refer to a gearhead with an input (attached to the motor shaft) and an output shaft.

Figure 12.1 shows the basic idea. The input shaft is attached to an input gear with N teeth, and the output shaft is attached to an output gear 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 12.2(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

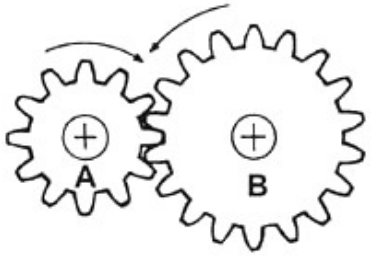


Figure 12.1: The input gear A has 12 teeth and the output gear B has 18, making the gear ratio $G = 1.5$.

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.

12.1.1 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 12.1.2) 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.

12.1.2 Examples

Figure 12.2 shows several different gear types. Not shown are cable, belt, and chain drives, which can also be used to implement a gear ratio while also transmitting torques over distances.

Spur gearhead. Figure 12.2(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 12.2(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 12.2(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 12.2(d), making for a large gear ratio in a compact package.

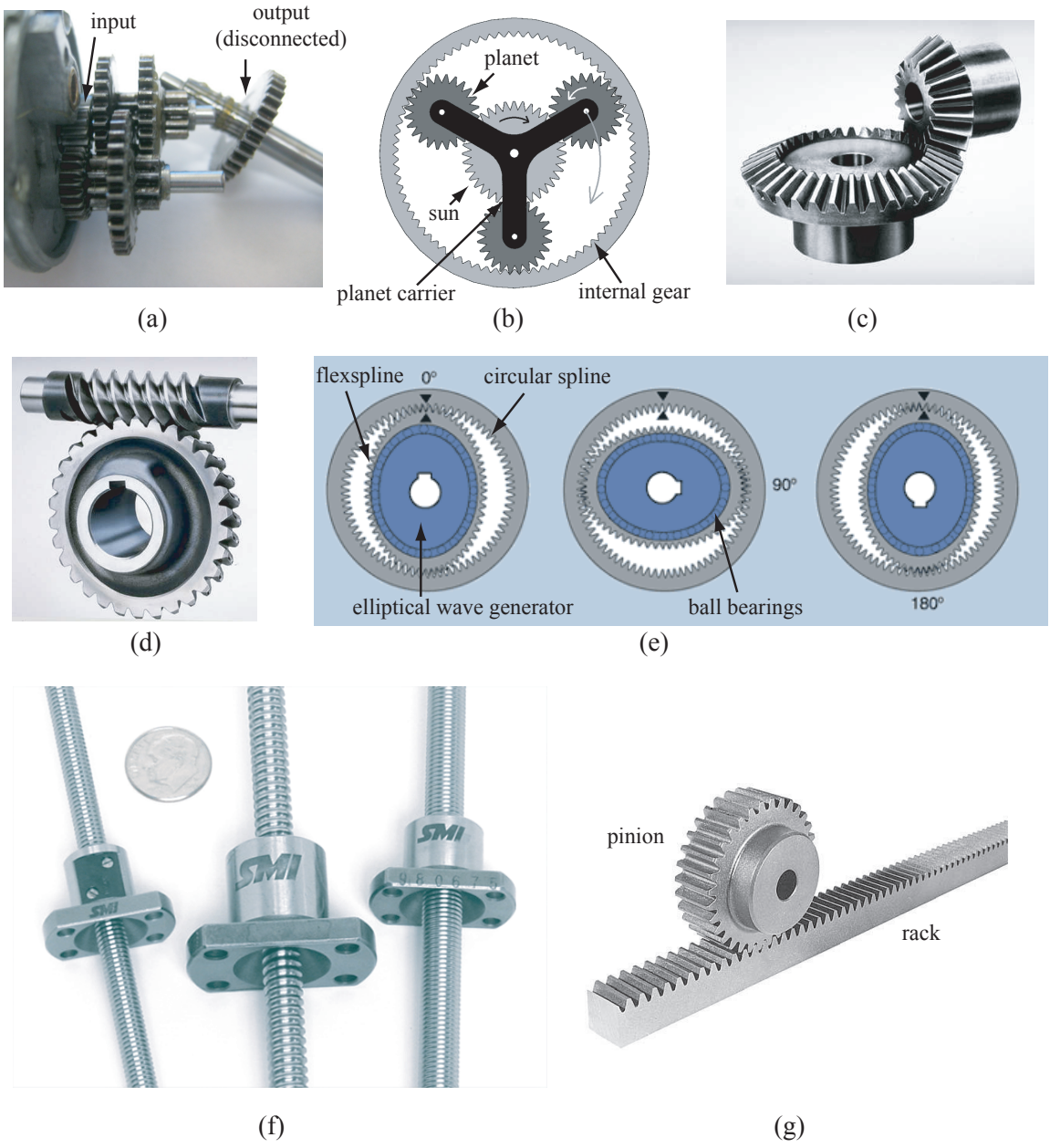


Figure 12.2: (a) Multi-stage spur gearhead. (b) A planetary gearhead. (c) Bevel gears. (d) Worm gears. (e) Harmonic drive gearhead. (f) Ball screws. (g) Rack and pinion.

Harmonic drive. The *harmonic drive* gearhead (Figure 12.2(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.

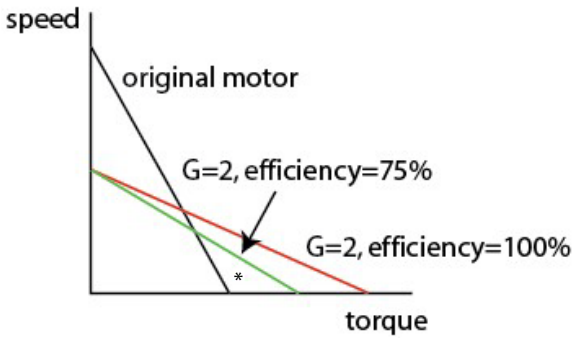


Figure 12.3: The effect of gearing on the speed-torque curve. The operating point * is possible with the gearhead, but not without.

Ball screw and lead screw. A ball screw or lead screw (Figure 12.2(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 12.2(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 12.2(g)) is another way to convert angular to linear motion. The rack is typically mounted to a part on a linear slide.

12.2 Choosing a Motor and Gearhead

12.2.1 Speed-Torque Curve

Figure 12.3 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.

12.2.2 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

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

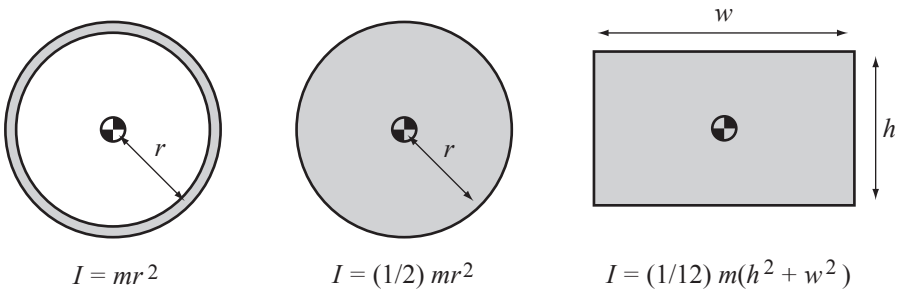


Figure 12.4: Inertia for an annulus, a solid disk, and a rectangle, each of mass m, about an axis out of the page and through the center of mass.

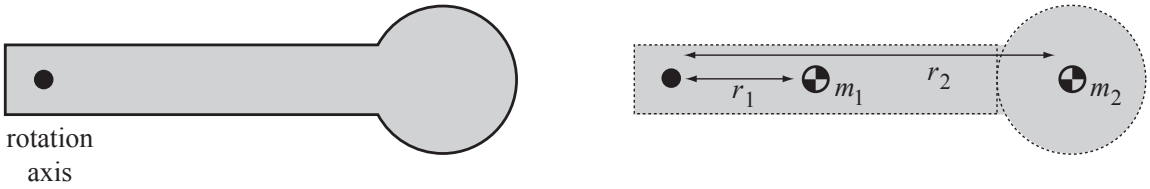


Figure 12.5: The body on the left can be approximated by the rectangle and disk on the right. If the inertias of the two bodies (about their centers of mass) are I_1 and I_2, then the approximate inertia of the compound body about the rotation axis is $I = I_1 + m_1 r_1^2 + I_2 + m_2 r_2^2$ by the parallel-axis 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 12.4 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 12.4, we can approximately calculate the inertia of a load consisting of multiple bodies (Figure 12.5). 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}.

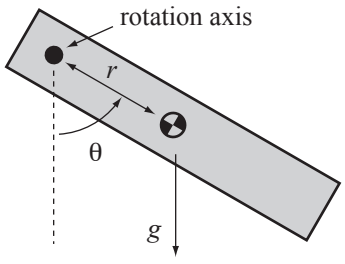


Figure 12.6: A load in gravity.

Given a load of mass m and inertia I_{load} (about the gearhead axis) in gravity as shown in Figure 12.6, 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).

12.2.3 Choosing a Motor and Gearhead

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

- The motor can be chosen based on the 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, the load inertia I_{load}, and the motor torque τ_m, our system is inertia matched if the gearing G is chosen so that the load acceleration α is maximized. 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.

Chapter 13

DC Motor Control

Driving a motor with variable torque requires variable high current. A microcontroller is capable of neither 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.

13.1 The H-bridge and Pulse Width Modulation

Figure 13.1 shows an H-bridge current amplifier used to drive an inductive load, like a DC motor. It consists of four switches, typically implemented with bipolar junction transistors or MOSFETs, and four flyback diodes. 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 (CCW rotation)
S2, S3	negative (CW rotation)
S1, S3	zero (short-circuit braking)
S2, S4	zero (short-circuit braking)
none or one	open circuit (free-wheeling)

Certain 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 usually easier to buy one packaged in an integrated circuit. Apart from reducing your component count, these ICs also make it impossible for

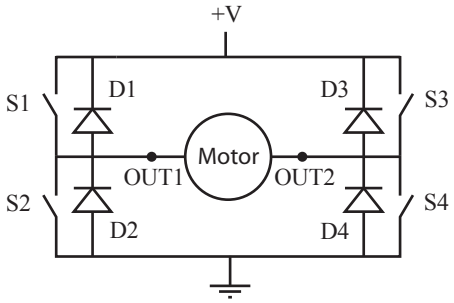


Figure 13.1: An H-bridge constructed of four switches and four flyback diodes. OUT1 and OUT2 are the H-bridge outputs, attached to the two terminals of the DC motor.

Input				Output		
IN1	IN2	PWM	STBY	OUT1	OUT2	Mode
H	H	H/L	H	L	L	Short brake
L	H	H	H	L	H	CCW
		L	H	L	L	Short brake
H	L	H	H	H	L	CW
		L	H	L	L	Short brake
L	L	H	H	OFF (High impedance)		Stop
H/L	H/L	H/L	L	OFF (High impedance)		Standby

Figure 13.2: The input-output truth table for the TB6612 H-bridge.

you to accidentally cause a short circuit. An example H-bridge IC is the Toshiba TB6612 (google it to find the datasheet). This chip consists of two full H-bridges, each one of which is capable of providing 1.2 A continuous and 2 A or more peak. It uses two voltage supplies: one to drive the motor (4.5 V to 15 V and high current) and one for the chip logic (2.7 V to 6 V, typically the same supply used for your microcontroller).

Let's consider a single H-bridge of the TB6612 (Figure 13.2). Instead of directly controlling the four switches S1-S4, our four logic inputs are IN1, IN2, PWM, and STBY (for "standby"). If STBY is logic low, the H-bridge outputs (at the OUT1 and OUT2 terminals in Figure 13.1) are high impedance, or effectively disconnected from the motor. If STBY is high, then IN1 and IN2 determine the rotation direction of the motor: IN1 = L and IN2 = H means CCW rotation of the motor, IN1 = H and IN2 = L means CW rotation, IN1 = IN2 = L causes high impedance outputs, and IN1 = IN2 = H causes the two outputs to be at the same voltage level, i.e., the motor is braked by its own short-circuit damping.

In the CCW mode, when the PWM signal is high, switches S1 and S4 are closed, OUT1 is connected to voltage VM and OUT2 is connected to GND, and current flows left to right through the motor. When the PWM signal is low, switches S2 and S4 are closed, OUT1 and OUT2 are connected to GND, and the motor is braked. In the CW mode, switches S2 and S3 are closed when PWM is high, OUT2 is VM and OUT1 is GND, and current flows right to left through the motor. When PWM is low, OUT1 and OUT2 are connected to GND and the motor is braked.

Figure 13.3 provides details of the behavior in CCW mode as PWM switches from high to low and back to high. When PWM is high, the MOSFET switches S1 and S4 are closed and conduct current from the power voltage VM, through the motor, to GND, as shown, during time t1. When PWM drops low, first the switch S1 is opened. Since the inductive motor continues to demand current, it will draw current from GND through the flyback diode D2. After time t2 elapses, the switch S2 is closed, and the motor is short-circuit braked. Now PWM goes high, so the switch S2 is opened again, and once again current is drawn through D2. Finally, after time t4, switch S1 closes again, and current is provided from VM. The "dead" times t2 and t4, where current is drawn through a diode, are included for a safety margin, to make sure that switches S1 and S2 are never closed simultaneously, which would cause a short circuit.

The TB6612 has the flyback diodes built in. Some H-bridge ICs require the diodes to be provided externally. In that case, you must choose flyback diodes capable of carrying the maximum current that can flow through the motor, and they must be fast switching from nonconducting to conducting. Schottky diodes are common due to their fast switching and low forward bias voltage, resulting in less power lost to heat.

Figure 13.4 shows the wiring of the TB6612 H-bridge to drive a single DC motor. The 6 V battery pack provides the motor voltage VM, and the TB6612's logic operates at the same 3.3 V as the PIC32.

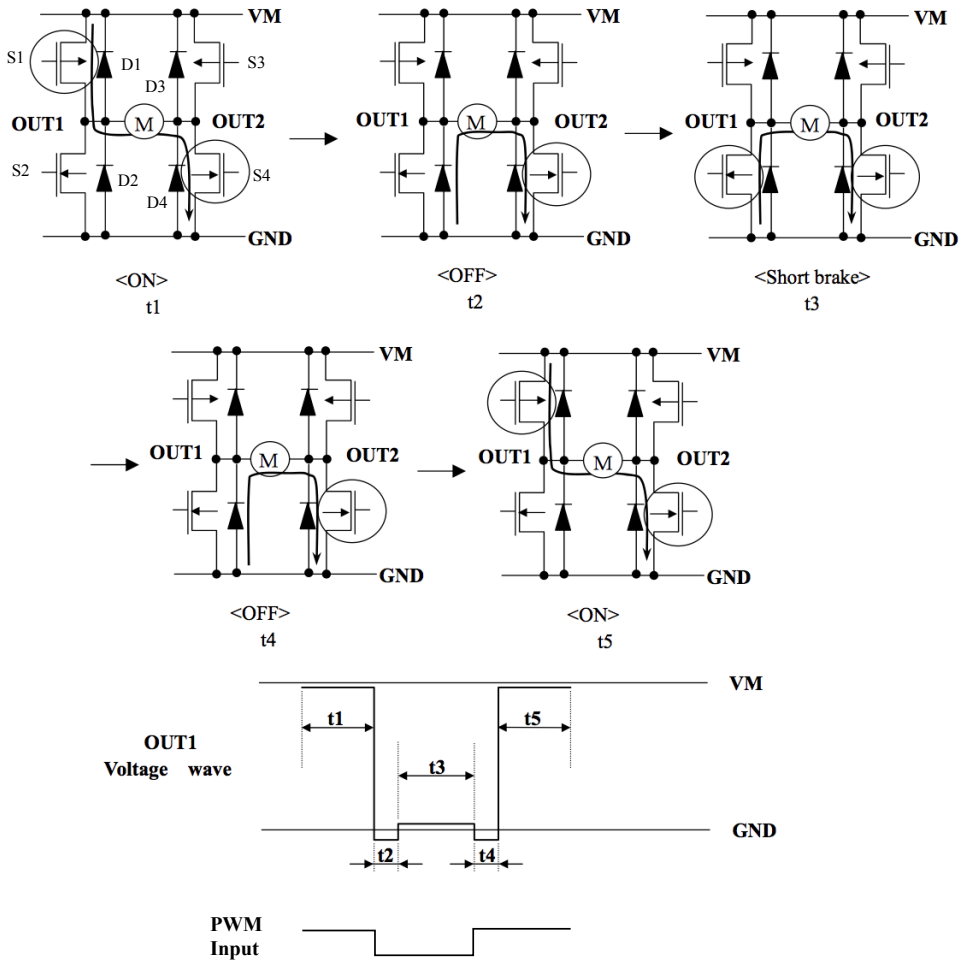


Figure 13.3: A detailed look at CCW mode as PWM switches high-to-low and then low-to-high.

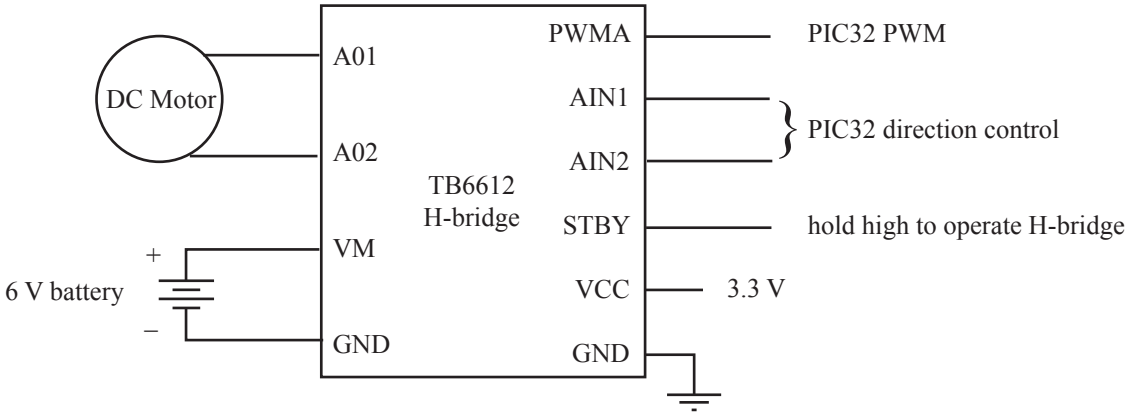


Figure 13.4: Connecting the TB6612 to a DC motor, motor battery pack, and the PIC32.

13.1.1 Control with PWM

Referring again to Figure 13.3, we see that we can use PWM to quickly alternate between positive voltage VM and zero voltage across the motor. Rapidly switching the voltage between VM and 0 has a similar effect

as an average voltage V_{ave} across the motor:

$$V_{\text{ave}} = \text{DC*VM}, \quad \text{where } 0 \leq \text{DC} \leq 1.$$

Let's look at this claim more closely. Assume that PWM is high and a current I_0 is flowing left to right through the motor. Now we switch PWM to low, switch S2 is closed, and the voltage across the motor drops from VM to 0. The motor, being inductive, continues to carry current, and we want to see how the current changes while PWM is low. Ignoring back-emf (it won't matter here), the motor equation

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

becomes

$$0 = I_0R + L\frac{dI}{dt}$$

or

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

with solution

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

The time constant of this first-order decay of current is the motor's electrical time constant, $T_e = L/R$. Assuming "typical" values of $L = 1$ mH and $R = 10\ \Omega$ for a small motor, the time constant is 0.1 ms. Thus the current will decay to about 37% of its original value in 0.1 ms. On the other hand, the mechanical time constant T_m is typically significantly larger (e.g., two orders of magnitude or more), particularly with a load attached to motor. Let's assume $T_m = 10$ ms. Then if the PWM has a 50% duty cycle at 10 kHz, we have a braking phase of duration $0.5/10^4$ s $= 50\ \mu$s during each PWM cycle, meaning that the speed of the shaft will drop to about

$$e^{-5\times10^{-5}/T_m} = 99.5\%$$

of its original value. Not much change. So the net effect of having a rapidly switching voltage, with average voltage V_{ave}, is almost the same as having a constant voltage V_{ave}.

We should choose the PWM frequency sufficiently high so as to avoid observable variation in the motor's speed during a PWM cycle. We should be careful not to choose the frequency so high, however, that the switches do not have time to fully activate during each cycle. The TB6612 datasheet suggests PWM up to 100 kHz. Common PWM frequencies are 10 to 40 kHz.

13.1.2 Other Practical Considerations

Motors are noisy devices, creating both electromagnetic interference (EMI) and voltage spikes on the power lines. These effects can disrupt the functioning of your microcontroller, cause erroneous readings on your sensor inputs, etc. EMI shielding is beyond our scope here, but it is easy to use *optoisolation* to separate noisy power and 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.

Another issue arises when the motor and load attain significant kinetic energy. When we brake the load, the energy must go somewhere. Some of it is lost to friction, and some of it is lost to I^2R winding heating. Remaining energy is dumped back into the power supply, essentially trying to "charge it up," whether it wants to be charged or not. Current passing through the flyback diodes D1 and D3, for example, are essentially charging the power supply and increasing the voltage. Some power supplies can handle this better than others; power supplies with large output capacitors may fare better than switching supplies, for example. You

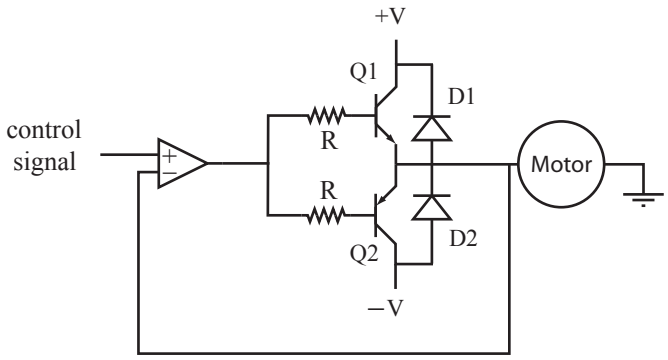


Figure 13.5: A simple linear push-pull amplifier.

can also address the problem by putting a high-capacitance, high-voltage capacitor across the power supply. This capacitor acts as a repository for the kinetic energy converted to electrical energy. If the capacitor voltage gets too high, a switch can allow the back-current to be redirected to a "regen" power resistor, which is designed to dump electrical energy as heat. "Regen" is short for "regenerative" or "regeneration."

13.1.3 Comparison to a Linear Push-Pull Amp

Another common method for driving a load is the linear push-pull amp, shown schematically in Figure 13.5. In this amplifier, the NPN transistor Q1 "pushes" current from +V, through the motor, to GND, while the PNP transistor Q2 "pulls" current from GND, through the motor, to −V. The control signal is a low-current analog signal. It is fed into an op-amp which is configured as a voltage follower. Since there is feedback from the output of the op-amp to the inverting input, the op-amp 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 the motor, the voltage across the motor should be equal to the control signal voltage. The circuit simply boosts the current available to drive the motor beyond the current available from the control signal (i.e., it has a high impedance input and a low impedance output).

As an example, if +V = 10 V, and the control signal is at 5 V, then 5 V should be across the motor. To double-check that our circuit works as we expect, we calculate the current that would flow through the motor (for example, when it is stalled); this is the current I_e that 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 $I_b = I_e/(\beta + 1)$, where β is the transistor gain. If so, we are in good shape. The voltage at the base of Q1 is a diode drop (or more) higher than the voltage across the motor, and the voltage at the op-amp output is that base voltage plus $I_b R$.

An example application would be controlling a motor speed based on a knob (potentiometer). The potentiometer could 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, we can connect the op-amp output 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 and 0.7 V, or whatever the base-emitter voltage is when the transistors are activated. We have a "deadband" or "crossover distortion" from the control signal to the motor voltage.

Comparing an H-bridge driver to a linear push-pull amp, we see the following advantages of the H-bridge:

- Only a unipolar power supply is required, as opposed to a bipolar power supply (plus and minus voltage in addition to ground).
- The H-bridge is driven by a PWM pulse train, which is easily generated by a microcontroller, as opposed to an analog signal.

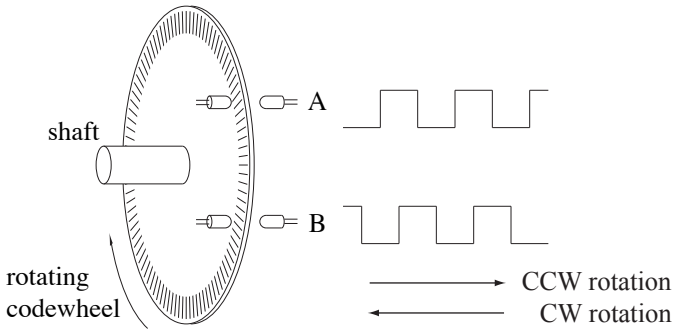


Figure 13.6: A rotating encoder creates 90 degree out-of-phase pulse trains on A and B using LEDs and phototransistors.

- Transistors spend most of their time in the off or saturated mode, with relatively small voltage drop across them, so that relatively little power is dissipated as heat by the transistors. This is in contrast to the linear amp, where the output transistors usually operate in the linear regime with significant voltage drops across them.

Linear amps are sometimes used in motor control when analog control signals and a bipolar power supply are available, and power dissipation and heat are not a concern. They are also preferred for better sound quality in speaker applications. There are many improvements to, and variations on, the basic circuit in Figure 13.5, 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. Since we can't hear sound below 20 Hz, and low-frequency currents simply heat up the speaker coils without producing audible sound, audio amplifiers typically cut off input frequencies below 10 Hz.

13.2 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.

13.2.1 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 is more common and is illustrated in Figure 13.6. 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 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 two out-of-phase 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). 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.

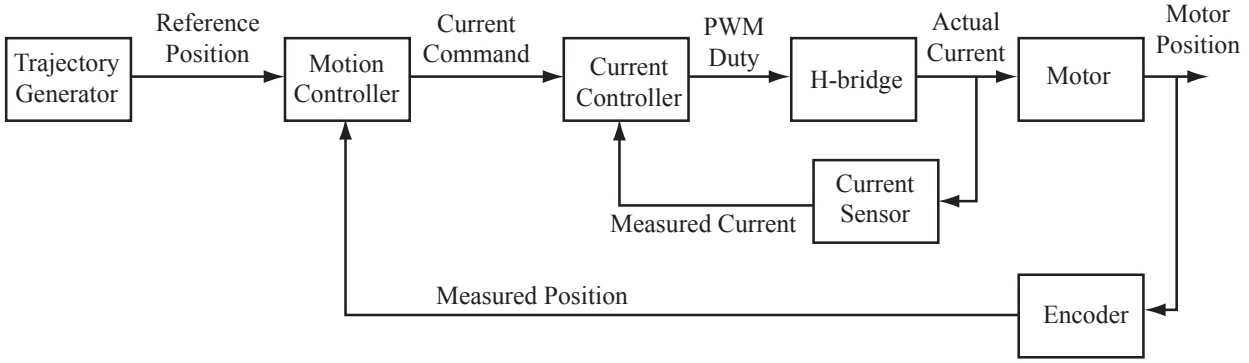


Figure 13.7: A block diagram for motion control.

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. In ME 333, we use a small PIC dedicated to counting encoder pulses and read the encoder count using SPI.

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

13.2.2 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 when you started watching the encoder signals. 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:

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.

13.3 Motion Control of a DC Motor

An example block diagram for control of a DC motor is shown in Figure 13.7.[1] A trajectory generator creates a reference position as a function of time. To drive the motor to follow this reference trajectory, we use two

[1] 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 13.7 is more typical of industrial implementations.

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 (derived from the 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 mechatronics engineers, so you will do both.

13.3.1 Motion Control

Feedback Control

Let θ and $\dot{\theta}$ be the actual position and velocity of the motor, and θ_d and $\dot{\theta}_d$ be the desired position and velocity. Define the error $e = \theta_d - \theta$, error rate of change $\dot{e} = \dot{\theta}_d - \dot{\theta}$, and error integral $e_{\text{int}} = \Delta t \sum_k e(k)$, where Δt is the controller time step. Then a reasonable choice of controller would be a *PID (proportional-integral-derivative) controller*,

$$I_{c,fb} = k_p e + k_i e_{\text{int}} + k_d \dot{e}, \tag{13.1}$$

where $I_{c,fb}$ is the commanded 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 is a little harder to interpret, but its job is to create a force proportional to the time integral of error. So if the motor has been sitting a long time with an error in angle, the error integral builds up and therefore the force builds up to try to overcome the error.

Pseudocode for the controller is given below, where `dt` is the controller timestep:

```
eprev = 0;                      // initial "previous error" is zero
eint = 0;                       // initial error integral is zero
now = 0;                        // "now" counts the timesteps

while(1) {                      // enter the control loop
  actual = readEncoder();       // get actual motor angle
  desired = desiredAng(now);    // get desired angle for time "now"
  e = desired - actual;         // calculate the angle error now
  edot = (e - eprev)/dt;        // estimate the error rate of change
  eint = eint + e*dt;           // update the integral of error
  u = kp*e + ki*eint + kd*edot; // calculate the control (current)
  sendCurrent(u);               // send the current to the motor
  eprev = e;                    // current error is now previous error
  now = now + 1;                // increment the time
}
```

To tune the controller, start with $k_d = k_i = 0$ and focus on finding a proportional gain k_p that provides approximate tracking without too much oscillation. Then add in a nonzero derivative gain k_d to damp out oscillations that arise from the pure spring-like proportional controller. The two gains can then be tuned together to further improve tracking of the desired trajectory. Typically higher gains yield better tracking,

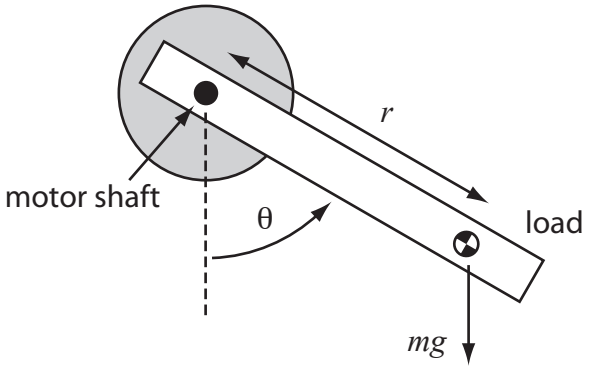


Figure 13.8: An unbalanced load in gravity.

but if the gains are too high, then the controller becomes unstable due to repeated overcompensation for errors.

PD (proportional-derivative) control, i.e., PID control with $k_i = 0$, is a common choice for motor position control. A nonzero k_i can be added at the end of the tuning process to try to compensate any persistent (steady-state) errors.

Other issues are discussed below.

- **Integrator anti-windup.** Imagine that the motor is prevented from moving for some time. During this time, the integrator error e_{int} builds up to a large value. This then causes the controller to try to dissipate the integrated error by causing error of the opposite sign. To limit the oscillation caused by this effect, the integrator error should be upper bounded. This can be implemented by adding two lines to the code above:

```
eint = eint + e*dt;                    // update the integral of error
if (eint > EINTMAX) eint = EINTMAX;    // ADDED:  integrator anti-windup
if (eint < -EINTMAX) eint = -EINTMAX;  // ADDED:  integrator anti-windup
```

  This is called *integrator anti-windup*.

- **Velocity estimates.** An estimate of velocity can be obtained from encoder position data by finite differencing, as in the pseudocode above. This may lead to a very jumpy, low-resolution signal, however, particularly if the encoder's resolution is not high. Digital low-pass filtering or smoothing of the estimate may be a good idea, perhaps by averaging `edot` estimates over the past several cycles. An alternative is to use a direct velocity measurement from a tachometer.

- **Velocity control.** When the goal is to track a desired velocity, then `actual` and `desired` in the pseudocode above are the actual and desired velocity, respectively. Just as a PD controller is common for motor position, the most common velocity controller is a PI controller ($k_d = 0$). Notice that the integral term in a PI velocity controller is equivalent to the proportional term in a PD position controller, and the proportional term in a PI velocity controller is equivalent to the derivative term in a PD position controller.

Feedforward Plus Feedback Control

A PID controller is called a *feedback* controller, since it uses sensor feedback. You could instead try a *feedforward* controller. A feedforward controller uses only a model of the system and the desired trajectory to choose a commanded current. For example, for an unbalanced load as in Figure 13.8, you could choose your current command to be

$$I_{c,ff} = \frac{1}{k_m}(J\ddot{\theta}_d + mgr\sin\theta + b_0\,\text{sgn}(\dot{\theta}) + b_1\dot{\theta}),$$

where k_m is the motor constant, J is the motor and load inertia, $\ddot{\theta}_d$ can be obtained by finite differencing the desired trajectory, mg is the weight of the load, r 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.

Feedforward control alone will never yield acceptable performance, as no model will be sufficiently accurate. However, feedback control can only respond to errors. Why wait for effects you can model to manifest themselves as error? If you have a good model of the mechanical properties of your system and you use a commanded current $I_c = I_{c,fb} + I_{c,ff}$, you should be able to get better tracking control than you can by either controller alone.

13.3.2 Current Sensing and Current Control

Current Sensor

We will use a 15 mΩ current-sensing resistor in conjunction with a MAX9918 current-sense amplifier to measure the current flowing through the motor. The current-sensing resistor is placed in series with the motor, so that a current of 1 A through the motor results in a 15 mV drop across the resistor. This voltage is then amplified and level shifted according to the user's choice of external resistors. (See the datasheet for more information.) Finally, the amplifier signal should be low-pass filtered (with a cutoff of a few hundred to 1000 Hz) to filter out current ripple due to PWM switching.

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

$$V_c = k_V I_c.$$

This would be a good choice if your load were only a resistance. Even if not, if you do not have a good mechanical model of your system, achieving a 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.

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 much larger if you do not measure the actual current in your current controller. More sophisticated current controller choices might be a mixed model-based and I feedback controller

$$V_c = I_c R + k_m \dot{\theta} + k_{I,i}\ e_{I,\text{int}}$$

or a PI feedback controller

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

where e_I is the error between the commanded current I_c and the measured current, $e_{I,\text{int}}$ is the integral of current error, R is the motor resistance, k_m is the motor 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.

13.3.3 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 13.9. 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 13.7. (Note: the duty cycle of a PWM input can be determined using the Input Capture peripheral on the PIC32, which we haven't discussed.)

Figure 13.9: The Copley Controls Accelus amplifier.

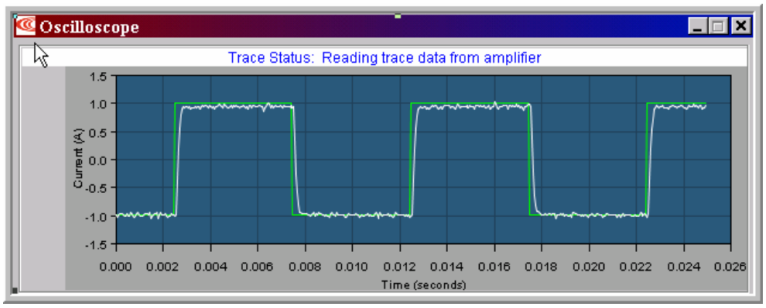


Figure 13.10: A plot of the reference square wave current and the actual measured current during PI current controller tuning.

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. 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 your PC, 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 current control loop.** Set a limit on the integrated current to avoid overheating the motor. This limit is based on the integral $\int I^2 dt$, which is related to how much energy the 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 the PI current controller. This tuning is assisted by plots of reference and actual currents as a function of time. See Figure 13.10. 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 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.

Chapter 14

Stepper Motors and RC Servos

Stepper motors and RC servo motors are options that do not require external feedback control. Stepper motors take discrete steps when their control inputs are changed, and they are commonly used in devices like copiers, scanners, and plotters where the load is predictable. This predictability ensures that the motor takes a step each time the control inputs are changed, eliminating the need for feedback.

For RC servos, the feedback control is built in. A control signal specifies the desired position of the servo, and internal feedback control achieves the position.

14.1 Stepper Motors

The basic operation of a stepper motor is shown in Figure 14.1. Current flows through coils 1 and 2 of the stator. Current through coil 1 creates an electromagnet of one sign (e.g., N) on coil 1 and the opposite sign (e.g., S) on its partner coil $\bar{1}$; coil 2 operates similarly. These magnetic fields attract permanent magnets on the rotor, causing the rotor to rotate (step) to a new equilibrium position. As the currents through the electromagnets proceed through a fixed sequence, the motor steps around.

A complete full-stepping sequence for the coils is shown below:

coil 1	+	−	−	+	+ (back to the beginning)
coil 2	+	+	−	−	+ (back to the beginning)

We also have the option of *half-stepping*, where the current to one of the coils is turned off before transitioning to the next full-step state. This doubles the number of steps we have available, but results in reduced holding torque at the half-steps. A complete half-stepping sequence is given below:

coil 1	+	0	−	−	−	0	+	+	+ (back to the beginning)
coil 2	+	+	+	0	−	−	−	0	+ (back to the beginning)

Figure 14.1(b) shows two half-steps making a full step.

The simple stepper shown in Figure 14.1(b) has only two *rotor poles*, and a full rotation of the motor consists of only four full steps (or eight half steps). Real stepper motors have many more rotor poles and *stator poles*, but always only two independently controlled sets of coils that energize these electromagnetic stator poles. The rotor poles are teeth that are attracted to stator pole teeth, and a large number of rotor teeth create much finer step sizes than the 90° of our simple example. For a rotor with N teeth, a full revolution of the rotor corresponds to $2N$ full steps, or $360°/(2N)$ degrees per step. Figure 14.2 shows a stepper motor that has been opened up, exposing the stator poles and rotor poles.

Stepper motors are characterized by the resistance and inductance of their coils; the current each coil can carry continuously without overheating (usually specified by the coil voltage); the *holding torque*, the amount of torque the rotor can resist before being moved out of its equilibrium position when the coils are energized; and the *detent torque*, the amount of torque needed to move the rotor out of its equilibrium position when the coils are off. The detent torque comes from the attraction of the permanent magnets to the stator teeth.

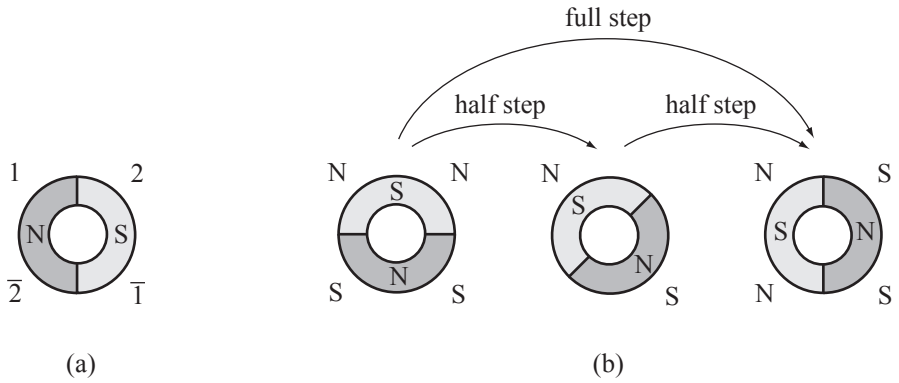


Figure 14.1: (a) A stepper motor with two rotor poles and four stator poles. The four stator poles correspond to two independent coils, 1 and 2. (b) The stepping sequence. As the coils change from current flowing one way, to off, to the other way, the rotor rotates to new equilibrium positions.

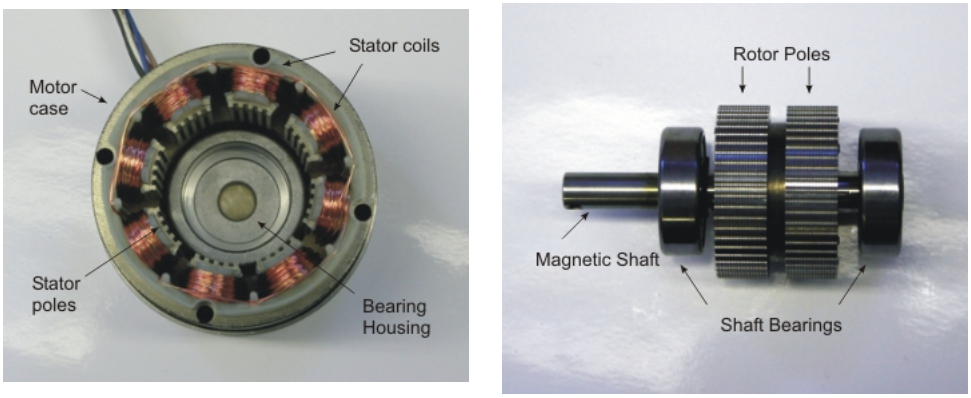


Figure 14.2: (Left) A stepper motor stator, showing the coils creating the electromagnets and the stator pole teeth. (Right) A stepper motor rotor, removed from the motor, showing the rotor pole teeth.

When a stepper is stepped slowly, it settles into its equilibrium position between steps. When the stepper is stepped quickly, it may never settle before the coils change, leading to continuous motion. If a stepper motor is stepped too quickly, it will not be able to keep up. Similarly, if a stepper is moving at a high speed and the stepping sequence stops suddenly, inertia may cause it to continue to rotate. Once either of these happens, it is impossible to know the angle of the rotor without an external sensing device, like an encoder. The speed at which the motor can be stepped reliably decreases as the torque it must provide increases. To ensure that the stepping sequence is followed, particularly in the presence of a significant load, it is a good idea to ramp up the stepping frequency at the beginning of a motion, and ramp down the frequency at the end, to place limits on the acceleration and deceleration of the load.

To save power when the stepper motor is not moving, the coils can be turned off, provided the detent torque is sufficient to prevent any unwanted motion.

Stepper motors come in two major types: bipolar and unipolar (Figure 14.3). With bipolar stepper motors, current can flow either direction (hence bipolar) through each coil. With unipolar stepper motors, each coil is broken into two subcoils, and current only flows one direction (hence unipolar) through each subcoil.

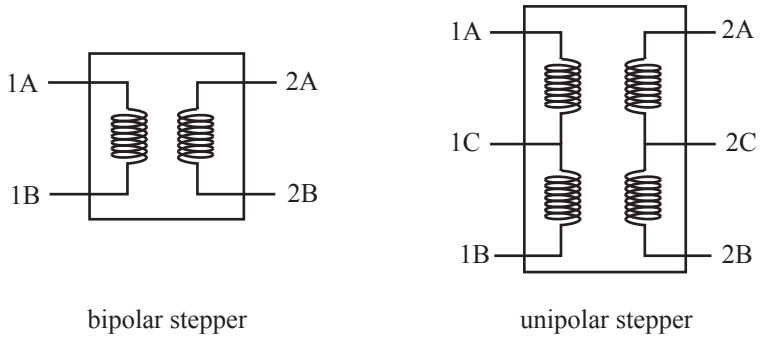


Figure 14.3: (Left) A bipolar stepper motor has four connections to the two coils. (Right) A unipolar stepper motor has six connections to the two coils, or five if 1C and 2C are combined.

14.1.1 Bipolar Stepper Motor

A bipolar stepper motor has four external wires to connect: two at either end of coil 1, and two at either end of coil 2. Let's call the two ends of coil 1 1A and 1B. To switch coil 1's current, we apply V and GND to 1A and 1B, then switch to GND and V at 1A and 1B, respectively. A bipolar stepper motor can be driven using two H-bridges, treating each coil separately as a DC motor that is driven either full forward or full backward.

If you are using an unknown bipolar stepper motor, you can determine which wires are connected to which coil using a multimeter set to resistance testing mode.

14.1.2 Unipolar Stepper Motor

A unipolar stepper motor typically has six external wires to connect, three for each coil. For coil 1, for example, the wires 1A and 1B are at either end of coil 1, as before, and the connection 1C is a "center tap." This center tap is commonly connected to the voltage V, and we switch current through the coil by alternating between 1A grounded and 1B left floating, and 1B grounded while 1A is left floating.

Five-wire unipolar stepper motors have a single center tap that is common to both coils.

One method for driving a unipolar stepper motor is to ignore the center taps, treating it as a bipolar stepper, and use one H-bridge for each coil.

If you are using an unknown unipolar stepper motor, you can determine which wires are connected to which coil using a multimeter set to resistance testing mode. The center taps are at half the resistance of the full coil resistance.

14.2 RC Servos

An RC servo consists of a DC motor, gearhead, position sensor (typically a potentiometer), and a feedback control circuit (typically implemented by a microcontroller), all in a single package (Figure 14.4). This makes an RC servo an excellent choice for high torque approximate positioning without requiring your own feedback controller. The servo's output shaft typically has a total rotation angle of less than 360 degrees, with 180 degrees being common.

An RC servo has three input connections: power (typically 5 or 6 V), GND, and a digital control signal. By convention, the control signal consists of a high pulse every 20 ms, and the duration of this pulse indicates the desired output shaft angle. A typical choice has a 0.5 ms pulse mapping to one end of the rotation range and a 2.5 ms pulse mapping to the other end of the range, with a linear relationship between pulse length and rotation angle in between (Figure 14.5).

Figure 14.4: An RC servo cutaway showing the gears and the microcontroller that receives the control signal, reads the potentiometer, and implements the feedback controller.

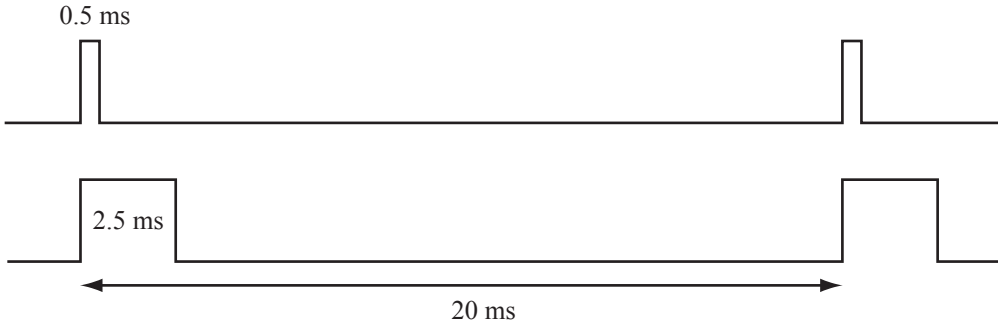


Figure 14.5: Typical RC servo control wave forms. These pulse widths typically drive the servo output shaft to the two ends of its rotation range.

Chapter 15

Digital Signal Processing

We have already used RC filters to low-pass filter high-frequency PWM signals, creating analog output signals. Filters find many other applications, like filtering out high-frequency or 60 Hz electrical noise from a measured signal, extracting high-frequency components from change-sensitive sensors, and integrating or differentiating a signal. If the signal is an analog voltage, these filters can be implemented by resistors, capacitors, and op-amps.

Filters can also be implemented in software. In this case, the signal is first converted to digital form, for example using an analog-to-digital converter to sample the signal at fixed time increments. Once in this form, a *digital filter* can be used to difference or integrate the signal, or to suppress, enhance, or extract different frequency components in the signal. Digital filters offer advantages over their analog electronic counterparts:

- No need for extra external components, such as resistors, capacitors, and op amps.
- Tremendous flexibility in the filter design. Filters with excellent properties can be implemented very easily in software.
- The ability to operate on signals that do not originate from analog voltage signals.

Digital filtering is one example of *digital signal processing* (DSP). We start this chapter by providing some background on sampled signal representation. We then provide an introduction to the *fast Fourier transform* (FFT), which can be used to decompose a digital signal into its frequency components. The FFT is among the most important and heavily used algorithms in video, audio, and many other signal processing and control applications. We then discuss a class of digital filters called *finite impulse response* (FIR) filters, which calculate their output values as weighted sums of their past input samples. We conclude with a brief description of *infinite impulse response* (IIR) filters, which calculate their output as weighted sums of their past inputs and outputs, and FFT-based filters.

This chapter is meant to provide a brief introduction and some practical hints on how to use FFTs, FIR filters, and IIR filters. We skip most of the mathematical underpinnings, which are covered in books and courses focusing solely on signal processing.

15.1 Sampled Signals and Aliasing

Let $x(t)$ be a periodic signal as a function of (continuous) time with period T ($x(t) = x(t+T)$), and therefore frequency $f = 1/T$. It can be shown that any periodic signal $x(T)$ can be expressed as a *Fourier series*, and particularly as the sum of a DC (constant) component and an infinite sequence of sinusoids at frequencies f, $2f$, $3f$, etc.:

$$x(t) = A_0 + \sum_{k=1}^{\infty} A_k \sin(2\pi k f t + \phi_k). \tag{15.1}$$

Thus the T-periodic signal $x(t)$ can be uniquely represented by the amplitudes $A_0, A_1, \ldots$ and the phases $\phi_1, \phi_2, \ldots$ of the component sinusoids. This is called the *frequency domain* representation of $x(t)$.

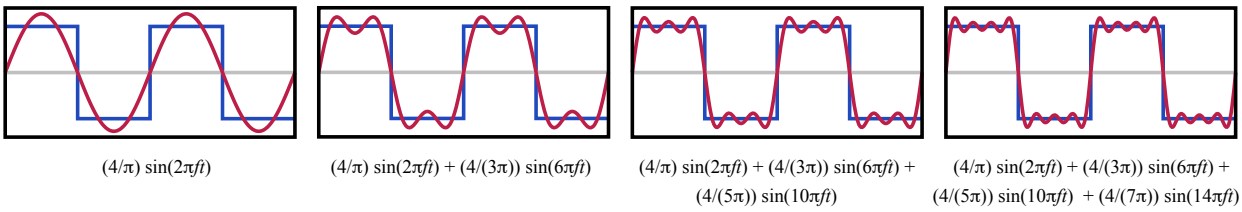


Figure 15.1: An illustration of the sum of the first four nonzero frequency components of the Fourier series of a square wave. The sum converges to the square wave as higher frequency components are included in the sum.

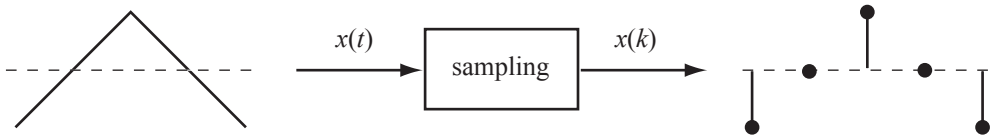


Figure 15.2: The sampling module converts the continuous-time signal $x(t)$ to a discrete-time signal $x(k)$.

An example is a square wave signal that swings between $+1$ and -1 at frequency f and 50% duty cycle. The Fourier series that creates this signal is given by $A_k = 0$ for even k and $A_k = 4/(k\pi)$ for odd k, with all phases $\phi_k = 0$. Figure 15.1 illustrates the first four components of the square wave.

In DSP, the first thing we do is sample the continuous-time signal $x(t)$ at time intervals T_s (or sampling frequency $f_s = 1/T_s$), yielding the samples $x(k) \equiv x(kT_s) = x(t)$ for $k = 0, 1, 2, \ldots$, as shown in Figure 15.2. The sampling process also quantizes the signal; for example, the PIC32's ADC module converts continuous voltage levels to one of 1024 levels. While quantization is an important consideration in DSP, in this chapter we will ignore quantization effects and assume the $x(k)$ can take arbitrary real values.

Suppose the original analog input signal is a sinusoid

$$x(t) = A\sin(2\pi ft + \phi),$$

where f is the frequency, $T = 1/f$ is the period, A is the amplitude, and ϕ is the phase. Given samples $x(k)$, and knowing the input is a sinusoid, it is possible to use the samples to uniquely determine A, f, and ϕ of the underlying signal, provided f is sufficiently low. As we increase the signal frequency f beyond $f_s/2$, however, something interesting happens, as illustrated in Figure 15.3. The sampling process renders a signal of frequency $f_1 = f_s/2 + \Delta, \Delta > 0$, with phase ϕ_1, indistinguishable from a signal with lower frequency $f_2 = f_s/2 - \Delta$ with phase ϕ_2. For example, for $\Delta = f_s/2$, an input signal of frequency $f_1 = f_s/2 + \Delta = f_s$ looks the same as a constant (DC) input signal ($f_2 = f_s/2 - \Delta = 0$), because our once-per-cycle sampling will return the same value each time.

The phenomenon of signals of frequency greater than $f_s/2$ "posing" as signals of frequency between 0 and $f_s/2$ is called *aliasing*. The frequency $f_s/2$, the highest frequency we can uniquely represent with a discrete-time signal, is known as the *Nyquist frequency* f_N. The relationship between input sinusoids of arbitrary frequency and the apparent frequency of the sampled signal is

actual frequency ($0 \le \Delta < f_N,\ k = 0, 1, 2, \ldots$)	apparent frequency
$2kf_N + \Delta$	Δ
$(2k+1)f_N + \Delta$	$f_N - \Delta$

Because we cannot distinguish higher-frequency signals from lower-frequency signals, we only concern ourselves with input frequencies in the range $0 \ldots f_N$. If the sampled signal is obtained from an analog voltage, it is common to put an analog low-pass *anti-aliasing* filter before the sampler to remove frequency components greater than f_N.

Aliasing is familiar from the spinning wheel optical illusion. Your eyes track a mark on the wheel as it speeds up at a constant rate, and initially you see the wheel spinning forward faster and faster (i.e., the

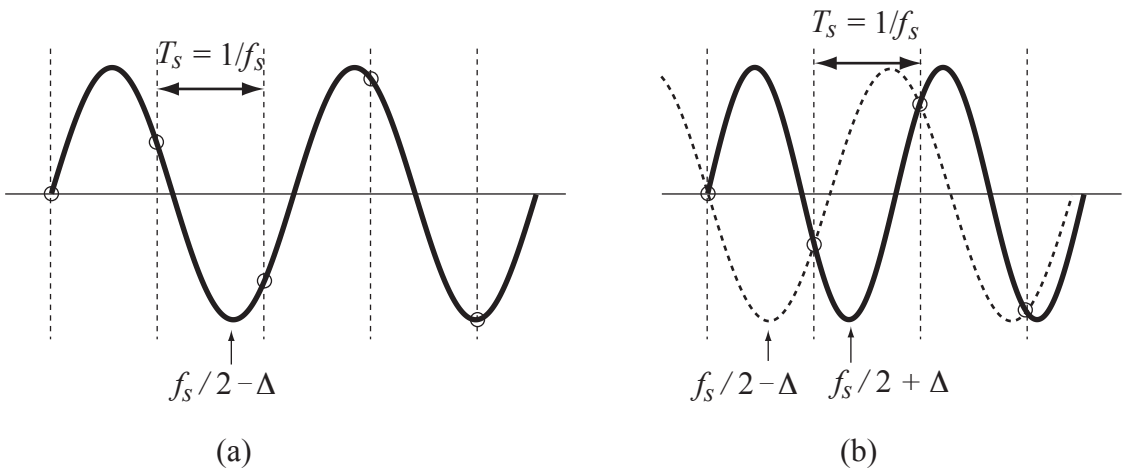


Figure 15.3: (a) The underlying sinusoid $x(t)$ with frequency $f = f_s/2 - \Delta, \Delta > 0$, can be reconstructed from its samples $x(k)$, shown as circles. (b) An input sinusoid of frequency $f = f_s/2 + \Delta$, however, appears to be a signal of frequency $f = f_s/2 - \Delta$ with a different phase.

input signal frequency is increasing). The motion becomes a blur, but as the wheel continues to speed up, it eventually appears to be rotating backward at a high speed. As its actual forward speed increases further, the apparent reverse speed begins to slow, until eventually the wheel appears to be at rest again. This effect is determined by the effective "sampling frequency" f_s of your visual system. As the wheel increases speed at a constant rate, the apparent input frequency of the signal increases linearly from 0 to $f_N = f_s/2$, then decreases linearly from f_N to 0, since signals at frequency $f_N + \Delta$ appear to have the same frequency as signals at $f_N - \Delta$. (The aliasing effect is even more obvious in a dark room with a strobe light at a fixed frequency.)

15.2 The Fast Fourier Transform

Given a sampled signal $x(k)$, we would like to determine its frequency representation. To do this, we calculate the *discrete Fourier transform* (DFT) of $x(k)$. For example, say that we've collected an even number of samples N at the sampling frequency $f_s = 1/T_s$. Then we can use the DFT to calculate $N/2 + 1$ amplitudes $A_k, k = 0 \ldots N/2$, and $N/2$ phases $\phi_k, k = 1 \ldots N/2$, corresponding to the magnitude and phase of sinusoid components at frequencies kf_s/N. The frequency spacing between sinusoidal components is $f_s/N = 1/(NT_s)$.

Of several numerical methods to compute the DFT, by far the most popular is the highly efficient *Fast Fourier Transform* (FFT). Many forms of the FFT require that N be a power of 2. To reach the next power of 2, we can simply "pad" our signal with "virtual" samples of value zero at the end. This is called "zero padding."

The mathematics of the DFT (and FFT) implicitly assume that the signal repeats every N samples. Thus, depending on the original analog signal and how it was sampled, the DFT may have a significant component at the lowest nonzero frequency, f_s/N, even if this frequency is not present in the original signal. This suggests the following:

- If the original signal is actually known to be periodic, our N samples should contain several complete cycles of the signal. This reduces the "phantom" amplitude at f_s/N and ensures that the lowest frequency component of the actual signal is at a frequency sufficiently higher than (isolated from) the frequency f_s/N. We should also have sufficiently many samples per cycle of the original signal so that we can adequately distinguish the different frequency components in the actual signal.

- If the original signal is not periodic, zero padding can be used to isolate the lowest nonzero frequency component of the actual signal from f_s/N.

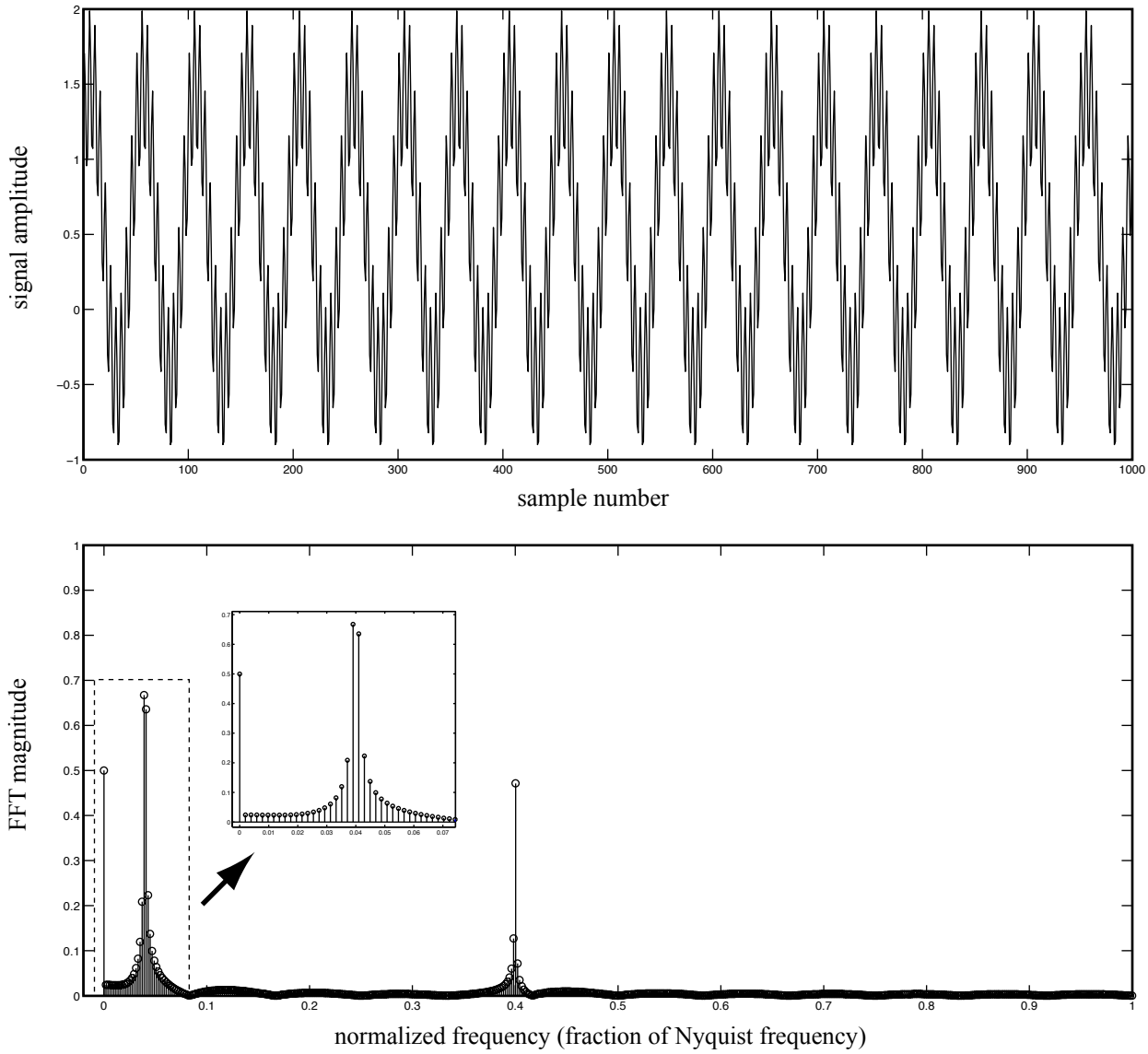


Figure 15.4: (Top) The original sampled signal. (Bottom) The FFT magnitude plot, with a portion of it magnified.

As an example, assume an underlying analog signal

$$x(t) = 0.5 + \sin(2\pi(20 \text{ Hz})t + \pi/4) + 0.5\sin(2\pi(200 \text{ Hz})t + \pi/2),$$

with components at DC, 20 Hz, and 200 Hz. We collect 1000 samples at $f_s = 1$ kHz (0.001 s intervals) and zero pad to get $N = 1024$. The signal and its FFT magnitude plot is shown in Figure 15.4. The magnitude components are spaced at frequency intervals of f_s/N, or 0.9766 Hz. The DC, 20 Hz (0.04 f_N), and 200 Hz (0.4 f_N) components are clearly picked out, though the numerical procedure has spread the 20 Hz component over several nearby frequencies so that the peak magnitude is less than 1, the amplitude of the actual 20 Hz component.

The phase plot is not shown, as the phase for small amplitude components (away from 20 Hz and 200 Hz) is meaningless.

The FFT itself takes $N = 2^n$ samples $x(k)$ as input and produces N complex coefficients $Y(k)$ as output, where k is a frequency index. These complex coefficients can be processed to produce magnitude plots as

shown in Figure 15.4. Below we discuss how to do this specifically in Matlab and with the PIC32.

15.2.1 Matlab

Given an even number N of samples in a row vector `x = [x(1) ... x(N)]` in Matlab, the command

```
Y = fft(x);
```

returns an N-vector `Y = [Y(1) ... Y(N)]` of complex numbers corresponding to the amplitude and phase at different frequency components. Let's try an FFT of $N = 200$ samples of a 50% duty cycle square wave, where each period consists of 10 samples equal to 2 and 10 samples equal to 0 (i.e., the square wave of Figure 15.1 plus a DC offset of 1). The frequency of the square wave is $f_s/20$, and our entire sampled signal consists of 10 full cycles. According to Figure 15.1, the square wave consists of frequency components at $f_s/20$, $3f_s/20$, $5f_s/20$, $7f_s/20$, etc. Thus we expect the frequency domain magnitude representation to consist of the DC component and these nonzero frequency components.

Let's build the signal and plot it (Figure 15.5(a)):

```
x=0;   % clear any array that might already be in x
x(1:10) = 2; x(11:20)= 0; x = [x x x x x x x x x x]; N = length(x);
plot(x,'Marker','o');  axis([-5 205 -0.1 2.1]);
```

Now let's take the FFT of it:

```
Y = fft(x);
```

Inspecting some of the entries of `Y`, we see that they are generally complex numbers. Since we are focusing on magnitude plots, let's plot the magnitudes of `Y`:

```
stem(abs(Y)); axis([-5 205 -10 210]);
```

The plot is shown in Figure 15.5(b). The entry `Y(1)` corresponds to the DC component discovered in the signal, while the entries `Y(2)` and `Y(N)` are equal and correspond to f_s/N, `Y(3)` and `Y(N-1)` are equal and correspond to $2f_s/N$, and so on, until finally entry `Y(N/2+1)` corresponds to $f_s/2 = f_N$, the Nyquist frequency. Thus there is one entry for the DC and Nyquist frequency components, and two for all other frequencies.

To convert the `Y(k)` to a magnitude plot, we add the magnitudes corresponding to each frequency and divide by the number of samples, i.e.,

```
mag(1)     = abs(Y(1))/N;       % this is the DC component
mag(k)     = 2*abs(Y(k))/N;     % for all k = 2 .. N/2; this is the component at frequencies (k-1)*fs/N
mag(N/2+1) = abs(Y(N/2+1))/N;   % this is the Nyquist frequency component
```

The second line arises from the fact that `Y(k) = Y(N-k+1)` for `1 < k` $\leq$ `N/2`, and both of these indexes refer to the frequency `(k-1)*fs/N`, where `fs` is the sampling frequency. Here's a simple line of code that converts `Y` to the frequency component magnitudes `mag` and plots it in Figure 15.5(c):

```
mag = 2*abs(Y(1:N/2+1))/N; mag(1) = mag(1)/2; mag(N/2+1) = mag(N/2+1)/2; stem(mag(1:N/2+1));
```

Now let's rescale the horizontal axis so that frequencies are expressed as a fraction of the Nyquist frequency:

```
freqs = linspace(0,1,N/2+1); stem(freqs,mag); axis([-0.05 1.05 -0.1 1.4])
```

The final FFT magnitude plot is shown in Figure 15.5(d). Notice that the FFT very clearly picks out the frequency components at DC, $0.1f_N, 0.3f_N, 0.5f_N, 0.7f_N$, and $0.9f_N$.

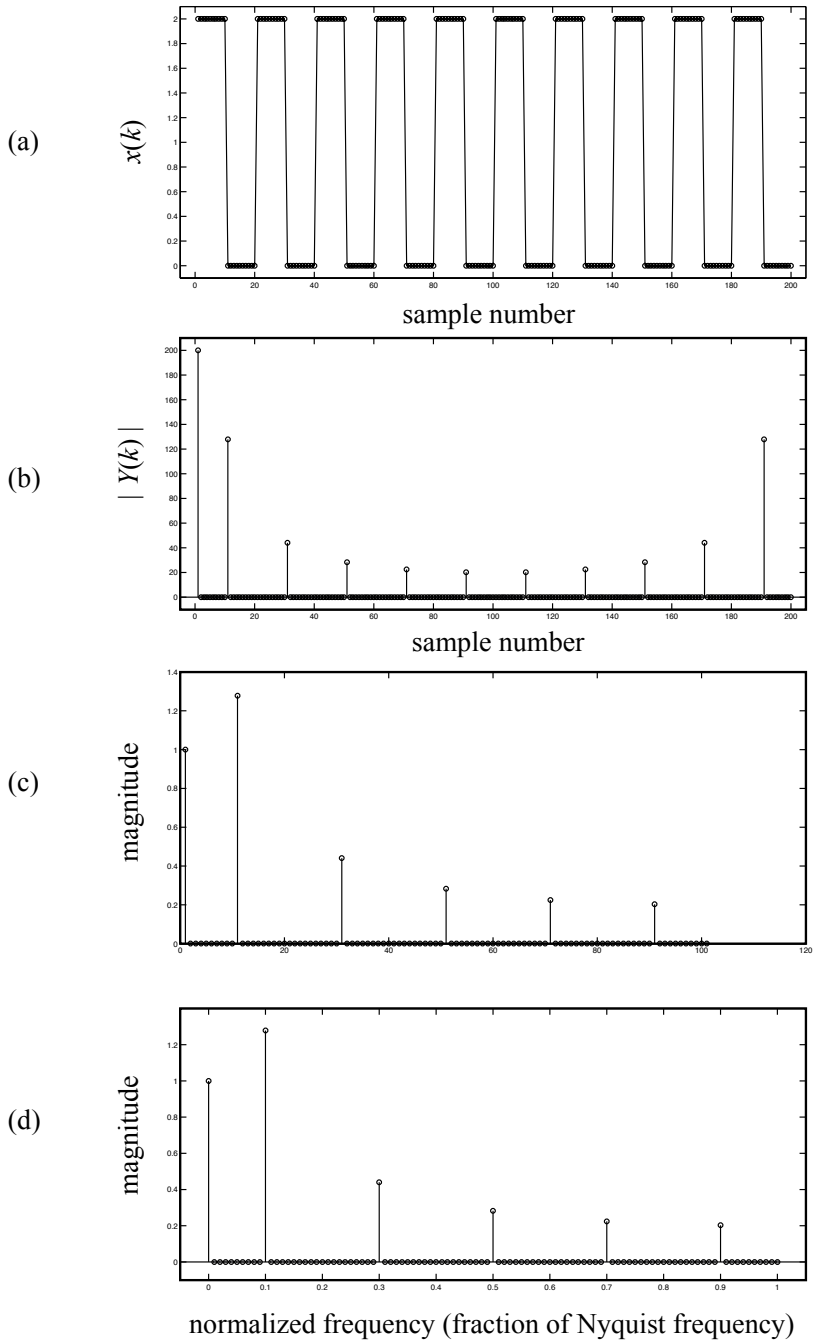


Figure 15.5: (a) The original sampled signal. (b) A plot of the magnitudes of the raw FFT output. (c) After scaling and adding the components at the same frequency. (d) After rescaling the frequency axis, we have our final FFT magnitude plot with frequencies expressed as a fraction of the Nyquist frequency.

FFT with $N = 2^n$ For efficiency reasons, on our PIC32 we will only perform FFTs on sampled signals that have a power-of-2 length. Let's do that now with Matlab. We will increase our samples from 200 to the next highest power of 2, $2^8 = 256$. We can either pad the original $x(k)$ with 56 zeros at the end, or we can take more samples in the first place.

Let's try the zero-padding option first:

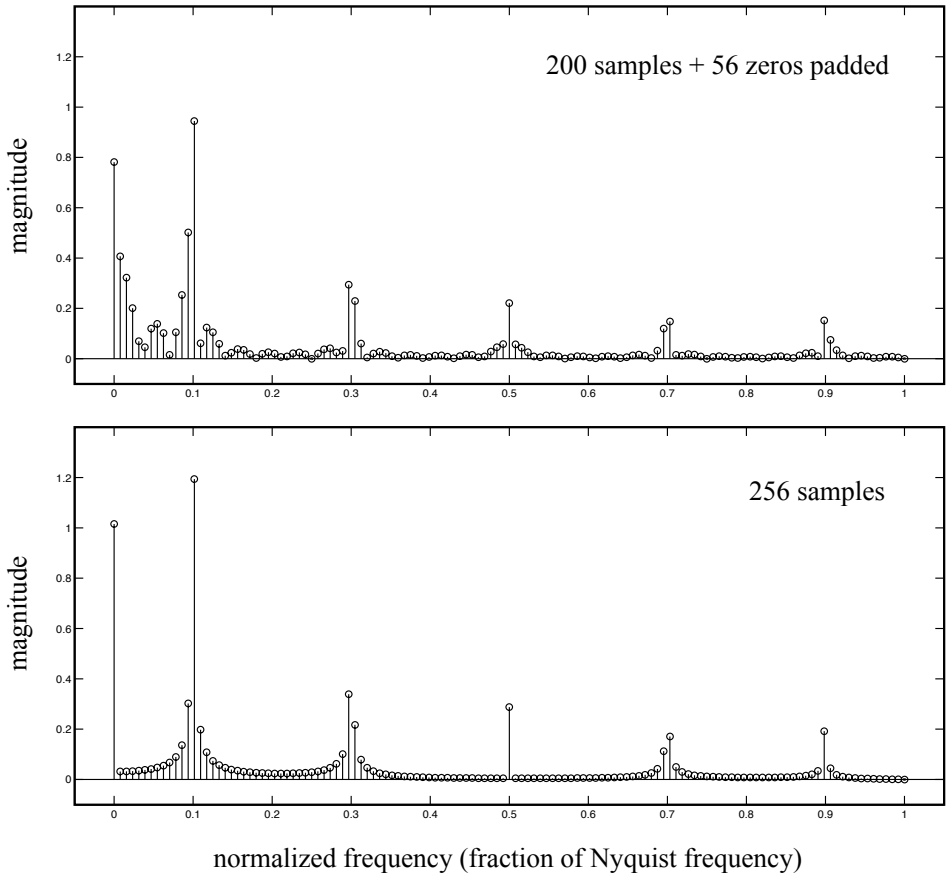


Figure 15.6: (Top) The FFT of the 200-sample square wave signal with 56 zeros padded. (Bottom) The FFT of the 256-sample square wave signal.

```
x = 0; x(1:10) = 2; x(11:20)= 0; x = [x x x x x x x x x x]; N = 2^nextpow2(length(x));
xpad = [x zeros(1,N-length(x))];
Y = fft(xpad);
mag = 2*abs(Y(1:N/2+1))/N; mag(1) = mag(1)/2; mag(N/2+1) = mag(N/2+1)/2;
freqs = linspace(0,1,N/2+1); stem(freqs,mag); axis([-0.05 1.05 -0.1 1.4]);
```

And now if the signal were sampled 256 times in the first place:

```
x = 0; x(1:10) = 2; x(11:20)= 0; x = [x x x x x x x x x x x 2*ones(1,10) zeros(1,6)];
N = length(x);
Y = fft(x);
mag = 2*abs(Y(1:N/2+1))/N; mag(1) = mag(1)/2; mag(N/2+1) = mag(N/2+1)/2;
freqs = linspace(0,1,N/2+1); stem(freqs,mag); axis([-0.05 1.05 -0.1 1.4]);
```

The results are plotted in Figure 15.6. The frequency components are still clearly visible, though the results are not as crystal clear as in Figure 15.5. A major reason for this is that the signal frequencies $0.1f_N, 0.3f_N$, etc., are not exactly represented in the FFT, as they were before. The frequency intervals are now $f_s/256$, not $f_s/200$. As a result, the FFT spreads the original frequency components across nearby frequencies rather than concentrating them in spikes at exact frequencies. This kind of spread is typical in most applications, as it is unlikely that the original signal will have component frequencies at exactly frequencies of the form kf_s/N.

Finally, we are sometimes interested in the *power* in the signal as a function of the frequency. The power is proportional to the square of the magnitude at each frequency, so we can plot the *power spectrum* using

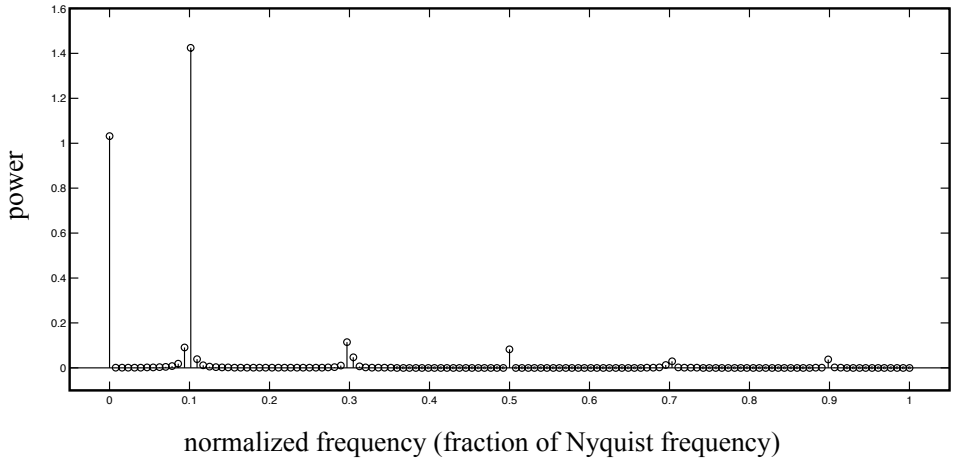


Figure 15.7: The power spectrum of Figure 15.6(bottom).

```
p = mag.^2; stem(freqs,p); axis([-0.05 1.05 -0.1 1.4]);
```

A plot of the power spectrum of the 256-sample FFT from Figure 15.6(bottom) is shown in Figure 15.7.

15.2.2 PIC32 FFT

The software library that comes with the PIC32 includes a DSP library with computationally efficient code for the FFT. For efficiency, it is written in assembly language, optimizing the number of instructions and taking advantage of primitive multiply and add operations which are not directly accessible through C. It also uses 16- and 32-bit integers to represent values, since, as we've seen, integer math is quite a bit faster than floating point math. To use the PIC32 functions correctly, you must first get your numbers in a reasonable range so that you don't lose too much resolution. This often involves multiplying all your data by some value `scale`, converting to integer types, performing the FFT, converting back to floating point, and then dividing the results by `scale`.

The DSP library also makes use of the new data types `int32c` and `int16c`, which are complex 32-bit and 16-bit numbers. The `int32c` is a `struct` with two fields, `re` and `im`, each 32-bit integers. Here's an example:

```
int32c complexnumber;
complexnumber.re = 3.2;  // set the real portion of the complex number
complexnumber.im = 1.7;  // set the imaginary portion
```

For more information on using the PIC32's FFT functions, see the sample code.

15.2.3 The Inverse Fast Fourier Transform

Given the frequency domain representation $Y(k)$ obtained from `Y = fft(x)`, the inverse FFT uses the FFT algorithm to recover the original signal $x(k)$. In Matlab, this is the procedure:

```
N = length(x); Y = fft(x); xrecovered = fft(conj(Y)/N); plot(real(xrecovered));
```

The inverse FFT is accomplished by applying `fft` to the complex conjugate of the frequency representation `Y` (the imaginary components of all entries are multiplied by -1), scaled by `1/N`. The vector `xrecovered` is equal to `x` up to numerical errors, so its entries have essentially zero imaginary components. The `real` operation ensures that they are exactly zero.

Figure 15.8: A digital filter produces filtered output $z(k)$ based on the inputs $x(k)$.

15.3 Finite Impulse Response (FIR) Digital Filters

Now that we have a basic understanding of frequency domain representations of sampled signals, we turn our attention to filtering those signals (Figure 15.8). A finite impulse response (FIR) filter produces a filtered signal $z(k)$ by multiplying the $P+1$ current and past inputs $x(k-j), j = 0 \ldots P$, by *filter coefficients* b_j and adding:

$$z(k) = \sum_{j=0}^{P} b_j x(k-j).$$

Such filters can be used for a number of operations, such as differencing a signal or suppressing low-frequency or high-frequency components. Since FIR filtering is a purely linear operation on the samples, filters in series can be performed in any order; for example, a differencing filter followed by a low-pass filter gives equivalent output to the low-pass filter followed by the differencing filter.

An FIR filter has $P+1$ coefficients, and P is called the *order* of the filter. The filter coefficients are directly evident in the *impulse response*, which is the response $z(k)$ to a unit impulse $\delta(k)$, where

$$\delta(k) = \begin{cases} 1 & \text{for } k = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The output is simply $z(0) = b_0, z(1) = b_1, z(2) = b_2$, etc. The impulse response is typically written as $h(k)$.

Since any input signal x can be represented as the sum of scaled and time-shifted impulses, e.g.,

$$x = 3\delta(k) - 2\delta(k-2),$$

and because the filter is linear, the output is simply the sum of the scaled and time-shifted impulse responses,

$$z = 3h(k) - 2h(k-2).$$

For the second-order filter with coefficients $b_0 = 3, b_1 = 2, b_2 = 1$, for example, the response to the input x is shown in Figure 15.9. When reading these signals, be aware that the leftmost samples are oldest; for example, the output $z(0)$ happened three timesteps before the output $z(3)$.

The output z is called the *convolution* of the input x and the filter's impulse response h, commonly written $z = x * h$. The filter response can be determined using Matlab's `conv` command. We collect the filter coefficients into the impulse response vector `h = b = [b0 b1 b2]` and the input into the vector `x = [3 0 -2]`, and then

```
z = conv(h,x)
```

produces `z = [9 6 -3 -4 -2]`.

"Finite Impulse Response" filters get their name from the fact that if the input goes to zero, the output goes to (exactly) zero in finite time. The shorter the filter's impulse response, the faster the output goes to zero. The output of an "Infinite Impulse Response" filter (Section 15.4) may *never* go to zero.

A filter is fully characterized by its impulse response. Often it is more convenient to look at the filter's *frequency response*, however. Because the filter is linear, a sinusoidal input will produce a sinusoidal output, and the filter's frequency response consists of its frequency-dependent gain (the ratio of the output amplitude to the input amplitude) and phase (the shift in the phase of the output sinusoid relative to the input sinusoid). To begin to understand the frequency response, we start with the simplest of FIR filters: the moving average filter.

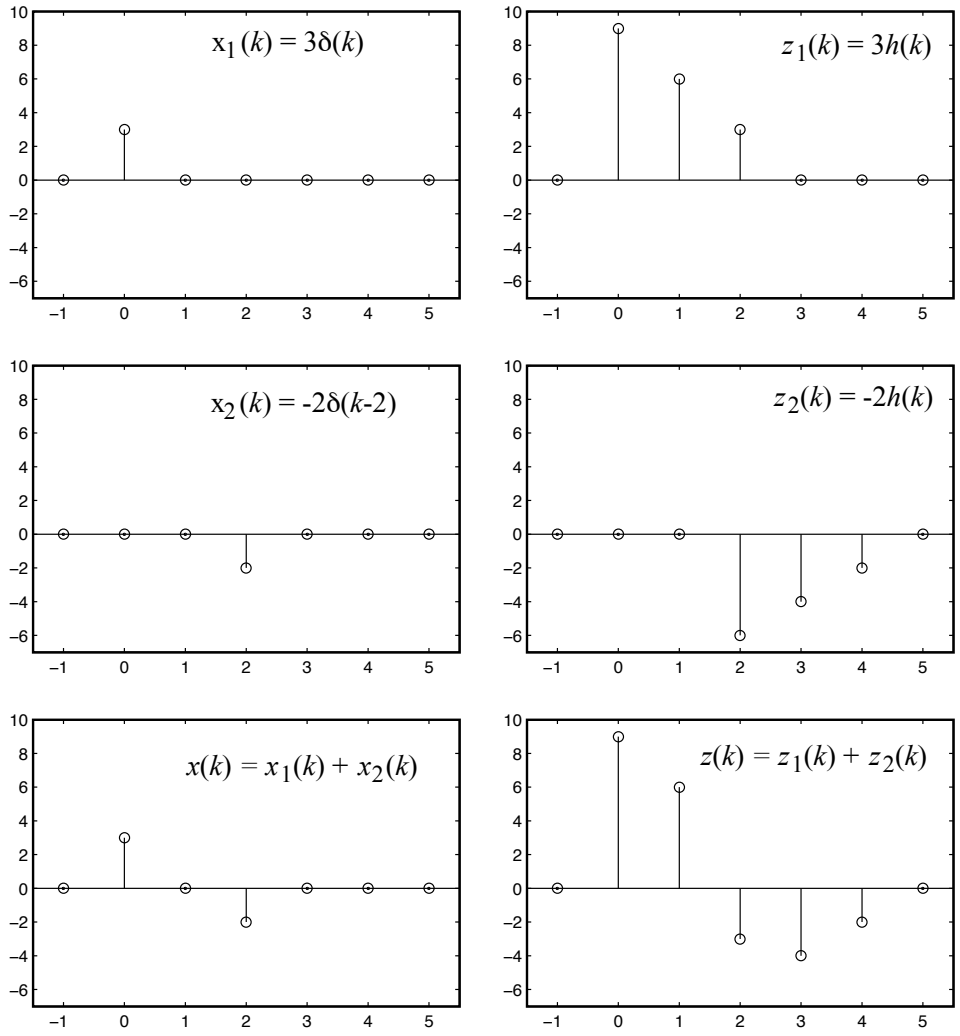


Figure 15.9: Any input signal can be expressed as the sum of scaled and time-shifted impulses, and the filter output is the sum of scaled and time-shifted impulse responses.

15.3.1 Moving Average Filter

Suppose we have a sensor signal $x(k)$ that has been corrupted by noise (Figure 15.10). We would like to find the low-frequency signal underneath the noise.

The simplest thing to try is a *moving average filter* (MAF). A moving average filter calulates the output $z(k)$ as a running average of the input signals $x(k)$,

$$z(k) = \frac{1}{P+1} \sum_{j=0}^{P} x(k-j), \tag{15.2}$$

i.e., the FIR filter coefficients are $b_0 = b_1 = \ldots = b_P = 1/(P+1)$. The output $y(k)$ is a smoothed and delayed version of $x(k)$. The more samples $P+1$ we average over, the smoother and more delayed the output. The delay is due to the fact that the output $z(k)$ is a function of only the current and previous inputs $x(k-j), 0 \leq j \leq P$. See Figure 15.10.

To find the frequency response of a third-order four-sample MAF, we can test it on some sinusoidal inputs at different frequencies (Figure 15.11). We find that the phase ϕ of the (reconstructed) output sinusoid

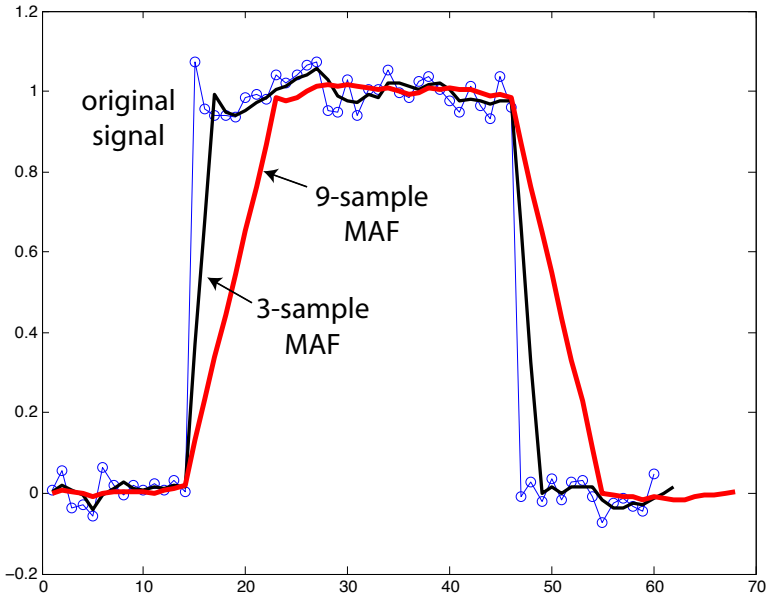


Figure 15.10: The original noisy signal with samples $x(k)$ given by the circles, the signal $z(k)$ resulting from filtering with a three-point MAF ($P = 2$), and the signal $z(k)$ from a nine-point MAF ($P = 8$). The signal gets smoother and more delayed as the number of samples in the MAF increases

relative to the input sinusoid, and the ratio G of the amplitude of their amplitudes, depend on the frequency. For the four test frequencies in Figure 15.11, we get the following table:

frequency	gain G	phase ϕ
$0.25f_N$	0.65	$-67.5°$
$0.5f_N$	0	NA
$0.67f_N$	0.25	$0°$
f_N	0	NA

Testing the response at many different frequencies, we can plot the frequency response in Figure 15.12. Two things to note about the gain plot:

- Gains are usually plotted on a log scale. This allows representation of a much wider range of gains.
- Gains are often expressed in decibels, which are related to gains by the following relationship:

$$M_{\text{dB}} = 20 \log_{10} G.$$

So $G = 1$ corresponds to 0 dB, $G = 0.1$ corresponds to -20 dB, and $G = 0.01$ corresponds to -40 dB.

Examining Figure 15.12 shows that low frequencies are passed with a gain of $G = 1$ and no phase shift. The gain drops monotonically as the frequency increases, until it reaches $G = 0$ ($-\infty$ dB) at input frequencies $f = 0.5f_N$. (The plot truncates the dip to $-\infty$.) The gain then begins to rise again, before falling once more to $G = 0$ at $f = f_N$. The MAF behaves somewhat like a low-pass filter, but not a very good one; high frequency signals can get through with gains of 0.25 or more. Still, it works reasonably well as a signal smoother.

Given a set of filter coefficients `b = [b0 b1 b2 ...]`, we can plot the frequency response in Matlab using

```
freqz(b);
```

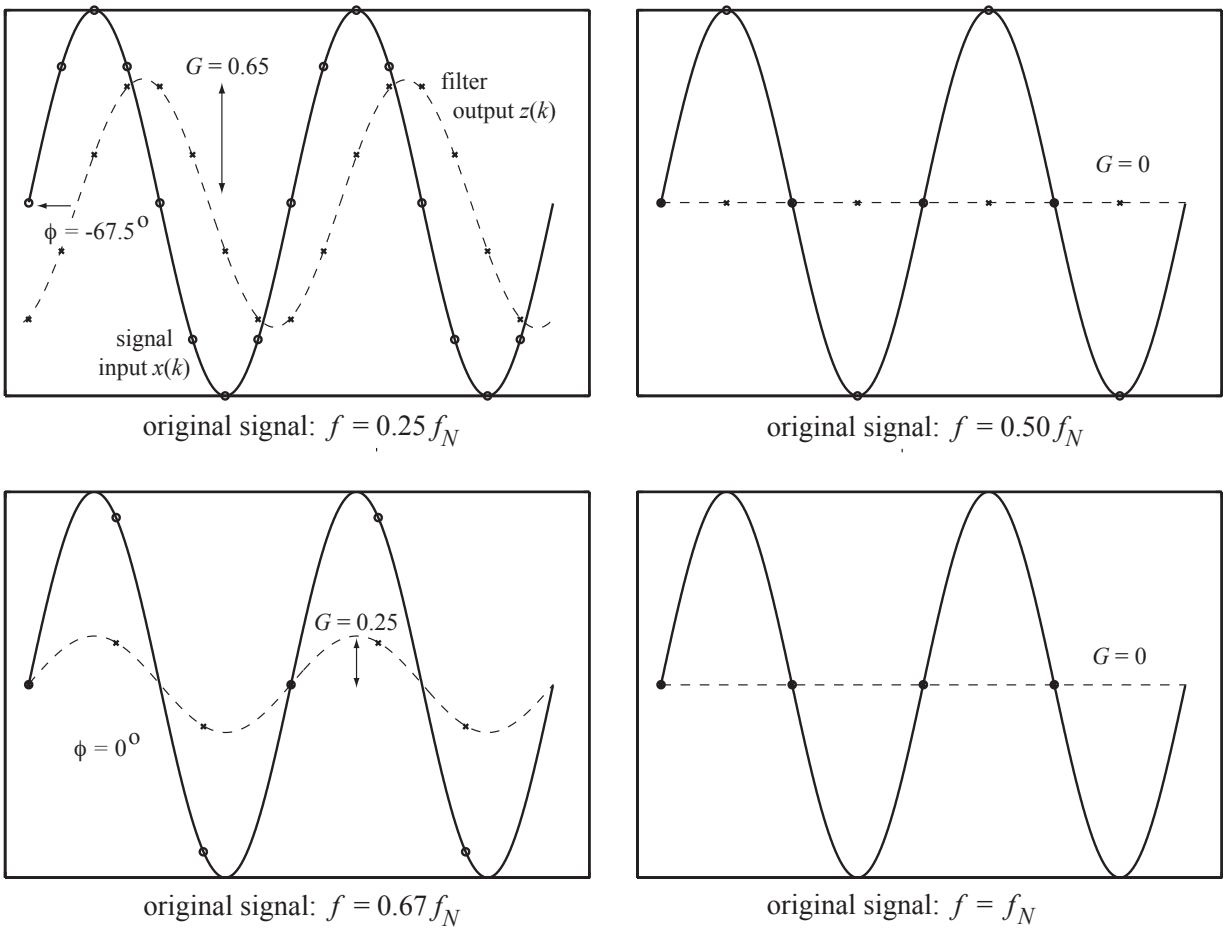


Figure 15.11: Testing the gain G of a four-sample MAF with different input frequencies.

Causal vs. Acausal Filters A filter is called *causal* if its output is the result of only current and past inputs, i.e., past inputs "cause" the current output. Causal filters are the only option for real-time implementation. If we are post-processing data, however, we can choose an *acausal* version of the filter, where the outputs at time k are a function of past as well as future inputs. This is useful for eliminating the delay associated with only using past inputs to calculate the current value. An example acausal filter is a five-sample MAF which uses the average of the past two inputs, the current input, and the next two inputs to calculate the output.

Zero Padding When a filter is first initialized, there are no past inputs. In this case we can assume the nonexistent past inputs were all zero. The output transient caused by this assumption will end at the $(P + 1)$th input.

15.3.2 FIR Filters Generally

FIR filters can be used for low-pass filtering, high-pass filtering, bandpass filtering, and bandstop (notch) filtering, among other things. Matlab provides a number of useful functions for filter design, such as `fir1` and `fdatool`. In this section we will work with `fir1`. See the Matlab documentation for more details.

A "good" filter is one that

- passes the frequencies we want with gain 1,
- highly attenuates the frequencies we don't want, and

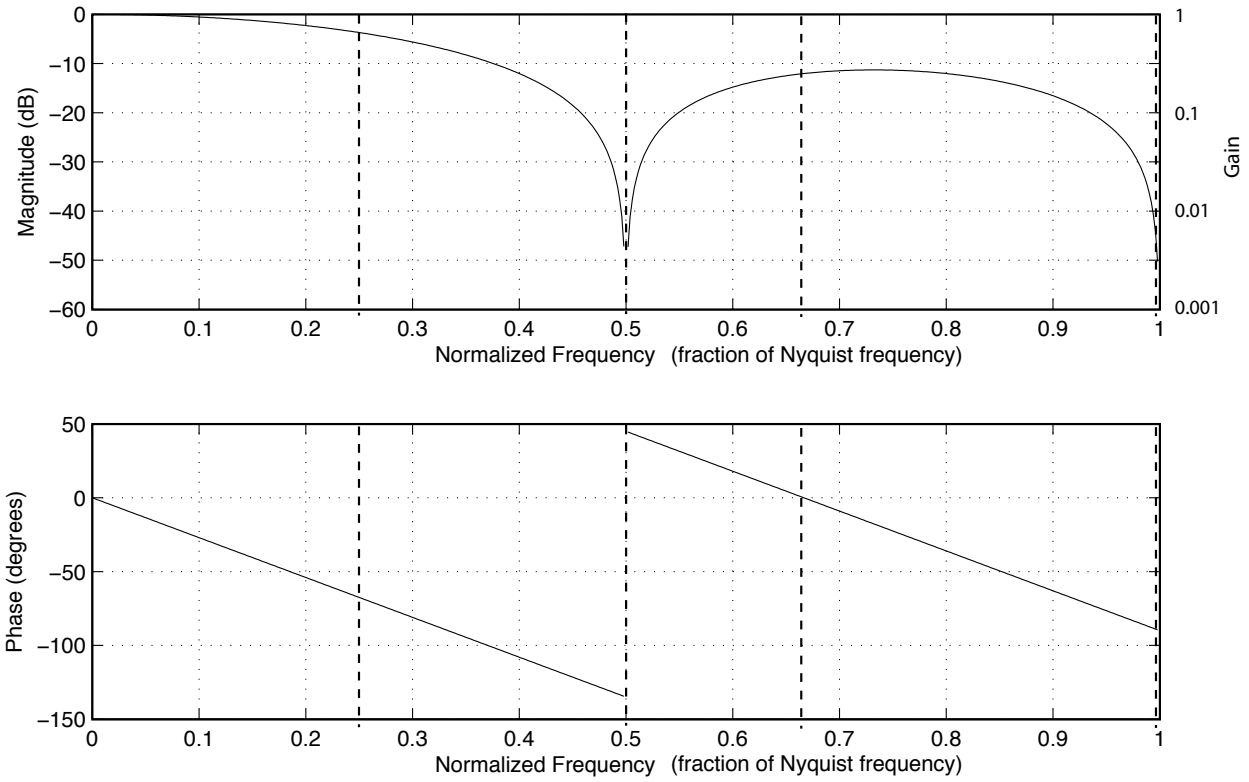


Figure 15.12: The frequency response of a four-sample MAF. Test frequencies from Figure 15.12 are shown as dotted lines.

- provides a sharp transition in gain between the passed and attenuated frequencies.

The order of the filter increases with the sharpness of the desired transition and the degree of attenuation needed in the stopped frequencies. This is a general principle: the sharper the transitions in the frequency domain, the longer the impulse response (i.e., the more coefficients that are needed in the filter). The converse is also true: the sharper the transition in the impulse response, the smoother the frequency response. We saw this with the moving average filter. It has a sharp transition between filter coefficients of 0 and $1/(P+1)$, and the resulting frequency response has only slow transitions.

High-order filters are fine for post-processing data or for non-time-critical applications such as audio applications, but they may not be appropriate for real-time control because of unacceptable delay. In audio applications, you don't care if the music arrives at your ear a tenth of a second after the bits are read from the CD.

The Matlab filter design function `fir1` takes the order of the filter, the frequencies you would like to pass or stop (expressed as a fraction of the Nyquist frequency), and other options, and returns a list of filter coefficients. Matlab considers the cutoff frequency to be where the gain is 0.5 (−6 dB). Here are some examples using `fir1`:

```
b = fir1(10,0.2);              % 10th order, 11-sample LPF with cutoff freq of 0.2 fN
b = fir1(10,0.2,'high');       % HPF cutting off frequencies below 0.2 fN
b = fir1(150,[0.1 0.2]);       % 150th order bandpass filter with passband 0.1 to 0.2 fN
b = fir1(50,[0.1 0.2],'stop'); % bandstop filter with notch at 0.1 to 0.2 fN
b = fir1(20,0.4,hann(21));     % you can use a "window" to roll off coeffs; hamming is default
```

You can then plot the frequency response of your designed filter using `freqz(b)`.

If the order of your specified filter is not high enough, you will not be able to meet your design criteria. For example, if you want a low-pass filter that cuts off frequencies at 0.1 f_N, and you only allow seven

coefficients (sixth order),

```
b = fir1(6,0.1);
```

you'll find that the filter coefficients that Matlab returns do not achieve your aims.

In the examples below we will work with a 1000-sample signal, with components at DC, $0.004f_N$, $0.04f_N$, and $0.8f_N$. The original signal `x` is plotted in Figure 15.13.

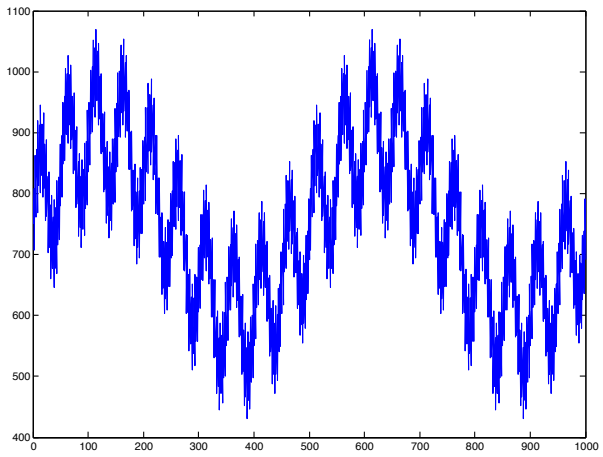


Figure 15.13: The original 1000-sample signal `x`.

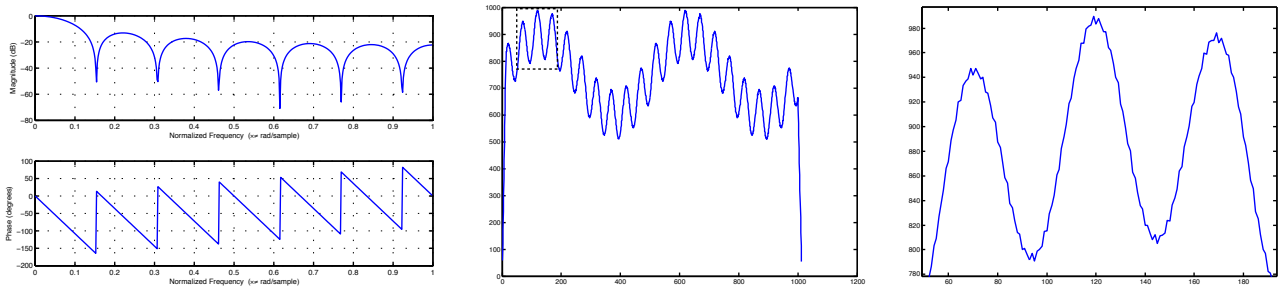


Figure 15.14: Moving average filter: `maf=ones(13,1)/13; freqz(maf); plot(conv(maf,x))`. **Left**: The frequency response of the 12th-order (13-sample) MAF. **Middle**: The result of the MAF applied to (convolved with) the original signal. Since the original signal has 1000 samples, and the MAF has 13 samples, the filtered signal has 1012 samples. (In general, if two signals of length j and k are convolved with each other, the result will have length $j + k - 1$.) This is equivalent to first "padding" the 1000 samples with 12 samples equal to zero on either end (sample numbers -11 to 0, and 1001 to 1012), then applying the 13-sample filter 1012 times, over samples -11 to 1, then -10 to 2, etc., up to samples 1000-1012. This zero-padding explains why the signal drops to close to zero at either end. **Right**: Zoomed in on the smoothed signal.

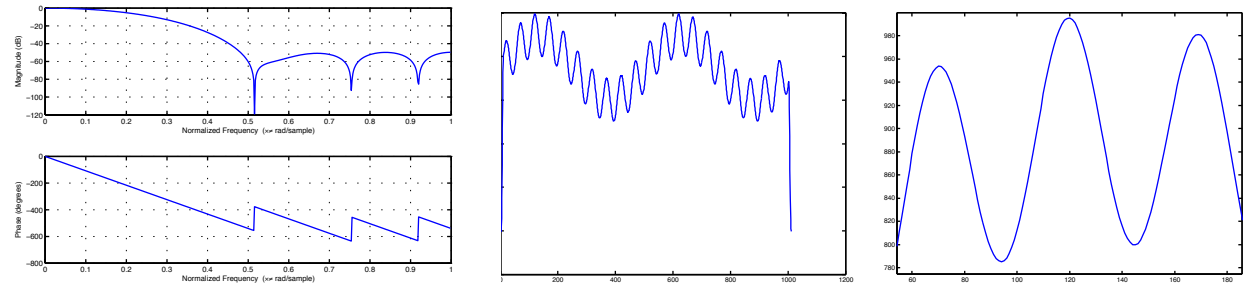


Figure 15.15: `lpf=fir1(12,0.2); freqz(lpf); plot(conv(lpf,x))`. **Left**: The frequency response of a 12th-order LPF with cutoff at $0.2 f_N$. **Middle**: The signal smoothed by the LPF. **Right**: A zoomed-in view, showing superior performance to the 12th-order MAF.

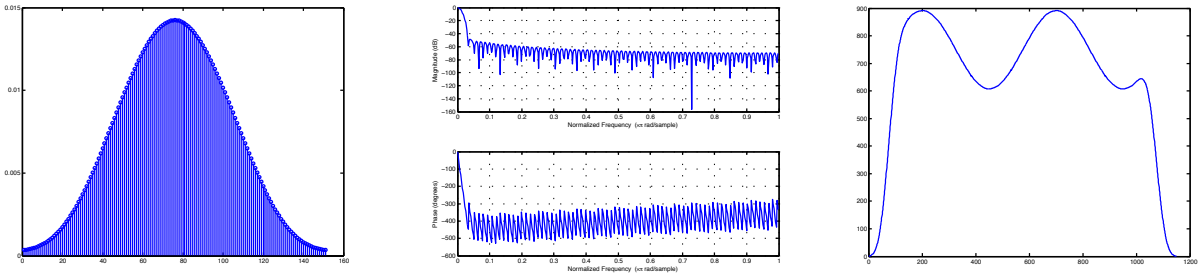


Figure 15.16: `lpf=fir1(150,0.01)`. **Left**: `stem(lpf)`. The coefficients of a 150th-order FIR LPF with a cutoff at $0.01 f_N$. **Middle**: `freqz(lpf)`. The frequency response. **Right**: `plot(conv(lpf,x))`. The smoothed signal, where only the $0.004 f_N$ and DC components get through.

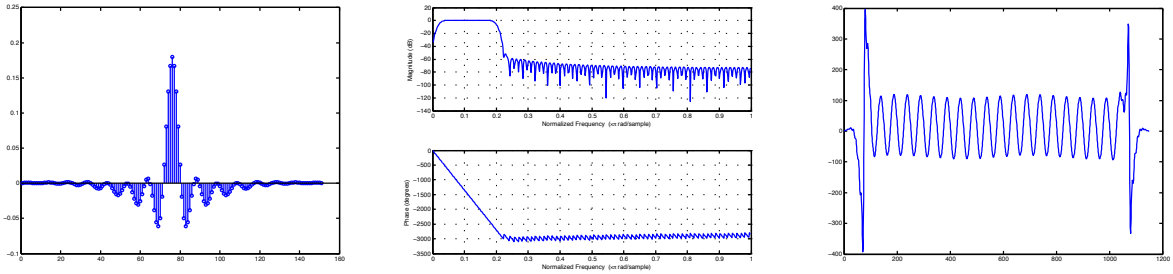


Figure 15.17: `bpf=fir1(150,[0.02,0.2])`. **Left**: `stem(bpf)`. The coefficients of a 150th-order bandpass filter). **Middle**: `freqz(bpf)`. The frequency response. **Right**: `plot(conv(bpf,x))`. The signal consisting mostly of the $0.04 f_N$ component, with small DC and $0.004 f_N$ components.

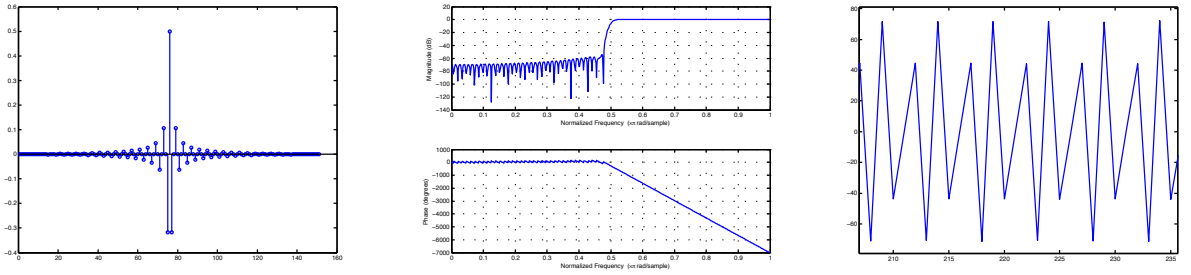


Figure 15.18: `hpf=fir1(150,0.5,'high')`. **Left**: `stem(hpf)`. The coefficients of a 150th-order high-pass filter. **Middle**: `freqz(hpf)`. The frequency response. **Right**: `plot(conv(hpf,x))`. Zoomed in on the high-passed signal.

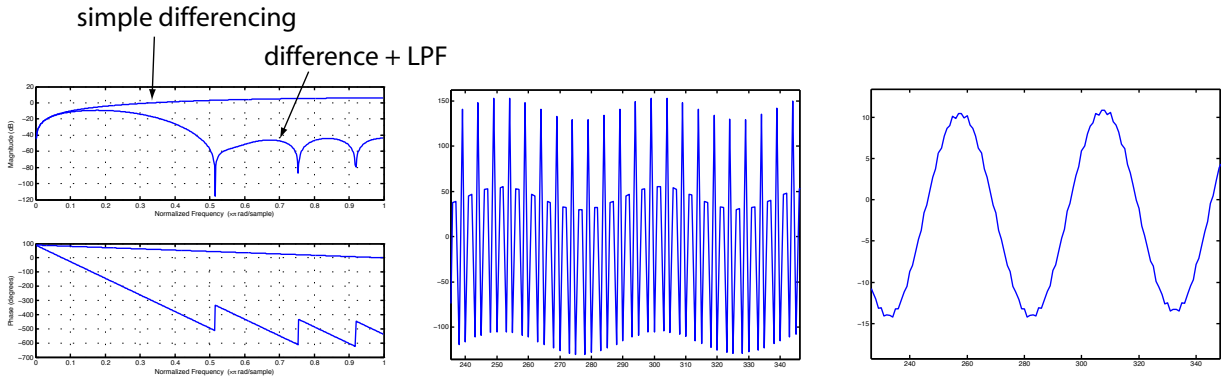


Figure 15.19: A simple differencing (or "velocity") filter has coefficients $b[0] = 1, b[1] = -1$, or written in Matlab, `b = [1 -1]`. (Note the order: the coefficient that goes with the most recent input is on the left.) A differencing filter responds more strongly to signals with larger slopes (i.e., higher frequency signals) and has zero response to constant (DC) signals. Usually the signal "velocities" we are interested in, though, are those at low frequency; higher-frequency signals tend to come from sensing noise. Thus a better filter is probably a differencing filter convolved with a low-pass filter. **Left**: `b1 = [1 -1]; b2 = conv(b1,fir1(12,0.2)); freqz(b1); hold on; freqz(b2)`. This plot shows the frequency response of the differencing filter, as well as a differencing filter convolved with a 12th-order FIR LPF with cutoff at $0.2f_N$. At low frequencies, where the signals we are interested in live, the two filters have the same response. At high frequencies, the simple differencing filter has a large (unwanted) response, while the other filter attenuates this noise. **Middle**: `plot(conv(b1,x))`. Zoomed in on the signal filtered by the simple difference filter. **Right**: `plot(conv(b2,x))`. Zoomed in on the signal filtered by the difference-plus-LPF.

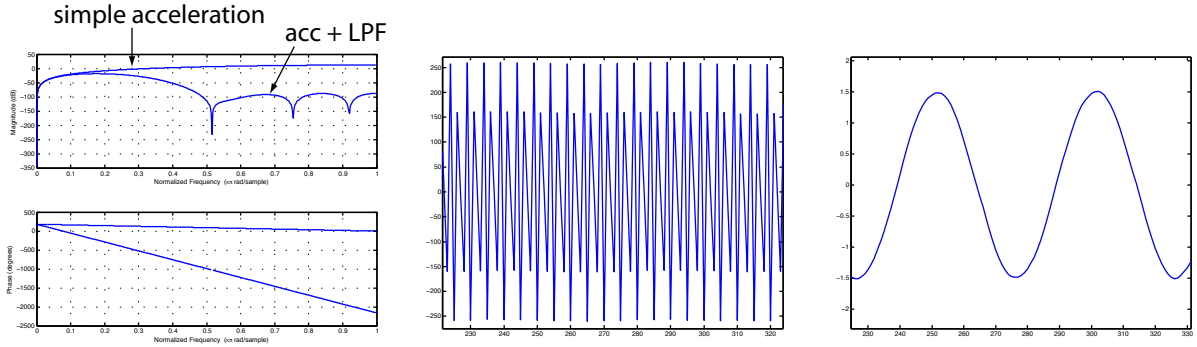


Figure 15.20: We can also make a double-differencing (or "acceleration") filter by taking the difference of two consecutive difference samples, i.e., convolving two differencing filters. This gives a simple filter with coefficients [1 -2 1]. This filter amplifies high frequency noise even more than a differencing filter. A better choice would be to use a filter that is the convolution of two difference-plus-low-pass filters from the previous example. **Left**: `bvel=[1 -1]; bacc=conv(bvel,bvel); bvellpf=conv(bvel,fir1(12,0.2)); bacclpf=conv(bvellpf,bvellpf); freqz(bacc); hold on; freqz(bacclpf)`. The low-frequency response of the two filters is identical, while the low-pass version attenuates high frequency noise. **Middle**: `plot(conv(bacc,x))`. Zoomed in on the second derivative of the signal, according to the simple acceleration filter. **Right**: `plot(conv(bacclpf,x))`. Zoomed in on the second derivative of the signal, according to the low-passed version of the acceleration filter.

15.4 Infinite Impulse Response (IIR) Digital Filters

The class of infinite impulse response (IIR) filters generalizes FIR filters to the following form:

$$\sum_{i=0}^{Q} a_i z(k-i) = \sum_{j=0}^{P} b_j x(k-j),$$

or, written in a more useful form for us,

$$z(k) = \frac{1}{a_0}\left(\sum_{j=0}^{P} b_j x(k-j) - \sum_{i=1}^{Q} a_i z(k-i)\right). \tag{15.3}$$

The response of an IIR filter to an impulse input may never die out, unlike the FIR filter. Some other differences between FIR and IIR filters are highlighted below:

- IIR filters may be *unstable*, that is, their output may grow unbounded even if the input is bounded. This is not possible with FIR filters.
- IIR filters often use many fewer coefficients to achieve the same magnitude response transition sharpness. Hence they can be more computationally efficient than FIR filters.
- IIR filters generally have a nonlinear phase response (phase does not change linearly with frequency, as with FIR filters). This may or may not be OK, depending on the application. A linear phase response ensures that the time (not phase) delay associated with signals at all frequencies is the same. A nonlinear phase response, on the other hand, may cause different time delays at different frequencies. This may result in unacceptable distortion in an audio application, for example.

Because of roundoff errors in computation, an IIR filter that is theoretically stable may be unstable when implemented directly in the form of Equation (15.3). Because of the possibility of instability, IIR filters with many coefficients are usually implemented as a cascade of filters with $P = 2$ and $Q = 2$. It is relatively easy to ensure that these low-order filters are stable, ensuring the stability of the cascade of filters.

Popular IIR filters include Chebyshev and Butterworth filters, which include low-pass, high-pass, bandpass, and bandstop versions. Matlab offers design tools for these filters. Given a set of coefficients `b` and `a` defining the IIR filter, the Matlab command `filter(b,a,signal)` returns the filtered version of `signal`.

Perhaps the simplest IIR filter is the integrator

```
z(k) = z(k-1) + x(k)*Ts
```

where `Ts` is the sample time. The coefficients are `a = [1 -1]` and `b = [Ts]`. The behavior of the integrator on the sample signal in Figure 15.13 is shown in Figure 15.21.

15.5 FFT-based Filters

One more option for filtering signals is to first FFT the signal, then set certain frequency components of the signal to zero, then do an inverse FFT. Let's say we're working with the 256-sample square wave we looked at in Section 15.2.1, and we want to extract only the component at $0.1f_N$. First let's build the signal and FFT it:

```
x = 0; x(1:10) = 2; x(11:20)= 0; x = [x x x x x x x x x x 2*ones(1,10) zeros(1,6)];
N = length(x);
Y = fft(x);
```

The element `Y(1)` is the DC component, `Y(2)` and `Y(256)` correspond to frequency f_s/N, `Y(3)` and `Y(255)` correspond to frequency $2f_s/N$, `Y(4)` and `Y(254)` correspond to frequency $3f_s/N$, etc., until `Y(129)` corresponds to frequency $128f_s/N = f_s/2 = f_N$. So the frequencies we care about are near index 14 and its counterpart $258 - 14 = 244$. To cancel other frequencies, we can do

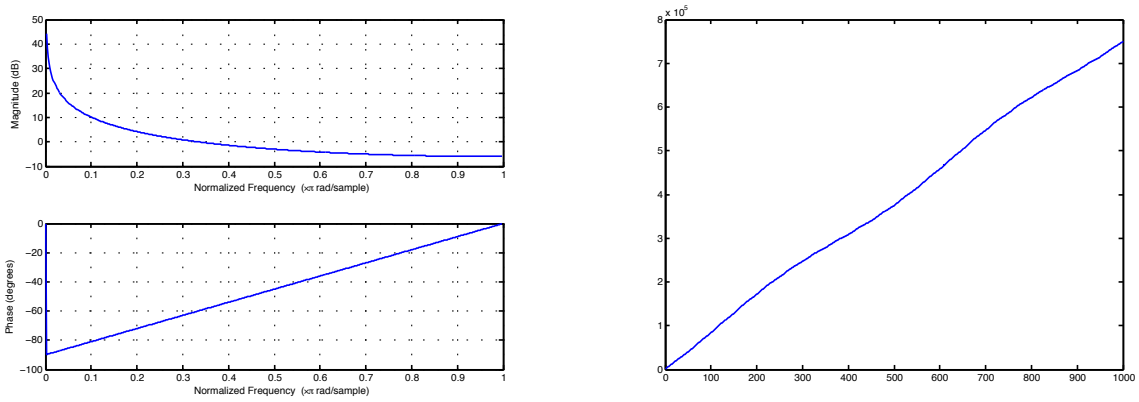


Figure 15.21: **Left**: `a=[1 -1]; b=[1]; freqz(b,a)`. Note that the frequency response of the integrator is infinite to DC signals (the integral of a nonzero constant signal goes to infinity) and low for high frequency signals. This is opposite of the differencing filter. **Right**: `plot(filter(b,a,x))`. The `filter` command applies the filter with coefficients `b` and `a` to `x`. This generalizes `conv` to IIR filters. (We can't simply use `conv` for IIR filters, since the impulse response is not finite.) The upward slope of the integral is due to the nonzero DC term. We can also see the wiggle due to the 2 Hz term. It is basically impossible to see the 20 Hz and 400 Hz terms in the signal.

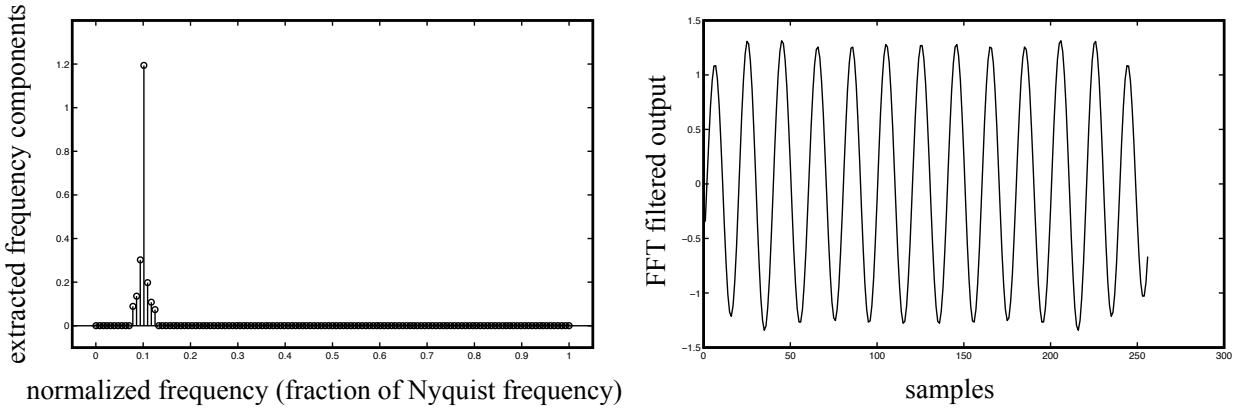


Figure 15.22: (Left) The extracted frequencies from the FFT magnitude plot in Figure 15.6. (Right) The FFT filter output, which is approximately a sinusoid at $0.1 f_N$.

```
halfwidth = 3;
Yfiltered = zeros(1,256);
Yfiltered(14-halfwidth:14+halfwidth) = Y(14-halfwidth:14+halfwidth);
Yfiltered(244-halfwidth:244+halfwidth) = Y(244-halfwidth:244+halfwidth);
xrecovered = fft(conj(Yfiltered)/N);
plot(real(xrecovered));
```

The result is the (approximate) sinusoidal component of the square wave at $0.1 f_N$, shown in Figure 15.22.

This is simple! (Of course the cost is in computing the FFT and inverse FFT.) FFT-based filter design tools allow you to specify an arbitrary frequency response (e.g., by drawing the magnitude response) and the size of the filter you are willing to accept, then use an FFT to find filter coefficients that best match the desired response. The more coefficients you allow, the closer the approximation. In Matlab, you can explore the use of `design`, `fdesign`, and `fdatool`.

A Crash Course in C, v1.0
DRAFT
Kevin Lynch
November 2012

Contents

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 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 C compiler, and a text editor. 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 or Cygwin (google them).[1] 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 well under 1 gigabyte and is more than sufficient for getting started with C.

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, the command line can be accessed through the `Start` button and searching for `cmd` or `command prompt`.

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);
}
```

[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.

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. Whatever 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 preceding `./` just indicates that `HelloWorld` sits in the current directory. The response should be

```
Hello world!
>
```

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:

- **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.

- **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*.

- **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 represents 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 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.

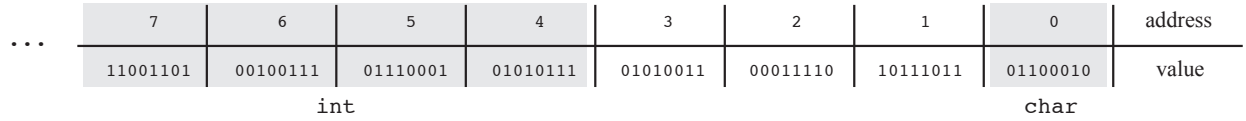
An example showing the first eight addresses in memory is shown below.

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

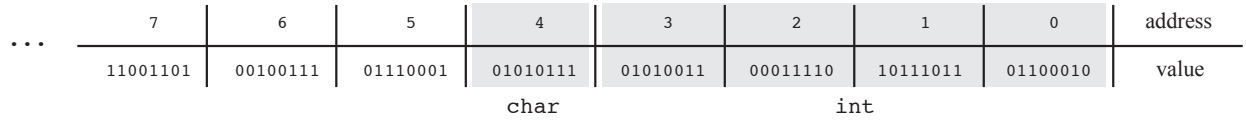
Now assume that the byte at address 4 is part of the representation of a variable. Do these 0's and 1's represent an integer, or part of an integer? A number with a fractional component? Something else?

[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."

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



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 and correct for the PIC32. 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;
```

[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.

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 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`, 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.

Using the data types. If your program calls for floating point calculations, you can choose between `float` and `double` data types. The advantages of `float`s are that they use less memory and

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

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

computations on `float`s (e.g., multiplies, square roots, etc.) are generally faster. The advantage of `double`s 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, 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 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 number is given below:

[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.

binary	signed base-10	unsigned 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.[5]

`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 $-2^n + 2$ to $2^n - 1$, where n and m depend on whether the type is `float`/`double`/`long double`. 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.

A.3.2 Memory, Addresses, and Pointers

Consider the following C syntax:

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

or equivalently

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

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

[5] 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. These phrases come from *Gulliver's Travels*, and the convention used depends on the processor.

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
```

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 memory that you have not allocated. The compiler is unlikely to recognize this during compilation, and you may end up with a "segmentation fault" when you execute the code.[6] 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:

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.

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

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 is not linked by default like other libraries are.

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 most 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';
  printf("Formatted string:  i = %4d  c = '%c'\n",i,c);
  printf("f = %25.23f  d = %25.23lf\n",f,d);
  return(0);
}
```

The `23lf` in the last `printf` statement is "twenty-three 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
```

On my laptop, here is the output:

```
Formatted string:  i =   32  c = 'k'
f = 4.27799987792968750000000  d = 4.27799999999999958077979
```

The main point of this program is to demonstrate formatted output from the code

```
printf("Formatted string:  i = %4d  c = %c\n",i,c);
printf("f = %25.23f  d = %25.23lf\n",f,d);
```

Inside a `printf` statement, everything inside the double quotes is printed to the screen, though 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 `%25.23f` means that a `float` will be printed right-justified over 25 spaces with 23 spaces after the decimal point. The `%25.23lf` means that a `double` (or "long float") will be printed right-justified over 25 spaces, with 23 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) {
  unsigned char iu = 100, ju = 240, sumu;
  signed char is = 100, js = 240, sums;
  char i = 100, j = 240, sum;

  sum = i+j; sumu = iu+ju; sums = is+js;
  printf("unsigned char:  %3d + %3d = %3d or ASCII character %c\n",iu,ju,sumu,sumu);
  printf("signed char:    %3d + %3d = %3d or ASCII character %c\n",is,js,sums,sums);
  printf("char:           %3d + %3d = %3d or ASCII character %c\n",i,j,sum,sum);
  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

```
unsigned char:  100 + 240 =  84 or ASCII character T
signed char:    100 + -16 =  84 or ASCII character T
char:           100 + -16 =  84 or ASCII character 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`.[7] 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

```
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.

[7] 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.

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 `double`.

We will see more on use of parentheses (Section A.4.1), scopes 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  *
15  * HISTORY:
16  * Dec 20, 2011   Created by Kevin Lynch
17  * Jan 4, 2012    Modified by Kevin Lynch (small changes, altered comments)
18  */
19
20 /*****************************************************************************
21  * 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

/******************************************************************************
 * 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
```

```
76  * value over the years, given the parameters in *invp.
77  */
78 void calculateGrowth(Investment *invp) {
79
80   int i;
81
82   // for loop ==SecA.4.14==
83   for (i=1; i <= invp->years; i=i+1) {   // relational operators ==SecA.4.10==
84                                          // struct access ==SecA.4.4==
85     invp->invarray[i] = invp->growth * invp->invarray[i-1];
86   }
87 } // ***** END calculateGrowth *****
88
89
90 /* getUserInput
91  *
92  * This reads the user's input into the struct pointed at by invp,
93  * and returns TRUE (1) if the input is valid, FALSE (0) if not.
94  */
95 int getUserInput(Investment *invp) {
96
97   int valid;      // int used as a boolean ==SecA.4.10==
98
99   // I/O functions in stdio.h ==SecA.4.15==
100  printf("Enter investment, growth rate, number of yrs (up to %d): ",MAX_YEARS);
101  scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));
102
103  // logical operators ==SecA.4.11==
104  valid = (invp->inv0 > 0) && (invp->growth > 0) &&
105    (invp->years > 0) && (invp->years <= MAX_YEARS);
106  printf("Valid input?  %d\n",valid);
107
108  // if-else ==SecA.4.13==
109  if (!valid) { // ! is logical NOT ==SecA.4.11==
110    printf("Invalid input; exiting.\n");
111  }
112  return(valid);
113 } // ***** END getUserInput *****
114
115
116 /* sendOutput
117  *
118  * This function takes the array of investment values (a pointer to the first
119  * element, which is a double) and the number of years (an int).  We could
120  * have just passed a pointer to the entire investment record, but we decided
121  * to demonstrate some different syntax.
122  */
123 void sendOutput(double *arr, int yrs) {
124
125  int i;
126  char outstring[100];  // defining a string ==SecA.4.9==
127
128  printf("\nRESULTS:\n\n");
129  for (i=0; i<=yrs; i++) {  // ++, +=, math in ==SecA.4.7==
```

```
130     sprintf(outstring,"Year %3d:  %10.2f\n",i,arr[i]);
131     printf("%s",outstring);
132   }
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 this example, 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 `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.

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 will be made available to us, as well as a set of function prototypes for input and output functions.

The preprocessor will replace our `#include <stdio.h>` command with the header file `stdio.h`.[8] 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 containing `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 macros that you may use in more than one place in your program. Here are two examples:

```
#define RAD_TO_DEG(x) ((x) * 57.29578)
#define MAX(A,B)      ((A) > (B) ? (A):(B))
```

The first macro is used to convert radians to degrees. 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

```
angle_deg = ((angle_rad) * 57.29578);
```

The second `#define` macro is used to return 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));
```

There are other uses of the preprocessor, but we won't go into them.

[8] 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>`.

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 `double`s. (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 `double`s, the `int`, and the array of `double`s. 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. A global variable is one that is defined in the `GLOBAL VARIABLES` section, before any function definition. Such variables can be referred to or altered in any function. Because of this special status, global variables are often Capitalized. Global variable usage should be minimized for program modularity and readability.

A local variable is one that is defined in a function. Such a variable is only usable inside that function. 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;
}
```

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 of `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 that extra space. 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.[9] With the C library `stdlib.h` included using the preprocessor command `#include <stdlib.h>`, the syntax

[9] Dynamic memory is allocated from the *heap*, a portion of memory set aside for dynamic allocation (and therefore is not available for global variables, local variables, and program code). You may have to adjust linker options setting the size of the heap.

```
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.[10] 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.");
```

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.

[10] 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.

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

$\sim$	bitwise NOT
`&`	bitwise AND
`|`	bitwise OR
$\wedge$	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.[11] 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`.

[11] 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 integer. Corresponding argument should be an integer data type.
- `%ld` Print a long integer. Corresponding argument should be a `long`.
- `%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 integer 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
```

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` (32,767 on the PIC32 and 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 Breaking a Program into Multiple Files

Often the same set of functions is useful for a number of different programs. For example, you could have helper files for mathematical calculations, graphics functions, user interfaces, etc., and use these over and over again in different programs. In this case, it would make sense to put these functions in their own "helper" C file, without a `main` function. When these functions are needed for a particular program with a `main` function, you just compile the `main` file and its helper files and link them into a single executable using a command such as

```
gcc main.c helper1.c helper2.c -o myprog
```

Each of the C files is compiled individually, and then their object codes are linked to make the final executable. Since the `main.c` file does not contain the helper functions, it at least needs prototypes of these functions so that the `main` object code can be created.[12] These prototypes are provided by header files associated with the helper C files. (Header files we have referred to previously include `stdio.h` and `math.h`, associated with the stdio and math libraries.) Typically each helper C file will have its own header file consisting of function prototypes and constants or macros that should be available to other files. These header files are `#include`d at the beginning of the `main.c` file. The preprocessor replaces the `#include` commands with the text of the `helper1.h` and `helper2.h` header files, providing the function prototypes that `main.c` needs.

Below is an example consisting of a `main.c` file, a `helper.c` file, and a `helper.h` file. `helper.c` has three functions, one of which is purely for internal use and two of which are meant to be used in other files. Therefore the header file `helper.h` provides prototypes for those two functions. `helper.h` also provides a definition of the constant `PI`. The final file is `main.c` which needs access to the helper functions and the constant `PI`. To get this access, it includes `helper.h` in its preprocessor commands. The file `helper.c` *also* includes `helper.h`, since it needs the constant `PI`.

```
// ***** helper.h *****

#ifndef HELPER_H  // "include guard"; don't include again if included already
#define HELPER_H  // second line of the "include guard"

#define PI 3.1415926
double radius2Volume(double r);    // function prototype
double radius2Surface(double r);   // function prototype

#endif            // third line, and end, of the "include guard"
```

[12] If, after compilation, the linker cannot find the required helper functions in the object code of any of the C files, the linker will report an error.

```
// ***** helper.c *****

#include <math.h>
#include "helper.h"  // if the file is in the same directory, enclose in "quotes"

double cuber(double x) {   // this function is not available externally
  return(pow(x,3.0));
}

double radius2Volume(double rad) {
  return((4.0/3.0)*PI*cuber(rad));
}

double radius2Surface(double rad) {
  return(4.0*PI*rad*rad);
}


// ***** main.c *****

#include <stdio.h>
#include "helper.h"

int main(void) {
  double radius = 3.0;
  printf("Pi is approximated as %25.23lf.\n",PI);
  printf("The surface area of the sphere is %8.4f.\n",radius2Surface(radius));
  printf("The volume of the sphere is %8.4f.\n",radius2Volume(radius));
  return 0;
}
```

Note the three lines making up the *include guard* in `helper.h`. During preprocessing of a C file, if `helper.h` is included, the flag (macro) `HELPER_H` is defined. If the same C file tries to include `helper.h` again, the include guard will recognize that `HELPER_H` already exists and therefore skip the prototypes and constant definition. Without include guards, if we wrote a `.c` file including both `header1.h` and `header2.h`, not knowing that `header2.h` already includes `header1.h`, we would get a compilation error due to duplicate declarations.

When we compile our source files (using `gcc helper.c main.c -o main`) and run, we get the output

```
Pi is approximated as 3.14159260000000006840537.
The surface area of the sphere is 113.0973.
The volume of the sphere is 113.0973.
```

Much more can be said about breaking a program into multiple files, the use of `makefile`s, etc., but we will stop here.

A.5 Problems

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 can be represented with a single byte. Now consider the code

    ```
    unsigned int i, j, *kp, *np;
    ```

    Let's assume that `i` occupies addresses A0–A3, `j` occupies A4–A7, `kp` is at A8, and `np` is at A9. The code continues as follows:

```
kp = &i;     // (a)
j = *kp;     // (b)
i = 0xAE;    // (c)
np = kp;     // (d)
*np = 0x12;  // (e)
j = *kp;     // (f)
```

For each of the comments (a)-(f) above, give the contents (in hexadecimal) at the address ranges A0–A3 (the `unsigned int i`), A4–A7 (the `unsigned int j`), A8 (the pointer `kp`), and A9 (the pointer `np`), at that point in the program, after executing the line containing the comment. If the contents are undefined, say so. (If it matters, you can 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 8 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`; (d) `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 space 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.

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.

    (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.

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 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);
  // put nested loops here to put the string in sorted order
  printResult(str);
  return(0);
}

// helper prototypes 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 blank space.

```
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.

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 descending order. Modify your bubble sort program to do this, taking name and score information until the first letter of the name entered is `0` (the number 0).

35. 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 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?

36. 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 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.

37. 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?

38. 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 should follow the template outlined below.

```
/*************************************************************************
 * 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
```